# Leaspy

***Release 1.2.0***

**Igor Koval, Raphael Couronne, Etienne Maheux, Arnaud Valladier,**

**Dec 18, 2021**

# GETTING STARTED


leaspy

CHAPTER

ONE

# INSTALLATION

## 1.1 Package installation

1. Leaspy requires Python >= 3.7
2. Create a dedicated environment (optional):

Using `conda`:

```
conda create --name leaspy python=3.7
conda activate leaspy
```

Or using `pyenv`:

```
pyenv virtualenv leaspy
pyenv local leaspy
```

3. Install `leaspy` with `pip`:

```
pip install leaspy
```

It will automatically install all needed dependencies.

## 1.2 Notebook configuration

After installation, you can run the examples in *Leaspy in a nutshell* and in *the Leaspy API*.
To do so, in your `leaspy` environment, you can download `ipykernel` to use `leaspy` with `jupyter` notebooks

```
conda install ipykernel
python -m ipykernel install --user --name=leaspy
```

Now, you can open `jupyter lab` or `jupyter notebook` and select the `leaspy` kernel.

CHAPTER

TWO

# LEASPY IN A NUTSHELL

## 2.1 Comprehensive example

We first load synthetic data from the *leaspy.datasets* to get of a grasp of longitudinal data.

```
>>> from leaspy import AlgorithmSettings, Data, Leaspy
>>> from leaspy.datasets import Loader
>>> alzheimer_df = Loader.load_dataset('alzheimer-multivariate')
>>> print(alzheimer_df.columns)
Index(['E-Cog Subject', 'E-Cog Study-partner', 'MMSE', 'RAVLT', 'FAQ',
       'FDG PET', 'Hippocampus volume ratio'], dtype='object')
>>> alzheimer_df = alzheimer_df[['MMSE', 'RAVLT', 'FAQ', 'FDG PET']]
>>> print(alzheimer_df.head())
                    MMSE      RAVLT     FAQ       FDG PET
ID      TIME
GS-001  73.973183  0.111998  0.510524  0.178827  0.454605
        74.573181  0.029991  0.749223  0.181327  0.450064
        75.173180  0.121922  0.779680  0.026179  0.662006
        75.773186  0.092102  0.649391  0.156153  0.585949
        75.973183  0.203874  0.612311  0.320484  0.634809
```

The data correspond to repeated visits (*TIME* index) of different participants (*ID* index). Each visit corresponds to the measurement of 4 different variables : the MMSE, the RAVLT, the FAQ and the FDG PET.

If plotted, the data would look like the following:

where each color corresponds to a variable, and the connected dots corresponds to the repeated visits of a single participant.

Not very engaging, right ? To go a step further, let's first encapsulate the data into the main *leaspy Data container*.

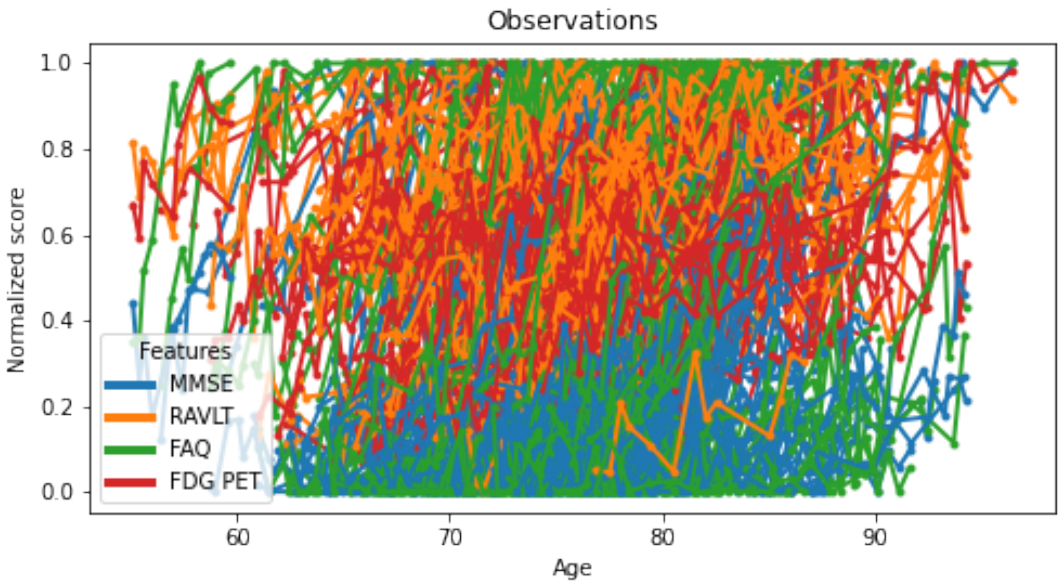
```
>>> data = Data.from_dataframe(alzheimer_df)
```

Leaspy core functionality is to estimate the group-average trajectory of the different variables that are measured in a population. Let's initialize the leaspy object

```
>>> leaspy_logistic = Leaspy('logistic', source_dimension=2)
```

as well as the algorithm needed to estimate the group-average trajectory:

```
>>> fit_settings = AlgorithmSettings('mcmc_saem', seed=0, n_iter=8000)
```




We then use the *Leaspy.fit* method to estimate the group average trajectory:

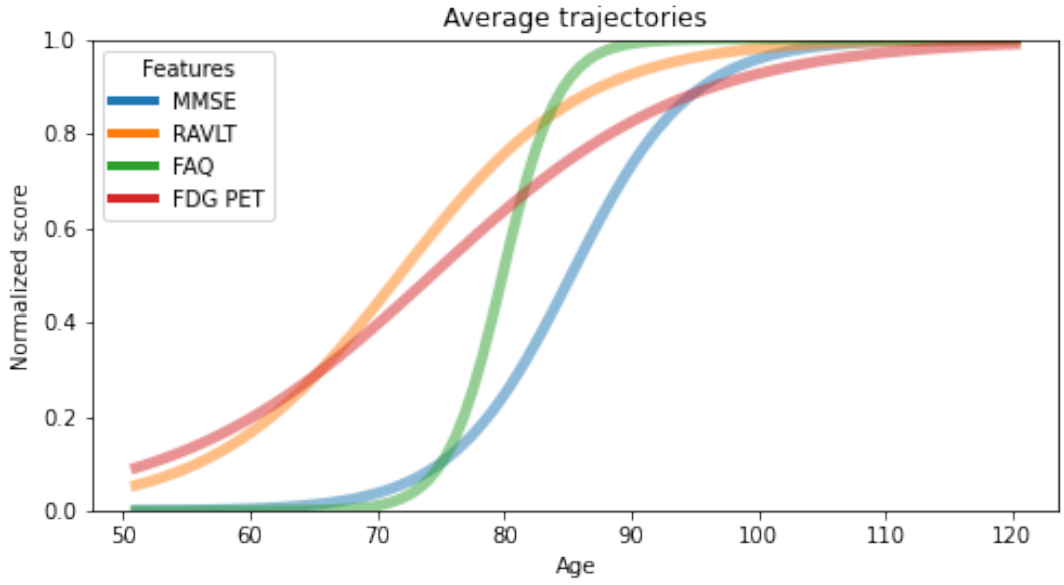
```
>>> leaspy_logistic.fit(data, fit_settings)
==> Setting seed to 0
|##################################################|   8000/8000 iterations
Fit with `mcmc_saem` took: 6m 57s
The standard deviation of the noise at the end of the fit is:
MMSE: 6.50%
RAVLT: 7.63%
FAQ: 6.67%
FDG PET: 7.87%
```

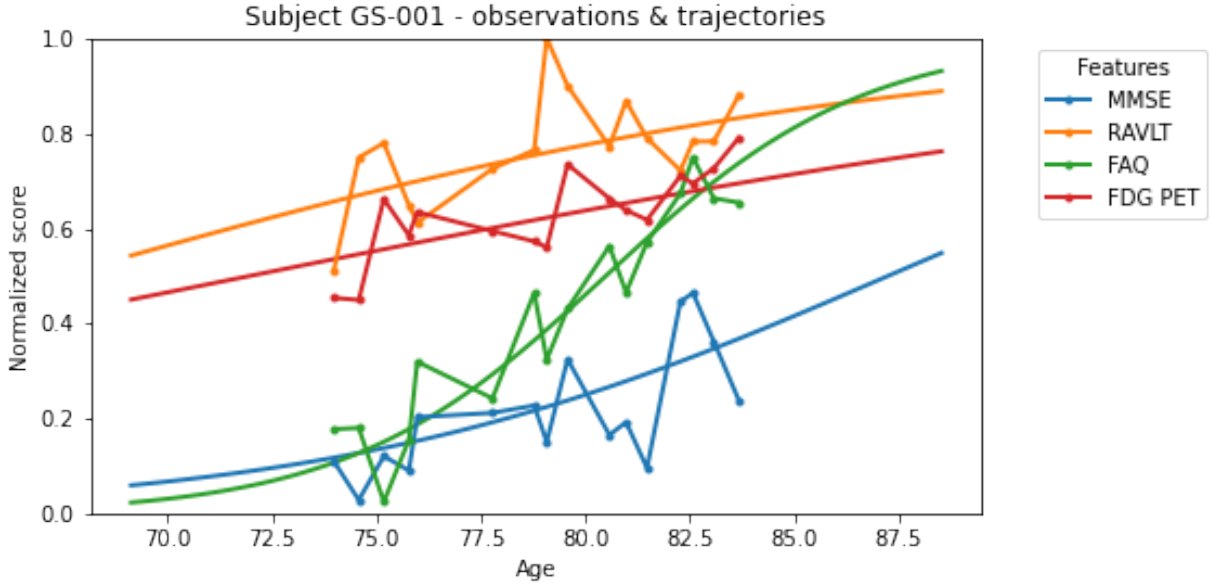
If we were to plot the measured average progression of the variables - see started example notebook for details - it would look like the following

We can also derive the individual trajectory of each subject. To do this, we use the *Leaspy.personalize* method, again by providing the proper settings.

```
>>> personalize_settings = AlgorithmSettings('scipy_minimize', seed=0)
>>> individual_parameters = leaspy_logistic.personalize(data, personalize_settings)
 ==> Setting seed to 0
|##################################################|   200/200 subjects
Personalize with `scipy_minimize` took: 9s
The standard deviation of the noise at the end of the personalize is:
MMSE: 6.32%
RAVLT: 7.27%
FAQ: 6.29%
FDG PET: 7.49%
```

Plotting the input participant data against its personalization would give the following - see started example notebook for details.


Average trajectories
Features
MMSE
RAVLT
FAQ
FDG PET
Normalized score
Age
1.0
0.8
0.6
0.4
0.2
0.0
50
60
70
80
90
100
110
120


Subject GS-001 - observations & trajectories
Features
MMSE
RAVLT
FAQ
FDG PET
Normalized score
Age
1.0
0.8
0.6
0.4
0.2
0.0
70.0
72.5
75.0
77.5
80.0
82.5
85.0
87.5

## 2.2 Using my own data

### 2.2.1 Data format

*Leaspy* uses its own data container. To use it properly, you need to provide a *csv* file or a *pandas.DataFrame* in the right format. Let's have a look at the data used in the previous example:

```
>>> print(alzheimer_df.head())
                    MMSE      RAVLT     FAQ       FDG PET
ID      TIME
GS-001  73.973183  0.111998  0.510524  0.178827  0.454605
        74.573181  0.029991  0.749223  0.181327  0.450064
        75.173180  0.121922  0.779680  0.026179  0.662006
        75.773186  0.092102  0.649391  0.156153  0.585949
        75.973183  0.203874  0.612311  0.320484  0.634809
```

You **MUST** have *ID* and *TIME*, either in index or in the columns. The other columns must be the observed variables (also named *features* or *endpoints*). In this fashion, you have one column per *feature* and one line per *visit*.

### 2.2.2 Data scale & constraints

*Leaspy* uses *linear* and *logistic* models. The features **MUST** be increasing with time. For the *logistic* model, you need to rescale your data between 0 and 1.

### 2.2.3 Missing data

*Leaspy* automatically handles missing data as long as they are encoded as `nan` in your *pandas.DataFrame*, or as empty values in your *csv* file.

## 2.3 Going further

You can check the *User guide* and the *full API documentation*. You can also dive into the started example of the Leaspy repository. The Disease Progression Modelling website also hosts a mathematical introduction and tutorials for Leaspy.

CHAPTER

# THREE

# API DOCUMENTATION

Full API documentation of the *Leaspy* Python package.

## 3.1 `leaspy.api`: Main API

The main class, from which you can instantiate and calibrate a model, personalize it to a given set a subjects, estimate trajectories and simulate synthetic data.

| | |
|---|---|
| *Leaspy*(model_name, **kwargs) | Main API used to fit models, run algorithms and simulations. |

### 3.1.1 `leaspy.api`.Leaspy

**class Leaspy**(*model_name: str*, ***kwargs*)

Bases: `object`

Main API used to fit models, run algorithms and simulations. This is the main class of the Leaspy package.

**Parameters**

**model_name** [str] The name of the model that will be used for the computations. The available models are:

- `'logistic'` - suppose that every modality follow a logistic curve across time.
- `'logistic_parallel'` - idem & suppose also that every modality have the same slope at inflexion point
- `'linear'` - suppose that every modality follow a linear curve across time.
- `'univariate_logistic'` - a 'logistic' model for a single modality.
- `'univariate_linear'` - idem with a 'linear' model.
- `'constant'` - benchmark model for constant predictions.
- `'lme'` - benchmark model for classical linear mixed-effects model.

****kwargs** Keyword arguments directly passed to the model for its initialization (through *ModelFactory.model()*). Refer to the corresponding model to know possible arguments.

**noise_model** [str] *For manifold-like models*. Define the noise structure of the model, can be either:

- `'gaussian_scalar'`: gaussian error, with same standard deviation for all features

- `'gaussian_diagonal'`: gaussian error, with one standard deviation parameter per feature (default)
- `'bernoulli'`: for binary data (Bernoulli realization)

**source_dimension** [int, optional] *For multivariate models only*. Set the degrees of freedom for _spatial_ variability. This number MUST BE strictly lower than the number of features. By default, this number is equal to square root of the number of features. One can interpret this hyperparameter as a way to reduce the dimension of inter-individual _spatial_ variability between progressions.

**See also:**

`leaspy.models`

**Attributes**

**model** [*AbstractModel*] Model used for computations, is an instance of *AbstractModel*.

**type** [str (read-only)] Name of the model - will be one of the names listed above.

## Methods

| | |
|---|---|
| *calibrate*(data, settings) | Duplicates of the *fit()* method. |
| *check_if_initialized*() | Check if model is initialized. |
| *estimate*(timepoints, individual_parameters, *) | Return the model values for individuals characterized by their individual parameters $z_i$ at time-points $(t_{i,j})_j$. |
| *estimate_ages_from_biomarker_values*(...[, ...]) | For individuals characterized by their individual parameters $z_i$, returns the age $t_{i,j}$ at which a given feature value $y_{i,j,k}$ is reached. |
| *fit*(data, settings) | Estimate the model's parameters $\theta$ for a given dataset and a given algorithm. |
| *load*(path_to_model_settings) | Instantiate a Leaspy object from json model parameter file or the corresponding dictionary. |
| *personalize*(data, settings, *[, return_noise]) | From a model, estimate individual parameters for each *ID* of a given dataset. |
| *save*(path, **kwargs) | Save Leaspy object as json model parameter file. |
| *simulate*(individual_parameters, data, settings) | Generate longitudinal synthetic patients data from a given model, a given collection of individual parameters and some given settings. |

`calibrate`(*data: Data*, *settings: AlgorithmSettings*)
Duplicates of the *fit()* method.

`check_if_initialized`()
Check if model is initialized.

**Raises**

**LeaspyInputError** Raise an error if the model has not been initialized.

`estimate`(*timepoints: Union[pd.MultiIndex, Dict[IDType, List[float]]]*, *individual_parameters: IndividualParameters*, *, *to_dataframe: bool = None*) → Union[pd.DataFrame, Dict[IDType, np.ndarray]]
Return the model values for individuals characterized by their individual parameters $z_i$ at time-points

$(t_{i,j})_j$.

**Parameters**

**timepoints** [dictionary {string/int: array_like[numeric]} or pandas.MultiIndex] Contains, for each individual, the time-points to estimate. It can be a unique time-point or a list of time-points.

**individual_parameters** [*IndividualParameters*] Corresponds to the individual parameters of individuals.

**to_dataframe** [bool or None (default)] Whether to output a dataframe of estimations? If None: default is to be True if and only if timepoints is a *pandas.MultiIndex*

**Returns**

**individual_trajectory** [pandas.DataFrame or dict (depending on *to_dataframe* flag)] Key: patient indices. Value: numpy.ndarray of the estimated value, in the shape (number of timepoints, number of features)

### Examples

Given the individual parameters of two subjects, estimate the features of the first at 70, 74 and 80 years old and at 71 and 72 years old for the second.

```
>>> from leaspy.datasets import Loader
>>> leaspy_logistic = Loader.load_leaspy_instance('parkinson-putamen-train')
>>> individual_parameters = Loader.load_individual_parameters('parkinson-
↪putamen-train')
>>> timepoints = {'GS-001': (70, 74, 80), 'GS-002': (71, 72)}
>>> estimations = leaspy_logistic.estimate(timepoints, individual_parameters)
```

**estimate_ages_from_biomarker_values**(*individual_parameters:* IndividualParameters, *biomarker_values: Dict[str, Union[List[float], float]]*, *feature: Optional[str] = None*) → Dict[str, Union[List[float], float]]

For individuals characterized by their individual parameters $z_i$, returns the age $t_{i,j}$ at which a given feature value $y_{i,j,k}$ is reached.

**Parameters**

**individual_parameters** [*IndividualParameters*] Corresponds to the individual parameters of individuals.

**biomarker_values** [Dict[Union[str, int], Union[List, float]]] Dictionary that associates to each patient (being a key of the dictionary) a value (float between 0 and 1, or a list of such floats) from which leaspy will estimate the age at which the value is reached.

**feature** [str] For multivariate models only: feature name (indicates to which model feature the biomarker values belongs)

**Returns**

**biomarker_ages :** Dictionary that associates to each patient (being a key of the dictionary) the corresponding age (or ages) for which the value(s) from biomarker_values have been reached. Same format as biomarker values.

**Raises**

**LeaspyTypeError** bad types for input

**LeaspyInputError** inconsistent inputs

### Examples

Given the individual parameters of two subjects, and the feature value of 0.2 for the first and 0.5 and 0.6 for the second, get the corresponding estimated ages at which these values will be reached.

```
>>> from leaspy.datasets import Loader
>>> leaspy_logistic = Loader.load_leaspy_instance('parkinson-putamen-train')
>>> individual_parameters = Loader.load_individual_parameters('parkinson-
↪putamen-train')
>>> biomarker_values = {'GS-001': [0.2], 'GS-002': [0.5, 0.6]}
# Here the 'feature' argument is optional, as the model is univariate
>>> estimated_ages = leaspy_logistic.estimate_ages_from_biomarker_
↪values(individual_parameters, biomarker_values,
>>> feature='PUTAMEN')
```

**fit**(*data: Data*, *settings: AlgorithmSettings*)

Estimate the model's parameters $\theta$ for a given dataset and a given algorithm.

These model's parameters correspond to the fixed-effects of the mixed-effects model.

**Parameters**

**data** [*Data*] Contains the information of the individuals, in particular the time-points $(t_{i,j})$ and the observations $(y_{i,j})$.

**settings** [*AlgorithmSettings*] Contains the algorithm's settings.

### Examples

Fit a logistic model on a longitudinal dataset, display the group parameters

```
>>> from leaspy import AlgorithmSettings, Data, Leaspy
>>> from leaspy.datasets import Loader
>>> putamen_df = Loader.load_dataset('parkinson-putamen')
>>> data = Data.from_dataframe(putamen_df)
>>> leaspy_logistic = Leaspy('univariate_logistic')
>>> settings = AlgorithmSettings('mcmc_saem', progress_bar=True, seed=0)
>>> leaspy_logistic.fit(data, settings)
 ==> Setting seed to 0
|##################################################|   10000/10000 iterations
The standard deviation of the noise at the end of the calibration is:
0.0213
Calibration took: 30s
>>> print(str(leaspy_logistic.model))
=== MODEL ===
g : tensor([-1.1744])
tau_mean : 68.56787872314453
tau_std : 10.12782096862793
xi_mean : -2.3396952152252197
xi_std : 0.5421289801597595
noise_std : 0.021265486255288124
```

**classmethod load**(*path_to_model_settings: str*)

Instantiate a Leaspy object from json model parameter file or the corresponding dictionary.

This function can be used to load a pre-trained model.

**Parameters**

**path_to_model_settings** [str or dict] Path to the model's settings json file or dictionary of model parameters

**Returns**

*Leaspy* An instanced Leaspy object with the given population parameters $\theta$.

### Examples

Load a univariate logistic pre-trained model.

```
>>> from leaspy import Leaspy
>>> from leaspy.datasets.loader import model_paths
>>> leaspy_logistic = Leaspy.load(model_paths['parkinson-putamen-train'])
>>> print(str(leaspy_logistic.model))
=== MODEL ===
g : tensor([-0.7901])
tau_mean : 64.18125915527344
tau_std : 10.199116706848145
xi_mean : -2.346343994140625
xi_std : 0.5663877129554749
noise_std : 0.021229960024356842
```

**personalize**(*data: Data*, *settings: AlgorithmSettings*, *, *return_noise: bool = False*)

From a model, estimate individual parameters for each *ID* of a given dataset. These individual parameters correspond to the random-effects $(z_{i,j})$ of the mixed-effects model.

**Parameters**

**data** [*Data*] Contains the information of the individuals, in particular the time-points $(t_{i,j})$ and the observations $(y_{i,j})$.

**settings** [*AlgorithmSettings*] Contains the algorithm's settings.

**return_noise** [bool (default False)] Returns a tuple (individual_parameters, noise_std) if True

**Returns**

**ips** [*IndividualParameters*] Contains individual parameters

**if return_noise is True** [tuple]

- ips : *IndividualParameters*
- noise_std : `torch.Tensor`

**Raises**

**LeaspyInputError** if model is not initialized.

### Examples

Compute the individual parameters for a given longitudinal dataset and calibrated model, then display the histogram of the log-acceleration:

```
>>> from leaspy import AlgorithmSettings, Data
>>> from leaspy.datasets import Loader
>>> leaspy_logistic = Loader.load_leaspy_instance('parkinson-putamen-train')
>>> putamen_df = Loader.load_dataset('parkinson-putamen')
>>> data = Data.from_dataframe(putamen_df)
>>> personalize_settings = AlgorithmSettings('scipy_minimize', progress_
↪bar=True, use_jacobian=True, seed=0)
>>> individual_parameters = leaspy_logistic.personalize(data, personalize_
↪settings)
 ==> Setting seed to 0
|##################################################|   200/200 subjects
The standard deviation of the noise at the end of the personalization is:
0.0191
Personalization scipy_minimize took: 5s
>>> ip_df = individual_parameters.to_dataframe()
>>> ip_df[['xi']].hist()
```

**save**(*path: str*, ***kwargs*)

Save Leaspy object as json model parameter file.

**Parameters**

**path** [str] Path to store the model's parameters.

****kwargs** Keyword arguments for *save()* (including those sent to `json.dump()` function).

### Examples

Load the univariate dataset `'parkinson-putamen'`, calibrate the model & save it:

```
>>> from leaspy import AlgorithmSettings, Data, Leaspy
>>> from leaspy.datasets import Loader
>>> putamen_df = Loader.load_dataset('parkinson-putamen')
>>> data = Data.from_dataframe(putamen_df)
>>> leaspy_logistic = Leaspy('univariate_logistic')
>>> settings = AlgorithmSettings('mcmc_saem', progress_bar=True, seed=0)
>>> leaspy_logistic.fit(data, settings)
 ==> Setting seed to 0
|##################################################|   10000/10000 iterations
The standard deviation of the noise at the end of the calibration is:
0.0213
Calibration took: 30s
>>> leaspy_logistic.save('leaspy-logistic-model_parameters-seed0.json')
```

**simulate**(*individual_parameters: IndividualParameters*, *data: Data*, *settings: AlgorithmSettings*)

Generate longitudinal synthetic patients data from a given model, a given collection of individual parameters and some given settings.

This procedure learn the joined distribution of the individual parameters and baseline age of the subjects present in `individual_parameters` and `data` respectively to sample new patients from this joined distribution. The model is used to compute for each patient their scores from the individual parameters. The

number of visits per patients is set in `settings['parameters']['mean_number_of_visits']` and `settings['parameters']['std_number_of_visits']` which are set by default to 6 and 3 respectively.

**Parameters**

**individual_parameters** [*IndividualParameters*] Contains the individual parameters.

**data** [*Data*] Data object

**settings** [*AlgorithmSettings*] Contains the algorithm's settings.

**Returns**

**simulated_data** [Result] Contains the generated individual parameters & the corresponding generated scores.

### Notes

To generate a new subject, first we estimate the joined distribution of the individual parameters and the reparametrized baseline ages. Then, we randomly pick a new point from this distribution, which define the individual parameters & baseline age of our new subjects. Then, we generate the timepoints following the baseline age. Then, from the model and the generated timepoints and individual parameters, we compute the corresponding values estimations. Then, we add some gaussian noise to these estimations. The level of noise is, by default, equal to the corresponding `'noise_std'` parameter of the model. You can choose to set your own noise value.

### Examples

Use a calibrated model & individual parameters to simulate new subjects similar to the ones you have:

```
>>> from leaspy import AlgorithmSettings, Data
>>> from leaspy.datasets import Loader
>>> putamen_df = Loader.load_dataset('parkinson-putamen-train_and_test')
>>> data = Data.from_dataframe(putamen_df.xs('train', level='SPLIT'))
>>> leaspy_logistic = Loader.load_leaspy_instance('parkinson-putamen-train')
>>> individual_parameters = Loader.load_individual_parameters('parkinson-
↪putamen-train')
>>> simulation_settings = AlgorithmSettings('simulation', seed=0)
>>> simulated_data = leaspy_logistic.simulate(individual_parameters, data,␣
↪simulation_settings)
 ==> Setting seed to 0
>>> print(simulated_data.data.to_dataframe().set_index(['ID', 'TIME']).head())
                                     PUTAMEN
ID                    TIME
Generated_subject_001 63.611107  0.556399
                      64.111107  0.571381
                      64.611107  0.586279
                      65.611107  0.615718
                      66.611107  0.644518
>>> print(simulated_data.get_dataframe_individual_parameters().tail())
                            tau        xi
ID
Generated_subject_096  46.771028 -2.483644
Generated_subject_097  73.189964 -2.513465
```

(continues on next page)

(continued from previous page)

```
Generated_subject_098  57.874967 -2.175362
Generated_subject_099  54.889400 -2.069300
Generated_subject_100  50.046972 -2.259841
```

By default, you have simulate 100 subjects, with an average number of visit at 6 & and standard deviation is the number of visits equal to 3. Let's say you want to simulate 200 subjects, everyone of them having ten visits exactly:

```
>>> simulation_settings = AlgorithmSettings('simulation', seed=0, number_of_
↪subjects=200, \
mean_number_of_visits=10, std_number_of_visits=0)
 ==> Setting seed to 0
>>> simulated_data = leaspy_logistic.simulate(individual_parameters, data,␣
↪simulation_settings)
>>> print(simulated_data.data.to_dataframe().set_index(['ID', 'TIME']).tail())
                                PUTAMEN
ID                    TIME
Generated_subject_200 72.119949  0.829185
                      73.119949  0.842113
                      74.119949  0.854271
                      75.119949  0.865680
                      76.119949  0.876363
```

By default, the generated subjects are named *'Generated_subject_001'*, *'Generated_subject_002'* and so on. Let's say you want a shorter name, for example *'GS-001'*. Furthermore, you want to set the level of noise around the subject trajectory when generating the observations:

```
>>> simulation_settings = AlgorithmSettings('simulation', seed=0, prefix='GS-',␣
↪noise=.2)
>>> simulated_data = leaspy_logistic.simulate(individual_parameters, data,␣
↪simulation_settings)
 ==> Setting seed to 0
>>> print(simulated_data.get_dataframe_individual_parameters().tail())
             tau        xi
ID
GS-096  46.771028 -2.483644
GS-097  73.189964 -2.513465
GS-098  57.874967 -2.175362
GS-099  54.889400 -2.069300
GS-100  50.046972 -2.259841
```

**class Leaspy**(*model_name: str*, ***kwargs*)

Main API used to fit models, run algorithms and simulations. This is the main class of the Leaspy package.

**Parameters**

**model_name** [str] The name of the model that will be used for the computations. The available models are:

- `'logistic'` - suppose that every modality follow a logistic curve across time.
- `'logistic_parallel'` - idem & suppose also that every modality have the same slope at inflexion point
- `'linear'` - suppose that every modality follow a linear curve across time.

- `'univariate_logistic'` - a 'logistic' model for a single modality.
- `'univariate_linear'` - idem with a 'linear' model.
- `'constant'` - benchmark model for constant predictions.
- `'lme'` - benchmark model for classical linear mixed-effects model.

****kwargs** Keyword arguments directly passed to the model for its initialization (through *ModelFactory.model()*). Refer to the corresponding model to know possible arguments.

**noise_model** [str] *For manifold-like models*. Define the noise structure of the model, can be either:

- `'gaussian_scalar'`: gaussian error, with same standard deviation for all features
- `'gaussian_diagonal'`: gaussian error, with one standard deviation parameter per feature (default)
- `'bernoulli'`: for binary data (Bernoulli realization)

**source_dimension** [int, optional] *For multivariate models only*. Set the degrees of freedom for _spatial_ variability. This number MUST BE strictly lower than the number of features. By default, this number is equal to square root of the number of features. One can interpret this hyperparameter as a way to reduce the dimension of inter-individual _spatial_ variability between progressions.

**See also:**

`leaspy.models`

**Attributes**

**model** [*AbstractModel*] Model used for computations, is an instance of *AbstractModel*.

**type** [str (read-only)] Name of the model - will be one of the names listed above.

### Methods

| | |
|---|---|
| *calibrate*(data, settings) | Duplicates of the *fit()* method. |
| *check_if_initialized*() | Check if model is initialized. |
| *estimate*(timepoints, individual_parameters, *) | Return the model values for individuals characterized by their individual parameters $z_i$ at time-points $(t_{i,j})_j$. |
| *estimate_ages_from_biomarker_values*(...[, ...]) | For individuals characterized by their individual parameters $z_i$, returns the age $t_{i,j}$ at which a given feature value $y_{i,j,k}$ is reached. |
| *fit*(data, settings) | Estimate the model's parameters $\theta$ for a given dataset and a given algorithm. |
| *load*(path_to_model_settings) | Instantiate a Leaspy object from json model parameter file or the corresponding dictionary. |
| *personalize*(data, settings, *[, return_noise]) | From a model, estimate individual parameters for each *ID* of a given dataset. |
| *save*(path, **kwargs) | Save Leaspy object as json model parameter file. |
| *simulate*(individual_parameters, data, settings) | Generate longitudinal synthetic patients data from a given model, a given collection of individual parameters and some given settings. |

## 3.2 leaspy.models: Models

Available models in *Leaspy*.

| | |
|---|---|
| model_factory.ModelFactory() | Return the wanted model given its name. |
| abstract_model.AbstractModel(name, **kwargs) | Contains the common attributes & methods of the different models. |
| univariate_model.UnivariateModel(name, **kwargs) | Univariate (logistic or linear) model for a single variable of interest. |
| abstract_multivariate_model.AbstractMultivariateModel(...) | Contains the common attributes & methods of the multivariate models. |
| multivariate_model.MultivariateModel(name, ...) | Manifold model for multiple variables of interest (logistic or linear formulation). |
| multivariate_parallel_model.MultivariateParallelModel(...) | Logistic model for multiple variables of interest, imposing same average evolution pace for all variables (logistic curves are only time-shifted). |
| lme_model.LMEModel(name, **kwargs) | LMEModel is a benchmark model that fits and personalize a linear mixed-effects model |
| constant_model.ConstantModel(name, **kwargs) | *ConstantModel* is a benchmark model that predicts constant values (no matter what the patient's ages are). |

### 3.2.1 leaspy.models.model_factory.ModelFactory

**class ModelFactory**

Bases: object

Return the wanted model given its name.

#### Methods

| | |
|---|---|
| model(name, **kwargs) | Return the model object corresponding to 'name' arg with possible *kwargs* |

**static model**(*name: str*, ***kwargs*) → *AbstractModel*

Return the model object corresponding to 'name' arg with possible *kwargs*

Check name type and value.

**Parameters**

**name** [str] The model's name.

****kwargs** Contains model's hyper-parameters. Raise an error if the keyword is inappropriate for the given model's name.

**Returns**

***AbstractModel*** A child class object of `models.AbstractModel` class object determined by 'name'.

**Raises**

**LeaspyModelInputError** if incorrect model requested.

**See also:**

*Leaspy*

### 3.2.2 leaspy.models.abstract_model.AbstractModel

**class AbstractModel**(*name: str*, ***kwargs*)

Bases: abc.ABC

Contains the common attributes & methods of the different models.

**Parameters**

**name** [str] The name of the model

****kwargs** Hyperparameters for the model

**Attributes**

**is_initialized** [bool] Indicates if the model is initialized

**name** [str] The model's name

**features** [list[str]] Names of the model features

**parameters** [dict] Contains the model's parameters

**noise_model** [str] The noise structure for the model. cf. `NoiseModel` to see possible values.

**regularization_distribution_factory** [function dist params -> `torch.distributions.Distribution`] Factory of torch distribution to compute log-likelihoods for regularization (gaussian by default)

#### Methods

| | |
|---|---|
| *compute_individual_ages_from_biomarker_values*(...) | For one individual, compute age(s) at which the given features values are reached (given the subject's individual parameters). |
| *compute_individual_ages_from_biomarker_values_tensorized*(...) | For one individual, compute age(s) at which the given features values are reached (given the subject's individual parameters), with tensorized inputs |
| *compute_individual_attachment_tensorized*(...) | Compute attachment term (per subject) |
| *compute_individual_attachment_tensorized_mcmc*(...) | Compute MCMC attachment of all subjects? One subject? One visit? TODO: complete |
| *compute_individual_tensorized*(timepoints, ...) | Compute the individual values at timepoints according to the model. |
| *compute_individual_trajectory*(timepoints, ...) | Compute scores values at the given time-point(s) given a subject's individual parameters. |
| *compute_jacobian_tensorized*(timepoints, ...) | Compute the jacobian of the model w.r.t. |
| *compute_regularity_realization*(realization) | Compute regularity term for a *Realization* instance. |
| *compute_regularity_variable*(value, mean, std) | Compute regularity term (Gaussian distribution), low-level. |
| *compute_sufficient_statistics*(data, realizations) | Compute sufficient statistics from realizations |
| *compute_sum_squared_per_ft_tensorized*(data, ...) | Compute the square of the residuals per subject per feature |

continues on next page

Table 6 – continued from previous page

| | |
|---|---|
| *compute_sum_squared_tensorized*(data, param_ind) | Compute the square of the residuals per subject |
| *get_individual_realization_names*() | Get names of individual variales of the model. |
| *get_individual_variable_name*() | Return list of names of the individual variables from the model. |
| *get_param_from_real*(realizations) | Get individual parameters realizations from all model realizations |
| *get_population_realization_names*() | Get names of population variales of the model. |
| *get_realization_object*(n_individuals) | Initialization of a *CollectionRealization* used during model fitting. |
| *initialize*(dataset[, method]) | Initialize the model given a dataset and an initialization method. |
| *load_hyperparameters*(hyperparameters) | Load model's hyperparameters |
| *load_parameters*(parameters) | Instantiate or update the model's parameters. |
| *random_variable_informations*() | Informations on model's random variables. |
| *save*(path, **kwargs) | Save Leaspy object as json model parameter file. |
| *smart_initialization_realizations*(data, ...) | Smart initialization of realizations if needed. |
| *time_reparametrization*(timepoints, xi, tau) | Tensorized time reparametrization formula |
| *update_model_parameters*(data, ...[, ...]) | Update model parameters (high-level function) |
| *update_model_parameters_burn_in*(data, ...) | Update model parameters (burn-in phase) |
| *update_model_parameters_normal*(data, suff_stats) | Update model parameters (after burn-in phase) |

**compute_individual_ages_from_biomarker_values**(*value: Union[float, List[float]]*, *individual_parameters: Dict[str, Any]*, *feature: Optional[str] = None*)

For one individual, compute age(s) at which the given features values are reached (given the subject's individual parameters).

Consistency checks are done in the main API layer.

**Parameters**

**value** [scalar or array_like[scalar] (list, tuple, `numpy.ndarray`)] Contains the biomarker value(s) of the subject.

**individual_parameters** [dict] Contains the individual parameters. Each individual parameter should be a scalar or array_like

**feature** [str (or None)] Name of the considered biomarker (optional for univariate models, compulsory for multivariate models).

**Returns**

`torch.Tensor` Contains the subject's ages computed at the given values(s) Shape of tensor is (1, n_values)

**Raises**

**LeaspyModelInputError** if computation is tried on more than 1 individual

**abstract compute_individual_ages_from_biomarker_values_tensorized**(*value: torch.FloatTensor*, *individual_parameters: Dict[str, torch.FloatTensor]*, *feature: Optional[str]*) → torch.FloatTensor

For one individual, compute age(s) at which the given features values are reached (given the subject's individual parameters), with tensorized inputs

**Parameters**

**value** [torch.Tensor of shape (1, n_values)] Contains the biomarker value(s) of the subject.

**individual_parameters** [dict] Contains the individual parameters. Each individual parameter should be a torch.Tensor

**feature** [str (or None)] Name of the considered biomarker (optional for univariate models, compulsory for multivariate models).

**Returns**

**torch.Tensor** Contains the subject's ages computed at the given values(s) Shape of tensor is (n_values, 1)

**compute_individual_attachment_tensorized**(*data: Dataset*, *param_ind: DictParamsTorch*, *attribute_type*) → torch.FloatTensor

Compute attachment term (per subject)

**Parameters**

**data** [*Dataset*] Contains the data of the subjects, in particular the subjects' time-points and the mask for nan values & padded visits

**param_ind** [dict] Contain the individual parameters

**attribute_type** [Any, optional] Flag to ask for MCMC attributes instead of model's attributes.

**Returns**

**attachment** [torch.Tensor] Negative Log-likelihood, shape = (n_subjects,)

**Raises**

**LeaspyModelInputError** If invalid *noise_model* for model

**compute_individual_attachment_tensorized_mcmc**(*data: Dataset*, *realizations: CollectionRealization*)

Compute MCMC attachment of all subjects? One subject? One visit? TODO: complete

**Parameters**

**data** [*Dataset*] Contains the data of the subjects, in particular the subjects' time-points and the mask (?)

**realizations** [*CollectionRealization*]

**Returns**

**attachment** [torch.Tensor] The subject attachment (?)

**abstract compute_individual_tensorized**(*timepoints: torch.FloatTensor*, *individual_parameters: Dict[str, torch.FloatTensor]*, *attribute_type=None*) → torch.FloatTensor

Compute the individual values at timepoints according to the model.

**Parameters**

**timepoints** [torch.Tensor of shape (n_individuals, n_timepoints)]

**individual_parameters** [dict[param_name: str, torch.Tensor of shape (n_individuals, n_dims_param)]]

**attribute_type** [Any (default None)] Flag to ask for MCMC attributes instead of model's attributes.

**Returns**

**torch.Tensor of shape (n_individuals, n_timepoints, n_features)**

**compute_individual_trajectory**(*timepoints*, *individual_parameters: Dict[str, Any]*, ***, *skip_ips_checks: bool = False*)

Compute scores values at the given time-point(s) given a subject's individual parameters.

**Parameters**

**timepoints** [scalar or array_like[scalar] (list, tuple, numpy.ndarray)] Contains the age(s) of the subject.

**individual_parameters** [dict] Contains the individual parameters. Each individual parameter should be a scalar or array_like

**skip_ips_checks** [bool (default: False)] Flag to skip consistency/compatibility checks and tensorization of individual_parameters when it was done earlier (speed-up)

**Returns**

**torch.Tensor** Contains the subject's scores computed at the given age(s) Shape of tensor is (1, n_tpts, n_features)

**Raises**

**LeaspyModelInputError** if computation is tried on more than 1 individual

**LeaspyIndividualParamsInputError** if invalid individual parameters

**abstract compute_jacobian_tensorized**(*timepoints: torch.FloatTensor*, *ind_parameters: Dict[str, torch.FloatTensor]*, *attribute_type=None*) → torch.FloatTensor

Compute the jacobian of the model w.r.t. each individual parameter.

This function aims to be used in *ScipyMinimize* to speed up optimization.

**Parameters**

**timepoints** [torch.Tensor of shape (n_individuals, n_timepoints)]

**individual_parameters** [dict[param_name: str, torch.Tensor of shape (n_individuals, n_dims_param)]]

**attribute_type** [Any (default None)] Flag to ask for MCMC attributes instead of model's attributes.

**Returns**

**dict[param_name: str, torch.Tensor of shape (n_individuals, n_timepoints, n_features, n_dims_param)]**

**compute_regularity_realization**(*realization: Realization*)
Compute regularity term for a *Realization* instance.

**Parameters**

**realization** [*Realization*]

**Returns**

**torch.Tensor**

**compute_regularity_variable**(*value: torch.FloatTensor*, *mean: torch.FloatTensor*, *std: torch.FloatTensor*) → torch.FloatTensor
Compute regularity term (Gaussian distribution), low-level.

**Parameters**

**value, mean, std** [torch.Tensor of same shapes]

**Returns**

**torch.Tensor of same shape than input**

**abstract compute_sufficient_statistics**(*data: Dataset*, *realizations: CollectionRealization*) → DictParamsTorch
Compute sufficient statistics from realizations

**Parameters**

**data** [*Dataset*]

**realizations** [*CollectionRealization*]

**Returns**

**dict[suff_stat: str, torch.Tensor]**

**compute_sum_squared_per_ft_tensorized**(*data: Dataset*, *param_ind: DictParamsTorch*, *attribute_type=None*) → torch.FloatTensor
Compute the square of the residuals per subject per feature

**Parameters**

**data** [*Dataset*] Contains the data of the subjects, in particular the subjects' time-points and the mask (?)

**param_ind** [dict] Contain the individual parameters

**attribute_type** [Any (default None)] Flag to ask for MCMC attributes instead of model's attributes.

**Returns**

**torch.Tensor of shape (n_individuals,dimension)** Contains L2 residual for each subject and each feature

**compute_sum_squared_tensorized**(*data: Dataset*, *param_ind: DictParamsTorch*, *attribute_type=None*) → torch.FloatTensor
Compute the square of the residuals per subject

**Parameters**

**data** [*Dataset*] Contains the data of the subjects, in particular the subjects' time-points and the mask (?)

**param_ind** [dict] Contain the individual parameters

**attribute_type** [Any (default None)] Flag to ask for MCMC attributes instead of model's attributes.

**Returns**

**torch.Tensor of shape (n_individuals,)** Contains L2 residual for each subject

`get_individual_realization_names()`
Get names of individual variales of the model.

**Returns**

**list[str]**

`get_individual_variable_name()`
Return list of names of the individual variables from the model.

Duplicate of *get_individual_realization_names()*

TODO delete one of them

**Returns**

**individual_variable_name** [list [str]] Contains the individual variables' names

`get_param_from_real(realizations: CollectionRealization)` → Dict[str, torch.FloatTensor]
Get individual parameters realizations from all model realizations

**Parameters**

**realizations** [*CollectionRealization*]

**Returns**

**dict[param_name: str, torch.Tensor [n_individuals, dims_param]]** Individual parameters

`get_population_realization_names()`
Get names of population variales of the model.

**Returns**

**list[str]**

`get_realization_object(n_individuals: int)` → *CollectionRealization*
Initialization of a *CollectionRealization* used during model fitting.

**Parameters**

**n_individuals** [int] Number of individuals to track

**Returns**

*CollectionRealization*

`abstract initialize(dataset: Dataset, method: str = 'default')`
Initialize the model given a dataset and an initialization method.

After calling this method `is_initialized` should be True and model should be ready for use.

**Parameters**

**dataset** [*Dataset*] The dataset we want to initialize from.

**method** [str] A custom method to initialize the model

`abstract load_hyperparameters(hyperparameters: Dict[str, Any])`
Load model's hyperparameters

**Parameters**

**hyperparameters** [dict[str, Any]] Contains the model's hyperparameters

**Raises**

**LeaspyModelInputError** If any of the consistency checks fail.

**load_parameters**(*parameters: Dict[str, Any]*)
Instantiate or update the model's parameters.

**Parameters**

**parameters** [dict[str, Any]] Contains the model's parameters

**abstract random_variable_informations**() → Dict[str, Any]
Informations on model's random variables.

**Returns**

**dict[str, Any]**

**abstract save**(*path: str*, ***kwargs*)
Save Leaspy object as json model parameter file.

**Parameters**

**path** [str] Path to store the model's parameters.

****kwargs** Keyword arguments for json.dump method.

**smart_initialization_realizations**(*data: Dataset*, *realizations: CollectionRealization*)
Smart initialization of realizations if needed.

Default behavior to return *realizations* as they are (no smart trick).

**Parameters**

**data** [*Dataset*]

**realizations** [*CollectionRealization*]

**Returns**

***CollectionRealization***

**static time_reparametrization**(*timepoints: torch.FloatTensor*, *xi: torch.FloatTensor*, *tau: torch.FloatTensor*) → torch.FloatTensor
Tensorized time reparametrization formula

<!> Shapes of tensors must be compatible between them.

**Parameters**

**timepoints** [torch.Tensor] Timepoints to reparametrize

**xi** [torch.Tensor] Log-acceleration of individual(s)

**tau** [torch.Tensor] Time-shift(s)

**Returns**

**torch.Tensor of same shape as *timepoints***

**update_model_parameters**(*data: Dataset*, *reals_or_suff_stats: Union[CollectionRealization, DictParamsTorch]*, *burn_in_phase=True*)
Update model parameters (high-level function)

Under-the-hood call *update_model_parameters_burn_in()* or *update_model_parameters_normal()* depending on the phase of the fit algorithm

**Parameters**

**data** [*Dataset*]

**reals_or_suff_stats :**

**If during burn-in phase will be realizations:** *CollectionRealization*

**If after burn-in phase will be sufficient statistics:** dict[suff_stat: str, `torch.Tensor`]

**abstract update_model_parameters_burn_in**(*data: Dataset*, *realizations: CollectionRealization*)
Update model parameters (burn-in phase)

**Parameters**

**data** [*Dataset*]

**realizations** [*CollectionRealization*]

**abstract update_model_parameters_normal**(*data: Dataset*, *suff_stats: DictParamsTorch*)
Update model parameters (after burn-in phase)

**Parameters**

**data** [*Dataset*]

**suff_stats** [dict[suff_stat: str, `torch.Tensor`]]

### 3.2.3 leaspy.models.univariate_model.UnivariateModel

**class UnivariateModel**(*name: str*, ***kwargs*)
Bases: *leaspy.models.abstract_model.AbstractModel*

Univariate (logistic or linear) model for a single variable of interest.

**Parameters**

**name** [str] Name of the model

****kwargs** Hyperparameters of the model

**Raises**

**LeaspyModelInputError**

- If *name* is not one of allowed sub-type: 'univariate_linear' or 'univariate_logistic'
- If hyperparameters are inconsistent

#### Methods

| | |
|---|---|
| *compute_individual_ages_from_biomarker_values*(...) | For one individual, compute age(s) at which the given features values are reached (given the subject's individual parameters). |
| *compute_individual_ages_from_biomarker_values_tensorized*(...) | For one individual, compute age(s) at which the given features values are reached (given the subject's individual parameters), with tensorized inputs |
| *compute_individual_attachment_tensorized*(...) | Compute attachment term (per subject) |

continues on next page

Table 7 – continued from previous page

| | |
|---|---|
| *compute_individual_attachment_tensorized_mcmc*(...) | Compute MCMC attachment of all subjects? One subject? One visit? TODO: complete |
| *compute_individual_tensorized*(timepoints, ...) | Compute the individual values at timepoints according to the model. |
| *compute_individual_tensorized_linear*(...[, ...]) | Compute the individual values at timepoints according to the model (linear). |
| *compute_individual_tensorized_logistic*(...) | Compute the individual values at timepoints according to the model (logistic). |
| *compute_individual_trajectory*(timepoints, ...) | Compute scores values at the given time-point(s) given a subject's individual parameters. |
| *compute_jacobian_tensorized*(timepoints, ...) | Compute the jacobian of the model w.r.t. |
| *compute_jacobian_tensorized_linear*(...[, ...]) | Compute the jacobian of the model (linear) w.r.t. |
| *compute_jacobian_tensorized_logistic*(...[, MCMC]) | Compute the jacobian of the model (logistic) w.r.t. |
| *compute_mean_traj*(timepoints) | Compute trajectory of the model with individual parameters being the group-average ones. |
| *compute_regularity_realization*(realization) | Compute regularity term for a *Realization* instance. |
| *compute_regularity_variable*(value, mean, std) | Compute regularity term (Gaussian distribution), low-level. |
| *compute_sufficient_statistics*(data, realizations) | Compute sufficient statistics from realizations |
| *compute_sum_squared_per_ft_tensorized*(data, ...) | Compute the square of the residuals per subject per feature |
| *compute_sum_squared_tensorized*(data, param_ind) | Compute the square of the residuals per subject |
| *get_individual_realization_names*() | Get names of individual variales of the model. |
| *get_individual_variable_name*() | Return list of names of the individual variables from the model. |
| *get_param_from_real*(realizations) | Get individual parameters realizations from all model realizations |
| *get_population_realization_names*() | Get names of population variales of the model. |
| *get_realization_object*(n_individuals) | Initialization of a *CollectionRealization* used during model fitting. |
| *initialize*(dataset[, method]) | Initialize the model given a dataset and an initialization method. |
| *initialize_MCMC_toolbox*() | Initialize Monte-Carlo Markov-Chain toolbox for calibration of model |
| *load_hyperparameters*(hyperparameters) | Load model's hyperparameters |
| *load_parameters*(parameters) | Instantiate or update the model's parameters. |
| *random_variable_informations*() | Informations on model's random variables. |
| *save*(path, **kwargs) | Save Leaspy object as json model parameter file. |
| *smart_initialization_realizations*(data, ...) | Smart initialization of realizations if needed. |
| *time_reparametrization*(timepoints, xi, tau) | Tensorized time reparametrization formula |
| *update_MCMC_toolbox*(...) | Update the MCMC toolbox with a collection of realizations of model population parameters. |
| *update_model_parameters*(data, ...[, ...]) | Update model parameters (high-level function) |
| *update_model_parameters_burn_in*(data, ...) | Update model parameters (burn-in phase) |
| *update_model_parameters_normal*(data, suff_stats) | Update model parameters (after burn-in phase) |

| compute_individual_ages_from_biomarker_values_tensorized_logistic | |
|---|---|

**compute_individual_ages_from_biomarker_values**(*value: Union[float, List[float]]*, *individual_parameters: Dict[str, Any]*, *feature: Optional[str] = None*)

For one individual, compute age(s) at which the given features values are reached (given the subject's individual parameters).

Consistency checks are done in the main API layer.

**Parameters**

**value** [scalar or array_like[scalar] (list, tuple, `numpy.ndarray`)] Contains the biomarker value(s) of the subject.

**individual_parameters** [dict] Contains the individual parameters. Each individual parameter should be a scalar or array_like

**feature** [str (or None)] Name of the considered biomarker (optional for univariate models, compulsory for multivariate models).

**Returns**

`torch.Tensor` Contains the subject's ages computed at the given values(s) Shape of tensor is (1, n_values)

**Raises**

**LeaspyModelInputError** if computation is tried on more than 1 individual

**compute_individual_ages_from_biomarker_values_tensorized**(*value: Tensor*, *individual_parameters: dict*, *feature: str*)

For one individual, compute age(s) at which the given features values are reached (given the subject's individual parameters), with tensorized inputs

**Parameters**

**value** [torch.Tensor of shape (1, n_values)] Contains the biomarker value(s) of the subject.

**individual_parameters** [dict] Contains the individual parameters. Each individual parameter should be a torch.Tensor

**feature** [str (or None)] Name of the considered biomarker (optional for univariate models, compulsory for multivariate models).

**Returns**

`torch.Tensor` Contains the subject's ages computed at the given values(s) Shape of tensor is (n_values, 1)

**compute_individual_attachment_tensorized**(*data: Dataset*, *param_ind: DictParamsTorch*, *attribute_type*) → torch.FloatTensor

Compute attachment term (per subject)

**Parameters**

**data** [*Dataset*] Contains the data of the subjects, in particular the subjects' time-points and the mask for nan values & padded visits

**param_ind** [dict] Contain the individual parameters

**attribute_type** [Any, optional] Flag to ask for MCMC attributes instead of model's attributes.

**Returns**

**attachment** [torch.Tensor] Negative Log-likelihood, shape = (n_subjects,)

**Raises**

**LeaspyModelInputError** If invalid *noise_model* for model

**compute_individual_attachment_tensorized_mcmc**(*data: Dataset*, *realizations: CollectionRealization*)

Compute MCMC attachment of all subjects? One subject? One visit? TODO: complete

**Parameters**

**data** [*Dataset*] Contains the data of the subjects, in particular the subjects' time-points and the mask (?)

**realizations** [*CollectionRealization*]

**Returns**

**attachment** [torch.Tensor] The subject attachment (?)

**compute_individual_tensorized**(*timepoints*, *ind_parameters*, *attribute_type=None*)

Compute the individual values at timepoints according to the model.

**Parameters**

**timepoints** [torch.Tensor of shape (n_individuals, n_timepoints)]

**individual_parameters** [dict[param_name: str, torch.Tensor of shape (n_individuals, n_dims_param)]]

**attribute_type** [Any (default None)] Flag to ask for MCMC attributes instead of model's attributes.

**Returns**

**torch.Tensor of shape (n_individuals, n_timepoints, n_features)**

**compute_individual_tensorized_linear**(*timepoints*, *ind_parameters*, *attribute_type=False*)

Compute the individual values at timepoints according to the model (linear).

**Parameters**

**timepoints** [torch.Tensor of shape (n_individuals, n_timepoints)]

**individual_parameters** [dict[param_name: str, torch.Tensor of shape (n_individuals, n_dims_param)]]

**attribute_type** [Any (default None)] Flag to ask for MCMC attributes instead of model's attributes.

**Returns**

**torch.Tensor of shape (n_individuals, n_timepoints, n_features)**

**compute_individual_tensorized_logistic**(*timepoints*, *ind_parameters*, *attribute_type=False*)

Compute the individual values at timepoints according to the model (logistic).

**Parameters**

**timepoints** [torch.Tensor of shape (n_individuals, n_timepoints)]

**individual_parameters** [dict[param_name: str, torch.Tensor of shape (n_individuals, n_dims_param)]]

**attribute_type** [Any (default None)] Flag to ask for MCMC attributes instead of model's attributes.

**Returns**

**torch.Tensor of shape (n_individuals, n_timepoints, n_features)**

**compute_individual_trajectory**(*timepoints*, *individual_parameters: Dict[str, Any]*, ***, *skip_ips_checks: bool = False*)

Compute scores values at the given time-point(s) given a subject's individual parameters.

**Parameters**

**timepoints** [scalar or array_like[scalar] (list, tuple, numpy.ndarray)] Contains the age(s) of the subject.

**individual_parameters** [dict] Contains the individual parameters. Each individual parameter should be a scalar or array_like

**skip_ips_checks** [bool (default: False)] Flag to skip consistency/compatibility checks and tensorization of individual_parameters when it was done earlier (speed-up)

**Returns**

**torch.Tensor** Contains the subject's scores computed at the given age(s) Shape of tensor is (1, n_tpts, n_features)

**Raises**

**LeaspyModelInputError** if computation is tried on more than 1 individual

**LeaspyIndividualParamsInputError** if invalid individual parameters

**compute_jacobian_tensorized**(*timepoints*, *ind_parameters*, *attribute_type=None*)

Compute the jacobian of the model w.r.t. each individual parameter.

This function aims to be used in *ScipyMinimize* to speed up optimization.

**Parameters**

**timepoints** [torch.Tensor of shape (n_individuals, n_timepoints)]

**individual_parameters** [dict[param_name: str, torch.Tensor of shape (n_individuals, n_dims_param)]]

**attribute_type** [Any (default None)] Flag to ask for MCMC attributes instead of model's attributes.

**Returns**

**dict[param_name: str, torch.Tensor of shape (n_individuals, n_timepoints, n_features, n_dims_param)]**

**compute_jacobian_tensorized_linear**(*timepoints*, *ind_parameters*, *attribute_type=None*)

Compute the jacobian of the model (linear) w.r.t. each individual parameter.

This function aims to be used in *ScipyMinimize* to speed up optimization.

**Parameters**

**timepoints** [torch.Tensor of shape (n_individuals, n_timepoints)]

**individual_parameters** [dict[param_name: str, torch.Tensor of shape (n_individuals, n_dims_param)]]

**attribute_type** [Any (default None)] Flag to ask for MCMC attributes instead of model's attributes.

**Returns**

**dict[param_name: str, torch.Tensor of shape (n_individuals, n_timepoints, n_features, n_dims_param)]**

`compute_jacobian_tensorized_logistic`(*timepoints*, *ind_parameters*, *MCMC=False*)
Compute the jacobian of the model (logistic) w.r.t. each individual parameter.

This function aims to be used in *ScipyMinimize* to speed up optimization.

**Parameters**

**timepoints** [torch.Tensor of shape (n_individuals, n_timepoints)]

**individual_parameters** [dict[param_name: str, torch.Tensor of shape (n_individuals, n_dims_param)]]

**attribute_type** [Any (default None)] Flag to ask for MCMC attributes instead of model's attributes.

**Returns**

**dict[param_name: str, torch.Tensor of shape (n_individuals, n_timepoints, n_features, n_dims_param)]**

`compute_mean_traj`(*timepoints*)
Compute trajectory of the model with individual parameters being the group-average ones.

TODO check dimensions of io? TODO generalize in abstract manifold model

**Parameters**

**timepoints** [torch.Tensor [1, n_timepoints]]

**Returns**

**torch.Tensor [1, n_timepoints, dimension]** The group-average values at given time-points

`compute_regularity_realization`(*realization: Realization*)
Compute regularity term for a *Realization* instance.

**Parameters**

**realization** [*Realization*]

**Returns**

**torch.Tensor**

`compute_regularity_variable`(*value: torch.FloatTensor*, *mean: torch.FloatTensor*, *std: torch.FloatTensor*) → torch.FloatTensor
Compute regularity term (Gaussian distribution), low-level.

**Parameters**

**value, mean, std** [torch.Tensor of same shapes]

**Returns**

**torch.Tensor of same shape than input**

`compute_sufficient_statistics`(*data*, *realizations*)
Compute sufficient statistics from realizations

**Parameters**

**data** [*Dataset*]

**realizations** [*CollectionRealization*]

**Returns**

**dict[suff_stat: str, torch.Tensor]**

**compute_sum_squared_per_ft_tensorized**(*data: Dataset*, *param_ind: DictParamsTorch*, *attribute_type=None*) → torch.FloatTensor

Compute the square of the residuals per subject per feature

**Parameters**

**data** [*Dataset*] Contains the data of the subjects, in particular the subjects' time-points and the mask (?)

**param_ind** [dict] Contain the individual parameters

**attribute_type** [Any (default None)] Flag to ask for MCMC attributes instead of model's attributes.

**Returns**

**torch.Tensor of shape (n_individuals,dimension)** Contains L2 residual for each subject and each feature

**compute_sum_squared_tensorized**(*data: Dataset*, *param_ind: DictParamsTorch*, *attribute_type=None*) → torch.FloatTensor

Compute the square of the residuals per subject

**Parameters**

**data** [*Dataset*] Contains the data of the subjects, in particular the subjects' time-points and the mask (?)

**param_ind** [dict] Contain the individual parameters

**attribute_type** [Any (default None)] Flag to ask for MCMC attributes instead of model's attributes.

**Returns**

**torch.Tensor of shape (n_individuals,)** Contains L2 residual for each subject

**get_individual_realization_names**()

Get names of individual variales of the model.

**Returns**

**list[str]**

**get_individual_variable_name**()

Return list of names of the individual variables from the model.

Duplicate of *get_individual_realization_names()*

TODO delete one of them

**Returns**

**individual_variable_name** [list [str]] Contains the individual variables' names

**get_param_from_real**(*realizations:* CollectionRealization) → Dict[str, torch.FloatTensor]

Get individual parameters realizations from all model realizations

**Parameters**

**realizations** [*CollectionRealization*]

**Returns**

**dict[param_name: str, torch.Tensor [n_individuals, dims_param]]** Individual parameters

`get_population_realization_names()`
Get names of population variales of the model.

**Returns**

**list[str]**

`get_realization_object(n_individuals: int) → CollectionRealization`
Initialization of a *CollectionRealization* used during model fitting.

**Parameters**

**n_individuals** [int] Number of individuals to track

**Returns**

*CollectionRealization*

`initialize(dataset, method='default')`
Initialize the model given a dataset and an initialization method.

After calling this method `is_initialized` should be True and model should be ready for use.

**Parameters**

**dataset** [*Dataset*] The dataset we want to initialize from.

**method** [str] A custom method to initialize the model

`initialize_MCMC_toolbox()`
Initialize Monte-Carlo Markov-Chain toolbox for calibration of model

TODO to move in a "MCMC-model interface"

`load_hyperparameters(hyperparameters: dict)`
Load model's hyperparameters

**Parameters**

**hyperparameters** [dict[str, Any]] Contains the model's hyperparameters

**Raises**

**LeaspyModelInputError** If any of the consistency checks fail.

`load_parameters(parameters)`
Instantiate or update the model's parameters.

**Parameters**

**parameters** [dict[str, Any]] Contains the model's parameters

`random_variable_informations()`
Informations on model's random variables.

**Returns**

**dict[str, Any]**

**save**(*path: str*, ***kwargs*)
Save Leaspy object as json model parameter file.

**Parameters**

**path** [str] Path to store the model's parameters.

****kwargs** Keyword arguments for json.dump method.

**smart_initialization_realizations**(*data: Dataset*, *realizations: CollectionRealization*)
Smart initialization of realizations if needed.

Default behavior to return *realizations* as they are (no smart trick).

**Parameters**

**data** [*Dataset*]

**realizations** [*CollectionRealization*]

**Returns**

*CollectionRealization*

**static time_reparametrization**(*timepoints: torch.FloatTensor*, *xi: torch.FloatTensor*, *tau: torch.FloatTensor*) → torch.FloatTensor
Tensorized time reparametrization formula

<!> Shapes of tensors must be compatible between them.

**Parameters**

**timepoints** [torch.Tensor] Timepoints to reparametrize

**xi** [torch.Tensor] Log-acceleration of individual(s)

**tau** [torch.Tensor] Time-shift(s)

**Returns**

**torch.Tensor of same shape as *timepoints***

**update_MCMC_toolbox**(*name_of_the_variables_that_have_been_changed*, *realizations*)
Update the MCMC toolbox with a collection of realizations of model population parameters.

TODO to move in a "MCMC-model interface"

**Parameters**

**name_of_the_variables_that_have_been_changed** [container[str] (list, tuple, ...)] Names of the population parameters to update in MCMC toolbox

**realizations** [*CollectionRealization*] All the realizations to update MCMC toolbox with

**update_model_parameters**(*data: Dataset*, *reals_or_suff_stats: Union[CollectionRealization, DictParamsTorch]*, *burn_in_phase=True*)
Update model parameters (high-level function)

Under-the-hood call *update_model_parameters_burn_in()* or *update_model_parameters_normal()* depending on the phase of the fit algorithm

**Parameters**

**data** [*Dataset*]

**reals_or_suff_stats :**

**If during burn-in phase will be realizations:** *CollectionRealization*

**If after burn-in phase will be sufficient statistics:** dict[suff_stat: str, torch.Tensor]

**update_model_parameters_burn_in**(*data*, *realizations*)
Update model parameters (burn-in phase)

**Parameters**

**data** [*Dataset*]

**realizations** [*CollectionRealization*]

**update_model_parameters_normal**(*data*, *suff_stats*)
Update model parameters (after burn-in phase)

**Parameters**

**data** [*Dataset*]

**suff_stats** [dict[suff_stat: str, torch.Tensor]]

### 3.2.4 leaspy.models.abstract_multivariate_model.AbstractMultivariateModel

**class AbstractMultivariateModel**(*name: str*, ***kwargs*)
Bases: *leaspy.models.abstract_model.AbstractModel*

Contains the common attributes & methods of the multivariate models.

**Parameters**

**name** [str] Name of the model

****kwargs** Hyperparameters for the model

**Raises**

**LeaspyModelInputError** if inconsistent hyperparameters

#### Methods

| | |
|---|---|
| *compute_individual_ages_from_biomarker_values*(...) | For one individual, compute age(s) at which the given features values are reached (given the subject's individual parameters). |
| *compute_individual_ages_from_biomarker_values_tensorized*(...) | For one individual, compute age(s) at which the given features values are reached (given the subject's individual parameters), with tensorized inputs |
| *compute_individual_attachment_tensorized*(...) | Compute attachment term (per subject) |
| *compute_individual_attachment_tensorized_mcmc*(...) | Compute MCMC attachment of all subjects? One subject? One visit? TODO: complete |
| *compute_individual_tensorized*(timepoints, ...) | Compute the individual values at timepoints according to the model. |
| *compute_individual_trajectory*(timepoints, ...) | Compute scores values at the given time-point(s) given a subject's individual parameters. |
| *compute_jacobian_tensorized*(timepoints, ...) | Compute the jacobian of the model w.r.t. |
| *compute_mean_traj*(timepoints) | Compute trajectory of the model with individual parameters being the group-average ones. |

continues on next page

Table 8 – continued from previous page

| | |
|---|---|
| *compute_regularity_realization*(realization) | Compute regularity term for a *Realization* instance. |
| *compute_regularity_variable*(value, mean, std) | Compute regularity term (Gaussian distribution), low-level. |
| *compute_sufficient_statistics*(data, realizations) | Compute sufficient statistics from realizations |
| *compute_sum_squared_per_ft_tensorized*(data, ...) | Compute the square of the residuals per subject per feature |
| *compute_sum_squared_tensorized*(data, param_ind) | Compute the square of the residuals per subject |
| *get_individual_realization_names*() | Get names of individual variales of the model. |
| *get_individual_variable_name*() | Return list of names of the individual variables from the model. |
| *get_param_from_real*(realizations) | Get individual parameters realizations from all model realizations |
| *get_population_realization_names*() | Get names of population variales of the model. |
| *get_realization_object*(n_individuals) | Initialization of a *CollectionRealization* used during model fitting. |
| *initialize*(dataset[, method]) | Initialize the model given a dataset and an initialization method. |
| *initialize_MCMC_toolbox*() | Initialize Monte-Carlo Markov-Chain toolbox for calibration of model |
| *load_hyperparameters*(hyperparameters) | Load model's hyperparameters |
| *load_parameters*(parameters) | Instantiate or update the model's parameters. |
| *random_variable_informations*() | Informations on model's random variables. |
| *save*(path[, with_mixing_matrix]) | Save Leaspy object as json model parameter file. |
| *smart_initialization_realizations*(data, ...) | Smart initialization of realizations if needed. |
| *time_reparametrization*(timepoints, xi, tau) | Tensorized time reparametrization formula |
| *update_MCMC_toolbox*(...) | Update the MCMC toolbox with a collection of realizations of model population parameters. |
| *update_model_parameters*(data, ...[, ...]) | Update model parameters (high-level function) |
| *update_model_parameters_burn_in*(data, ...) | Update model parameters (burn-in phase) |
| *update_model_parameters_normal*(data, suff_stats) | Update model parameters (after burn-in phase) |

**compute_individual_ages_from_biomarker_values**(*value: Union[float, List[float]]*, *individual_parameters: Dict[str, Any]*, *feature: Optional[str] = None*)

For one individual, compute age(s) at which the given features values are reached (given the subject's individual parameters).

Consistency checks are done in the main API layer.

**Parameters**

**value** [scalar or array_like[scalar] (list, tuple, `numpy.ndarray`)] Contains the biomarker value(s) of the subject.

**individual_parameters** [dict] Contains the individual parameters. Each individual parameter should be a scalar or array_like

**feature** [str (or None)] Name of the considered biomarker (optional for univariate models, compulsory for multivariate models).

**Returns**

torch.Tensor Contains the subject's ages computed at the given values(s) Shape of tensor is (1, n_values)

**Raises**

**LeaspyModelInputError** if computation is tried on more than 1 individual

**abstract compute_individual_ages_from_biomarker_values_tensorized**(*value: torch.FloatTensor*, *individual_parameters: Dict[str, torch.FloatTensor]*, *feature: Optional[str]*) → torch.FloatTensor

For one individual, compute age(s) at which the given features values are reached (given the subject's individual parameters), with tensorized inputs

**Parameters**

**value** [torch.Tensor of shape (1, n_values)] Contains the biomarker value(s) of the subject.

**individual_parameters** [dict] Contains the individual parameters. Each individual parameter should be a torch.Tensor

**feature** [str (or None)] Name of the considered biomarker (optional for univariate models, compulsory for multivariate models).

**Returns**

torch.Tensor Contains the subject's ages computed at the given values(s) Shape of tensor is (n_values, 1)

**compute_individual_attachment_tensorized**(*data: Dataset*, *param_ind: DictParamsTorch*, *attribute_type*) → torch.FloatTensor

Compute attachment term (per subject)

**Parameters**

**data** [*Dataset*] Contains the data of the subjects, in particular the subjects' time-points and the mask for nan values & padded visits

**param_ind** [dict] Contain the individual parameters

**attribute_type** [Any, optional] Flag to ask for MCMC attributes instead of model's attributes.

**Returns**

**attachment** [torch.Tensor] Negative Log-likelihood, shape = (n_subjects,)

**Raises**

**LeaspyModelInputError** If invalid *noise_model* for model

**compute_individual_attachment_tensorized_mcmc**(*data: Dataset*, *realizations: CollectionRealization*)

Compute MCMC attachment of all subjects? One subject? One visit? TODO: complete

**Parameters**

**data** [*Dataset*] Contains the data of the subjects, in particular the subjects' time-points and the mask (?)

**realizations** [*CollectionRealization*]

**Returns**

**attachment** [torch.Tensor] The subject attachment (?)

**abstract compute_individual_tensorized**(*timepoints*, *individual_parameters*, *attribute_type=None*)

Compute the individual values at timepoints according to the model.

**Parameters**

**timepoints** [torch.Tensor of shape (n_individuals, n_timepoints)]

**individual_parameters** [dict[param_name: str, torch.Tensor of shape (n_individuals, n_dims_param)]]

**attribute_type** [Any (default None)] Flag to ask for MCMC attributes instead of model's attributes.

**Returns**

**torch.Tensor of shape (n_individuals, n_timepoints, n_features)**

**compute_individual_trajectory**(*timepoints*, *individual_parameters: Dict[str, Any]*, ***, *skip_ips_checks: bool = False*)

Compute scores values at the given time-point(s) given a subject's individual parameters.

**Parameters**

**timepoints** [scalar or array_like[scalar] (list, tuple, numpy.ndarray)] Contains the age(s) of the subject.

**individual_parameters** [dict] Contains the individual parameters. Each individual parameter should be a scalar or array_like

**skip_ips_checks** [bool (default: False)] Flag to skip consistency/compatibility checks and tensorization of individual_parameters when it was done earlier (speed-up)

**Returns**

**torch.Tensor** Contains the subject's scores computed at the given age(s) Shape of tensor is (1, n_tpts, n_features)

**Raises**

**LeaspyModelInputError** if computation is tried on more than 1 individual

**LeaspyIndividualParamsInputError** if invalid individual parameters

**abstract compute_jacobian_tensorized**(*timepoints: torch.FloatTensor*, *ind_parameters: Dict[str, torch.FloatTensor]*, *attribute_type=None*) → torch.FloatTensor

Compute the jacobian of the model w.r.t. each individual parameter.

This function aims to be used in *ScipyMinimize* to speed up optimization.

**Parameters**

**timepoints** [torch.Tensor of shape (n_individuals, n_timepoints)]

**individual_parameters** [dict[param_name: str, torch.Tensor of shape (n_individuals, n_dims_param)]]

**attribute_type** [Any (default None)] Flag to ask for MCMC attributes instead of model's attributes.

**Returns**

**dict[param_name: str, torch.Tensor of shape (n_individuals, n_timepoints, n_features, n_dims_param)]**

**compute_mean_traj**(*timepoints*)

Compute trajectory of the model with individual parameters being the group-average ones.

TODO check dimensions of io?

**Parameters**

**timepoints** [torch.Tensor [1, n_timepoints]]

**Returns**

**torch.Tensor [1, n_timepoints, dimension]** The group-average values at given time-points

**compute_regularity_realization**(*realization: Realization*)

Compute regularity term for a *Realization* instance.

**Parameters**

**realization** [*Realization*]

**Returns**

**torch.Tensor**

**compute_regularity_variable**(*value: torch.FloatTensor*, *mean: torch.FloatTensor*, *std: torch.FloatTensor*) → torch.FloatTensor

Compute regularity term (Gaussian distribution), low-level.

**Parameters**

**value, mean, std** [torch.Tensor of same shapes]

**Returns**

**torch.Tensor of same shape than input**

**abstract compute_sufficient_statistics**(*data: Dataset*, *realizations: CollectionRealization*) → DictParamsTorch

Compute sufficient statistics from realizations

**Parameters**

**data** [*Dataset*]

**realizations** [*CollectionRealization*]

**Returns**

**dict[suff_stat: str, torch.Tensor]**

**compute_sum_squared_per_ft_tensorized**(*data: Dataset*, *param_ind: DictParamsTorch*, *attribute_type=None*) → torch.FloatTensor

Compute the square of the residuals per subject per feature

**Parameters**

**data** [*Dataset*] Contains the data of the subjects, in particular the subjects' time-points and the mask (?)

**param_ind** [dict] Contain the individual parameters

**attribute_type** [Any (default None)] Flag to ask for MCMC attributes instead of model's attributes.

**Returns**

**torch.Tensor of shape (n_individuals,dimension)** Contains L2 residual for each subject and each feature

**compute_sum_squared_tensorized**(*data: Dataset*, *param_ind: DictParamsTorch*, *attribute_type=None*) → torch.FloatTensor

Compute the square of the residuals per subject

**Parameters**

**data** [*Dataset*] Contains the data of the subjects, in particular the subjects' time-points and the mask (?)

**param_ind** [dict] Contain the individual parameters

**attribute_type** [Any (default None)] Flag to ask for MCMC attributes instead of model's attributes.

**Returns**

**torch.Tensor of shape (n_individuals,)** Contains L2 residual for each subject

**get_individual_realization_names**()

Get names of individual variales of the model.

**Returns**

**list[str]**

**get_individual_variable_name**()

Return list of names of the individual variables from the model.

Duplicate of *get_individual_realization_names()*

TODO delete one of them

**Returns**

**individual_variable_name** [list [str]] Contains the individual variables' names

**get_param_from_real**(*realizations:* CollectionRealization) → Dict[str, torch.FloatTensor]

Get individual parameters realizations from all model realizations

**Parameters**

**realizations** [*CollectionRealization*]

**Returns**

**dict[param_name: str, torch.Tensor [n_individuals, dims_param]]** Individual parameters

**get_population_realization_names**()

Get names of population variales of the model.

**Returns**

**list[str]**

**get_realization_object**(*n_individuals: int*) → *CollectionRealization*

Initialization of a *CollectionRealization* used during model fitting.

**Parameters**

**n_individuals** [int] Number of individuals to track

**Returns**

*CollectionRealization*

**initialize**(*dataset*, *method='default'*)
Initialize the model given a dataset and an initialization method.

After calling this method `is_initialized` should be True and model should be ready for use.

**Parameters**

**dataset** [*Dataset*] The dataset we want to initialize from.

**method** [str] A custom method to initialize the model

**abstract initialize_MCMC_toolbox**()
Initialize Monte-Carlo Markov-Chain toolbox for calibration of model

TODO to move in a "MCMC-model interface"

**load_hyperparameters**(*hyperparameters*)
Load model's hyperparameters

**Parameters**

**hyperparameters** [dict[str, Any]] Contains the model's hyperparameters

**Raises**

**LeaspyModelInputError** If any of the consistency checks fail.

**load_parameters**(*parameters: Dict[str, Any]*)
Instantiate or update the model's parameters.

**Parameters**

**parameters** [dict[str, Any]] Contains the model's parameters

**abstract random_variable_informations**() → Dict[str, Any]
Informations on model's random variables.

**Returns**

**dict[str, Any]**

**save**(*path*, *with_mixing_matrix=True*, ***kwargs*)
Save Leaspy object as json model parameter file.

**Parameters**

**path** [str] Path to store the model's parameters.

**with_mixing_matrix** [bool (default True)] Save the mixing matrix in the exported file in its 'parameters' section. <!> It is not a real parameter and its value will be overwritten at model loading (orthonormal basis is recomputed from other "true" parameters and mixing matrix is then deduced from this orthonormal basis and the betas)! It was integrated historically because it is used for convenience in browser webtool and only there…

****kwargs** Keyword arguments for json.dump method. Default to: dict(indent=2)

**smart_initialization_realizations**(*data: Dataset*, *realizations: CollectionRealization*)
Smart initialization of realizations if needed.

Default behavior to return *realizations* as they are (no smart trick).

**Parameters**

**data** [*Dataset*]

**realizations** [*CollectionRealization*]

**Returns**

*CollectionRealization*

**static time_reparametrization**(*timepoints: torch.FloatTensor*, *xi: torch.FloatTensor*, *tau: torch.FloatTensor*) → torch.FloatTensor

Tensorized time reparametrization formula

<!> Shapes of tensors must be compatible between them.

**Parameters**

**timepoints** [torch.Tensor] Timepoints to reparametrize

**xi** [torch.Tensor] Log-acceleration of individual(s)

**tau** [torch.Tensor] Time-shift(s)

**Returns**

**torch.Tensor of same shape as *timepoints***

**abstract update_MCMC_toolbox**(*name_of_the_variables_that_have_been_changed*, *realizations*)

Update the MCMC toolbox with a collection of realizations of model population parameters.

TODO to move in a "MCMC-model interface"

**Parameters**

**name_of_the_variables_that_have_been_changed** [container[str] (list, tuple, ...)] Names of the population parameters to update in MCMC toolbox

**realizations** [*CollectionRealization*] All the realizations to update MCMC toolbox with

**update_model_parameters**(*data: Dataset*, *reals_or_suff_stats: Union[CollectionRealization, DictParamsTorch]*, *burn_in_phase=True*)

Update model parameters (high-level function)

Under-the-hood call *update_model_parameters_burn_in()* or *update_model_parameters_normal()* depending on the phase of the fit algorithm

**Parameters**

**data** [*Dataset*]

**reals_or_suff_stats :**

**If during burn-in phase will be realizations:** *CollectionRealization*

**If after burn-in phase will be sufficient statistics:** dict[suff_stat: str, torch.Tensor]

**abstract update_model_parameters_burn_in**(*data: Dataset*, *realizations: CollectionRealization*)

Update model parameters (burn-in phase)

**Parameters**

**data** [*Dataset*]

**realizations** [*CollectionRealization*]

**abstract update_model_parameters_normal**(*data: Dataset*, *suff_stats: DictParamsTorch*)

Update model parameters (after burn-in phase)

**Parameters**

**data** [*Dataset*]

**suff_stats** [dict[suff_stat: str, torch.Tensor]]

### 3.2.5 leaspy.models.multivariate_model.MultivariateModel

**class MultivariateModel**(*name: str*, ***kwargs*)

Bases: *leaspy.models.abstract_multivariate_model.AbstractMultivariateModel*

Manifold model for multiple variables of interest (logistic or linear formulation).

**Parameters**

**name** [str] Name of the model

****kwargs** Hyperparameters of the model

**Raises**

**LeaspyModelInputError**

- If *name* is not one of allowed sub-type: 'univariate_linear' or 'univariate_logistic'
- If hyperparameters are inconsistent

#### Methods

| | |
|---|---|
| *compute_individual_ages_from_biomarker_values*(...) | For one individual, compute age(s) at which the given features values are reached (given the subject's individual parameters). |
| *compute_individual_ages_from_biomarker_values_tensorized*(...) | For one individual, compute age(s) at which the given features values are reached (given the subject's individual parameters), with tensorized inputs |
| *compute_individual_attachment_tensorized*(...) | Compute attachment term (per subject) |
| *compute_individual_attachment_tensorized_mcmc*(...) | Compute MCMC attachment of all subjects? One subject? One visit? TODO: complete |
| *compute_individual_tensorized*(timepoints, ...) | Compute the individual values at timepoints according to the model. |
| *compute_individual_tensorized_linear*(...[, ...]) | Compute the individual values at timepoints according to the model (linear). |
| *compute_individual_tensorized_logistic*(...) | Compute the individual values at timepoints according to the model (logistic). |
| *compute_individual_trajectory*(timepoints, ...) | Compute scores values at the given time-point(s) given a subject's individual parameters. |
| *compute_jacobian_tensorized*(timepoints, ...) | Compute the jacobian of the model w.r.t. |
| *compute_jacobian_tensorized_linear*(...[, ...]) | Compute the jacobian of the model (linear) w.r.t. |
| *compute_jacobian_tensorized_logistic*(...[, ...]) | Compute the jacobian of the model (logistic) w.r.t. |
| *compute_mean_traj*(timepoints) | Compute trajectory of the model with individual parameters being the group-average ones. |
| *compute_regularity_realization*(realization) | Compute regularity term for a *Realization* instance. |
| *compute_regularity_variable*(value, mean, std) | Compute regularity term (Gaussian distribution), low-level. |

continues on next page

Table 9 – continued from previous page

| | |
|---|---|
| compute_sufficient_statistics(data, realizations) | Compute sufficient statistics from realizations |
| compute_sum_squared_per_ft_tensorized(data, ...) | Compute the square of the residuals per subject per feature |
| compute_sum_squared_tensorized(data, param_ind) | Compute the square of the residuals per subject |
| get_individual_realization_names() | Get names of individual variales of the model. |
| get_individual_variable_name() | Return list of names of the individual variables from the model. |
| get_param_from_real(realizations) | Get individual parameters realizations from all model realizations |
| get_population_realization_names() | Get names of population variales of the model. |
| get_realization_object(n_individuals) | Initialization of a CollectionRealization used during model fitting. |
| initialize(dataset[, method]) | Initialize the model given a dataset and an initialization method. |
| initialize_MCMC_toolbox([set_v0_prior]) | Initialize Monte-Carlo Markov-Chain toolbox for calibration of model |
| load_hyperparameters(hyperparameters) | Load model's hyperparameters |
| load_parameters(parameters) | Instantiate or update the model's parameters. |
| random_variable_informations() | Informations on model's random variables. |
| save(path[, with_mixing_matrix]) | Save Leaspy object as json model parameter file. |
| smart_initialization_realizations(data, ...) | Smart initialization of realizations if needed. |
| time_reparametrization(timepoints, xi, tau) | Tensorized time reparametrization formula |
| update_MCMC_toolbox(...) | Update the MCMC toolbox with a collection of realizations of model population parameters. |
| update_model_parameters(data, ...[, ...]) | Update model parameters (high-level function) |
| update_model_parameters_burn_in(data, ...) | Update model parameters (burn-in phase) |
| update_model_parameters_normal(data, suff_stats) | Update model parameters (after burn-in phase) |

**compute_individual_ages_from_biomarker_values_tensorized_logistic**

**compute_individual_ages_from_biomarker_values**(*value: Union[float, List[float]]*, *individual_parameters: Dict[str, Any]*, *feature: Optional[str] = None*)

For one individual, compute age(s) at which the given features values are reached (given the subject's individual parameters).

Consistency checks are done in the main API layer.

**Parameters**

**value** [scalar or array_like[scalar] (list, tuple, numpy.ndarray)] Contains the biomarker value(s) of the subject.

**individual_parameters** [dict] Contains the individual parameters. Each individual parameter should be a scalar or array_like

**feature** [str (or None)] Name of the considered biomarker (optional for univariate models, compulsory for multivariate models).

**Returns**

torch.Tensor Contains the subject's ages computed at the given values(s) Shape of tensor is (1, n_values)

**Raises**

**LeaspyModelInputError** if computation is tried on more than 1 individual

**compute_individual_ages_from_biomarker_values_tensorized**(*value: Tensor*, *individual_parameters: dict*, *feature: str*)

For one individual, compute age(s) at which the given features values are reached (given the subject's individual parameters), with tensorized inputs

**Parameters**

**value** [torch.Tensor of shape (1, n_values)] Contains the biomarker value(s) of the subject.

**individual_parameters** [dict] Contains the individual parameters. Each individual parameter should be a torch.Tensor

**feature** [str (or None)] Name of the considered biomarker (optional for univariate models, compulsory for multivariate models).

**Returns**

torch.Tensor Contains the subject's ages computed at the given values(s) Shape of tensor is (n_values, 1)

**compute_individual_attachment_tensorized**(*data: Dataset*, *param_ind: DictParamsTorch*, *attribute_type*) → torch.FloatTensor

Compute attachment term (per subject)

**Parameters**

**data** [*Dataset*] Contains the data of the subjects, in particular the subjects' time-points and the mask for nan values & padded visits

**param_ind** [dict] Contain the individual parameters

**attribute_type** [Any, optional] Flag to ask for MCMC attributes instead of model's attributes.

**Returns**

**attachment** [torch.Tensor] Negative Log-likelihood, shape = (n_subjects,)

**Raises**

**LeaspyModelInputError** If invalid *noise_model* for model

**compute_individual_attachment_tensorized_mcmc**(*data: Dataset*, *realizations: CollectionRealization*)

Compute MCMC attachment of all subjects? One subject? One visit? TODO: complete

**Parameters**

**data** [*Dataset*] Contains the data of the subjects, in particular the subjects' time-points and the mask (?)

**realizations** [*CollectionRealization*]

**Returns**

**attachment** [torch.Tensor] The subject attachment (?)

**compute_individual_tensorized**(*timepoints*, *ind_parameters*, *attribute_type=None*)
Compute the individual values at timepoints according to the model.

**Parameters**

**timepoints** [torch.Tensor of shape (n_individuals, n_timepoints)]

**individual_parameters** [dict[param_name: str, torch.Tensor of shape (n_individuals, n_dims_param)]]

**attribute_type** [Any (default None)] Flag to ask for MCMC attributes instead of model's attributes.

**Returns**

**torch.Tensor of shape (n_individuals, n_timepoints, n_features)**

**compute_individual_tensorized_linear**(*timepoints*, *ind_parameters*, *attribute_type=None*)
Compute the individual values at timepoints according to the model (linear).

**Parameters**

**timepoints** [torch.Tensor of shape (n_individuals, n_timepoints)]

**individual_parameters** [dict[param_name: str, torch.Tensor of shape (n_individuals, n_dims_param)]]

**attribute_type** [Any (default None)] Flag to ask for MCMC attributes instead of model's attributes.

**Returns**

**torch.Tensor of shape (n_individuals, n_timepoints, n_features)**

**compute_individual_tensorized_logistic**(*timepoints*, *ind_parameters*, *attribute_type=None*)
Compute the individual values at timepoints according to the model (logistic).

**Parameters**

**timepoints** [torch.Tensor of shape (n_individuals, n_timepoints)]

**individual_parameters** [dict[param_name: str, torch.Tensor of shape (n_individuals, n_dims_param)]]

**attribute_type** [Any (default None)] Flag to ask for MCMC attributes instead of model's attributes.

**Returns**

**torch.Tensor of shape (n_individuals, n_timepoints, n_features)**

**compute_individual_trajectory**(*timepoints*, *individual_parameters: Dict[str, Any]*, ***, *skip_ips_checks: bool = False*)
Compute scores values at the given time-point(s) given a subject's individual parameters.

**Parameters**

**timepoints** [scalar or array_like[scalar] (list, tuple, numpy.ndarray)] Contains the age(s) of the subject.

**individual_parameters** [dict] Contains the individual parameters. Each individual parameter should be a scalar or array_like

**skip_ips_checks** [bool (default: False)] Flag to skip consistency/compatibility checks and tensorization of individual_parameters when it was done earlier (speed-up)

**Returns**

torch.Tensor Contains the subject's scores computed at the given age(s) Shape of tensor is (1, n_tpts, n_features)

**Raises**

**LeaspyModelInputError** if computation is tried on more than 1 individual

**LeaspyIndividualParamsInputError** if invalid individual parameters

**compute_jacobian_tensorized**(*timepoints*, *ind_parameters*, *attribute_type=None*)

Compute the jacobian of the model w.r.t. each individual parameter.

This function aims to be used in *ScipyMinimize* to speed up optimization.

**Parameters**

**timepoints** [torch.Tensor of shape (n_individuals, n_timepoints)]

**individual_parameters** [dict[param_name: str, torch.Tensor of shape (n_individuals, n_dims_param)]]

**attribute_type** [Any (default None)] Flag to ask for MCMC attributes instead of model's attributes.

**Returns**

**dict[param_name: str, torch.Tensor of shape (n_individuals, n_timepoints, n_features, n_dims_param)]**

**compute_jacobian_tensorized_linear**(*timepoints*, *ind_parameters*, *attribute_type=None*)

Compute the jacobian of the model (linear) w.r.t. each individual parameter.

This function aims to be used in *ScipyMinimize* to speed up optimization.

**Parameters**

**timepoints** [torch.Tensor of shape (n_individuals, n_timepoints)]

**individual_parameters** [dict[param_name: str, torch.Tensor of shape (n_individuals, n_dims_param)]]

**attribute_type** [Any (default None)] Flag to ask for MCMC attributes instead of model's attributes.

**Returns**

**dict[param_name: str, torch.Tensor of shape (n_individuals, n_timepoints, n_features, n_dims_param)]**

**compute_jacobian_tensorized_logistic**(*timepoints*, *ind_parameters*, *attribute_type=None*)

Compute the jacobian of the model (logistic) w.r.t. each individual parameter.

This function aims to be used in *ScipyMinimize* to speed up optimization.

**Parameters**

**timepoints** [torch.Tensor of shape (n_individuals, n_timepoints)]

**individual_parameters** [dict[param_name: str, torch.Tensor of shape (n_individuals, n_dims_param)]]

**attribute_type** [Any (default None)] Flag to ask for MCMC attributes instead of model's attributes.

**Returns**

**dict[param_name: str, torch.Tensor of shape (n_individuals, n_timepoints, n_features, n_dims_param)]**

**compute_mean_traj**(*timepoints*)

Compute trajectory of the model with individual parameters being the group-average ones.

TODO check dimensions of io?

**Parameters**

**timepoints** [torch.Tensor [1, n_timepoints]]

**Returns**

**torch.Tensor [1, n_timepoints, dimension]** The group-average values at given time-points

**compute_regularity_realization**(*realization: Realization*)

Compute regularity term for a *Realization* instance.

**Parameters**

**realization** [*Realization*]

**Returns**

**torch.Tensor**

**compute_regularity_variable**(*value: torch.FloatTensor*, *mean: torch.FloatTensor*, *std: torch.FloatTensor*) → torch.FloatTensor

Compute regularity term (Gaussian distribution), low-level.

**Parameters**

**value, mean, std** [torch.Tensor of same shapes]

**Returns**

**torch.Tensor of same shape than input**

**compute_sufficient_statistics**(*data*, *realizations*)

Compute sufficient statistics from realizations

**Parameters**

**data** [*Dataset*]

**realizations** [*CollectionRealization*]

**Returns**

**dict[suff_stat: str, torch.Tensor]**

**compute_sum_squared_per_ft_tensorized**(*data: Dataset*, *param_ind: DictParamsTorch*, *attribute_type=None*) → torch.FloatTensor

Compute the square of the residuals per subject per feature

**Parameters**

**data** [*Dataset*] Contains the data of the subjects, in particular the subjects' time-points and the mask (?)

**param_ind** [dict] Contain the individual parameters

**attribute_type** [Any (default None)] Flag to ask for MCMC attributes instead of model's attributes.

**Returns**

**torch.Tensor of shape (n_individuals,dimension)** Contains L2 residual for each subject and each feature

**compute_sum_squared_tensorized**(*data: Dataset*, *param_ind: DictParamsTorch*, *attribute_type=None*) → torch.FloatTensor

Compute the square of the residuals per subject

**Parameters**

**data** [*Dataset*] Contains the data of the subjects, in particular the subjects' time-points and the mask (?)

**param_ind** [dict] Contain the individual parameters

**attribute_type** [Any (default None)] Flag to ask for MCMC attributes instead of model's attributes.

**Returns**

**torch.Tensor of shape (n_individuals,)** Contains L2 residual for each subject

**get_individual_realization_names**()

Get names of individual variales of the model.

**Returns**

**list[str]**

**get_individual_variable_name**()

Return list of names of the individual variables from the model.

Duplicate of *get_individual_realization_names()*

TODO delete one of them

**Returns**

**individual_variable_name** [list [str]] Contains the individual variables' names

**get_param_from_real**(*realizations:* CollectionRealization) → Dict[str, torch.FloatTensor]

Get individual parameters realizations from all model realizations

**Parameters**

**realizations** [*CollectionRealization*]

**Returns**

**dict[param_name: str, torch.Tensor [n_individuals, dims_param]]** Individual parameters

**get_population_realization_names**()

Get names of population variales of the model.

**Returns**

**list[str]**

**get_realization_object**(*n_individuals: int*) → *CollectionRealization*

Initialization of a *CollectionRealization* used during model fitting.

**Parameters**

**n_individuals** [int] Number of individuals to track

**Returns**

*CollectionRealization*

`initialize`(*dataset*, *method='default'*)
Initialize the model given a dataset and an initialization method.

After calling this method `is_initialized` should be True and model should be ready for use.

**Parameters**

**dataset** [*Dataset*] The dataset we want to initialize from.

**method** [str] A custom method to initialize the model

`initialize_MCMC_toolbox`(*set_v0_prior=False*)
Initialize Monte-Carlo Markov-Chain toolbox for calibration of model

TODO to move in a "MCMC-model interface"

`load_hyperparameters`(*hyperparameters*)
Load model's hyperparameters

**Parameters**

**hyperparameters** [dict[str, Any]] Contains the model's hyperparameters

**Raises**

**LeaspyModelInputError** If any of the consistency checks fail.

`load_parameters`(*parameters*)
Instantiate or update the model's parameters.

**Parameters**

**parameters** [dict[str, Any]] Contains the model's parameters

`random_variable_informations`()
Informations on model's random variables.

**Returns**

**dict[str, Any]**

`save`(*path*, *with_mixing_matrix=True*, ***kwargs*)
Save Leaspy object as json model parameter file.

**Parameters**

**path** [str] Path to store the model's parameters.

**with_mixing_matrix** [bool (default True)] Save the mixing matrix in the exported file in its 'parameters' section. <!> It is not a real parameter and its value will be overwritten at model loading (orthonormal basis is recomputed from other "true" parameters and mixing matrix is then deduced from this orthonormal basis and the betas)! It was integrated historically because it is used for convenience in browser webtool and only there…

****kwargs** Keyword arguments for json.dump method. Default to: dict(indent=2)

`smart_initialization_realizations`(*data: Dataset*, *realizations: CollectionRealization*)
Smart initialization of realizations if needed.

Default behavior to return *realizations* as they are (no smart trick).

**Parameters**

**data** [*Dataset*]

**realizations** [*CollectionRealization*]

**Returns**

*CollectionRealization*

**static time_reparametrization**(*timepoints: torch.FloatTensor*, *xi: torch.FloatTensor*, *tau: torch.FloatTensor*) → torch.FloatTensor

Tensorized time reparametrization formula

<!> Shapes of tensors must be compatible between them.

**Parameters**

**timepoints** [torch.Tensor] Timepoints to reparametrize

**xi** [torch.Tensor] Log-acceleration of individual(s)

**tau** [torch.Tensor] Time-shift(s)

**Returns**

**torch.Tensor of same shape as *timepoints***

**update_MCMC_toolbox**(*name_of_the_variables_that_have_been_changed*, *realizations*)

Update the MCMC toolbox with a collection of realizations of model population parameters.

TODO to move in a "MCMC-model interface"

**Parameters**

**name_of_the_variables_that_have_been_changed** [container[str] (list, tuple, ...)] Names of the population parameters to update in MCMC toolbox

**realizations** [*CollectionRealization*] All the realizations to update MCMC toolbox with

**update_model_parameters**(*data: Dataset*, *reals_or_suff_stats: Union[CollectionRealization, DictParamsTorch]*, *burn_in_phase=True*)

Update model parameters (high-level function)

Under-the-hood call *update_model_parameters_burn_in()* or *update_model_parameters_normal()* depending on the phase of the fit algorithm

**Parameters**

**data** [*Dataset*]

**reals_or_suff_stats :**

**If during burn-in phase will be realizations:** *CollectionRealization*

**If after burn-in phase will be sufficient statistics:** dict[suff_stat: str, torch.Tensor]

**update_model_parameters_burn_in**(*data*, *realizations*)

Update model parameters (burn-in phase)

**Parameters**

**data** [*Dataset*]

**realizations** [*CollectionRealization*]

**update_model_parameters_normal**(*data*, *suff_stats*)

Update model parameters (after burn-in phase)

**Parameters**

**data** [*Dataset*]

**suff_stats** [dict[suff_stat: str, torch.Tensor]]

### 3.2.6 leaspy.models.multivariate_parallel_model.MultivariateParallelModel

**class** MultivariateParallelModel(*name: str*, ***kwargs*)

Bases: *leaspy.models.abstract_multivariate_model.AbstractMultivariateModel*

Logistic model for multiple variables of interest, imposing same average evolution pace for all variables (logistic curves are only time-shifted).

**Parameters**

**name** [str] Name of the model

****kwargs** Hyperparameters of the model

#### Methods

| | |
|---|---|
| *compute_individual_ages_from_biomarker_values*(...) | For one individual, compute age(s) at which the given features values are reached (given the subject's individual parameters). |
| *compute_individual_ages_from_biomarker_values_tensorized*(...) | For one individual, compute age(s) at which the given features values are reached (given the subject's individual parameters), with tensorized inputs |
| *compute_individual_attachment_tensorized*(...) | Compute attachment term (per subject) |
| *compute_individual_attachment_tensorized_mcmc*(...) | Compute MCMC attachment of all subjects? One subject? One visit? TODO: complete |
| *compute_individual_tensorized*(timepoints, ...) | Compute the individual values at timepoints according to the model. |
| *compute_individual_trajectory*(timepoints, ...) | Compute scores values at the given time-point(s) given a subject's individual parameters. |
| *compute_jacobian_tensorized*(timepoints, ...) | Compute the jacobian of the model w.r.t. |
| *compute_mean_traj*(timepoints) | Compute trajectory of the model with individual parameters being the group-average ones. |
| *compute_regularity_realization*(realization) | Compute regularity term for a *Realization* instance. |
| *compute_regularity_variable*(value, mean, std) | Compute regularity term (Gaussian distribution), low-level. |
| *compute_sufficient_statistics*(data, realizations) | Compute sufficient statistics from realizations |
| *compute_sum_squared_per_ft_tensorized*(data, ...) | Compute the square of the residuals per subject per feature |
| *compute_sum_squared_tensorized*(data, param_ind) | Compute the square of the residuals per subject |
| *get_individual_realization_names*() | Get names of individual variales of the model. |
| *get_individual_variable_name*() | Return list of names of the individual variables from the model. |
| *get_param_from_real*(realizations) | Get individual parameters realizations from all model realizations |
| *get_population_realization_names*() | Get names of population variales of the model. |
| *get_realization_object*(n_individuals) | Initialization of a *CollectionRealization* used during model fitting. |
| *initialize*(dataset[, method]) | Initialize the model given a dataset and an initialization method. |

continues on next page

Table 10 – continued from previous page

| | |
|---|---|
| *initialize_MCMC_toolbox*() | Initialize Monte-Carlo Markov-Chain toolbox for calibration of model |
| *load_hyperparameters*(hyperparameters) | Load model's hyperparameters |
| *load_parameters*(parameters) | Instantiate or update the model's parameters. |
| *random_variable_informations*() | Informations on model's random variables. |
| *save*(path[, with_mixing_matrix]) | Save Leaspy object as json model parameter file. |
| *smart_initialization_realizations*(data, ...) | Smart initialization of realizations if needed. |
| *time_reparametrization*(timepoints, xi, tau) | Tensorized time reparametrization formula |
| *update_MCMC_toolbox*(...) | Update the MCMC toolbox with a collection of realizations of model population parameters. |
| *update_model_parameters*(data, ...[, ...]) | Update model parameters (high-level function) |
| *update_model_parameters_burn_in*(data, ...) | Update model parameters (burn-in phase) |
| *update_model_parameters_normal*(data, suff_stats) | Update model parameters (after burn-in phase) |

**compute_individual_ages_from_biomarker_values**(*value: Union[float, List[float]]*, *individual_parameters: Dict[str, Any]*, *feature: Optional[str] = None*)

For one individual, compute age(s) at which the given features values are reached (given the subject's individual parameters).

Consistency checks are done in the main API layer.

**Parameters**

**value** [scalar or array_like[scalar] (list, tuple, `numpy.ndarray`)] Contains the biomarker value(s) of the subject.

**individual_parameters** [dict] Contains the individual parameters. Each individual parameter should be a scalar or array_like

**feature** [str (or None)] Name of the considered biomarker (optional for univariate models, compulsory for multivariate models).

**Returns**

`torch.Tensor` Contains the subject's ages computed at the given values(s) Shape of tensor is (1, n_values)

**Raises**

**LeaspyModelInputError** if computation is tried on more than 1 individual

**compute_individual_ages_from_biomarker_values_tensorized**(*value*, *individual_parameters*, *feature*)

For one individual, compute age(s) at which the given features values are reached (given the subject's individual parameters), with tensorized inputs

**Parameters**

**value** [torch.Tensor of shape (1, n_values)] Contains the biomarker value(s) of the subject.

**individual_parameters** [dict] Contains the individual parameters. Each individual parameter should be a torch.Tensor

**feature** [str (or None)] Name of the considered biomarker (optional for univariate models, compulsory for multivariate models).

**Returns**

torch.Tensor Contains the subject's ages computed at the given values(s) Shape of tensor is (n_values, 1)

**compute_individual_attachment_tensorized**(*data: Dataset*, *param_ind: DictParamsTorch*, *attribute_type*) → torch.FloatTensor

Compute attachment term (per subject)

**Parameters**

**data** [*Dataset*] Contains the data of the subjects, in particular the subjects' time-points and the mask for nan values & padded visits

**param_ind** [dict] Contain the individual parameters

**attribute_type** [Any, optional] Flag to ask for MCMC attributes instead of model's attributes.

**Returns**

**attachment** [torch.Tensor] Negative Log-likelihood, shape = (n_subjects,)

**Raises**

**LeaspyModelInputError** If invalid *noise_model* for model

**compute_individual_attachment_tensorized_mcmc**(*data: Dataset*, *realizations: CollectionRealization*)

Compute MCMC attachment of all subjects? One subject? One visit? TODO: complete

**Parameters**

**data** [*Dataset*] Contains the data of the subjects, in particular the subjects' time-points and the mask (?)

**realizations** [*CollectionRealization*]

**Returns**

**attachment** [torch.Tensor] The subject attachment (?)

**compute_individual_tensorized**(*timepoints*, *ind_parameters*, *attribute_type=None*)

Compute the individual values at timepoints according to the model.

**Parameters**

**timepoints** [torch.Tensor of shape (n_individuals, n_timepoints)]

**individual_parameters** [dict[param_name: str, torch.Tensor of shape (n_individuals, n_dims_param)]]

**attribute_type** [Any (default None)] Flag to ask for MCMC attributes instead of model's attributes.

**Returns**

**torch.Tensor of shape (n_individuals, n_timepoints, n_features)**

**compute_individual_trajectory**(*timepoints*, *individual_parameters: Dict[str, Any]*, ***, *skip_ips_checks: bool = False*)

Compute scores values at the given time-point(s) given a subject's individual parameters.

**Parameters**

**timepoints** [scalar or array_like[scalar] (list, tuple, numpy.ndarray)] Contains the age(s) of the subject.

**individual_parameters** [dict] Contains the individual parameters. Each individual parameter should be a scalar or array_like

**skip_ips_checks** [bool (default: False)] Flag to skip consistency/compatibility checks and tensorization of individual_parameters when it was done earlier (speed-up)

**Returns**

`torch.Tensor` Contains the subject's scores computed at the given age(s) Shape of tensor is (1, n_tpts, n_features)

**Raises**

`LeaspyModelInputError` if computation is tried on more than 1 individual

`LeaspyIndividualParamsInputError` if invalid individual parameters

`compute_jacobian_tensorized`(*timepoints*, *ind_parameters*, *attribute_type=None*)

Compute the jacobian of the model w.r.t. each individual parameter.

This function aims to be used in *ScipyMinimize* to speed up optimization.

**Parameters**

**timepoints** [`torch.Tensor` of shape (n_individuals, n_timepoints)]

**individual_parameters** [dict[param_name: str, `torch.Tensor` of shape (n_individuals, n_dims_param)]]

**attribute_type** [Any (default None)] Flag to ask for MCMC attributes instead of model's attributes.

**Returns**

**dict[param_name: str, `torch.Tensor` of shape (n_individuals, n_timepoints, n_features, n_dims_param)]**

`compute_mean_traj`(*timepoints*)

Compute trajectory of the model with individual parameters being the group-average ones.

TODO check dimensions of io?

**Parameters**

**timepoints** [`torch.Tensor` [1, n_timepoints]]

**Returns**

`torch.Tensor` **[1, n_timepoints, dimension]** The group-average values at given timepoints

`compute_regularity_realization`(*realization: Realization*)

Compute regularity term for a *Realization* instance.

**Parameters**

**realization** [*Realization*]

**Returns**

`torch.Tensor`

`compute_regularity_variable`(*value: torch.FloatTensor*, *mean: torch.FloatTensor*, *std: torch.FloatTensor*) → torch.FloatTensor

Compute regularity term (Gaussian distribution), low-level.

**Parameters**

**value, mean, std** [torch.Tensor of same shapes]

**Returns**

**torch.Tensor of same shape than input**

**compute_sufficient_statistics**(*data*, *realizations*)
Compute sufficient statistics from realizations

**Parameters**

**data** [*Dataset*]

**realizations** [*CollectionRealization*]

**Returns**

**dict[suff_stat: str, torch.Tensor]**

**compute_sum_squared_per_ft_tensorized**(*data: Dataset*, *param_ind: DictParamsTorch*, *attribute_type=None*) → torch.FloatTensor
Compute the square of the residuals per subject per feature

**Parameters**

**data** [*Dataset*] Contains the data of the subjects, in particular the subjects' time-points and the mask (?)

**param_ind** [dict] Contain the individual parameters

**attribute_type** [Any (default None)] Flag to ask for MCMC attributes instead of model's attributes.

**Returns**

**torch.Tensor of shape (n_individuals,dimension)** Contains L2 residual for each subject and each feature

**compute_sum_squared_tensorized**(*data: Dataset*, *param_ind: DictParamsTorch*, *attribute_type=None*) → torch.FloatTensor
Compute the square of the residuals per subject

**Parameters**

**data** [*Dataset*] Contains the data of the subjects, in particular the subjects' time-points and the mask (?)

**param_ind** [dict] Contain the individual parameters

**attribute_type** [Any (default None)] Flag to ask for MCMC attributes instead of model's attributes.

**Returns**

**torch.Tensor of shape (n_individuals,)** Contains L2 residual for each subject

**get_individual_realization_names**()
Get names of individual variales of the model.

**Returns**

**list[str]**

**get_individual_variable_name**()
Return list of names of the individual variables from the model.

Duplicate of *get_individual_realization_names()*

TODO delete one of them

**Returns**

**individual_variable_name** [list [str]] Contains the individual variables' names

`get_param_from_real`(*realizations:* CollectionRealization) → Dict[str, torch.FloatTensor]
Get individual parameters realizations from all model realizations

**Parameters**

**realizations** [*CollectionRealization*]

**Returns**

**dict[param_name: str, `torch.Tensor` [n_individuals, dims_param]]** Individual parameters

`get_population_realization_names`()
Get names of population variales of the model.

**Returns**

**list[str]**

`get_realization_object`(*n_individuals: int*) → *CollectionRealization*
Initialization of a *CollectionRealization* used during model fitting.

**Parameters**

**n_individuals** [int] Number of individuals to track

**Returns**

*CollectionRealization*

`initialize`(*dataset*, *method='default'*)
Initialize the model given a dataset and an initialization method.

After calling this method `is_initialized` should be True and model should be ready for use.

**Parameters**

**dataset** [*Dataset*] The dataset we want to initialize from.

**method** [str] A custom method to initialize the model

`initialize_MCMC_toolbox`()
Initialize Monte-Carlo Markov-Chain toolbox for calibration of model

TODO to move in a "MCMC-model interface"

`load_hyperparameters`(*hyperparameters*)
Load model's hyperparameters

**Parameters**

**hyperparameters** [dict[str, Any]] Contains the model's hyperparameters

**Raises**

**`LeaspyModelInputError`** If any of the consistency checks fail.

`load_parameters`(*parameters*)
Instantiate or update the model's parameters.

**Parameters**

**parameters** [dict[str, Any]] Contains the model's parameters

**random_variable_informations**()
Informations on model's random variables.

**Returns**

**dict[str, Any]**

**save**(*path*, *with_mixing_matrix=True*, ***kwargs*)
Save Leaspy object as json model parameter file.

**Parameters**

**path** [str] Path to store the model's parameters.

**with_mixing_matrix** [bool (default True)] Save the mixing matrix in the exported file in its 'parameters' section. <!> It is not a real parameter and its value will be overwritten at model loading (orthonormal basis is recomputed from other "true" parameters and mixing matrix is then deduced from this orthonormal basis and the betas)! It was integrated historically because it is used for convenience in browser webtool and only there…

****kwargs** Keyword arguments for json.dump method. Default to: dict(indent=2)

**smart_initialization_realizations**(*data: Dataset*, *realizations: CollectionRealization*)
Smart initialization of realizations if needed.

Default behavior to return *realizations* as they are (no smart trick).

**Parameters**

**data** [*Dataset*]

**realizations** [*CollectionRealization*]

**Returns**

*CollectionRealization*

**static time_reparametrization**(*timepoints: torch.FloatTensor*, *xi: torch.FloatTensor*, *tau: torch.FloatTensor*) → torch.FloatTensor
Tensorized time reparametrization formula

<!> Shapes of tensors must be compatible between them.

**Parameters**

**timepoints** [torch.Tensor] Timepoints to reparametrize

**xi** [torch.Tensor] Log-acceleration of individual(s)

**tau** [torch.Tensor] Time-shift(s)

**Returns**

**torch.Tensor of same shape as *timepoints***

**update_MCMC_toolbox**(*name_of_the_variables_that_have_been_changed*, *realizations*)
Update the MCMC toolbox with a collection of realizations of model population parameters.

TODO to move in a "MCMC-model interface"

**Parameters**

**name_of_the_variables_that_have_been_changed** [container[str] (list, tuple, …)] Names of the population parameters to update in MCMC toolbox

**realizations** [*CollectionRealization*] All the realizations to update MCMC toolbox with

**update_model_parameters**(*data: Dataset*, *reals_or_suff_stats: Union[CollectionRealization, DictParamsTorch]*, *burn_in_phase=True*)
Update model parameters (high-level function)

Under-the-hood call *update_model_parameters_burn_in()* or *update_model_parameters_normal()* depending on the phase of the fit algorithm

**Parameters**

**data** [*Dataset*]

**reals_or_suff_stats :**

**If during burn-in phase will be realizations:** *CollectionRealization*

**If after burn-in phase will be sufficient statistics:** dict[suff_stat: str, torch.Tensor]

**update_model_parameters_burn_in**(*data*, *realizations*)
Update model parameters (burn-in phase)

**Parameters**

**data** [*Dataset*]

**realizations** [*CollectionRealization*]

**update_model_parameters_normal**(*data*, *suff_stats*)
Update model parameters (after burn-in phase)

**Parameters**

**data** [*Dataset*]

**suff_stats** [dict[suff_stat: str, torch.Tensor]]

### 3.2.7 leaspy.models.lme_model.LMEModel

**class LMEModel**(*name: str*, ***kwargs*)
Bases: leaspy.models.generic_model.GenericModel

LMEModel is a benchmark model that fits and personalize a linear mixed-effects model

The model specification is the following:

$$y_{ij} = fixed_{intercept} + random_{intercept_i} + (fixed_{slopeAge} + random_{slopeAge_i}) * age_{ij} + \epsilon_{ij}$$

**with:**

- $y_{ij}$: value of the feature of the i-th subject at his j-th visit,
- $age_{ij}$: age of the i-th subject at his j-th visit.
- $\epsilon_{ij}$: residual Gaussian noise (independent between visits)

<!> This model must be fitted on one feature only (univariate model).

TODO? add some covariates in this very simple model.

**Parameters**

**name** [str] The model's name

****kwargs**

**Model hyperparameters:**

- with_random_slope_age : bool (default True)

**See also:**

*LMEFitAlgorithm*

*LMEPersonalizeAlgorithm*

**Attributes**

**name** [str] The model's name

**is_initialized** [bool] Is the model initialized?

**with_random_slope_age** [bool (default True)] Has the LME a random slope for subject's age? Otherwise it only has a random intercept per subject

**features** [list[str]] List of the model features <!> LME has only one feature.

**dimension** [int] Will always be 1 (univariate)

**parameters** [dict]

**Contains the model parameters. In particular:**

- **ages_mean** [float] Mean of ages (for normalization)
- **ages_std** [float] Std-dev of ages (for normalization)
- **fe_params** [np.ndarray[float]] Fixed effects
- **cov_re** [np.ndarray[float, float]] Variance-covariance matrix of random-effects
- **cov_re_unscaled_inv** [np.ndarray[float, float]] Inverse of unscaled (= divided by variance of noise) variance-covariance matrix of random-effects. This matrix is used for personalization to new subjects.
- **noise_std** [float] Std-dev of Gaussian noise
- **bse_fe, bse_re** [np.ndarray[float]] Standard errors on fixed-effects and random-effects respectively (not used in Leaspy).

### Methods

| | |
|---|---|
| *compute_individual_trajectory*(timepoints, ip) | Compute scores values at the given time-point(s) given a subject's individual parameters. |
| *get_hyperparameters*(*[, with_features, ...]) | Get all model hyperparameters |
| *hyperparameters_ok*() | Check all model hyperparameters are ok |
| *initialize*(dataset[, method]) | Initialize the model given a dataset and an initialization method. |
| *load_hyperparameters*(hyperparameters, *[, ...]) | Load model hyperparameters from a dict |
| *load_parameters*(parameters, *[, list_converter]) | Instantiate or update the model's parameters. |
| *save*(path, **kwargs) | Save Leaspy object as json model parameter file. |
| *validate_compatibility_of_dataset*(dataset) | Raise if the given dataset is not compatible with the current model. |

**compute_individual_trajectory**(*timepoints*, *ip: dict*)

Compute scores values at the given time-point(s) given a subject's individual parameters.

**Parameters**

**timepoints** [array-like of ages (not normalized)] Timepoints to compute individual trajectory at

**ip** [dict]

**Individual parameters:**

- random_intercept
- random_slope_age (if `with_random_slope_age == True`)

**Returns**

**`torch.Tensor` of float of shape (n_individuals == 1, n_tpts == len(timepoints), n_features == 1)**

`get_hyperparameters`(*, *with_features=True*, *with_properties=True*, *default=None*) → Dict[str, Any]
Get all model hyperparameters

**Parameters**

**with_features, with_properties** [bool (default True)] Whether to include *features* and respectively all *_properties* (i.e. _dynamic_ hyperparameters) in the returned dictionary

**default** [Any] Default value is something is an hyperparameter is missing (should not!)

**Returns**

**dict { hyperparam_name** [str -> hyperparam_value]

`hyperparameters_ok`() → bool
Check all model hyperparameters are ok

**Returns**

**bool**

`initialize`(*dataset: Dataset*, *method: str = None*)
Initialize the model given a dataset and an initialization method.

After calling this method `is_initialized` should be True and model should be ready for use.

**Parameters**

**dataset** [`Dataset`] The dataset we want to initialize from.

**method** [str, optional (default None)] A custom method to initialize the model

`load_hyperparameters`(*hyperparameters: Dict[str, Any]*, *, *with_defaults: bool = False*) → None
Load model hyperparameters from a dict

**Parameters**

**hyperparameters** [dict[str, Any]] Contains the model's hyperparameters

**with_defaults** [bool (default False)] If true, it also resets hyperparameters that are part of the model but not included in *hyperparameters* to their default value.

**Raises**

**`LeaspyModelInputError`** if inconsistent hyperparameters

`load_parameters`(*parameters*, *, *list_converter=<built-in function array>*) → None
Instantiate or update the model's parameters.

**Parameters**

**parameters** [dict] Contains the model's parameters

**save**(*path: str*, ***kwargs*)
Save Leaspy object as json model parameter file.

Default save method: it can be overwritten in child class but should be generic…

**Parameters**

**path** [str] Path to store the model's parameters.

****kwargs** Keyword arguments for json.dump method.

**validate_compatibility_of_dataset**(*dataset: Dataset*)
Raise if the given dataset is not compatible with the current model.

**Parameters**

**dataset** [*Dataset*] The dataset we want to model.

**Raises**

**LeaspyDataInputError :** if data is not univariate.

### 3.2.8 leaspy.models.constant_model.ConstantModel

**class ConstantModel**(*name: str*, ***kwargs*)
Bases: leaspy.models.generic_model.GenericModel

*ConstantModel* is a benchmark model that predicts constant values (no matter what the patient's ages are).

These constant values depend on the algorithm setting and the patient's values provided during calibration. It could predict:

- *last*: last value seen during calibration (even if NaN),
- *last_known*: last non NaN value seen during calibration*§,
- *max*: maximum (=worst) value seen during calibration*§,
- *mean*: average of values seen during calibration§.

* <!> depending on features, the *last_known* / *max* value may correspond to different visits.
§ <!> for a given feature, value will be NaN if and only if all values for this feature were NaN.

**Parameters**

**name** [str] The model's name

****kwargs** Hyperparameters for the model. None supported for now.

**See also:**

*ConstantPredictionAlgorithm*

**Attributes**

**name** [str] The model's name

**is_initialized** [bool] Always True (no true initialization needed for constant model)

**features** [list[str]] List of the model features. Unlike most models features will be determined at *personalization* only (because it does not needed any *fit*)

**dimension** [int] Number of features (read-only)

**parameters** [dict] Model has no parameters: empty dictionary. The *prediction_type* parameter should be defined during *personalization*. Example:

```
>>> AlgorithmSettings('constant_prediction', prediction_type='last_
↪known')
```

### Methods

| | |
|---|---|
| *compute_individual_trajectory*(timepoints, ip) | Compute scores values at the given time-point(s) given a subject's individual parameters. |
| *get_hyperparameters*(*[, with_features, ...]) | Get all model hyperparameters |
| *hyperparameters_ok*() | Check all model hyperparameters are ok |
| *initialize*(dataset[, method]) | Initialize the model given a dataset and an initialization method. |
| *load_hyperparameters*(hyperparameters, *[, ...]) | Load model hyperparameters from a dict |
| *load_parameters*(parameters, *[, list_converter]) | Instantiate or update the model's parameters. |
| *save*(path, **kwargs) | Save Leaspy object as json model parameter file. |
| *validate_compatibility_of_dataset*(dataset) | Raise if the given dataset is not compatible with the current model. |

**compute_individual_trajectory**(*timepoints*, *ip*)
Compute scores values at the given time-point(s) given a subject's individual parameters.

**Parameters**

**timepoints** [scalar or array_like[scalar] (list, tuple, `numpy.ndarray`)] Contains the age(s) of the subject.

**individual_parameters** [dict] Contains the individual parameters. Each individual parameter should be a scalar or array_like

****kws** extra model specific keyword-arguments

**Returns**

`torch.Tensor` Contains the subject's scores computed at the given age(s) Shape of tensor is (1, n_tpts, n_features)

**get_hyperparameters**(*, *with_features=True*, *with_properties=True*, *default=None*) → Dict[str, Any]
Get all model hyperparameters

**Parameters**

**with_features, with_properties** [bool (default True)] Whether to include *features* and respectively all *_properties* (i.e. _dynamic_ hyperparameters) in the returned dictionary

**default** [Any] Default value is something is an hyperparameter is missing (should not!)

**Returns**

**dict { hyperparam_name** [str -> hyperparam_value]

**hyperparameters_ok**() → bool
Check all model hyperparameters are ok

**Returns**

**bool**

**initialize**(*dataset: Dataset*, *method: str = None*)
Initialize the model given a dataset and an initialization method.

After calling this method `is_initialized` should be True and model should be ready for use.

**Parameters**

**dataset** [*Dataset*] The dataset we want to initialize from.

**method** [str, optional (default None)] A custom method to initialize the model

**load_hyperparameters**(*hyperparameters: Dict[str, Any]*, ***, *with_defaults: bool = False*) → None
Load model hyperparameters from a dict

**Parameters**

**hyperparameters** [dict[str, Any]] Contains the model's hyperparameters

**with_defaults** [bool (default False)] If true, it also resets hyperparameters that are part of the model but not included in *hyperparameters* to their default value.

**Raises**

**LeaspyModelInputError** if inconsistent hyperparameters

**load_parameters**(*parameters*, ***, *list_converter=<built-in function array>*) → None
Instantiate or update the model's parameters.

**Parameters**

**parameters** [dict] Contains the model's parameters

**save**(*path: str*, ***kwargs*)
Save Leaspy object as json model parameter file.

Default save method: it can be overwritten in child class but should be generic…

**Parameters**

**path** [str] Path to store the model's parameters.

****kwargs** Keyword arguments for json.dump method.

**validate_compatibility_of_dataset**(*dataset: Dataset*)
Raise if the given dataset is not compatible with the current model.

**Parameters**

**dataset** [*Dataset*] The dataset we want to model.

**Raises**

**LeaspyDataInputError** If and only if data is incompatible with model.

### 3.2.9 leaspy.models.utils.attributes: Models' attributes

Attributes used by the models.

| | |
|---|---|
| *attributes_factory.AttributesFactory*() | Return an *Attributes* class object based on the given parameters. |
| *abstract_attributes.AbstractAttributes*(name) | Abstract base class for attributes of models. |

continues on next page

Table 13 – continued from previous page

| | |
|---|---|
| *abstract_manifold_model_attributes.AbstractManifoldModelAttributes*(...) | Abstract base class for attributes of leaspy manifold models. |
| *linear_attributes.LinearAttributes*(name, ...) | Attributes of leaspy linear models. |
| *logistic_attributes.LogisticAttributes*(name, ...) | Attributes of leaspy logistic models. |
| *logistic_parallel_attributes.LogisticParallelAttributes*(...) | Attributes of leaspy logistic parallel models. |

### leaspy.models.utils.attributes.attributes_factory.AttributesFactory

**class AttributesFactory**

Bases: object

Return an *Attributes* class object based on the given parameters.

#### Methods

| | | |
|---|---|---|
| *attributes*(name, dimension[, source_dimension]) | | Class method to build correct model attributes depending on model *name*. |

**classmethod attributes**(*name: str*, *dimension: int*, *source_dimension: Optional[int] = None*) → *AbstractAttributes*

Class method to build correct model attributes depending on model *name*.

**Parameters**

**name** [str]

**dimension** [int]

**source_dimension** [int, optional (default None)]

**Returns**

*AbstractAttributes*

**Raises**

**LeaspyModelInputError** if any inconsistent parameter.

### leaspy.models.utils.attributes.abstract_attributes.AbstractAttributes

**class AbstractAttributes**(*name: str*, *dimension: Optional[int] = None*, *source_dimension: Optional[int] = None*)

Bases: abc.ABC

Abstract base class for attributes of models.

Contains the common attributes & methods of the different attributes classes. Such classes are used to update the models' attributes.

**Parameters**

**name** [str]

**dimension** [int (default None)]

**source_dimension** [int (default None)]

**Raises**

**LeaspyModelInputError** if any inconsistent parameter.

**Attributes**

**name** [str] Name of the associated leaspy model.

**dimension** [int] Number of features of the model

**source_dimension** [int] Number of sources of the model TODO? move to AbstractManifoldModelAttributes?

**univariate** [bool] Whether model is univariate or not (i.e. dimension == 1)

**has_sources** [bool] Whether model has sources or not (not univariate and source_dimension >= 1) TODO? move to AbstractManifoldModelAttributes?

**update_possibilities** [tuple[str] (default empty)] Contains the available parameters to update. Different models have different parameters.

### Methods

| | |
|---|---|
| *get_attributes*() | Returns the essential attributes of a given model. |
| *update*(names_of_changes_values, values) | Update model group average parameter(s). |

**abstract get_attributes**() → Tuple[torch.FloatTensor, ...]
Returns the essential attributes of a given model.

**Returns**

**Depends on the subclass, please refer to each specific class.**

**abstract update**(*names_of_changes_values: Tuple[str, ...]*, *values: Dict[str, torch.FloatTensor]*) → None
Update model group average parameter(s).

**Parameters**

**names_of_changed_values** [list [str]] Values to be updated

**values** [dict [str, *torch.Tensor*]] New values used to update the model's group average parameters

**Raises**

**LeaspyModelInputError** If *names_of_changed_values* contains unknown values to update.

## leaspy.models.utils.attributes.abstract_manifold_model_attributes.AbstractManifoldModelAttributes

**class AbstractManifoldModelAttributes**(*name: str*, *dimension: int*, *source_dimension: Optional[int] = None*)

Bases: *leaspy.models.utils.attributes.abstract_attributes.AbstractAttributes*

Abstract base class for attributes of leaspy manifold models.

Contains the common attributes & methods of the different attributes classes. Such classes are used to update the models' attributes.

**Parameters**

**name** [str]

**dimension** [int]

**source_dimension** [int (default None)]

**Raises**

**LeaspyModelInputError** if any inconsistent parameter.

**Attributes**

**name** [str (default None)] Name of the associated leaspy model.

**dimension** [int]

**source_dimension** [int]

**univariate** [bool] Whether model is univariate or not (i.e. dimension == 1)

**has_sources** [bool] Whether model has sources or not (not univariate and source_dimension >= 1)

**update_possibilities** [tuple [str], (default ('all', 'g', 'v0', 'betas') )] Contains the available parameters to update. Different models have different parameters.

**positions** [torch.Tensor [dimension] (default None)] <!> Depending on the submodel it does not correspond to the same thing.

**velocities** [torch.Tensor [dimension] (default None)] Vector of velocities for each feature (positive components).

**orthonormal_basis** [torch.Tensor [dimension, dimension - 1] (default None)]

**betas** [torch.Tensor [dimension - 1, source_dimension] (default None)]

**mixing_matrix** [torch.Tensor [dimension, source_dimension] (default None)] Matrix A such that w_i = A * s_i.

### Methods

| | |
|---|---|
| *get_attributes*() | Returns the following attributes: positions, velocities & mixing_matrix. |
| *update*(names_of_changes_values, values) | Update model group average parameter(s). |

**get_attributes**()
Returns the following attributes: positions, velocities & mixing_matrix.

**Returns**

**For univariate models:** positions: *torch.Tensor*

**For not univariate models:**

- positions: *torch.Tensor*
- velocities: *torch.Tensor*
- mixing_matrix: *torch.Tensor*

**abstract update**(*names_of_changes_values: Tuple[str, ...]*, *values: Dict[str, torch.FloatTensor]*) → None
Update model group average parameter(s).

**Parameters**

**names_of_changed_values** [list [str]] Values to be updated

**values** [dict [str, *torch.Tensor*]] New values used to update the model's group average parameters

**Raises**

**LeaspyModelInputError** If *names_of_changed_values* contains unknown values to update.

### leaspy.models.utils.attributes.linear_attributes.LinearAttributes

**class LinearAttributes**(*name*, *dimension*, *source_dimension*)

Bases: *leaspy.models.utils.attributes.abstract_manifold_model_attributes.AbstractManifoldModelAttributes*

Attributes of leaspy linear models.

Contains the common attributes & methods to update the linear model's attributes.

**Parameters**

**name** [str]

**dimension** [int]

**source_dimension** [int]

**See also:**

*UnivariateModel*

*MultivariateModel*

**Attributes**

**name** [str (default 'linear')] Name of the associated leaspy model.

**dimension** [int]

**source_dimension** [int]

**univariate** [bool] Whether model is univariate or not (i.e. dimension == 1)

**has_sources** [bool] Whether model has sources or not (not univariate and source_dimension >= 1)

**update_possibilities** [tuple [str] (default ('all', 'g', 'v0', 'betas') )] Contains the available parameters to update. Different models have different parameters.

**positions** [torch.Tensor [dimension] (default None)] positions = realizations['g'] such that "p0" = positions

**velocities** [torch.Tensor [dimension] (default None)] Always positive: exp(realizations['v0'])

**orthonormal_basis** [torch.Tensor [dimension, dimension - 1] (default None)]

**betas** [torch.Tensor [dimension - 1, source_dimension] (default None)]

**mixing_matrix** [torch.Tensor [dimension, source_dimension] (default None)] Matrix A such that w_i = A * s_i.

### Methods

| | |
|---|---|
| *get_attributes*() | Returns the following attributes: `positions`, `velocities` & `mixing_matrix`. |
| *update*(names_of_changed_values, values) | Update model group average parameter(s). |

**get_attributes**()
Returns the following attributes: `positions`, `velocities` & `mixing_matrix`.

**Returns**

**For univariate models:** positions: *torch.Tensor*

**For not univariate models:**

- positions: *torch.Tensor*
- velocities: *torch.Tensor*
- mixing_matrix: *torch.Tensor*

**update**(*names_of_changed_values*, *values*)
Update model group average parameter(s).

**Parameters**

**names_of_changed_values** [list [str]]

**Elements of list must be either:**

- `all` (update everything)
- `g` correspond to the attribute `positions`.
- `v0` (`xi_mean` if univariate) correspond to the attribute `velocities`.
- `betas` correspond to the linear combinaison of columns from the orthonormal basis so to derive the `mixing_matrix`.

**values** [dict [str, *torch.Tensor*]] New values used to update the model's group average parameters

**Raises**

**LeaspyModelInputError** If *names_of_changed_values* contains unknown parameters.

## leaspy.models.utils.attributes.logistic_attributes.LogisticAttributes

**class LogisticAttributes**(*name*, *dimension*, *source_dimension*)
Bases: *leaspy.models.utils.attributes.abstract_manifold_model_attributes.AbstractManifoldModelAttributes*

Attributes of leaspy logistic models.

Contains the common attributes & methods to update the logistic model's attributes.

**Parameters**

**name** [str]

**dimension** [int]

**source_dimension** [int]

**See also:**

*UnivariateModel*

*MultivariateModel*

**Attributes**

**name** [str (default 'logistic')] Name of the associated leaspy model.

**dimension** [int]

**source_dimension** [int]

**univariate** [bool] Whether model is univariate or not (i.e. dimension == 1)

**has_sources** [bool] Whether model has sources or not (not univariate and source_dimension >= 1)

**update_possibilities** [tuple [str] (default ('all', 'g', 'v0', 'betas') )] Contains the available parameters to update. Different models have different parameters.

**positions** [torch.Tensor [dimension] (default None)] positions = exp(realizations['g']) such that "p0" = 1 / (1 + positions)

**velocities** [torch.Tensor [dimension] (default None)] Always positive: exp(realizations['v0'])

**orthonormal_basis** [torch.Tensor [dimension, dimension - 1] (default None)]

**betas** [torch.Tensor [dimension - 1, source_dimension] (default None)]

**mixing_matrix** [torch.Tensor [dimension, source_dimension] (default None)] Matrix A such that w_i = A * s_i.

## Methods

| | |
|---|---|
| *get_attributes*() | Returns the following attributes: `positions`, `velocities` & `mixing_matrix`. |
| *update*(names_of_changed_values, values) | Update model group average parameter(s). |

`get_attributes`()
Returns the following attributes: `positions`, `velocities` & `mixing_matrix`.

**Returns**

**For univariate models:** positions: *torch.Tensor*

**For not univariate models:**

- positions: *torch.Tensor*
- velocities: *torch.Tensor*
- mixing_matrix: *torch.Tensor*

`update`(*names_of_changed_values*, *values*)
Update model group average parameter(s).

**Parameters**

**names_of_changed_values** [list [str]]

**Elements of list must be either:**

- `all` (update everything)
- `g` correspond to the attribute `positions`.
- `v0` (`xi_mean` if univariate) correspond to the attribute `velocities`.
- `betas` correspond to the linear combinaison of columns from the orthonormal basis so to derive the `mixing_matrix`.

**values** [dict [str, *torch.Tensor*]] New values used to update the model's group average parameters

**Raises**

**LeaspyModelInputError** If *names_of_changed_values* contains unknown parameters.

## leaspy.models.utils.attributes.logistic_parallel_attributes.LogisticParallelAttributes

**class LogisticParallelAttributes**(*name*, *dimension*, *source_dimension*)

Bases: *leaspy.models.utils.attributes.abstract_manifold_model_attributes.AbstractManifoldModelAttributes*

Attributes of leaspy logistic parallel models.

Contains the common attributes & methods of the logistic parallel models' attributes.

**Parameters**

**name** [str]

**dimension** [int]

**source_dimension** [int]

**Raises**

**LeaspyModelInputError** if any inconsistent parameters for the model.

**See also:**

*MultivariateParallelModel*

**Attributes**

**name** [str (default 'logistic_parallel')] Name of the associated leaspy model.

**dimension** [int]

**source_dimension** [int]

**has_sources** [bool] Whether model has sources or not (source_dimension >= 1)

**update_possibilities** [tuple [str] (default ('all', 'g', 'xi_mean', 'deltas', 'betas') )] Contains the available parameters to update. Different models have different parameters.

**positions** [torch.Tensor (scalar) (default None)] positions = exp(realizations['g']) such that "p0" = 1 / (1 + positions * exp(-deltas))

**deltas** [torch.Tensor [dimension] (default None)] deltas = [0, delta_2_realization, ..., delta_n_realization]

**velocities** [torch.Tensor (scalar) (default None)] Always positive: exp(realizations['xi_mean'])

**orthonormal_basis** [torch.Tensor [dimension, dimension - 1] (default None)]

**betas** [torch.Tensor [dimension - 1, source_dimension] (default None)]

**mixing_matrix** [torch.Tensor [dimension, source_dimension] (default None)] Matrix A such that w_i = A * s_i.

### Methods

| | |
|---|---|
| *get_attributes*() | Returns the following attributes: positions, deltas & mixing_matrix. |
| *update*(names_of_changed_values, values) | Update model group average parameter(s). |

**get_attributes**()
Returns the following attributes: positions, deltas & mixing_matrix.

**Returns**

**positions:** *torch.Tensor*

**deltas:** *torch.Tensor*

**mixing_matrix:** *torch.Tensor*

**update**(*names_of_changed_values*, *values*)
Update model group average parameter(s).

**Parameters**

**names_of_changed_values** [list [str]]

**Elements of list must be either:**

- all (update everything)
- g correspond to the attribute positions.
- xi_mean correspond to the attribute velocities.
- deltas correspond to the attribute deltas.
- betas correspond to the linear combinaison of columns from the orthonormal basis so to derive the mixing_matrix.

**values** [dict [str, *torch.Tensor*]] New values used to update the model's group average parameters

**Raises**

**LeaspyModelInputError** If *names_of_changed_values* contains unknown parameters.

### 3.2.10 leaspy.models.utils.initialization: Initialization methods

Available methods to initialize model parameters before a fit.

| | |
|---|---|
| *model_initialization.initialize_parameters*(...) | Initialize the model's group parameters given its name & the scores of all subjects. |

#### leaspy.models.utils.initialization.model_initialization.initialize_parameters

**initialize_parameters**(*model*, *dataset*, *method='default'*)

Initialize the model's group parameters given its name & the scores of all subjects.

**Under-the-hood it calls an initialization function dedicated for the *model*:**

- `initialize_linear()` (including when *univariate*)
- `initialize_logistic()` (including when *univariate*)
- `initialize_logistic_parallel()`

It is automatically called during *Leaspy.fit()*.

**Parameters**

**model** [*AbstractModel*] The model to initialize.

**dataset** [*Dataset*] Contains the individual scores.

**method** [str]

**Must be one of:**

- `'default'`: initialize at mean.
- `'random'`: initialize with a gaussian realization with same mean and variance.

**Returns**

**parameters** [dict [str, `torch.Tensor`]] Contains the initialized model's group parameters.

**Raises**

**LeaspyInputError** If no initialization method is known for model type / method

## 3.3 leaspy.algo: Algorithms

Contains all algorithms used in the package.

| | |
|---|---|
| *abstract_algo.AbstractAlgo*(settings) | Abstract class containing common methods for all algorithm classes. |
| *algo_factory.AlgoFactory*() | Return the wanted algorithm given its name. |

### 3.3.1 leaspy.algo.abstract_algo.AbstractAlgo

**class AbstractAlgo**(*settings*)

Bases: abc.ABC

Abstract class containing common methods for all algorithm classes. These classes are child classes of *AbstractAlgo*.

**Parameters**

**name** [str]

**family** [str] cf. attributes

**parameters** [KwargsType] cf. attribute *algo_parameters*

**Attributes**

**name** [str] Name of the algorithm.

**family** [str]

**Family of the algorithm. For now, valid families are:**

- 'fit'``
- 'personalize'``
- 'simulate'

**deterministic** [bool] True, if and only if algorithm does not involve in randomness. Setting a seed and such algorithms will be useless.

**algo_parameters** [dict] Contains the algorithm's parameters. These ones are set by a *AlgorithmSettings* class object.

**seed** [int, optional] Seed used by numpy and torch.

**output_manager** [FitOutputManager] Optional output manager of the algorithm

#### Methods

| | |
|---|---|
| *load_parameters*(parameters) | Update the algorithm's parameters by the ones in the given dictionary. |
| *run*(model, *args[, return_noise]) | Main method, run the algorithm. |
| *run_impl*(model, *args, **extra_kwargs) | Run the algorithm (actual implementation), to be implemented in children classes. |
| *set_output_manager*(output_settings) | Set a FitOutputManager object for the run of the algorithm |

**load_parameters**(*parameters: dict*)

Update the algorithm's parameters by the ones in the given dictionary. The keys in the io which does not belong to the algorithm's parameters keys are ignored.

**Parameters**

**parameters** [dict] Contains the pairs (key, value) of the wanted parameters

### Examples

```
>>> settings = leaspy.io.settings.algorithm_settings.AlgorithmSettings("mcmc_
↪saem")
>>> my_algo = leaspy.algo.fit.tensor_mcmcsaem.TensorMCMCSAEM(settings)
>>> my_algo.algo_parameters
{'n_iter': 10000,
 'n_burn_in_iter': 9000,
 'eps': 0.001,
 'L': 10,
 'sampler_ind': 'Gibbs',
 'sampler_pop': 'Gibbs',
 'annealing': {'do_annealing': False,
  'initial_temperature': 10,
  'n_plateau': 10,
  'n_iter': 200}}
>>> parameters = {'n_iter': 5000, 'n_burn_in_iter': 4000}
>>> my_algo.load_parameters(parameters)
>>> my_algo.algo_parameters
{'n_iter': 5000,
 'n_burn_in_iter': 4000,
 'eps': 0.001,
 'L': 10,
 'sampler_ind': 'Gibbs',
 'sampler_pop': 'Gibbs',
 'annealing': {'do_annealing': False,
  'initial_temperature': 10,
  'n_plateau': 10,
  'n_iter': 200}}
```

**run**(*model: AbstractModel*, **args*, *return_noise: bool = False*, ***extra_kwargs*) → Any

Main method, run the algorithm.

TODO fix proper abstract class method: input depends on algorithm… (esp. simulate != from others…)

**Parameters**

**model** [*AbstractModel*] The used model.

**dataset** [*Dataset*] Contains all the subjects' observations with corresponding timepoints, in torch format to speed up computations.

**return_noise** [bool (default False), keyword only] Should the algorithm return main output and optional noise output as a 2-tuple?

**Returns**

**Depends on algorithm class: TODO change?**

**See also:**

*AbstractFitAlgo*

*AbstractPersonalizeAlgo*

*SimulationAlgorithm*

**abstract run_impl**(*model: AbstractModel*, **args*, ***extra_kwargs*) → Tuple[Any, Optional[torch.FloatTensor]]

Run the algorithm (actual implementation), to be implemented in children classes.

TODO fix proper abstract class

**Parameters**

**model** [*AbstractModel*] The used model.

**dataset** [*Dataset*] Contains all the subjects' observations with corresponding timepoints, in torch format to speed up computations.

**Returns**

**A 2-tuple containing:**

- the result to send back to user
- optional float tensor representing noise std-dev (to be printed)

**See also:**

*AbstractFitAlgo*

*AbstractPersonalizeAlgo*

*SimulationAlgorithm*

**set_output_manager**(*output_settings*)

Set a `FitOutputManager` object for the run of the algorithm

**Parameters**

**output_settings** [*OutputsSettings*] Contains the logs settings for the computation run (console print periodicity, plot periodicity ...)

### Examples

```
>>> from leaspy import AlgorithmSettings
>>> from leaspy.io.settings.outputs_settings import OutputsSettings
>>> from leaspy.algo.fit.tensor_mcmcsaem import TensorMCMCSAEM
>>> algo_settings = AlgorithmSettings("mcmc_saem")
>>> my_algo = TensorMCMCSAEM(algo_settings)
>>> settings = {'path': 'brouillons',
                'console_print_periodicity': 50,
                'plot_periodicity': 100,
                'save_periodicity': 50
                }
>>> my_algo.set_output_manager(OutputsSettings(settings))
```

### 3.3.2 leaspy.algo.algo_factory.AlgoFactory

**class AlgoFactory**

Bases: object

Return the wanted algorithm given its name.

#### Notes

For developers: add your new algorithm in corresponding category of `_algos` dictionary.

#### Methods

| | |
|---|---|
| *algo*(algorithm_family, settings) | Return the wanted algorithm given its name. |
| *get_class*(name) | Get the class of the algorithm identified as *name*. |

**classmethod algo**(*algorithm_family: str*, *settings*) → AbstractAlgo

Return the wanted algorithm given its name.

**Parameters**

**algorithm_family** [str] Task name, used to check if the algorithm within the input *settings* is compatible with this task. Must be one of the following api's name:

- *fit*
- *personalize*
- *simulate*

**settings** [*AlgorithmSettings*] The algorithm settings.

**Returns**

**algorithm** [child class of *AbstractAlgo*] The wanted algorithm if it exists and is compatible with algorithm family.

**Raises**

**LeaspyAlgoInputError**

- if the algorithm family is unknown
- if the algorithm name is unknown / does not belong to the wanted algorithm family

**classmethod get_class**(*name: str*) → Type[AbstractAlgo]

Get the class of the algorithm identified as *name*.

### 3.3.3 leaspy.algo.fit: Fit algorithms

Algorithms used to calibrate (fit) a model.

| | |
|---|---|
| *abstract_fit_algo.AbstractFitAlgo*(settings) | Abstract class containing common method for all *fit* algorithm classes. |
| *abstract_mcmc.AbstractFitMCMC*(settings) | Abstract class containing common method for all *fit* algorithm classes based on *Monte-Carlo Markov Chains* (MCMC). |
| *tensor_mcmcsaem.TensorMCMCSAEM*(settings) | Main algorithm for MCMC-SAEM. |

#### leaspy.algo.fit.abstract_fit_algo.AbstractFitAlgo

**class AbstractFitAlgo**(*settings*)

Bases: *leaspy.algo.abstract_algo.AbstractAlgo*

Abstract class containing common method for all *fit* algorithm classes.

**See also:**

*Leaspy.fit()*

**Attributes**

**current_iteration** [int, default 0] The number of the current iteration

**Inherited attributes** From *AbstractAlgo*

##### Methods

| | |
|---|---|
| *iteration*(dataset, model, realizations) | Update the parameters (abstract method). |
| *load_parameters*(parameters) | Update the algorithm's parameters by the ones in the given dictionary. |
| *run*(model, *args[, return_noise]) | Main method, run the algorithm. |
| *run_impl*(model, dataset) | Main method, run the algorithm. |
| *set_output_manager*(output_settings) | Set a `FitOutputManager` object for the run of the algorithm |

**abstract iteration**(*dataset*, *model*, *realizations*)

Update the parameters (abstract method).

**Parameters**

**dataset** [*Dataset*] Contains the subjects' obersvations in torch format to speed up computation.

**model** [*AbstractModel*] The used model.

**realizations** [*CollectionRealization*] The parameters.

**load_parameters**(*parameters: dict*)

Update the algorithm's parameters by the ones in the given dictionary. The keys in the io which does not belong to the algorithm's parameters keys are ignored.

**Parameters**

**parameters** [dict] Contains the pairs (key, value) of the wanted parameters

### Examples

```
>>> settings = leaspy.io.settings.algorithm_settings.AlgorithmSettings("mcmc_
↪saem")
>>> my_algo = leaspy.algo.fit.tensor_mcmcsaem.TensorMCMCSAEM(settings)
>>> my_algo.algo_parameters
{'n_iter': 10000,
 'n_burn_in_iter': 9000,
 'eps': 0.001,
 'L': 10,
 'sampler_ind': 'Gibbs',
 'sampler_pop': 'Gibbs',
 'annealing': {'do_annealing': False,
  'initial_temperature': 10,
  'n_plateau': 10,
  'n_iter': 200}}
>>> parameters = {'n_iter': 5000, 'n_burn_in_iter': 4000}
>>> my_algo.load_parameters(parameters)
>>> my_algo.algo_parameters
{'n_iter': 5000,
 'n_burn_in_iter': 4000,
 'eps': 0.001,
 'L': 10,
 'sampler_ind': 'Gibbs',
 'sampler_pop': 'Gibbs',
 'annealing': {'do_annealing': False,
  'initial_temperature': 10,
  'n_plateau': 10,
  'n_iter': 200}}
```

**run**(*model: AbstractModel*, **args*, *return_noise: bool = False*, ***extra_kwargs*) → Any

Main method, run the algorithm.

TODO fix proper abstract class method: input depends on algorithm… (esp. simulate != from others…)

**Parameters**

**model** [*AbstractModel*] The used model.

**dataset** [*Dataset*] Contains all the subjects' observations with corresponding timepoints, in torch format to speed up computations.

**return_noise** [bool (default False), keyword only] Should the algorithm return main output and optional noise output as a 2-tuple?

**Returns**

**Depends on algorithm class: TODO change?**

**See also:**

*AbstractFitAlgo*

*AbstractPersonalizeAlgo*

*SimulationAlgorithm*

**run_impl**(*model*, *dataset*)
Main method, run the algorithm.

Basically, it initializes the *CollectionRealization* object, updates it using the *iteration* method then returns it.

TODO fix proper abstract class

**Parameters**

**model** [*AbstractModel*] The used model.

**dataset** [*Dataset*] Contains the subjects' observations in torch format to speed up computation.

**Returns**

**2-tuple:**

- **realizations** [*CollectionRealization*] The optimized parameters.
- None : placeholder for noise-std

**set_output_manager**(*output_settings*)
Set a `FitOutputManager` object for the run of the algorithm

**Parameters**

**output_settings** [*OutputsSettings*] Contains the logs settings for the computation run (console print periodicity, plot periodicity ...)

### Examples

```
>>> from leaspy import AlgorithmSettings
>>> from leaspy.io.settings.outputs_settings import OutputsSettings
>>> from leaspy.algo.fit.tensor_mcmcsaem import TensorMCMCSAEM
>>> algo_settings = AlgorithmSettings("mcmc_saem")
>>> my_algo = TensorMCMCSAEM(algo_settings)
>>> settings = {'path': 'brouillons',
                'console_print_periodicity': 50,
                'plot_periodicity': 100,
                'save_periodicity': 50
                }
>>> my_algo.set_output_manager(OutputsSettings(settings))
```

## leaspy.algo.fit.abstract_mcmc.AbstractFitMCMC

**class AbstractFitMCMC**(*settings*)
Bases: `leaspy.algo.utils.samplers.algo_with_samplers.AlgoWithSamplersMixin`, *leaspy.algo.fit.abstract_fit_algo.AbstractFitAlgo*

Abstract class containing common method for all *fit* algorithm classes based on *Monte-Carlo Markov Chains* (MCMC).

**Parameters**

**settings** [*AlgorithmSettings*] MCMC fit algorithm settings

**See also:**

**leaspy.algo.utils.samplers**

**Attributes**

**samplers** [dict[ str, *AbstractSampler* ]] Dictionary of samplers per each variable

**TODO add missing**

### Methods

| | |
|---|---|
| *iteration*(data, model, realizations) | MCMC-SAEM iteration. |
| *load_parameters*(parameters) | Update the algorithm's parameters by the ones in the given dictionary. |
| *run*(model, *args[, return_noise]) | Main method, run the algorithm. |
| *run_impl*(model, dataset) | Main method, run the algorithm. |
| *set_output_manager*(output_settings) | Set a `FitOutputManager` object for the run of the algorithm |

**iteration**(*data*, *model*, *realizations*)

MCMC-SAEM iteration.

1. Sample : MC sample successively of the population and individual variales
2. Maximization step : update model parameters from current population/individual variables values.

**Parameters**

**data** [*Dataset*]

**model** [*AbstractModel*]

**realizations** [*CollectionRealization*]

**load_parameters**(*parameters: dict*)

Update the algorithm's parameters by the ones in the given dictionary. The keys in the io which does not belong to the algorithm's parameters keys are ignored.

**Parameters**

**parameters** [dict] Contains the pairs (key, value) of the wanted parameters

### Examples

```
>>> settings = leaspy.io.settings.algorithm_settings.AlgorithmSettings("mcmc_
↪saem")
>>> my_algo = leaspy.algo.fit.tensor_mcmcsaem.TensorMCMCSAEM(settings)
>>> my_algo.algo_parameters
{'n_iter': 10000,
 'n_burn_in_iter': 9000,
 'eps': 0.001,
 'L': 10,
 'sampler_ind': 'Gibbs',
 'sampler_pop': 'Gibbs',
 'annealing': {'do_annealing': False,
  'initial_temperature': 10,
```

(continues on next page)

(continued from previous page)

```
  'n_plateau': 10,
  'n_iter': 200}}
>>> parameters = {'n_iter': 5000, 'n_burn_in_iter': 4000}
>>> my_algo.load_parameters(parameters)
>>> my_algo.algo_parameters
{'n_iter': 5000,
 'n_burn_in_iter': 4000,
 'eps': 0.001,
 'L': 10,
 'sampler_ind': 'Gibbs',
 'sampler_pop': 'Gibbs',
 'annealing': {'do_annealing': False,
  'initial_temperature': 10,
  'n_plateau': 10,
  'n_iter': 200}}
```

**run**(*model: AbstractModel*, **args*, *return_noise: bool = False*, ***extra_kwargs*) → Any

Main method, run the algorithm.

TODO fix proper abstract class method: input depends on algorithm… (esp. simulate != from others…)

**Parameters**

**model** [*AbstractModel*] The used model.

**dataset** [*Dataset*] Contains all the subjects' observations with corresponding timepoints, in torch format to speed up computations.

**return_noise** [bool (default False), keyword only] Should the algorithm return main output and optional noise output as a 2-tuple?

**Returns**

**Depends on algorithm class: TODO change?**

**See also:**

*AbstractFitAlgo*

*AbstractPersonalizeAlgo*

*SimulationAlgorithm*

**run_impl**(*model*, *dataset*)

Main method, run the algorithm.

Basically, it initializes the *CollectionRealization* object, updates it using the *iteration* method then returns it.

TODO fix proper abstract class

**Parameters**

**model** [*AbstractModel*] The used model.

**dataset** [*Dataset*] Contains the subjects' observations in torch format to speed up computation.

**Returns**

**2-tuple:**

- **realizations** [*CollectionRealization*] The optimized parameters.
- None : placeholder for noise-std

**set_output_manager**(*output_settings*)
Set a `FitOutputManager` object for the run of the algorithm

**Parameters**

**output_settings** [*OutputsSettings*] Contains the logs settings for the computation run (console print periodicity, plot periodicity ...)

### Examples

```
>>> from leaspy import AlgorithmSettings
>>> from leaspy.io.settings.outputs_settings import OutputsSettings
>>> from leaspy.algo.fit.tensor_mcmcsaem import TensorMCMCSAEM
>>> algo_settings = AlgorithmSettings("mcmc_saem")
>>> my_algo = TensorMCMCSAEM(algo_settings)
>>> settings = {'path': 'brouillons',
                'console_print_periodicity': 50,
                'plot_periodicity': 100,
                'save_periodicity': 50
                }
>>> my_algo.set_output_manager(OutputsSettings(settings))
```

## leaspy.algo.fit.tensor_mcmcsaem.TensorMCMCSAEM

**class TensorMCMCSAEM**(*settings*)
Bases: *leaspy.algo.fit.abstract_mcmc.AbstractFitMCMC*

Main algorithm for MCMC-SAEM.

**Parameters**

**settings** [*AlgorithmSettings*] MCMC fit algorithm settings

**See also:**

*AbstractFitMCMC*

### Methods

| | |
|---|---|
| *iteration*(data, model, realizations) | MCMC-SAEM iteration. |
| *load_parameters*(parameters) | Update the algorithm's parameters by the ones in the given dictionary. |
| *run*(model, *args[, return_noise]) | Main method, run the algorithm. |
| *run_impl*(model, dataset) | Main method, run the algorithm. |
| *set_output_manager*(output_settings) | Set a `FitOutputManager` object for the run of the algorithm |

**iteration**(*data*, *model*, *realizations*)
MCMC-SAEM iteration.

1. Sample : MC sample successively of the population and individual variales
2. Maximization step : update model parameters from current population/individual variables values.

**Parameters**

**data** [*Dataset*]

**model** [*AbstractModel*]

**realizations** [*CollectionRealization*]

**load_parameters**(*parameters: dict*)

Update the algorithm's parameters by the ones in the given dictionary. The keys in the io which does not belong to the algorithm's parameters keys are ignored.

**Parameters**

**parameters** [dict] Contains the pairs (key, value) of the wanted parameters

### Examples

```
>>> settings = leaspy.io.settings.algorithm_settings.AlgorithmSettings("mcmc_
↪saem")
>>> my_algo = leaspy.algo.fit.tensor_mcmcsaem.TensorMCMCSAEM(settings)
>>> my_algo.algo_parameters
{'n_iter': 10000,
 'n_burn_in_iter': 9000,
 'eps': 0.001,
 'L': 10,
 'sampler_ind': 'Gibbs',
 'sampler_pop': 'Gibbs',
 'annealing': {'do_annealing': False,
  'initial_temperature': 10,
  'n_plateau': 10,
  'n_iter': 200}}
>>> parameters = {'n_iter': 5000, 'n_burn_in_iter': 4000}
>>> my_algo.load_parameters(parameters)
>>> my_algo.algo_parameters
{'n_iter': 5000,
 'n_burn_in_iter': 4000,
 'eps': 0.001,
 'L': 10,
 'sampler_ind': 'Gibbs',
 'sampler_pop': 'Gibbs',
 'annealing': {'do_annealing': False,
  'initial_temperature': 10,
  'n_plateau': 10,
  'n_iter': 200}}
```

**run**(*model: AbstractModel*, **args*, *return_noise: bool = False*, ***extra_kwargs*) → Any

Main method, run the algorithm.

TODO fix proper abstract class method: input depends on algorithm… (esp. simulate != from others…)

**Parameters**

**model** [*AbstractModel*] The used model.

**dataset** [*Dataset*] Contains all the subjects' observations with corresponding timepoints, in torch format to speed up computations.

**return_noise** [bool (default False), keyword only] Should the algorithm return main output and optional noise output as a 2-tuple?

**Returns**

**Depends on algorithm class: TODO change?**

**See also:**

*AbstractFitAlgo*

*AbstractPersonalizeAlgo*

*SimulationAlgorithm*

**run_impl**(*model*, *dataset*)

Main method, run the algorithm.

Basically, it initializes the *CollectionRealization* object, updates it using the *iteration* method then returns it.

TODO fix proper abstract class

**Parameters**

**model** [*AbstractModel*] The used model.

**dataset** [*Dataset*] Contains the subjects' observations in torch format to speed up computation.

**Returns**

**2-tuple:**

- **realizations** [*CollectionRealization*] The optimized parameters.
- None : placeholder for noise-std

**set_output_manager**(*output_settings*)

Set a `FitOutputManager` object for the run of the algorithm

**Parameters**

**output_settings** [*OutputsSettings*] Contains the logs settings for the computation run (console print periodicity, plot periodicity ...)

### Examples

```
>>> from leaspy import AlgorithmSettings
>>> from leaspy.io.settings.outputs_settings import OutputsSettings
>>> from leaspy.algo.fit.tensor_mcmcsaem import TensorMCMCSAEM
>>> algo_settings = AlgorithmSettings("mcmc_saem")
>>> my_algo = TensorMCMCSAEM(algo_settings)
>>> settings = {'path': 'brouillons',
                'console_print_periodicity': 50,
                'plot_periodicity': 100,
                'save_periodicity': 50
```

(continues on next page)

(continued from previous page)

```
                }
>>> my_algo.set_output_manager(OutputsSettings(settings))
```

### 3.3.4 leaspy.algo.personalize: Personalization algorithms

Algorithms used to personalize a model to given subjects.

| | |
|---|---|
| *abstract_personalize_algo.AbstractPersonalizeAlgo*(...) | Abstract class for *personalize* algorithm. |
| *scipy_minimize.ScipyMinimize*(settings) | Gradient descent based algorithm to compute individual parameters, *i.e.* personalize a model to a given set of subjects. |

#### leaspy.algo.personalize.abstract_personalize_algo.AbstractPersonalizeAlgo

**class AbstractPersonalizeAlgo**(*settings*)

Bases: *leaspy.algo.abstract_algo.AbstractAlgo*

Abstract class for *personalize* algorithm. Estimation of individual parameters of a given *Data* file with a frozen model (already estimated, or loaded from known parameters).

**Parameters**

**settings** [*AlgorithmSettings*] Settings of the algorithm.

**See also:**

*Leaspy.personalize()*

**Attributes**

**name** [str] Algorithm's name.

**seed** [int, optional] Algorithm's seed (default None).

**algo_parameters** [dict] Algorithm's parameters.

#### Methods

| | |
|---|---|
| *load_parameters*(parameters) | Update the algorithm's parameters by the ones in the given dictionary. |
| *run*(model, *args[, return_noise]) | Main method, run the algorithm. |
| *run_impl*(model, dataset) | Main personalize function, wraps the abstract `_get_individual_parameters()` method. |
| *set_output_manager*(output_settings) | Set a `FitOutputManager` object for the run of the algorithm |

**load_parameters**(*parameters: dict*)

Update the algorithm's parameters by the ones in the given dictionary. The keys in the io which does not belong to the algorithm's parameters keys are ignored.

**Parameters**

**parameters** [dict] Contains the pairs (key, value) of the wanted parameters

### Examples

```
>>> settings = leaspy.io.settings.algorithm_settings.AlgorithmSettings("mcmc_
↪saem")
>>> my_algo = leaspy.algo.fit.tensor_mcmcsaem.TensorMCMCSAEM(settings)
>>> my_algo.algo_parameters
{'n_iter': 10000,
 'n_burn_in_iter': 9000,
 'eps': 0.001,
 'L': 10,
 'sampler_ind': 'Gibbs',
 'sampler_pop': 'Gibbs',
 'annealing': {'do_annealing': False,
  'initial_temperature': 10,
  'n_plateau': 10,
  'n_iter': 200}}
>>> parameters = {'n_iter': 5000, 'n_burn_in_iter': 4000}
>>> my_algo.load_parameters(parameters)
>>> my_algo.algo_parameters
{'n_iter': 5000,
 'n_burn_in_iter': 4000,
 'eps': 0.001,
 'L': 10,
 'sampler_ind': 'Gibbs',
 'sampler_pop': 'Gibbs',
 'annealing': {'do_annealing': False,
  'initial_temperature': 10,
  'n_plateau': 10,
  'n_iter': 200}}
```

**run**(*model: AbstractModel*, **args*, *return_noise: bool = False*, ***extra_kwargs*) → Any

Main method, run the algorithm.

TODO fix proper abstract class method: input depends on algorithm… (esp. simulate != from others…)

**Parameters**

**model** [*AbstractModel*] The used model.

**dataset** [*Dataset*] Contains all the subjects' observations with corresponding timepoints, in torch format to speed up computations.

**return_noise** [bool (default False), keyword only] Should the algorithm return main output and optional noise output as a 2-tuple?

**Returns**

**Depends on algorithm class: TODO change?**

**See also:**

*AbstractFitAlgo*

*AbstractPersonalizeAlgo*

*SimulationAlgorithm*

**run_impl**(*model*, *dataset*)

Main personalize function, wraps the abstract `_get_individual_parameters()` method.

**Parameters**

**model** [*AbstractModel*] A subclass object of leaspy *AbstractModel*.

**dataset** [*Dataset*] Dataset object build with leaspy class objects Data, algo & model

**Returns**

**individual_parameters** [*IndividualParameters*] Contains individual parameters.

**noise_std** [float or `torch.FloatTensor`] The estimated noise (is a tensor if *model.noise_model* is `'gaussian_diagonal'`)

$$= \frac{1}{n_{visits} \times n_{dim}} \sqrt{\sum_{i,j \in [1,n_{visits}] \times [1,n_{dim}]} \varepsilon_{i,j}}$$

where $\varepsilon_{i,j} = (f(\theta, (z_{i,j}), (t_{i,j})) - (y_{i,j}))^2$ , where $\theta$ are the model's fixed effect, $(z_{i,j})$ the model's random effects, $(t_{i,j})$ the time-points and $f$ the model's estimator.

**set_output_manager**(*output_settings*)

Set a `FitOutputManager` object for the run of the algorithm

**Parameters**

**output_settings** [*OutputsSettings*] Contains the logs settings for the computation run (console print periodicity, plot periodicity ...)

### Examples

```
>>> from leaspy import AlgorithmSettings
>>> from leaspy.io.settings.outputs_settings import OutputsSettings
>>> from leaspy.algo.fit.tensor_mcmcsaem import TensorMCMCSAEM
>>> algo_settings = AlgorithmSettings("mcmc_saem")
>>> my_algo = TensorMCMCSAEM(algo_settings)
>>> settings = {'path': 'brouillons',
                'console_print_periodicity': 50,
                'plot_periodicity': 100,
                'save_periodicity': 50
                }
>>> my_algo.set_output_manager(OutputsSettings(settings))
```

## leaspy.algo.personalize.scipy_minimize.ScipyMinimize

**class ScipyMinimize**(*settings*)

Bases: *leaspy.algo.personalize.abstract_personalize_algo.AbstractPersonalizeAlgo*

Gradient descent based algorithm to compute individual parameters, *i.e.* personalize a model to a given set of subjects.

**Parameters**

**settings** [*AlgorithmSettings*] Settings of the algorithm.

**Attributes**

**print_convergence_issues** [bool] Should we display all convergence issues returned by *scipy.optimize*? By default display convergences issues iff not BFGS method Note that it is not used if custom *logger* is defined in settings.

**minimize_kwargs** [kwargs] Keyword arguments passed to `scipy.optimize.minimize()`

### Methods

| | |
|---|---|
| *is_jacobian_implemented*(model) | Check that the jacobian of model is implemented. |
| *load_parameters*(parameters) | Update the algorithm's parameters by the ones in the given dictionary. |
| *obj*(x, *args) | Objective loss function to minimize in order to get patient's individual parameters |
| *run*(model, *args[, return_noise]) | Main method, run the algorithm. |
| *run_impl*(model, dataset) | Main personalize function, wraps the abstract `_get_individual_parameters()` method. |
| *set_output_manager*(output_settings) | Set a `FitOutputManager` object for the run of the algorithm |

**is_jacobian_implemented**(*model*) → bool
Check that the jacobian of model is implemented.

**load_parameters**(*parameters: dict*)
Update the algorithm's parameters by the ones in the given dictionary. The keys in the io which does not belong to the algorithm's parameters keys are ignored.

**Parameters**

**parameters** [dict] Contains the pairs (key, value) of the wanted parameters

### Examples

```
>>> settings = leaspy.io.settings.algorithm_settings.AlgorithmSettings("mcmc_
↪saem")
>>> my_algo = leaspy.algo.fit.tensor_mcmcsaem.TensorMCMCSAEM(settings)
>>> my_algo.algo_parameters
{'n_iter': 10000,
 'n_burn_in_iter': 9000,
 'eps': 0.001,
 'L': 10,
 'sampler_ind': 'Gibbs',
 'sampler_pop': 'Gibbs',
 'annealing': {'do_annealing': False,
  'initial_temperature': 10,
  'n_plateau': 10,
  'n_iter': 200}}
>>> parameters = {'n_iter': 5000, 'n_burn_in_iter': 4000}
>>> my_algo.load_parameters(parameters)
>>> my_algo.algo_parameters
{'n_iter': 5000,
 'n_burn_in_iter': 4000,
 'eps': 0.001,
```

(continues on next page)

(continued from previous page)

```
'L': 10,
'sampler_ind': 'Gibbs',
'sampler_pop': 'Gibbs',
'annealing': {'do_annealing': False,
 'initial_temperature': 10,
 'n_plateau': 10,
 'n_iter': 200}}
```

**obj**(*x*, **args*)

Objective loss function to minimize in order to get patient's individual parameters

**Parameters**

**x** [array-like [float]] Individual **standardized** parameters At initialization `x = [xi_mean/xi_std, tau_mean/tau_std] (+ [0.] * n_sources if multivariate model)`

***args**

- **model** [*AbstractModel*] Model used to compute the group average parameters.
- **timepoints** [`torch.Tensor` [1,n_tpts]] Contains the individual ages corresponding to the given `values`
- **values** [`torch.Tensor` [n_tpts, n_fts]] Contains the individual true scores corresponding to the given `times`.
- **with_gradient** [bool]
  - If True: return (objective, gradient_objective)
  - Else: simply return objective

**Returns**

**objective** [float] Value of the loss function (opposite of log-likelihood).

**if *with_gradient* is True:**

**2-tuple (as expected by `scipy.optimize.minimize()` when `jac=True`)**

- objective : float
- gradient : array-like[float] of length n_dims_params

**Raises**

**`LeaspyAlgoInputError`** if noise model is not currently supported by algorithm. TODO: everything that is not generic here concerning noise structure should be handle by model/NoiseModel directly!!!!

**run**(*model: AbstractModel*, **args*, *return_noise: bool = False*, ***extra_kwargs*) → Any

Main method, run the algorithm.

TODO fix proper abstract class method: input depends on algorithm… (esp. simulate != from others…)

**Parameters**

**model** [*AbstractModel*] The used model.

**dataset** [*Dataset*] Contains all the subjects' observations with corresponding timepoints, in torch format to speed up computations.

**return_noise** [bool (default False), keyword only] Should the algorithm return main output and optional noise output as a 2-tuple?

**Returns**

**Depends on algorithm class: TODO change?**

**See also:**

*AbstractFitAlgo*

*AbstractPersonalizeAlgo*

*SimulationAlgorithm*

**run_impl**(*model*, *dataset*)

Main personalize function, wraps the abstract `_get_individual_parameters()` method.

**Parameters**

**model** [*AbstractModel*] A subclass object of leaspy *AbstractModel*.

**dataset** [*Dataset*] Dataset object build with leaspy class objects Data, algo & model

**Returns**

**individual_parameters** [*IndividualParameters*] Contains individual parameters.

**noise_std** [float or `torch.FloatTensor`] The estimated noise (is a tensor if *model.noise_model* is `'gaussian_diagonal'`)

$$= \frac{1}{n_{visits} \times n_{dim}} \sqrt{\sum_{i,j \in [1, n_{visits}] \times [1, n_{dim}]} \varepsilon_{i,j}}$$

where $\varepsilon_{i,j} = \left(f(\theta, (z_{i,j}), (t_{i,j})) - (y_{i,j})\right)^2$ , where $\theta$ are the model's fixed effect, $(z_{i,j})$ the model's random effects, $(t_{i,j})$ the time-points and $f$ the model's estimator.

**set_output_manager**(*output_settings*)

Set a `FitOutputManager` object for the run of the algorithm

**Parameters**

**output_settings** [*OutputsSettings*] Contains the logs settings for the computation run (console print periodicity, plot periodicity ...)

### Examples

```
>>> from leaspy import AlgorithmSettings
>>> from leaspy.io.settings.outputs_settings import OutputsSettings
>>> from leaspy.algo.fit.tensor_mcmcsaem import TensorMCMCSAEM
>>> algo_settings = AlgorithmSettings("mcmc_saem")
>>> my_algo = TensorMCMCSAEM(algo_settings)
>>> settings = {'path': 'brouillons',
                'console_print_periodicity': 50,
                'plot_periodicity': 100,
                'save_periodicity': 50
                }
>>> my_algo.set_output_manager(OutputsSettings(settings))
```

### 3.3.5 leaspy.algo.simulate: Simulation algorithms

Algorithm to simulate synthetic observations and individual parameters.

| | |
|---|---|
| *simulate.SimulationAlgorithm*(settings) | To simulate new data given existing one by learning the individual parameters joined distribution. |

#### leaspy.algo.simulate.simulate.SimulationAlgorithm

**class SimulationAlgorithm**(*settings*)

Bases: *leaspy.algo.abstract_algo.AbstractAlgo*

To simulate new data given existing one by learning the individual parameters joined distribution.

You can choose to only learn the distribution of a group of patient. To do so, choose the cofactor(s) and the cofactor(s) state of the wanted patient in the settings. For instance, for an Alzheimer's disease patient, you can load a genetic cofactor informative of the APOE4 carriers. Choose cofactor ['genetic'] and cofactor_state ['APOE4'] to simulate only APOE4 carriers.

**Parameters**

**settings** [*AlgorithmSettings*] The algorithm settings. They may include the following parameters, described in _Attributes_ section:

- *noise*
- *bandwidth_method*
- *cofactor*
- *cofactor_state*
- *number_of_subjects*
- *mean_number_of_visits*, *std_number_of_visits*, *min_number_of_visits*, *max_number_of_visits*
- *delay_btw_visits*
- *reparametrized_age_bounds*
- *sources_method*
- *prefix*
- *features_bounds*
- *features_bounds_nb_subjects_factor*

**Raises**

**LeaspyAlgoInputError** If algorithm parameters are of bad type or do not comply to detailed requirements.

### Notes

The baseline ages are no more jointly learnt with individual parameters. Instead, we jointly learn the _reparametrized_ baseline ages, together with individual parameters. The baseline ages are then reconstructed from the simulated reparametrized baseline ages and individual parameters.

By definition, the relation between age and reparametrized age is:

$$\psi_i(t) = e^{\xi_i}(t - \tau_i) + \bar{\tau}$$

with $t$ the real age, $\psi_i(t)$ the reparametrized age, $\xi_i$ the individual log-acceleration parameter, $\tau_i$ the individual time-shift parameter and $\bar{\tau}$ the mean conversion age derivated by the *model* object.

One can restrict the interval of the baseline reparametrized age to be _learnt_ in kernel, by setting bounds in *reparametrized_age_bounds*. Note that the simulated reparametrized baseline ages are unconstrained and thus could, theoretically (but very unlikely), be out of these prescribed bounds.

**Attributes**

**name** `['simulation']` Algorithm's name.

**seed** [int] Used by `numpy.random` & `torch.random` for reproducibility.

**algo_parameters** [dict] Contains the algorithm's parameters.

**bandwidth_method** [float or str or callable, optional] Bandwidth argument used in `scipy.stats.gaussian_kde` in order to learn the patients' distribution.

**cofactor** [list[str], optional (default = None)] The list of cofactors included used to select the wanted group of patients (ex - ['genetic']). All of them must correspond to an existing cofactor in the attribute *Data* of the input *result* of the *run()* method. TODO? should we allow to learn joint distribution of individual parameters and numeric/categorical cofactors (not fixed)?

**cofactor_state** [list[str], optional (default None)] The cofactors states used to select the wanted group of patients (ex - ['APOE4']). There is exactly one state per cofactor in *cofactor* (same order). It must correspond to an existing cofactor state in the attribute *Data* of the input *result* of the *run()* method. TODO? it could be replaced by methods to easily sub-select individual having certain cofactors PRIOR to running this algorithm + the functionality described just above (included varying cofactors as part of the distribution to estimate).

**features_bounds** [bool or dict[str, (float, float)] (default False)] Specify if the scores of the generated subjects must be bounded. This parameter can express in two way:

- *bool* : the bounds are the maximum and minimum scores observed in the baseline data (TODO: "baseline" instead?).
- *dict* : the user has to set the min and max bounds for every features. For example: `{'feature1': (score_min, score_max), 'feature2': (score_min, score_max), ...}`

**features_bounds_nb_subjects_factor** [float > 1 (default 10)] Only used if *features_bounds* is not False. The ratio of simulated subjects (> 1) so that there is at least *number_of_subjects* that comply to features bounds constraint.

**mean_number_of_visits** [int or float (default 6)] Average number of visits of the simulated patients. Examples - choose 5 => in average, a simulated patient will have 5 visits.

**std_number_of_visits** [int or float > 0, or None (default 3)] Standard deviation used into the generation of the number of visits per simulated patient. If <= 0 or None: number of visits will be deterministic

**min_number_of_visits, max_number_of_visits** [int (optional for max)] Minimum (resp. maximum) number of visits. Only used when *std_number_of_visits* > 0. *min_number_of_visits* should be >= 1 (default), *max_number_of_visits* can be None (no limit, default).

**delay_btw_visits :**

**Control by how many years consecutive visits of a patient are delayed. Multiple options are possible:**

- float > 0 : regular spacing between all visits
- dictionary : {'min': float > 0, 'mean': float >= min, 'std': float > 0 [, 'max': float >= mean]}

Specify a Gaussian random spacing (truncated between min, and max if given) * function : n (int >= 1) => 1D numpy.ndarray[float > 0] of length *n* giving delay between visits (e.g.: 3 => [0.5, 1.5, 1.])

**noise** [str or float or array-like[float], optional]

**Wanted level of gaussian noise in the generated scores:**

- Set noise to `None` will lead to patients follow the model exactly (no noise added).
- Set to `'inherit_struct'` (or deprecated `'default'`), the noise added will follow the model noise structure and for Gaussian noise it will be computed from reconstruction errors on data & individual parameters provided.
- Set noise to `'model'`, the noise added will follow the model noise structure as well as its values.
- Set to `'bernoulli'`, to simulate Bernoulli realizations.
- Set a float will add for each feature's scores a noise of standard deviation the given float ('gaussian_scalar' noise).
- Set an array-like[float] (1D of length *n_features*) will add for the feature *j* a noise of standard deviation `noise[j]` ('gaussian_diagonal' noise).

**number_of_subjects** [int > 0] Number of subject to simulate.

**reparametrized_age_bounds** [tuple[float, float], optional (default None)] Define the minimum and maximum reparametrized ages of subjects included in the kernel estimation. See Notes section. Example: reparametrized_age_bounds = (65, 70)

**sources_method** [str in {'full_kde', 'normal_sources'}]

- `'full_kde'` : the sources are also learned with the gaussian kernel density estimation.
- `'normal_sources'` : the sources are generated as multivariate normal distribution linked with the other individual parameters.

**prefix** [str] Prefix appended to simulated patients' identifiers

### Methods

| | |
|---|---|
| *load_parameters*(parameters) | Update the algorithm's parameters by the ones in the given dictionary. |
| *run*(model, *args[, return_noise]) | Main method, run the algorithm. |
| *run_impl*(model, individual_parameters, data) | Run simulation - learn joined distribution of patients' individual parameters and return a results object containing the simulated individual parameters and the simulated scores. |
| *set_output_manager*(output_settings) | Set a `FitOutputManager` object for the run of the algorithm |

**load_parameters**(*parameters: dict*)

Update the algorithm's parameters by the ones in the given dictionary. The keys in the io which does not belong to the algorithm's parameters keys are ignored.

**Parameters**

**parameters** [dict] Contains the pairs (key, value) of the wanted parameters

### Examples

```
>>> settings = leaspy.io.settings.algorithm_settings.AlgorithmSettings("mcmc_
↪saem")
>>> my_algo = leaspy.algo.fit.tensor_mcmcsaem.TensorMCMCSAEM(settings)
>>> my_algo.algo_parameters
{'n_iter': 10000,
 'n_burn_in_iter': 9000,
 'eps': 0.001,
 'L': 10,
 'sampler_ind': 'Gibbs',
 'sampler_pop': 'Gibbs',
 'annealing': {'do_annealing': False,
  'initial_temperature': 10,
  'n_plateau': 10,
  'n_iter': 200}}
>>> parameters = {'n_iter': 5000, 'n_burn_in_iter': 4000}
>>> my_algo.load_parameters(parameters)
>>> my_algo.algo_parameters
{'n_iter': 5000,
 'n_burn_in_iter': 4000,
 'eps': 0.001,
 'L': 10,
 'sampler_ind': 'Gibbs',
 'sampler_pop': 'Gibbs',
 'annealing': {'do_annealing': False,
  'initial_temperature': 10,
  'n_plateau': 10,
  'n_iter': 200}}
```

**run**(*model: AbstractModel*, **args*, *return_noise: bool = False*, ***extra_kwargs*) → Any

Main method, run the algorithm.

TODO fix proper abstract class method: input depends on algorithm... (esp. simulate != from others...)

**Parameters**

**model** [*AbstractModel*] The used model.

**dataset** [*Dataset*] Contains all the subjects' observations with corresponding time-points, in torch format to speed up computations.

**return_noise** [bool (default False), keyword only] Should the algorithm return main output and optional noise output as a 2-tuple?

**Returns**

**Depends on algorithm class: TODO change?**

**See also:**

*AbstractFitAlgo*

*AbstractPersonalizeAlgo*

*SimulationAlgorithm*

**run_impl**(*model: AbstractModel*, *individual_parameters: IndividualParameters*, *data: Data*) → Tuple[Result, Optional[torch.FloatTensor]]

Run simulation - learn joined distribution of patients' individual parameters and return a results object containing the simulated individual parameters and the simulated scores.

<!> The *AbstractAlgo.run* signature is not respected for simulation algorithm... TODO: respect it... at least use (model, dataset, individual_parameters) signature...

**Parameters**

**model** [*AbstractModel*] Subclass object of *AbstractModel*. Model used to compute the population & individual parameters. It contains the population parameters.

**individual_parameters** [*IndividualParameters*] Object containing the computed individual parameters.

**data** [*Data*] The data object.

**Returns**

**Result** Contains the simulated individual parameters & individual scores.

### Notes

In simulation_settings, one can specify in the parameters the cofactor & cofactor_state. By doing so, one can simulate based only on the subject for the given cofactor & cofactor's state.

By default, all the subjects provided are used to estimate the joined distribution.

**set_output_manager**(*output_settings*)

Set a `FitOutputManager` object for the run of the algorithm

**Parameters**

**output_settings** [*OutputsSettings*] Contains the logs settings for the computation run (console print periodicity, plot periodicity ...)

#### Examples

```
>>> from leaspy import AlgorithmSettings
>>> from leaspy.io.settings.outputs_settings import OutputsSettings
>>> from leaspy.algo.fit.tensor_mcmcsaem import TensorMCMCSAEM
>>> algo_settings = AlgorithmSettings("mcmc_saem")
>>> my_algo = TensorMCMCSAEM(algo_settings)
>>> settings = {'path': 'brouillons',
                'console_print_periodicity': 50,
                'plot_periodicity': 100,
                'save_periodicity': 50
                }
>>> my_algo.set_output_manager(OutputsSettings(settings))
```

### 3.3.6 `leaspy.algo.others`: Other algorithms

Reference algorithms to use with reference models (for benchmarks).

| | |
|---|---|
| *constant_prediction_algo.ConstantPredictionAlgorithm*(...) | ConstantPredictionAlgorithm is the algorithm that outputs a constant prediction |
| *lme_fit.LMEFitAlgorithm*(settings) | Calibration algorithm associated to *LMEModel* |
| *lme_personalize.LMEPersonalizeAlgorithm*(settings) | Personalization algorithm associated to *LMEModel* |

#### leaspy.algo.others.constant_prediction_algo.ConstantPredictionAlgorithm

**class ConstantPredictionAlgorithm**(*settings*)

Bases: *leaspy.algo.abstract_algo.AbstractAlgo*

ConstantPredictionAlgorithm is the algorithm that outputs a constant prediction

It is associated to *ConstantModel*

TODO: it should be a child of *AbstractPersonalizeAlgorithm* (refactoring needed)

**Parameters**

**settings** [*AlgorithmSettings*] The settings of constant prediction algorithm. It may define *prediction_type* (str):

- `'last'`: last value seen during calibration (even if NaN) [default],
- `'last_known'`: last non NaN value seen during calibration*§,
- `'max'`: maximum (=worst) value seen during calibration*§,
- `'mean'`: average of values seen during calibration§.

* <!> depending on features, the *last_known* / *max* value may correspond to different visits.

§ <!> for a given feature, value will be NaN if and only if all values for this feature were NaN.

**Raises**

**LeaspyAlgoInputError** If any invalid setting for the algorithm

### Methods

| | |
|---|---|
| *load_parameters*(parameters) | Update the algorithm's parameters by the ones in the given dictionary. |
| *run*(model, *args[, return_noise]) | Main method, run the algorithm. |
| *run_impl*(model, dataset) | Main method, refer to abstract definition in *run()*. |
| *set_output_manager*(settings) | Not implemented. |

**load_parameters**(*parameters: dict*)

Update the algorithm's parameters by the ones in the given dictionary. The keys in the io which does not belong to the algorithm's parameters keys are ignored.

**Parameters**

**parameters** [dict] Contains the pairs (key, value) of the wanted parameters

### Examples

```
>>> settings = leaspy.io.settings.algorithm_settings.AlgorithmSettings("mcmc_
↪saem")
>>> my_algo = leaspy.algo.fit.tensor_mcmcsaem.TensorMCMCSAEM(settings)
>>> my_algo.algo_parameters
{'n_iter': 10000,
 'n_burn_in_iter': 9000,
 'eps': 0.001,
 'L': 10,
 'sampler_ind': 'Gibbs',
 'sampler_pop': 'Gibbs',
 'annealing': {'do_annealing': False,
  'initial_temperature': 10,
  'n_plateau': 10,
  'n_iter': 200}}
>>> parameters = {'n_iter': 5000, 'n_burn_in_iter': 4000}
>>> my_algo.load_parameters(parameters)
>>> my_algo.algo_parameters
{'n_iter': 5000,
 'n_burn_in_iter': 4000,
 'eps': 0.001,
 'L': 10,
 'sampler_ind': 'Gibbs',
 'sampler_pop': 'Gibbs',
 'annealing': {'do_annealing': False,
  'initial_temperature': 10,
  'n_plateau': 10,
  'n_iter': 200}}
```

**run**(*model: AbstractModel*, **args*, *return_noise: bool = False*, ***extra_kwargs*) → Any

Main method, run the algorithm.

TODO fix proper abstract class method: input depends on algorithm… (esp. simulate != from others…)

**Parameters**

**model** [*AbstractModel*] The used model.

**dataset** [*Dataset*] Contains all the subjects' observations with corresponding time-points, in torch format to speed up computations.

**return_noise** [bool (default False), keyword only] Should the algorithm return main output and optional noise output as a 2-tuple?

**Returns**

**Depends on algorithm class: TODO change?**

**See also:**

*AbstractFitAlgo*

*AbstractPersonalizeAlgo*

*SimulationAlgorithm*

**run_impl**(*model: ConstantModel*, *dataset: Dataset*)
Main method, refer to abstract definition in *run()*.

**Parameters**

**model** [*ConstantModel*] A subclass object of leaspy *ConstantModel*.

**dataset** [*Dataset*] Dataset object build with leaspy class objects Data, algo & model

**Returns**

**individual_parameters** [*IndividualParameters*] Contains individual parameters.

**noise_std** [float] TODO: always 0 for now

**set_output_manager**(*settings*)
Not implemented.

### leaspy.algo.others.lme_fit.LMEFitAlgorithm

**class LMEFitAlgorithm**(*settings*)
Bases: *leaspy.algo.abstract_algo.AbstractAlgo*

Calibration algorithm associated to *LMEModel*

**Parameters**

**settings** [*AlgorithmSettings*]

- **with_random_slope_age** [bool] If False: only varying intercepts If True: random intercept & random slope w.r.t ages

  Deprecated since version 1.2.

  You should rather define this directly as an hyperparameter of LME model.

- **force_independent_random_effects** [bool] Force independence of random intercept & random slope
- other keyword arguments passed to `statsmodels.regression.mixed_linear_model.MixedLM.fit()`

**See also:**

`statsmodels.regression.mixed_linear_model.MixedLM`

### Methods

| | |
|---|---|
| *load_parameters*(parameters) | Update the algorithm's parameters by the ones in the given dictionary. |
| *run*(model, *args[, return_noise]) | Main method, run the algorithm. |
| *run_impl*(model, dataset) | Main method, refer to abstract definition in *run()*. |
| *set_output_manager*(settings) | Not implemented. |

**load_parameters**(*parameters: dict*)

Update the algorithm's parameters by the ones in the given dictionary. The keys in the io which does not belong to the algorithm's parameters keys are ignored.

**Parameters**

**parameters** [dict] Contains the pairs (key, value) of the wanted parameters

### Examples

```
>>> settings = leaspy.io.settings.algorithm_settings.AlgorithmSettings("mcmc_
↪saem")
>>> my_algo = leaspy.algo.fit.tensor_mcmcsaem.TensorMCMCSAEM(settings)
>>> my_algo.algo_parameters
{'n_iter': 10000,
 'n_burn_in_iter': 9000,
 'eps': 0.001,
 'L': 10,
 'sampler_ind': 'Gibbs',
 'sampler_pop': 'Gibbs',
 'annealing': {'do_annealing': False,
  'initial_temperature': 10,
  'n_plateau': 10,
  'n_iter': 200}}
>>> parameters = {'n_iter': 5000, 'n_burn_in_iter': 4000}
>>> my_algo.load_parameters(parameters)
>>> my_algo.algo_parameters
{'n_iter': 5000,
 'n_burn_in_iter': 4000,
 'eps': 0.001,
 'L': 10,
 'sampler_ind': 'Gibbs',
 'sampler_pop': 'Gibbs',
 'annealing': {'do_annealing': False,
  'initial_temperature': 10,
  'n_plateau': 10,
  'n_iter': 200}}
```

**run**(*model: AbstractModel*, **args*, *return_noise: bool = False*, ***extra_kwargs*) → Any

Main method, run the algorithm.

TODO fix proper abstract class method: input depends on algorithm… (esp. simulate != from others…)

**Parameters**

**model** [*AbstractModel*] The used model.

**dataset** [*Dataset*] Contains all the subjects' observations with corresponding time-points, in torch format to speed up computations.

**return_noise** [bool (default False), keyword only] Should the algorithm return main output and optional noise output as a 2-tuple?

**Returns**

**Depends on algorithm class: TODO change?**

**See also:**

*AbstractFitAlgo*

*AbstractPersonalizeAlgo*

*SimulationAlgorithm*

**run_impl**(*model: LMEModel*, *dataset: Dataset*)
Main method, refer to abstract definition in *run()*.

TODO fix proper inheritance

**Parameters**

**model** [*LMEModel*] A subclass object of leaspy *LMEModel*.

**dataset** [*Dataset*] Dataset object build with leaspy class objects Data, algo & model

**Returns**

**2-tuple:**

- None
- noise scale (std-dev), scalar

**set_output_manager**(*settings*)
Not implemented.

### leaspy.algo.others.lme_personalize.LMEPersonalizeAlgorithm

**class LMEPersonalizeAlgorithm**(*settings*)
Bases: *leaspy.algo.abstract_algo.AbstractAlgo*

Personalization algorithm associated to *LMEModel*

TODO: it should be a child of *AbstractPersonalizeAlgorithm* (refactoring needed)

**Parameters**

**settings** [*AlgorithmSettings*] Algorithm settings (none yet). Most LME parameters are defined within LME model and LME fit algorithm.

**Attributes**

**name** ['lme_personalize']

### Methods

| | |
|---|---|
| *load_parameters*(parameters) | Update the algorithm's parameters by the ones in the given dictionary. |
| *run*(model, *args[, return_noise]) | Main method, run the algorithm. |
| *run_impl*(model, dataset) | Main method, refer to abstract definition in *run()*. |
| *set_output_manager*(settings) | Not implemented. |

**load_parameters**(*parameters: dict*)

Update the algorithm's parameters by the ones in the given dictionary. The keys in the io which does not belong to the algorithm's parameters keys are ignored.

**Parameters**

**parameters** [dict] Contains the pairs (key, value) of the wanted parameters

### Examples

```
>>> settings = leaspy.io.settings.algorithm_settings.AlgorithmSettings("mcmc_
↪saem")
>>> my_algo = leaspy.algo.fit.tensor_mcmcsaem.TensorMCMCSAEM(settings)
>>> my_algo.algo_parameters
{'n_iter': 10000,
 'n_burn_in_iter': 9000,
 'eps': 0.001,
 'L': 10,
 'sampler_ind': 'Gibbs',
 'sampler_pop': 'Gibbs',
 'annealing': {'do_annealing': False,
  'initial_temperature': 10,
  'n_plateau': 10,
  'n_iter': 200}}
>>> parameters = {'n_iter': 5000, 'n_burn_in_iter': 4000}
>>> my_algo.load_parameters(parameters)
>>> my_algo.algo_parameters
{'n_iter': 5000,
 'n_burn_in_iter': 4000,
 'eps': 0.001,
 'L': 10,
 'sampler_ind': 'Gibbs',
 'sampler_pop': 'Gibbs',
 'annealing': {'do_annealing': False,
  'initial_temperature': 10,
  'n_plateau': 10,
  'n_iter': 200}}
```

**run**(*model: AbstractModel*, **args*, *return_noise: bool = False*, ***extra_kwargs*) → Any

Main method, run the algorithm.

TODO fix proper abstract class method: input depends on algorithm… (esp. simulate != from others…)

**Parameters**

**model** [*AbstractModel*] The used model.

**dataset** [*Dataset*] Contains all the subjects' observations with corresponding time-points, in torch format to speed up computations.

**return_noise** [bool (default False), keyword only] Should the algorithm return main output and optional noise output as a 2-tuple?

**Returns**

**Depends on algorithm class: TODO change?**

**See also:**

*AbstractFitAlgo*

*AbstractPersonalizeAlgo*

*SimulationAlgorithm*

**run_impl**(*model*, *dataset*)
Main method, refer to abstract definition in *run()*.

TODO fix proper inheritance

**Parameters**

**model** [*LMEModel*] A subclass object of leaspy *LMEModel*.

**dataset** [*Dataset*] Dataset object build with leaspy class objects Data, algo & model

**Returns**

**individual_parameters** [*IndividualParameters*] Contains individual parameters.

**noise_std** [float] The estimated noise

**set_output_manager**(*settings*)
Not implemented.

### 3.3.7 leaspy.algo.utils.samplers: Samplers

Samplers used by the MCMC algorithms.

| | |
|---|---|
| *abstract_sampler.AbstractSampler*(info, ...) | Abstract sampler class. |
| *gibbs_sampler.GibbsSampler*(info, n_patients) | Gibbs sampler class. |

#### leaspy.algo.utils.samplers.abstract_sampler.AbstractSampler

**class AbstractSampler**(*info: Dict[str, Any]*, *n_patients: int*)
Bases: object

Abstract sampler class.

**Parameters**

**info** [dict[str, Any]] The dictionary describing the random variable to sample. It should contains the following entries:

- name : str
- shape : tuple[int, ...]
- type : 'population' or 'individual'

**n_patients** [int > 0] Number of patients (useful for individual variables)

**Raises**

**LeaspyModelInputError**

**Attributes**

**acceptation_temp** [torch.Tensor] Acceptation rate for the sampler in MCMC-SAEM algorithm Keep the history of the last *temp_length* last steps

**name** [str] Name of variable

**shape** [tuple] Shape of variable

**temp_length** [int] Deepness of the history kept in the acceptation rate *acceptation_temp* Length of the *acceptation_temp* torch tensor

## leaspy.algo.utils.samplers.gibbs_sampler.GibbsSampler

**class GibbsSampler**(*info*, *n_patients*)

Bases: *leaspy.algo.utils.samplers.abstract_sampler.AbstractSampler*

Gibbs sampler class.

**Parameters**

**info** [dict[str, Any]] The dictionary describing the random variable to sample. It should contains the following entries:

- name : str
- shape : tuple[int, ...]
- type : 'population' or 'individual'

**n_patients** [int > 0] Number of patients (useful for individual variables)

**Raises**

**LeaspyModelInputError**

### Methods

| | |
|---|---|
| *sample*(data, model, realizations, ...) | Sample either as population or individual. |

**sample**(*data*, *model*, *realizations*, *temperature_inv*)

Sample either as population or individual.

Modifies in-place the realizations object.

**Parameters**

**data** [*Dataset*]

**model** [*AbstractModel*]

**realizations** [*CollectionRealization*]

**temperature_inv** [float > 0]

## 3.4 leaspy.dataset: Datasets

Give access to some synthetic longitudinal observations mimicking cohort of subjects with neurodegenerative disorders, as well as calibrated models and computed individual parameters.

| | |
|---|---|
| *loader.Loader*() | Contains static methods to load synthetic longitudinal dataset, calibrated *Leaspy* instances & *IndividualParameters*. |

### 3.4.1 leaspy.datasets.loader.Loader

**class Loader**

Bases: object

Contains static methods to load synthetic longitudinal dataset, calibrated *Leaspy* instances & *IndividualParameters*.

#### Notes

- A *Leaspy* instance named <name> have been calibrated on the dataset <name>.
- An *IndividualParameters* name <name> have been computed by personalizing the *Leaspy* instance named <name> on the dataset <name>.

See the documentation of each method to get their respective available names.

**Attributes**

**data_paths** [dict [str, str]] Contains the datasets' names and their respective path within `leaspy.datasets` subpackage.

**model_paths** [dict [str, str]] Contains the *Leaspy* instances' names and their respective path within `leaspy.datasets` subpackage.

**ip_paths** [dict [str, str]] Contains the individual parameters' names and their respective path within `leaspy.datasets` subpackage.

#### Methods

| | |
|---|---|
| *load_dataset*(dataset_name) | Load synthetic longitudinal observations mimicking cohort of subjects with neurodegenerative disorders. |
| *load_individual_parameters*(ip_name) | Load a Leaspy instance with a model already calibrated on the synthetic dataset corresponding to the name of the instance. |
| *load_leaspy_instance*(instance_name) | Load a Leaspy instance with a model already calibrated on the synthetic dataset corresponding to the name of the instance. |

**static load_dataset**(*dataset_name*)

Load synthetic longitudinal observations mimicking cohort of subjects with neurodegenerative disorders.

**Parameters**

**dataset_name** [{'parkinson-multivariate', 'alzheimer-multivariate', 'parkinson-putamen', 'parkinson-putamen-train_and_test'}] Name of the dataset.

**Returns**

**pandas.DataFrame** DataFrame containing the IDs, timepoints and observations.

### Notes

All *DataFrames* have the same structures.

- Index: a `pandas.MultiIndex` - `['ID', 'TIME']` which contain IDs and timepoints. The *DataFrame* is sorted by index. So, one line corresponds to one visit for one subject. The *DataFrame* having *'train_and_test'* in their name also have `'SPLIT'` as the third index level. It differenciate *train* and *test* data.
- Columns: One column correspond to one feature (or score).

**static load_individual_parameters**(*ip_name*)

Load a Leaspy instance with a model already calibrated on the synthetic dataset corresponding to the name of the instance.

**Parameters**

**ip_name** [{'parkinson-putamen-train'}] Name of the individual parameters.

**Returns**

***IndividualParameters*** Leaspy instance with a model already calibrated.

**static load_leaspy_instance**(*instance_name*)

Load a Leaspy instance with a model already calibrated on the synthetic dataset corresponding to the name of the instance.

**Parameters**

**instance_name** [{'parkinson-putamen-train'}] Name of the instance.

**Returns**

***Leaspy*** Leaspy instance with a model already calibrated.

## 3.5 `leaspy.io`: Inputs / Outputs

Containers classes used as input / outputs in the *Leaspy* package.

### 3.5.1 `leaspy.io.data`: Data containers

| | |
|---|---|
| *data.Data*() | Main data container, initialized from a *csv file* or a `pandas.DataFrame`. |
| *dataset.Dataset*(data[, model, algo]) | Data container based on `torch.Tensor`, used to run algorithms. |

### leaspy.io.data.data.Data

**class Data**

Bases: object

Main data container, initialized from a *csv file* or a pandas.DataFrame.

#### Methods

| | |
|---|---|
| *from_csv_file*(path, **kws) | Create a *Data* object from a CSV file. |
| *from_dataframe*(df, **kws) | Create a *Data* object from a pandas.DataFrame. |
| *from_individuals*(indices, timepoints, ...) | Create a *Data* class object from lists of *ID*, *timepoints* and the corresponding *values*. |
| *get_by_idx*(idx) | Get the IndividualData of a an individual identified by its ID. |
| *load_cofactors*(df, cofactors) | Load cofactors from a *pandas.DataFrame* to the *Data* object |
| *to_dataframe*([cofactors]) | Return the subjects' observations in a pandas.DataFrame along their ID and ages at all visits. |

**static from_csv_file**(*path: str*, ***kws*)

Create a *Data* object from a CSV file.

**Parameters**

**path** [str] Path to the CSV file to load (with extension)

****kws** Keyword arguments that are sent to CSVDataReader

**Returns**

*Data*

**static from_dataframe**(*df: DataFrame*, ***kws*)

Create a *Data* object from a pandas.DataFrame.

**Parameters**

**df** [pandas.DataFrame] Dataframe containing ID, TIME and features.

****kws** Keyword arguments that are sent to DataframeDataReader

**Returns**

***Data***

**static from_individuals**(*indices: List[str]*, *timepoints: List[List]*, *values: List[List]*, *headers: List[str]*)

Create a *Data* class object from lists of *ID*, *timepoints* and the corresponding *values*.

**Parameters**

**indices** [list[str]] Contains the individuals' ID.

**timepoints** [list[array-like 1D]] For each individual i, list of ages at visits. Number of timepoints is referred below as n_timepoints_i

**values** [list[array-like 2D]] For each individual i, all values at visits. Shape is (n_timepoints_i, n_features).

**headers** [list[str]] Contains the features' names.

**Returns**

***Data*** Data class object with all ID, timepoints, values and features' names.

`get_by_idx`(*idx: str*)
Get the `IndividualData` of a an individual identified by its ID.

**Returns**

`IndividualData`

`load_cofactors`(*df: DataFrame*, *cofactors: List[str]*)
Load cofactors from a *pandas.DataFrame* to the *Data* object

**Parameters**

**df** [`pandas.DataFrame`] the index is the list of subject ids

**cofactors** [list[str]] names of the column(s) of df which shall be loaded as cofactors

**Raises**

`LeaspyDataInputError`

`to_dataframe`(*cofactors=None*)
Return the subjects' observations in a `pandas.DataFrame` along their ID and ages at all visits.

**Parameters**

**cofactors** [str, list [str], optional (default None)] Contains the cofactors' names to be included in the DataFrame. By default, no cofactors are returned. If cofactors == "all", all the available cofactors are returned.

**Returns**

`pandas.DataFrame` Contains the subjects's ID, age and scores (optional - and cofactors) for each timepoint.

**Raises**

`LeaspyDataInputError`

## leaspy.io.data.dataset.Dataset

`class Dataset`(*data: Data*, *model: AbstractModel = None*, *algo: AbstractAlgo = None*)
Bases: `object`

Data container based on `torch.Tensor`, used to run algorithms.

**Parameters**

**data** [*Data*] Create *Dataset* from *Data* object

**model** [*AbstractModel* (optional)] If not None, will check compatibility of model and data

**algo** [*AbstractAlgo* (optional)] If not None, will check compatibility of algo and data

**Raises**

`LeaspyInputError` if data, model or algo are not compatible together.

**Attributes**

**headers** [list[str]] Features names

**dimension** [int] Number of features

**n_individuals** [int] Number of individuals

**indices** [list[ID]] Order of patients

**n_visits_per_individual** [list[int]] Number of visits per individual

**n_visits_max** [int] Maximum number of visits for one individual

**n_visits** [int] Total number of visits

**n_observations_per_ind_per_ft** [`torch.LongTensor`, shape (n_individuals, dimension)] Number of observations (not taking into account missing values) per individual per feature

**n_observations_per_ft** [`torch.LongTensor`, shape (dimension,)] Total number of observations per feature

**n_observations** [int] Total number of observations

**timepoints** [`torch.FloatTensor`, shape (n_individuals, n_visits_max)] Ages of patients at their different visits

**values** [`torch.FloatTensor`, shape (n_individuals, n_visits_max, dimension)] Values of patients for each visit for each feature

**mask** [`torch.FloatTensor`, shape (n_individuals, n_visits_max, dimension)] Binary mask associated to values. If 1: value is meaningful If 0: value is meaningless (either was nan or does not correspond to a real visit - only here for padding)

**L2_norm_per_ft** [`torch.FloatTensor`, shape (dimension,)] Sum of all non-nan squared values, feature per feature

**L2_norm** [scalar `torch.FloatTensor`] Sum of all non-nan squared values

### Methods

| | |
|---|---|
| *get_times_patient*(i) | Get ages for patient number `i` |
| *get_values_patient*(i) | Get values for patient number `i` |
| *to_pandas*() | Convert dataset to a *DataFrame*. |

**get_times_patient**(*i: int*) → torch.FloatTensor
Get ages for patient number `i`

**Returns**

**torch.Tensor, shape (n_obs_of_patient,)** Contains float

**get_values_patient**(*i: int*) → torch.FloatTensor
Get values for patient number `i`

**Returns**

**torch.Tensor, shape (n_obs_of_patient, dimension)** Contains float or nans

**to_pandas**() → DataFrame
Convert dataset to a *DataFrame*.

**Returns**

**pandas.DataFrame**

**class Data**
Main data container, initialized from a *csv file* or a `pandas.DataFrame`.

#### Methods

| | |
|---|---|
| *from_csv_file*(path, **kws) | Create a *Data* object from a CSV file. |
| *from_dataframe*(df, **kws) | Create a *Data* object from a `pandas.DataFrame`. |
| *from_individuals*(indices, timepoints, ...) | Create a *Data* class object from lists of *ID*, *timepoints* and the corresponding *values*. |
| *get_by_idx*(idx) | Get the `IndividualData` of a an individual identified by its ID. |
| *load_cofactors*(df, cofactors) | Load cofactors from a *pandas.DataFrame* to the *Data* object |
| *to_dataframe*([cofactors]) | Return the subjects' observations in a `pandas.DataFrame` along their ID and ages at all visits. |

### 3.5.2 `leaspy.io.settings`: Settings classes

| | |
|---|---|
| *model_settings.ModelSettings*(...) | Used in *Leaspy.load()* to create a *Leaspy* class object from a *json* file. |
| *algorithm_settings.AlgorithmSettings*(name, ...) | Used to set the algorithms' settings. |
| *outputs_settings.OutputsSettings*(settings) | Used to create the *logs* folder to monitor the convergence of the calibration algorithm. |

#### `leaspy.io.settings.model_settings`.ModelSettings

**class** ModelSettings(*path_to_model_settings_or_dict: Union[str, dict]*)

Bases: `object`

Used in *Leaspy.load()* to create a *Leaspy* class object from a *json* file.

**Parameters**

**path_to_model_settings_or_dict** [dict or str]

- If a str: path to a json file containing model settings
- If a dict: content of model settings

**Raises**

`LeaspyModelInputError`

#### `leaspy.io.settings.algorithm_settings`.AlgorithmSettings

**class** AlgorithmSettings(*name: str*, ***kwargs*)

Bases: `object`

Used to set the algorithms' settings. All parameters, except the choice of the algorithm, is set by default. The user can overwrite all default settings.

**Parameters**

**name** [str]

**The algorithm's name. Must be in:**

- **For *fit* algorithms:**

- `'mcmc_saem'`
- `'lme_fit'` (for LME model only)

- **For *personalize* algorithms:**
  - `'scipy_minimize'`
  - `'mean_real'`
  - `'mode_real'`
  - `'constant_prediction'` (for constant model only)
  - `'lme_personalize'` (for LME model only)
- **For *simulate* algorithms:**
  - `'simulation'`

**model_initialization_method** [str, optional] For **fit** algorithms only, give a model initialization method, according to those possible in *initialize_parameters()*.

**algo_initialization_method** [str, optional] Personalize the algorithm initialization method, according to those possible for the given algorithm (refer to its documentation in `leaspy.algo`).

**n_iter** [int, optional] Number of iteration. There is no stopping criteria for the all the MCMC SAEM algorithms.

**n_burn_in_iter** [int, optional] Number of iteration during burning phase, used for the MCMC SAEM algorithms.

**seed** [int, optional, default None] Used for stochastic algorithms.

**use_jacobian** [bool, optional, default False] Used in `scipy_minimize` algorithm to perform a *L-BFGS* instead of a *Powell* algorithm.

**n_jobs** [int, optional, default 1] Used in `scipy_minimize` algorithm to accelerate calculation with parallel derivation using joblib.

**progress_bar** [bool, optional, default False] Used to display a progress bar during computation.

**Raises**

`LeaspyAlgoInputError`

**See also:**

`leaspy.algo`

### Notes

For developers: use `_dynamic_default_parameters` to dynamically set some default parameters, depending on other parameters that were set, while these *dynamic* parameters were not set.

**Example:** you could want to set burn in iterations or annealing iterations as fractions of non-default number of iterations given.

Format:

```
{algo_name: [
    (functional_condition_to_trigger_dynamic_setting(kwargs),
    {
        nested_keys_of_dynamic_setting: dynamic_value(kwargs)
```

(continues on next page)

(continued from previous page)

```
    })
]}
```

**Attributes**

**name** [str] The algorithm's name.

**model_initialization_method** [str, optional] For fit algorithms, give a model initialization method, according to those possible in *initialize_parameters()*.

**algo_initialization_method** [str, optional] Personalize the algorithm initialization method, according to those possible for the given algorithm (refer to its documentation in `leaspy.algo`).

**seed** [int, optional, default None] Used for stochastic algorithms.

**parameters** [dict] Contains the other parameters: *n_iter*, *n_burn_in_iter*, *use_jacobian*, *n_jobs* & *progress_bar*.

**logs** [*OutputsSettings*, optional] Used to create a `logs` file during a model calibration containing convergence information.

### Methods

| | |
|---|---|
| *check_consistency*() | Check internal consistency of algorithm settings and warn or raise a *LeaspyAlgoInputError* if not. |
| *load*(path_to_algorithm_settings) | Instantiate a AlgorithmSettings object a from json file. |
| *save*(path, **kwargs) | Save an AlgorithmSettings object in a json file. |
| *set_logs*(path, **kwargs) | Use this method to monitor the convergence of a model callibration. |

**property algo_class**
Class of the algorithm derived from its name (shorthand).

**check_consistency**() → None
Check internal consistency of algorithm settings and warn or raise a *LeaspyAlgoInputError* if not.

**classmethod load**(*path_to_algorithm_settings: str*)
Instantiate a AlgorithmSettings object a from json file.

**Parameters**

**path_to_algorithm_settings** [str] Path of the json file.

**Returns**

***AlgorithmSettings*** An instanced of AlgorithmSettings with specified parameters.

**Raises**

**LeaspyAlgoInputError** if anything is invalid in algo settings

### Examples

```
>>> from leaspy import AlgorithmSettings
>>> leaspy_univariate = AlgorithmSettings.load('outputs/leaspy-univariate_
↪model-settings.json')
```

**save**(*path: str*, ***kwargs*)

Save an AlgorithmSettings object in a json file.

TODO? save leaspy version as well for retro/future-compatibility issues?

**Parameters**

**path** [str] Path to store the AlgorithmSettings.

****kwargs** Keyword arguments for json.dump method. Default: dict(indent=2)

### Examples

```
>>> from leaspy import AlgorithmSettings
>>> settings = AlgorithmSettings('scipy_minimize', seed=42, n_jobs=-1, use_
↪jacobian=True, progress_bar=True)
>>> settings.save('outputs/scipy_minimize-settings.json')
```

**set_logs**(*path*, ***kwargs*)

Use this method to monitor the convergence of a model callibration.

It create graphs and csv files of the values of the population parameters (fixed effects) during the callibration

**Parameters**

**path** [str] The path of the folder to store the graphs and csv files.

****kwargs**

- **console_print_periodicity: int, optional, default 50** Display logs in the console/terminal every N iterations.
- **plot_periodicity: int, optional, default 100** Saves the values to display in pdf every N iterations.
- **save_periodicity: int, optional, default 50** Saves the values in csv files every N iterations.
- **overwrite_logs_folder: bool, optional, default False** Set it to `True` to overwrite the content of the folder in `path`.

**Raises**

**LeaspyAlgoInputError** If the folder given in `path` already exists and if `overwrite_logs_folder` is set to `False`.

### Notes

By default, if the folder given in `path` already exists, the method will raise an error. To overwrite the content of the folder, set `overwrite_logs_folder` it to `True`.

## leaspy.io.settings.outputs_settings.OutputsSettings

**class** OutputsSettings(*settings*)

Bases: object

Used to create the *logs* folder to monitor the convergence of the calibration algorithm.

**Parameters**

**settings** [dict[str, Any]]

**Parameters of the object. It may be in:**

- **console_print_periodicity** [int] Flag to log into console convergence data every N iterations
- **plot_periodicity** [int] Flag to plot convergence data every N iterations
- **save_periodicity** [int] Flag to save convergence data every N iterations
- **overwrite_logs_folder** [bool] Flag to remove all previous logs if existing (default False)
- **path** [str] Where to store logs (default to './_outputs/')

**Raises**

LeaspyAlgoInputError

**class** AlgorithmSettings(*name: str*, ***kwargs*)

Used to set the algorithms' settings. All parameters, except the choice of the algorithm, is set by default. The user can overwrite all default settings.

**Parameters**

**name** [str]

**The algorithm's name. Must be in:**

- **For *fit* algorithms:**
  - `'mcmc_saem'`
  - `'lme_fit'` (for LME model only)
- **For *personalize* algorithms:**
  - `'scipy_minimize'`
  - `'mean_real'`
  - `'mode_real'`
  - `'constant_prediction'` (for constant model only)
  - `'lme_personalize'` (for LME model only)
- **For *simulate* algorithms:**
  - `'simulation'`

**model_initialization_method** [str, optional] For **fit** algorithms only, give a model initialization method, according to those possible in *initialize_parameters()*.

**algo_initialization_method** [str, optional] Personalize the algorithm initialization method, according to those possible for the given algorithm (refer to its documentation in `leaspy.algo`).

**n_iter** [int, optional] Number of iteration. There is no stopping criteria for the all the MCMC SAEM algorithms.

**n_burn_in_iter** [int, optional] Number of iteration during burning phase, used for the MCMC SAEM algorithms.

**seed** [int, optional, default None] Used for stochastic algorithms.

**use_jacobian** [bool, optional, default False] Used in `scipy_minimize` algorithm to perform a *L-BFGS* instead of a *Powell* algorithm.

**n_jobs** [int, optional, default 1] Used in `scipy_minimize` algorithm to accelerate calculation with parallel derivation using joblib.

**progress_bar** [bool, optional, default False] Used to display a progress bar during computation.

**Raises**

`LeaspyAlgoInputError`

**See also:**

`leaspy.algo`

### Notes

For developers: use `_dynamic_default_parameters` to dynamically set some default parameters, depending on other parameters that were set, while these *dynamic* parameters were not set.
**Example:** you could want to set burn in iterations or annealing iterations as fractions of non-default number of iterations given.
Format:

```
{algo_name: [
    (functional_condition_to_trigger_dynamic_setting(kwargs),
    {
        nested_keys_of_dynamic_setting: dynamic_value(kwargs)
    })
]}
```

**Attributes**

**name** [str] The algorithm's name.

**model_initialization_method** [str, optional] For fit algorithms, give a model initialization method, according to those possible in *initialize_parameters()*.

**algo_initialization_method** [str, optional] Personalize the algorithm initialization method, according to those possible for the given algorithm (refer to its documentation in `leaspy.algo`).

**seed** [int, optional, default None] Used for stochastic algorithms.

**parameters** [dict] Contains the other parameters: *n_iter*, *n_burn_in_iter*, *use_jacobian*, *n_jobs* & *progress_bar*.

**logs** [*OutputsSettings*, optional] Used to create a `logs` file during a model calibration containing convergence information.

#### Methods

| | |
|---|---|
| *check_consistency*() | Check internal consistency of algorithm settings and warn or raise a *LeaspyAlgoInputError* if not. |
| *load*(path_to_algorithm_settings) | Instantiate a AlgorithmSettings object a from json file. |
| *save*(path, **kwargs) | Save an AlgorithmSettings object in a json file. |
| *set_logs*(path, **kwargs) | Use this method to monitor the convergence of a model callibration. |

### 3.5.3 leaspy.io.outputs: Outputs classes

| | |
|---|---|
| *individual_parameters.IndividualParameters*() | Data container for individual parameters, contains IDs, timepoints and observations values. |

#### leaspy.io.outputs.individual_parameters.IndividualParameters

**class IndividualParameters**

Bases: object

Data container for individual parameters, contains IDs, timepoints and observations values. Output of the *Leaspy.personalize()* method, contains the *random effects*.

There are used as output of the *personalization algorithms* and as input/output of the *simulation algorithm*, to provide an initial distribution of individual parameters.

**Attributes**

**_indices** [list] List of the patient indices

**_individual_parameters** [dict] Individual indices (key) with their corresponding individual parameters {parameter name: parameter value}

**_parameters_shape** [dict] Shape of each individual parameter

**_default_saving_type** [str] Default extension for saving when none is provided

#### Methods

| | |
|---|---|
| *add_individual_parameters*(index, ...) | Add the individual parameter of an individual to the IndividualParameters object |
| *from_dataframe*(df) | Static method that returns an IndividualParameters object from the dataframe |
| *from_pytorch*(indices, dict_pytorch) | Static method that returns an IndividualParameters object from the indices and pytorch dictionary |
| *get_aggregate*(parameter, function) | Returns the result of aggregation by *function* of parameter values across all patients |
| *get_mean*(parameter) | Returns the mean value of a parameter across all patients |
| *get_std*(parameter) | Returns the stardard deviation of a parameter across all patients |

continues on next page

Table 49 – continued from previous page

| | |
|---|---|
| *items*() | Get items of dict _individual_parameters. |
| *load*(path) | Static method that loads the individual parameters (json or csv) existing at the path locatio |
| *save*(path, **kwargs) | Saves the individual parameters (json or csv) at the path location |
| *subset*(indices, *[, copy]) | Returns IndividualParameters object with a subset of the initial individuals |
| *to_dataframe*() | Returns the dataframe of individual parameters |
| *to_pytorch*() | Returns the indices and pytorch dictionary of individual parameters |

**add_individual_parameters**(*index: str*, *individual_parameters: Dict[str, Any]*)

Add the individual parameter of an individual to the IndividualParameters object

**Parameters**

**index** [str] Index of the individual

**individual_parameters** [dict] Individual parameters of the individual {name: value:}

**Raises**

**LeaspyIndividualParamsInputError**

- If the index is not a string or has already been added
- Or if the individual parameters is not a dict.
- Or if individual parameters are not self-consistent.

### Examples

Add two individual with tau, xi and sources parameters

```
>>> ip = IndividualParameters()
>>> ip.add_individual_parameters('index-1', {"xi": 0.1, "tau": 70, "sources":␣
↪[0.1, -0.3]})
>>> ip.add_individual_parameters('index-2', {"xi": 0.2, "tau": 73, "sources":␣
↪[-0.4, -0.1]})
```

**static from_dataframe**(*df: DataFrame*)

Static method that returns an IndividualParameters object from the dataframe

**Parameters**

**df** [pandas.DataFrame] Dataframe of the invidual parameters. Each row must correspond to one individual. The index corresponds to the individual index. The columns are the names of the parameters.

**Returns**

***IndividualParameters***

**static from_pytorch**(*indices: List[str]*, *dict_pytorch: Dict[str, torch.FloatTensor]*)

Static method that returns an IndividualParameters object from the indices and pytorch dictionary

**Parameters**

**indices** [list[ID]] List of the patients indices

**dict_pytorch** [dict[parameter:str, *torch.Tensor*]] Dictionary of the individual parameters

**Returns**

*IndividualParameters*

**Raises**

**LeaspyIndividualParamsInputError**

### Examples

```
>>> indices = ['index-1', 'index-2', 'index-3']
>>> ip_pytorch = {
>>>    "xi": torch.tensor([[0.1], [0.2], [0.3]], dtype=torch.float32),
>>>    "tau": torch.tensor([[70], [73], [58.]], dtype=torch.float32),
>>>    "sources": torch.tensor([[0.1, -0.3], [-0.4, 0.1], [-0.6, 0.2]],␣
↪dtype=torch.float32)
>>> }
>>> ip_pytorch = IndividualParameters.from_pytorch(indices, ip_pytorch)
```

**get_aggregate**(*parameter: str*, *function: Callable*) → List

Returns the result of aggregation by *function* of parameter values across all patients

**Parameters**

**parameter** [str] Name of the parameter

**function** [callable] A function operating on iterables and supporting axis keyword, and outputing an iterable supporting the *tolist* method.

**Returns**

**list or float (depending on parameter shape)** Resulting value of the parameter

**Raises**

**LeaspyIndividualParamsInputError**

- If individual parameters are empty,
- or if the parameter is not in the IndividualParameters.

### Examples

```
>>> ip = IndividualParameters.load("path/to/individual_parameters")
>>> tau_median = ip.get_aggregate("tau", np.median)
```

**get_mean**(*parameter: str*)

Returns the mean value of a parameter across all patients

**Parameters**

**parameter** [str] Name of the parameter

**Returns**

**list or float (depending on parameter shape)** Mean value of the parameter

**Raises**

**LeaspyIndividualParamsInputError**

- If individual parameters are empty,
- or if the parameter is not in the IndividualParameters.

### Examples

```
>>> ip = IndividualParameters.load("path/to/individual_parameters")
>>> tau_mean = ip.get_mean("tau")
```

**get_std**(*parameter: str*)

Returns the stardard deviation of a parameter across all patients

**Parameters**

**parameter** [str] Name of the parameter

**Returns**

**list or float (depending on parameter shape)** Standard-deviation value of the parameter

**Raises**

**LeaspyIndividualParamsInputError**

- If individual parameters are empty,
- or if the parameter is not in the IndividualParameters.

### Examples

```
>>> ip = IndividualParameters.load("path/to/individual_parameters")
>>> tau_std = ip.get_std("tau")
```

**items**()

Get items of dict `_individual_parameters`.

**classmethod load**(*path: str*)

Static method that loads the individual parameters (json or csv) existing at the path locatio

**Parameters**

**path** [str] Path and file name of the individual parameters.

**Returns**

***IndividualParameters*** Individual parameters object load from the file

**Raises**

**LeaspyIndividualParamsInputError** If the provided extension is not *csv* or not *json*.

### Examples

```
>>> ip = IndividualParameters.load('/path/to/individual_parameters_1.json')
>>> ip2 = IndividualParameters.load('/path/to/individual_parameters_2.csv')
```

**save**(*path: str*, ***kwargs*)

Saves the individual parameters (json or csv) at the path location

TODO? save leaspy version as well for retro/future-compatibility issues?

**Parameters**

**path** [str] Path and file name of the individual parameters. The extension can be json or csv. If no extension, default extension (csv) is used

****kwargs** Additional keyword arguments to pass to either: * `pandas.DataFrame.to_csv()` * `json.dump()` depending on saving format requested

**Raises**

**LeaspyIndividualParamsInputError**

- If extension not supported for saving
- If individual parameters are empty

**subset**(*indices: Iterable[str]*, ***, *copy: bool = True*)

Returns IndividualParameters object with a subset of the initial individuals

**Parameters**

**indices** [list[ID]] List of strings that corresponds to the indices of the individuals to return

**copy** [bool, optional (default True)] Should we copy underlying parameters or not?

**Returns**

***IndividualParameters*** An instance of the IndividualParameters object with the selected list of individuals

**Raises**

**LeaspyIndividualParamsInputError** Raise an error if one of the index is not in the IndividualParameters

### Examples

```
>>> ip = IndividualParameters()
>>> ip.add_individual_parameters('index-1', {"xi": 0.1, "tau": 70, "sources":␣
↪[0.1, -0.3]})
>>> ip.add_individual_parameters('index-2', {"xi": 0.2, "tau": 73, "sources":␣
↪[-0.4, -0.1]})
>>> ip.add_individual_parameters('index-3', {"xi": 0.3, "tau": 58, "sources":␣
↪[-0.6, 0.2]})
>>> ip_sub = ip.subset(['index-1', 'index-3'])
```

**to_dataframe**() → DataFrame

Returns the dataframe of individual parameters

**Returns**

**pandas.DataFrame** Each row corresponds to one individual. The index corresponds to the individual index ('ID'). The columns are the names of the parameters.

#### Examples

Convert the individual parameters object into a dataframe

```
>>> ip = IndividualParameters.load("path/to/individual_parameters")
>>> ip_df = ip.to_dataframe()
```

**to_pytorch**() → Tuple[List[str], Dict[str, torch.FloatTensor]]
Returns the indices and pytorch dictionary of individual parameters

**Returns**

**indices: list[ID]** List of patient indices

**pytorch_dict: dict[parameter:str, *torch.Tensor*]** Dictionary of the individual parameters {parameter name: pytorch tensor of values across individuals}

#### Examples

Convert the individual parameters object into a dataframe

```
>>> ip = IndividualParameters.load("path/to/individual_parameters")
>>> indices, ip_pytorch = ip.to_pytorch()
```

### 3.5.4 leaspy.io.realizations: Realizations classes

Internal classes used for random variables in MCMC algorithms.

| | |
|---|---|
| *realization.Realization*(name, shape, ...) | Contains the realization of a given parameter. |
| *collection_realization.CollectionRealization*() | Realizations of population and individual parameters. |

#### leaspy.io.realizations.realization.Realization

**class Realization**(*name: str*, *shape: Tuple[int, ...]*, *variable_type: str*)
Bases: object

Contains the realization of a given parameter.

**Parameters**

**name** [str] Variable name

**shape** [tuple of int] Shape of variable (multiple dimensions allowed)

**variable_type** [str] 'individual' or 'population' variable?

**Attributes**

**name** [str] Variable name

**shape** [tuple of int] Shape of variable (multiple dimensions allowed)

**variable_type** [str] 'individual' or 'population' variable?

**tensor_realizations** [torch.Tensor] Actual realizations, whose shape is given by *shape*

### Methods

| | |
|---|---|
| *copy*() | Copy the Realization object |
| *from_tensor*(name, shape, variable_type, ...) | Create realization from variable infos and torch tensor object |
| *initialize*(n_individuals, model[, ...]) | Initialize realization from a given model. |
| *set_autograd*() | Set autograd for tensor of realizations |
| *set_tensor_realizations_element*(element, dim) | Manually change the value (in-place) of *tensor_realizations* at dimension *dim*. |
| *unset_autograd*() | Unset autograd for tensor of realizations |

**copy**()
Copy the Realization object

#### Notes

**From PyTorch torch.Tensor.clone() doc:** Unlike copy_(), this function is recorded in the computation graph. Gradients propagating to the cloned tensor will propagate to the original tensor.

**classmethod from_tensor**(*name: str*, *shape: Tuple[int, ...]*, *variable_type: str*, *tensor_realization: torch.FloatTensor*)
Create realization from variable infos and torch tensor object

**Parameters**

**name** [str] Variable name

**shape** [tuple of int] Shape of variable (multiple dimensions allowed)

**variable_type** [str] 'individual' or 'population' variable?

**tensor_realization** [torch.Tensor] Actual realizations, whose shape is given by *shape*

**Returns**

*Realization*

**initialize**(*n_individuals: int*, *model: AbstractModel*, *scale_individual: float = 1.0*)
Initialize realization from a given model.

**Parameters**

**n_individuals** [int > 0] Number of individuals

**model** [*AbstractModel*] The model you want realizations for.

**scale_individual** [float > 0] Multiplicative factor to scale the std-dev as given by model parameters

**Raises**

**LeaspyModelInputError** if unknown variable type

**set_autograd**()
Set autograd for tensor of realizations

**Raises**

`ValueError` if inconsistent internal request

**See also:**

`torch.Tensor.requires_grad_`

**set_tensor_realizations_element**(*element*, *dim: int*)
Manually change the value (in-place) of *tensor_realizations* at dimension *dim*.

**unset_autograd**()
Unset autograd for tensor of realizations

**Raises**

`ValueError` if inconsistent internal request

**See also:**

`torch.Tensor.requires_grad_`

### leaspy.io.realizations.collection_realization.CollectionRealization

**class CollectionRealization**
Bases: `object`

Realizations of population and individual parameters.

#### Methods

| | |
|---|---|
| *copy*() | Copy of self instance |
| *initialize*(n_individuals, model, *[, ...]) | Initialize the Collection Realization with a model. |
| *initialize_from_values*(n_individuals, model) | cf. |
| *keys*() | Return all variable names |
| *to_dict*() | Returns 2 dictionaries with realizations |

**copy**()
Copy of self instance

**Returns**

***CollectionRealization***

**initialize**(*n_individuals: int*, *model: AbstractModel*, *, *scale_individual: float = 1.0*)
Initialize the Collection Realization with a model.

Idem that *initialize_from_values()* method except it calls *Realization.initialize()* with `scale_individual=1` by default.

**Parameters**

**n_individuals** [int] Number of individuals modelled

**model** [*AbstractModel*] Model we initialize from

**scale_individual** [float] Individual scale, cf. *Realization.initialize()*

**initialize_from_values**(*n_individuals: int*, *model: AbstractModel*)
cf. *initialize*

**keys**()
Return all variable names

**to_dict**()
Returns 2 dictionaries with realizations

**Returns**

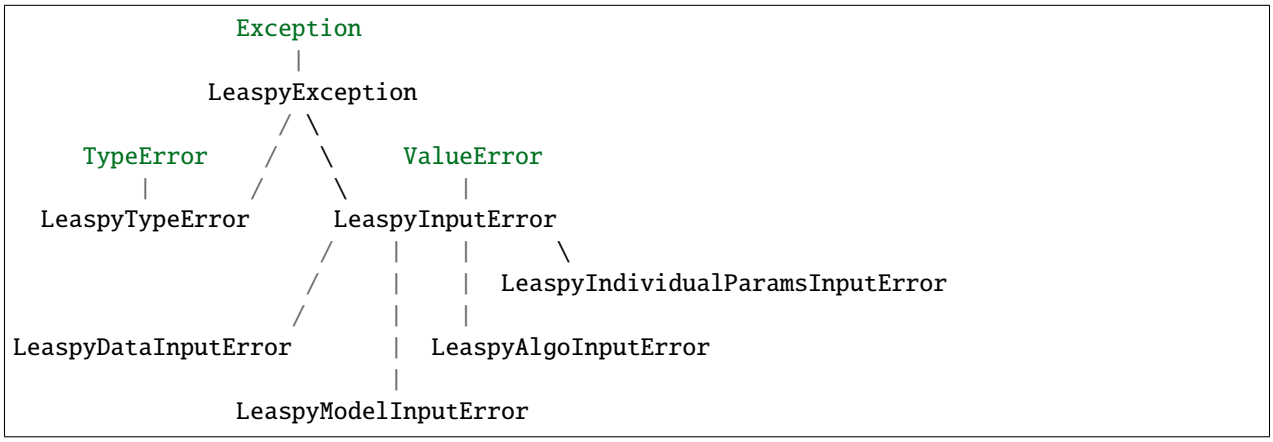
**reals_pop** [dict[var_name: str, `torch.FloatTensor`]] Realizations of population variables

**reals_ind** [dict[var_name: str, `torch.FloatTensor`]] Realizations of individual variables

## 3.6 `leaspy.exceptions`: Exceptions

Define custom Leaspy exceptions for better downstream handling.

Exceptions classes are nested so to handle in the most convenient way for users:




**For I/O operations, non-Leaspy specific errors may be raised, in particular:**

- `FileNotFoundError`
- `NotADirectoryError`

| | |
|---|---|
| `LeaspyException` | Base of all Leaspy exceptions. |
| `LeaspyTypeError` | Leaspy Exception, deriving from *TypeError*. |
| `LeaspyInputError` | Leaspy Exception, deriving from *ValueError*. |
| `LeaspyDataInputError` | Leaspy Input Error for data related issues. |
| `LeaspyModelInputError` | Leaspy Input Error for model related issues. |
| `LeaspyAlgoInputError` | Leaspy Input Error for algorithm related issues. |
| `LeaspyIndividualParamsInputError` | Leaspy Input Error for individual parameters related issues. |

### 3.6.1 leaspy.exceptions.LeaspyException

**exception LeaspyException**
Bases: Exception

Base of all Leaspy exceptions.

### 3.6.2 leaspy.exceptions.LeaspyTypeError

**exception LeaspyTypeError**
Bases: leaspy.exceptions.LeaspyException, TypeError

Leaspy Exception, deriving from *TypeError*.

### 3.6.3 leaspy.exceptions.LeaspyInputError

**exception LeaspyInputError**
Bases: leaspy.exceptions.LeaspyException, ValueError

Leaspy Exception, deriving from *ValueError*.

### 3.6.4 leaspy.exceptions.LeaspyDataInputError

**exception LeaspyDataInputError**
Bases: leaspy.exceptions.LeaspyInputError

Leaspy Input Error for data related issues.

### 3.6.5 leaspy.exceptions.LeaspyModelInputError

**exception LeaspyModelInputError**
Bases: leaspy.exceptions.LeaspyInputError

Leaspy Input Error for model related issues.

### 3.6.6 leaspy.exceptions.LeaspyAlgoInputError

**exception LeaspyAlgoInputError**
Bases: leaspy.exceptions.LeaspyInputError

Leaspy Input Error for algorithm related issues.

### 3.6.7 leaspy.exceptions.LeaspyIndividualParamsInputError

**exception LeaspyIndividualParamsInputError**

Bases: leaspy.exceptions.LeaspyInputError

Leaspy Input Error for individual parameters related issues.

CHAPTER

FOUR

# USER GUIDE

TODO

## 4.1 Mathematical aspects

### 4.1.1 Introduction

TODO

### 4.1.2 Mathematical formulation

TODO

### 4.1.3 Riemanian framework

TODO

### 4.1.4 Missing data

TODO

## 4.2 Leaspy's tutorial

### 4.2.1 What do I need?

TODO

### 4.2.2 Derive the population parameters

TODO

### 4.2.3 Derive the individual parameters

TODO

### 4.2.4 Cofactor analysis

TODO

### 4.2.5 What about missing values?

TODO

### 4.2.6 Predictions

TODO

### 4.2.7 Simulations

TODO

CHAPTER

# FIVE

# INDEX

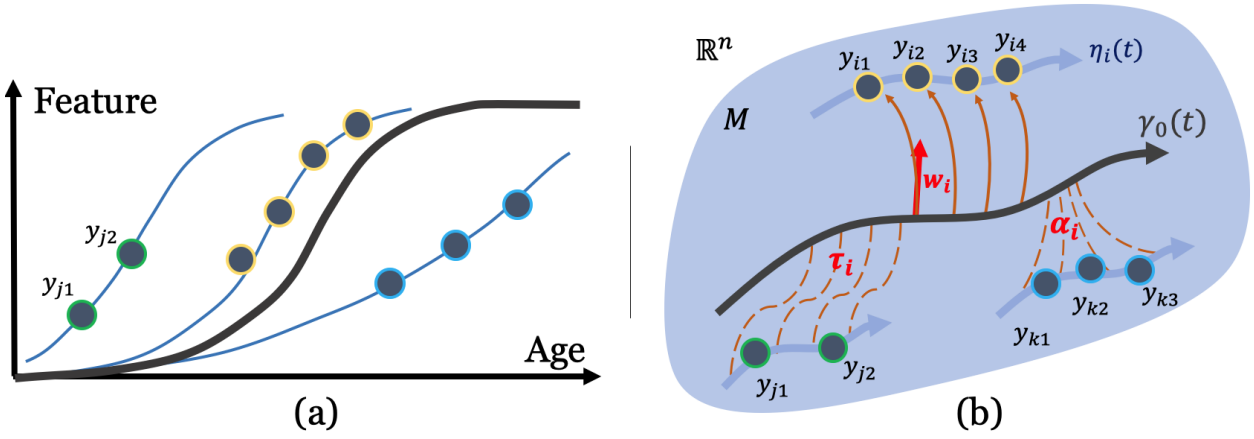
CHAPTER

SIX

# LEARNING SPATIOTEMPORAL PATTERNS IN PYTHON

## 6.1 Description

**Leaspy** is a software package for the statistical analysis of **longitudinal data**, particularly **medical** data that comes in a form of **repeated observations** of patients at different time-points.




Considering these series of short-term data, the software aims at :

- Recombining them to reconstruct the long-term spatio-temporal trajectory of evolution
- Positioning each patient observations relatively to the group-average timeline, in term of both temporal differences (time shift and acceleration factor) and spatial differences (different sequences of events, spatial pattern of progression, ...)
- Quantifying impact of cofactors (gender, genetic mutation, environmental factors, ...) on the evolution of the signal
- Imputing missing values
- Predicting future observations
- Simulating virtual patients to un-bias the initial cohort or mimic its characteristics

The software package can be used with scalar multivariate data whose progression can be modelled by a logistic shape, an exponential decay or a linear progression. The simplest type of data handled by the software are scalar data: they

correspond to one (univariate) or multiple (multivariate) measurement(s) per patient observation. This includes, for instance, clinical scores, cognitive assessments, physiological measurements (e.g. blood markers, radioactive markers) but also imaging-derived data that are rescaled, for instance, between 0 and 1 to describe a logistic progression.

## 6.2 Getting started

Information to install, test, and contribute to the package.

## 6.3 API Documentation

The exact API of all functions and classes, as given in the docstrings. The API documents expected types and allowed features for all functions, and all parameters available for the algorithms.

## 6.4 User Guide

The main documentation. This contains an in-depth description of all algorithms and how to apply them.

## 6.5 License

The package is distributed under the BSD 3-Clause license.

## 6.6 Further information

More detailed explanations about the models themselves and about the estimation procedure can be found in the following articles :

- **Mathematical framework**: *A Bayesian mixed-effects model to learn trajectories of changes from repeated manifold-valued observations.* Jean-Baptiste Schiratti, Stéphanie Allassonnière, Olivier Colliot, and Stanley Durrleman. The Journal of Machine Learning Research, 18:1–33, December 2017. Open Access
- **Application to imaging data**: *Statistical learning of spatiotemporal patterns from longitudinal manifold-valued networks.* I. Koval, J.-B. Schiratti, A. Routier, M. Bacci, O. Colliot, S. Allassonnière and S. Durrleman. MICCAI, September 2017. Open Access
- **Application to imaging data**: *Spatiotemporal Propagation of the Cortical Atrophy: Population and Individual Patterns.* Igor Koval, Jean-Baptiste Schiratti, Alexandre Routier, Michael Bacci, Olivier Colliot, Stéphanie Allassonnière, and Stanley Durrleman. Front Neurol. 2018 May 4;9:235. Open Access
- **Application to data with missing values**: *Learning disease progression models with longitudinal data and missing values*. R. Couronne, M. Vidailhet, JC. Corvol, S. Lehéricy, S. Durrleman. ISBI, April 2019. Open Access
- **Intensive application for Alzheimer's Disease progression**: *AD Course Map charts Alzheimer's disease progression*, I. Koval, A. Bone, M. Louis, S. Bottani, A. Marcoux, J. Samper-Gonzalez, N. Burgos, B. Charlier, A. Bertrand, S. Epelbaum, O. Colliot, S. Allassonniere & S. Durrleman, Scientific Reports, 2021. 11(1):1-16 Open Access
- www.digital-brain.org : Website related to the application of the model for Alzheimer's disease.

- Disease Course Mapping webpage by Igor Koval

# INDEX

## A



## C

## K



## L



## M



## O

## U



## V