MDRefine Documentation

Contents

Module MDRefine

A package to perform refinement of MD simulation trajectories.

Source code is on GitHub[1]. A test pdf manual is available here[2].

Examples

In the examples[3] directory you can find a number of notebooks that can be used as a source of inspiration.

Sub-modules

- MDRefine.MDRefinement
- MDRefine.bayesian
- MDRefine.data_loading
- MDRefine.hyperminimizer
- MDRefine.loss_and_minimizer

Functions

Function get_version

```
def get_version()
```

Module MDRefine.MDRefinement

Main tool: MDRefinement(). It refines MD-generated trajectories with customizable refinement.

Functions

Function MDRefinement

```
def MDRefinement(
    data,
    *,
    regularization: dict = None,
    stride: int = 1,
    starting_alpha: float = inf,
    starting_beta: float = inf,
    starting_gamma: float = inf,
    random_states=5,
    which_set: str = 'validation',
    gtol: float = 0.5,
    ftol: float = 0.05,
    results_folder_name: str = 'results',
    n_parallel_jobs: int = None,
    id_code: str = None
```

[1] https://github.com/bussilab/MDRefine
[2] ../MDRefine.pdf
[3] ../examples

)

This is the main tool of the package: it loads data, searches for the optimal hyperparameters and minimizes the loss function on the whole data set by using the opimized hyperparameters. The output variables are then saved in a folder; they include input values, min_lambdas (optimal lambda coefficients for Ensemble Refinement, when performed), result, hyper_search (steps in the search for optimal hyperparameters) (.csv files) and the .npy arrays with the new weights determined in the refinement.

Parameters

`data` : class instance The data set: an instance of the data_loading.my_data class or split_dataset.my_data_trainvali class (in the case with test observables). It can also be infos, a dictionary of information used to load data with load_data (see in the Examples directory).

`regularization` : dict A dictionary which can include two keys: force_field_reg and forward_model_reg, to specify the regularizations to the force-field correction and the forward model, respectively; the first key is either a string (among plain l2, constraint 1, constraint 2, KL divergence) or a user-defined function which takes as input pars_ff and returns the regularization term to be multiplied by the hyperparameter beta; the second key is a user-defined function which takes as input pars_fm and forward_coeffs_0 (current and refined forward-model coefficients) and returns the regularization term to be multiplied by the hyperparameter gamma.

`stride` : int The stride of the frames used to load data employed in search for optimal hyperparameters (used only when passing infos rather than data). In order to reduce the computational cost, at the price of a lower representativeness of the ensembles.

`starting_alpha, starting_beta, starting_gamma` : floats Starting values of the hyperparameters (np.inf by default, namely no refinement in that direction).

`random_states` : int or list of integers Random states (i.e., seeds) used to split the data set in cross validation (if integer, then `random_states = np.arange(random_states)`.

`which_set` : str String chosen among `'training'`, `'valid_frames'` or `'validation'`, which specifies how to determine optimal hyperparameters: if minimizing the (average) chi2 on the training set for `'training'`, on training observables and validation frames for `'valid_frames'`, on validation observables and validation frames for `'validation'`, on validation observables and all frames for `'valid_obs'`.

`gtol` : float Tolerance gtol (on the gradient) of scipy.optimize.minimize (0.5 by default).

`ftol` : float Tolerance ftol of scipy.optimize.minimize (0.05 by default).

`results_folder_name` : str String for the prefix of the folder where to save results; the complete folder name is `results_folder_name + '_' + time` where time is the current time when the algorithm has finished, in order to uniquely identify the folder with the results.

`n_parallel_jobs` : int How many jobs are run in parallel (None by default).

`id_code` : None or str Identificative code (suffix) of the folder name where output data (result) will be saved. If None, then the current date will be used.

Function `compute_chi2_test`

```
def compute_chi2_test(
    data_test,
    regularization,
    pars_ff: jax.Array = None,
    pars_fm: jax.Array = None,
    lambdas: dict = None,
    which_set: str = 'validation'
)
```

Compute the chi2 on the test set, after the optimal solution has been found.

Parameters ——= data_test : object Object for the test dataset, as returned by split_dataset. pars_ff : np.ndarray Numpy 1d array for the force-field correction parameters. pars_fm : np.ndarray Numpy 1d array for the forward model parameters. lambdas : dict Dictionary for the lambda coefficients (of ensemble refinement). which_set : str String variable, if `'validation'` compute the chi2 on validation observables and validation frames.

Return ——= red_chi2 : float Reduced chi2 (chi2 / n. of observables)

Function `save_txt`

```
def save_txt(
    input_values,
    Result,
    coeff_names,
    folder_name='Result',
    id_code: str = None
)
```

This is an internal tool of MDRefinement() used to save input_values and output Result as csv and npy files in a folder whose name is `folder_name + '_' + date` where date is the current time when the computation ended (it uses date_time to generate unique file name, on the assumption of a single folder name at given time).

Parameters

`input_values` : dict Dictionary with input values of the refinement, such as stride, starting values of the hyperparameters, random_states, which_set, tolerances (see MDRefinement()).

`Result` : class instance Class instance with the results of minimizer and the search for the optimal hyperparameters.

`coeff_names` : list List with the names of the coefficients (force-field and forward-model corrections).

`folder_name` : str String for the prefix of the folder name (by default, `'Result'`).

`id_code` : None or str Identificative code (suffix) of the folder name where output data (result) will be saved. If None, then the current date will be used.

Function `unwrap_2dict`

```
def unwrap_2dict(
    my_2dict
)
```

Tool to unwrap a 2-layer dictionary my_2dict into list of values and list of keys.

Function `unwrap_dict`

```
def unwrap_dict(
    d
)
```

Module `MDRefine.bayesian`

Tools for the sampling of the posterior distribution, defined over a set of ensembles, by using a suitable uninformative prior (namely, a prescription on the counting of the ensembles).

Functions

Function `block_analysis`

```
def block_analysis(
    x,
    size_blocks=None,
    n_conv=50
)
```

This function performs the block analysis of a (correlated) time series x, cycling over different block sizes. It includes also a numerical search of the optimal estimated error epsilon, by smoothing epsilon and searching for the first time it decreases, which should correspond to a plateau region.

It returns an instance of the Block_analysis_Result class.

Parameters

x : numpy.ndarray Numpy array with the time series of which you do block analysis.

size_blocks : list, int or None The list with the block sizes used in the analysis; you can either pass an integer value, in this case the list of sizes is given by `np.arange(1, np.int64(size/2) + size_blocks, size_blocks)`; further, if size_blocks is None, the list of sizes is `np.arange(1, np.int64(size/2) + 1, 1)`.

n_conv : int Length (as number of elements in the block-size list) of the kernel used to smooth the epsilon function (estimated error vs. block size) in order to search for the optimal epsilon, corresponding to the plateau.

Function `energy_fun`

```
def energy_fun(
    x,
    data,
    regularization,
    alpha=inf,
    beta=inf,
    which_measure='uniform'
)
```

This is the energy function defined for running the usual sampling algorithms, corresponding to -log of the posterior distribution (a part from a normalization factor and with the optional inclusion of the entropic contribution, as prescribed by which_measure). Depending on which hyperparameter is infinite (alpha or beta), it corresponds either to ensemble refinement or force-field fitting.

Parameters

x : numpy.ndarray Numpy array with the lambda coefficients (for ensemble refinement) or the force-field correction coefficients (for force-field refinement), in the same order required by loss_and_minimizer.loss_function.

data : data_loading.my_data An instance of the class data_loading.my_data class, with all the data for the molecules of interest.

regularization : dict Dictionary for the regularization (None for ensemble refinement), as described for MDRefinement.

alpha, beta : float Values of the hyperparameters alpha (ensemble refinement) or beta (force-field fitting): either one of them must be infinite (the sampling has been implemented either for ensemble or force-field refinement).

which_measure : dict Dictionary indicating the measure used for sampling the posterior (choose among: `'uniform'`, `'jeffreys'`, `'average'`, `'dirichlet'`).

Returns

energy : float Float value for the energy used in the sampling, as defined by the input variables.

qs : MyQuantities An instance of the `'MyQuantities` class containing loss, average observables and regularization values.

Function `langevin_sampling`

```
def langevin_sampling(
    energy_fun,
    starting_x,
    n_iter: int = 10000,
    gamma: float = 0.1,
    dt: float = 0.005,
    kT: float = 1.0,
    seed: int = 1,
    if_tqdm: bool = True
```

)

A function to perform a Langevin sampling of energy_fun() at temperature kT (with the Euler-Maruyama scheme).

Parameters

energy_fun : function The energy function, written with jax.numpy in order to do automatic differentiation through jax.grad (this requires energy_fun() to return a scalar value and not an array, otherwise you should use jax.jacfwd for example; to this aim, you can do jnp.sum(energy_fun()(x))).
starting_x : numpy.ndarray The starting configuration of the Langevin sampling.
n_iter : int Number of iterations.
gamma : float Friction coefficient.
dt : float Time step.
kT : float The temperature.
seed : int Integer value for the seed.
if_tqdm : Bool Boolean variable, if True use tqdm (default choice).

Returns

traj : np.ndarray Numpy array with the trajectory.
ene : np.ndarray Numpy array with the energies.
force_list : list List with the forces.
check : dict Dictionary with `'dif'` for np.ediff1d(traj), together with its mean and standard deviation.

Function `local_density`

```
def local_density(
    variab,
    weights,
    which_measure='jeffreys'
)
```

This function computes the local density of ensembles in the cases of ensemble refinement or force-field fitting.

This density can be defined through the Jeffreys "uninformative" prior (`which_measure = 'jeffreys'`): in these two cases, the Jeffreys prior is given by the square root of the determinant of the covariance matrix (of the observables in Ensemble Refinement or the generalized forces in Force-Field Fitting, where the generalized forces are the derivatives of the force-field correction with respect to the fitting coefficients).

It includes also the possibility for the computation of the local density of ensembles with plain Dirichlet if `which_measure = 'dirichlet'`, or with the variation of the average observables if `which_measure = 'average'`.

Since we are anyway dealing with a real-value, symmetric and semi-positive definite matrix, its determinant is computed through the Cholesky decomposition (which is faster for big matrices): triang is such that `metric = triang * triang.T`, so `sqrt(det metric) = det(triang)`.

Parameters

variab : numpy.ndarray, dict or tuple For Ensemble Refinement, variab is either the dictionary data.mol[name_mol].g to be unwrapped or directly the numpy array with the observables defined in each frame.

For Force-Field Fitting and `which_measure == 'jeffreys' or 'dirichlet'`, variab is the tuple (fun_forces, pars, f) where: - fun_forces is the function for the gradient of the force-field correction with respect to pars (defined through Jax as `fun_forces = jax.jacfwd(ff_correction, argnums=0)` where `ff_correction = data.mol[name_mol].ff_correction`; you can compute it just once at the beginning of the MC sampling); - pars is the numpy.ndarray of parameters for the force-field correction; - f is the numpy.ndarray data.mol[name_mol].f with the terms required to compute the force-field correction. If `which_measure = 'average'`, then the observables are required, too, and variab is the tuple (fun_forces, pars, f, g).

See documentation of MDRefine at https://www.bussilab.org/doc-MDRefine/MDRefine/index.html for further details about the data object.

weights : numpy.ndarray Numpy array with the normalized weights of each frame; this is the probability distribution at which you want to compute the Jeffreys prior, corresponding to the local density of ensembles.

which_measure : str String variable, chosen among: jeffreys, dirichlet or average, indicating the prescription for the local density of ensembles (Jeffreys prior, plain Dirichlet, average observables).

Returns

measure : float The local density of ensembles at the given distribution weights, computed as specified by which_measure (Jeffreys prior by default).

cov : numpy.ndarray The metric tensor for the chosen metrics defined by which_measure if `which_measure = 'jeffreys'` or `'dirichlet'`; the covariance matrix if `which_measure = 'average'`.

Function `posterior_sampling`

```
def posterior_sampling(
    starting_point,
    data,
    regularization=None,
    alpha: float = inf,
    beta: float = inf,
    which_measure=MDRefine.bayesian.Which_measure,
    proposal_move='default',
    n_steps_MC: int = 10000,
    seed: int = 1
)
```

Main function of the bayesian module, it is the algorithm that samples from the posterior distribution exp(-L(P)) with the specified uninformative prior, either in the case of ensemble refinement or force-field refinement.

Parameters

starting_point

data : data_loading.my_data An instance of the class data_loading.my_data class, with all the data for the molecules of interest.

regularization : dict Dictionary for the regularization to the force-field correction.

alpha, beta : float Float values for the hyperparameters (either one of them must be infinite or None).

which_measure : Which_measure An instance of the Which_measure class, to specify the entropic measure used in the sampling (chosen among `FLAT = 'uniform'`, `JEFFREYS = 'jeffreys'`, `AVERAGE = 'average'`, `DIRICHLET = 'dirichlet'`).

proposal_move : str or function or float or tuple Variable used to specify the move employed in the Metropolis algorithm, as indicated in run_Metropolis(); if it is `'default'`, then a Gaussian move is used with standard deviation `proposal_move = 0.1`.

n_steps_MC : int Integer for the number of steps in the Metropolis algorithm.

seed : int Integer for the random state (seed) used in the Metropolis algorithm.

Returns

sampling : Result_MyQuantities An instance of the Result_MyQuantities class, which merges the quantities returned by each step of the MCMC sampling (as indicated in MyQuantities).

Function `run_Metropolis`

```
def run_Metropolis(
```

```
    x0,
    proposal,
    energy_function,
    quantity_function=None,
    *,
    kT=1.0,
    n_steps=100,
    seed=1,
    i_print=10000,
    if_tqdm=True,
    saving=None
)
```

This function runs a Metropolis sampling algorithm.

Parameters

`x0` : numpy.ndarray Numpy array for the initial configuration.

`proposal` : function or float or tuple Function for the proposal move, which takes as input variables just the starting configuration x0 and returns the new proposed configuration (trial move of Metropolis algorithm). Alternatively, float variable for the standard deviation of a (zero-mean) multi-variate Gaussian variable representing the proposed step (namely, the stride). Another possibility is the tuple `('one-by-one', step)` where step is a float or int variable; in this case, the proposal is done on each coordinate one at a time, following a cycle.

`energy_function` : function Function for the energy, which takes as input variables just a configuration (x0 for instance) and returns its energy; energy_function can return also some quantities of interest, defined on the input configuration. If your energy function energy_fun() has more than one input variables, just redefine it as `energy_function = lambda x : energy_fun(x, simple_model, 'dirichlet')` before passing energy_function to run_Metropolis().

`quantity_function` : function Function used to compute some quantities of interest on the initial configuration. If energy_function has more than one output, quantity_function is ignored and the quantities of interest are the 2nd output of energy_function (in this way, they are computed together with the energy, avoiding the need for running twice the same function). Notice that quantity_function does not support other input parameters beyond the configuration; otherwise, you can use energy_function.

`kT` : float Temperature of the Metropolis sampling algorithm.

`n_steps` : int Number of steps of Metropolis.

`seed` : int Seed for the random generation.

`i_print` : int How many steps to print an indicator of the running algorithm (current n. of steps).

`if_tqdm` : Bool Boolean variable, if True then use tqdm.

`saving` : None or float or Saving_function An instance of the Saving_function class, used to save the results during Metropolis run (or in the end). If saving is None do not save, if it is `'yes'` use default object of class Saving_function.

Returns

`obj_result` : Result_run_Metropolis An instance of the Result_run_Metropolis class with trajectory, energy, average acceptance and computed quantities.

Classes

Class `Block_analysis_Result`

```
class Block_analysis_Result(
    mean: float,
    std: float,
    opt_epsilon: float,
    epsilons: numpy.ndarray,
    epsilons_smooth: numpy.ndarray,
    n_blocks: numpy.ndarray,
```

```
        size_blocks: numpy.ndarray
    )
```

Result of a block_analysis() calculation.

Ancestors (in MRO)

- MDRefine.bayesian.Result
- builtins.dict

Instance variables

Variable `epsilons` list with the associated error epsilon for each block size.

Variable `epsilons_smooth` list with the associated error epsilon for each block size (smooth time series).

Variable `mean` float with the mean value of the time series.

Variable `n_blocks` list with the number of blocks in the time series, for each analysed block size.

Variable `opt_epsilon` float with the optimal estimate of the associated error epsilon.

Variable `size_blocks` list with the block sizes initially defined.

Variable `std` float with the standard deviation of the time series (assuming independent frames).

Class `MyQuantities`

```
    class MyQuantities(
        loss,
        reg,
        avs
    )
```

Class with the evaluated quantities for each step of the MCMC sampling, beyond energy and trajectory.

Instance variables

Variable `avs` float with the average values.

Variable `loss` float with the loss value (excluding the entropic contribution).

Variable `reg` float with the regularization value.

Static methods

`Method merge`

```
    def merge(
        instances
    )
```

Function to merge multiple instances of MyQuantities in a single one (to be run in the end of the MCMC sampling to collect quantities).

Class `Proposal_onebyone`

```
class Proposal_onebyone(
    step_width=1.0,
    index=0,
    rng=None
)
```

Class for a proposal move which updates one coordinate per time (it includes the attribute index to take in memory which coordinate to update)

Class `Result`

```
class Result(
    *args,
    **kwargs
)
```

Base class for objects returning results.

It allows one to create a return type that is similar to those created by scipy.optimize.minimize. The string representation of such an object contains a list of attributes and values and is easy to visualize on notebooks.

Examples The simplest usage is this one:

```
from bussilab import coretools

class MytoolResult(coretools.Result):
    """Result of a mytool calculation."""
    pass

def mytool():
    a = 3
    b = "ciao"
    return MytoolResult(a=a, b=b)

m=mytool()
print(m)
```

Notice that the class variables are dynamic: any keyword argument provided in the class constructor will be processed. If you want to enforce the class attributes you should add an explicit constructor. This will also allow you to add pdoc docstrings. The recommended usage is thus:

```
from bussilab import coretools

class MytoolResult(coretools.Result):
    """Result of a mytool calculation."""
    def __init__(a, b):
        super().__init__()
        self.a = a
        """Documentation for attribute a."""
        self.b = b
        """Documentation for attribute b."""

def mytool():
    a = 3
    b = "ciao"
    return MytoolResult(a=a, b=b)

m = mytool()
print(m)
```

Ancestors (in MRO)

- builtins.dict

Descendants

- MDRefine.bayesian.Block_analysis_Result
- MDRefine.bayesian.Result_MyQuantities
- MDRefine.bayesian.Result_run_Metropolis

Class `Result_MyQuantities`

```
class Result_MyQuantities(
    my_quantities_concat: MDRefine.bayesian.MyQuantities
)
```

Class with the merged quantities from MyQuantities (MyQuantities.merge()).

Ancestors (in MRO)

- MDRefine.bayesian.Result
- builtins.dict

Class `Result_run_Metropolis`

```
class Result_run_Metropolis(
    traj,
    ene,
    av_acceptance,
    quantities=None
)
```

Result of a run_Metropolis() calculation.

Ancestors (in MRO)

- MDRefine.bayesian.Result
- builtins.dict

Instance variables

Variable `av_acceptance` Float value for the average acceptance

Variable `ene` Energy

Variable `traj` Trajectory

Class `Saving_function`

```
class Saving_function(
    values: dict = {},
    t0: float = 0.0,
    date: str = '',
    path: str = '.',
    i_save: int = 10000
)
```

Class `Which_measure`

```
class Which_measure(
    *args,
    **kwds
)
```

Class with the strings for which_measure variable.

Ancestors (in MRO)

- enum.Enum

Class variables

Variable `AVERAGE`

Variable `DIRICHLET`

Variable `FLAT`

Variable `JEFFREYS`

Module `MDRefine.data_loading`

Tools n. 1: data_loading. It loads data into the data object.

Functions

Function `check_and_skip`

```
def check_and_skip(
    data,
    *,
    stride=1
)
```

This function is an internal tool used in load_data() to modify input data:

- weights are normalized;
- it appends observables computed through forward models (if any) to data.mol[name_sys].g;
- if `hasattr(data.mol[name_sys], 'selected_obs')`: it removes non-selected observables from data.mol[name_sys].forward_qs;
- select frames with given stride;
- count n. experiments and n. frames (data.mol[name_sys].n_frames and data.mol[name_sys].n_experiments) and check corresponding matching.

Function `load_data`

```
def load_data(
    infos,
    *,
    stride=1
)
```

This tool loads data from specified directory as indicated by the user in infos to a dictionary data of classes, which includes data.properties (global properties) and data[system_name]; for alchemical calculations, there is also data[cycle_name].

Classes

Class `data_class`

```
class data_class(
    info,
    path_directory,
    name_sys
)
```

Data object of a molecular system.

Parameters

`info` : dict Dictionary for the information about the data of name_sys molecular system in path_directory.

`path_directory` : str String for the path of the directory with data of the molecular system name_sys.

`name_sys` : str Name of the molecular system taken into account.

Returns

`temperature` : float Value for the temperature at which the trajectory is simulated.

`gexp` : dict Dictionary of Numpy 2-dimensional arrays (N x 2); gexp[j,0] is the experimental value of the j-th observable, gexp[j,1] is the corresponding uncertainty; the size N depends on the type of observable.

`names` : dict Dictionary of Numpy 1-dimensional arrays of length N with the names of the observables of each type.

`ref` : dict Dictionary of strings with signs ''=', '>', '<', '><' used to define the chi2 to compute, depending on the observable type.

`g` : dict Dictionary of Numpy 2-dimensional arrays (M x N), where g[name][i,j] is the j-th observable of that type computed in the i-th frame.

`forward_qs` : dict Dictionary of Numpy 2-dimensional arrays (M x N) with the quantities required for the forward model.

`forward_model` : function Function for the forward model, whose input variables are the forward-model coefficients fm_coeffs and the forward_qs dictionary; a third optional argument is the selected_obs (dictionary with indices of selected observables).

`weights` : array_like Numpy 1-dimensional array of length M with the weights (not required to be normalized).

`f` : array_like Numpy 2-dimensional array (M x P) of terms required to compute the force-field correction, where P is the n. of parameters pars and M is the n. of frames.

`ff_correction` : function Function for the force-field correction, whose input variables are the force-field correction parameters pars and the f array (sorted consistently with each other).

Class `data_cycle_class`

```
class data_cycle_class(
    cycle_name,
    DDGs_exp,
    info
)
```

Data object of a thermodynamic cycle.

Parameters

`cycle_name` : str String with the name of the thermodynamic cycle taken into account.

`DDGs_exp` : pandas.DataFrame Pandas.DataFrame with the experimental values and uncertainties of Delta Delta G in labelled thermodynamic cycles.

`info` : dict Dictionary for the information about the temperature of cycle_name thermodynamic cycle.

####
Returns

####
Returns
system_names
: list : List
of names of
the
investigated
molecular
systems.
forward_coeffs_0
: list : List
of the
forward-
model
coefficients.
names_ff_pars
: list : List
of names of
the
force-field
correction
parameters.
cycle_names
: list : List
of names of
the
investigated
thermody-
namic
cycles.
####
Methods
#####
Method
`tot_n_experiments`
{#MDRefine.data_loading.datapropertiesclass.tot_n_experiments}
> def
tot_n_experiments(
> self, >
data >)
This method
computes
the total
n. of
experiments.
Class
`my_data`
{#MDRefine.data_loading.my_data}
> class
my_data(>
infos >)
Module
`MDRefine.hyperminimizer`
{#MDRefine.hyperminimizer}

Returns

Tools n. 3: hyperminimizer. It performs the automatic search for the optimal hyperparameters.

Functions

Function `compute_chi2_tot`

{#MDRefine.hyperminimizer.compute_chi2_tot}

```
> def compute_chi2_tot(
>
pars_ff_fm,
> lambdas,
> data, >
regularization, >
alpha, >
beta, >
gamma, >
which_set,
>
data_train
> )
```

####
Returns

This function is an internal tool used in compute_hypergradient() and hyper_minimizer() to compute the total chi2 (float variable) for training or validation data set and its derivatives (with respect to pars_ff_fm and lambdas). The choice of the data set is indicated by which_set (`which_set = 'training'` for chi2 on the training set, `'valid_frames'` for chi2 on training observables and validation frames, `'validation'` for chi2 on validation observables and validation frames, through validation function).

######
Parameters

####
Returns

pars_ff_fm, **lambdas** : array_like : Numpy arrays for (force-field + forward-model) parameters and lambdas parameters, respectively.

data : dict : Dictionary of data set object.

regularization : dict : Specified regularizations of force-field and forward-model corrections (see in MDRefinement).

alpha, **beta**, **gamma** : float : Values of the hyperparameters.

which_set : str : String variable, chosen among `'training'`, `'valid_frames'`, `'valid_obs'` or `'validation'` as explained above.

####
Returns

`data_train`
: dict :
Dictionary
of training
dataset
objects,
required if
`which_set`
`=`
`'valid_obs'`
to compute
the chi2 on
validating
obvervables
including
also training
frames.
###
Function
`compute_hyperderivatives`
{#MDRefine.hyperminimizer.compute_hyperderivatives}
> def com-
pute_hyperderivatives(
>
pars_ff_fm,
> lambdas,
> data, >
regulariza-
tion, >
deriva-
tives_funs,
>
log10_alpha=inf,
>
log10_beta=inf,
>
log10_gamma=inf
>)

####
Returns
This is an internal tool of compute_hypergradient() which computes the derivatives of parameters with respect to hyperparameters, which are going to be used later to compute the derivatives of chi2 w.r.t. hyperparameters. It returns an instance of the class derivatives, which includes as attributes the numerical values of the derivatives dlambdas_dlogalpha, dlambdas_dpars, dpars_dlogalpha, dpars_dlogbeta, dpars_dloggamma.
######
Parameters
pars_ff_fm : array_like : Numpy array for force-field and forward-model coefficients.

####
Returns

---

lambdas
: array_like
: Numpy
array for
lambdas
coefficients
(those for
ensemble
refinement).
data : dict :
The data
object.
regularization
: dict : The
regulariza-
tion of
force-field
and forward-
model
corrections
(see in
MDRefine-
ment).
derivatives_funs
: class
instance :
Instance of
the deriva-
tives_funs_class
class of
derivatives
functions
computed by
Jax.
log10_alpha,
log10_beta,
log10_gamma
: floats :
Logarithms
(in base 10)
of the corre-
sponding
hyperparam-
eters alpha,
beta,
gamma
(np.inf by
default).
###
Function
`compute_hypergradient`
{#MDRefine.hyperminimizer.compute_hypergradient}

####
Returns

> def com-
pute_hypergradient(
>
pars_ff_fm,
> lambdas,
>
log10_alpha,
>
log10_beta,
>
log10_gamma,
>
data_train,
> regulariza-
tion, >
which_set,
>
data_valid,
> deriva-
tives_funs >
)
This is an
internal tool
of
mini_and_chi2_and_grad(),
which
employs
previously
defined
functions
(com-
pute_hyperderivatives(),
com-
pute_chi2_tot(),
put_together())
to return
selected chi2
and its
gradient
w.r.t hyper-
parameters.
######
Parameters
pars_ff_fm
: array_like
: Numpy
array of
(force-field
and forward-
model)
parameters.

####
Returns

lambdas : dict : Dictionary of dictionaries with lambda coefficients (corresponding to Ensemble Refinement).
log10_alpha, **log10_beta**, **log10_gamma** : floats : Logarithms (in base 10) of the hyperparameters alpha, beta, gamma.
data_train : class instance : The training data set object, which is anyway required to compute the derivatives of parameters w.r.t. hyperparameters.
regularization : dict : Specified regularizations (see in MDRefinement).
which_set : str : String indicating which set defines the chi2 to minimize in order to get the optimal hyperparameters (see in compute_chi2_tot()).

####
Returns

---

data_valid
: class instance : The validation data set object, which is required to compute the chi2 on the validation set (when **which_set** **==** **'valid_frames'** **or** **'validation'**; otherwise, if **which_set** **=** **'training'**, it is useless, so it can be set to None).
derivatives_funs
: class instance : Instance of the derivatives_funs_class class of derivatives functions computed by Jax Autodiff (they include those employed in compute_hyperderivatives() and dchi2_dpars and/or dchi2_dlambdas). If None (default value), do not compute the derivatives of chi2.

###
Function
hyper_function
{#MDRefine.hyperminimizer.hyper_function}

####
Returns

> def hyper_function(
>
log10_hyperpars,
>
map_hyperpars,
> data, >
regularization, >
valid_obs, >
valid_frames,
> which_set,
> derivatives_funs,
> starting_pars, >
n_parallel_jobs
>)
This
function is
an internal
tool of hyper_minimizer()
which
determines
the optimal
parameters
by
minimizing
the loss
function at
given hyper-
parameters;
then, it
computes
chi2 and its
gradient
w.r.t hyper-
parameters
(for the
optimal
parameters).
######
Parameters
log10_hyperpars
: array_like
: Numpy
array for
log10 hyper-
parameters
alpha, beta,
gamma (in
this order,
when
present).

####
Returns

---

map_hyperpars : list : Legend for log10_hyperpars (they refer to alpha, beta, gamma in this order, but some of them may not be present, if fixed to `+np.inf`).

data : class instance : Class instance for data object.

regularization : dict : Dictionaries for regularization object.

valid_obs, **valid_frames** : dicts : Dictionaries for validation observables and validation frames, indicized by seeds.

which_set : str : String, see for compute_chi2_tot().

derivatives_funs : class instance : Derivative functions computed by Jax and employed in compute_hypergradient().

####
Returns

starting_pars : float : Starting values of the parameters, if user-defined; None otherwise.
n_parallel_jobs : int : Number of parallel jobs.

Returns

tot_chi2 : float Float value of total chi2.
tot_gradient : array_like Numpy array for gradient of total chi2 with respect to the hyperparameters.
Results : class instance Results given by minimizer.

Global variable: hyper_intermediate, in order to follow steps of minimization.

######
Returns
out : class instance : Class instance whose attributes can include dchi2_dlogalpha, dchi2_dlogbeta, dchi2_dloggamma, depending on which hyperparameters are not fixed to +np.inf.
Module MDRefine.loss_and_minimizer {#MDRefine.loss_and_minimizer}

Global variable: hyper_intermediate, in order to follow steps of minimization.

Tools n. 2: loss_and_minimizer. It defines the loss functions and minimizes it. It includes also split_dataset() and validation().

Functions

Function `compute_D_KL` {#MDRefine.loss_and_minimizer.compute_D_KL}

> def compute_D_KL(
>
weights_P: numpy.ndarray,
> correction_ff: numpy.ndarray,
> temperature: float,
>
logZ_P: float
>)

Global variable: hyper_intermediate, in order to follow steps of minimization.

This tool computes the Kullback-Leibler divergence of P(x) = 1/Z P_0 (x) e^(-V(x)/T) with respect to P_0 as `av(V)/T + log Z` where av(V) is the average value of the potential V(x) over P(x).
######
Parameters
`weights_P` : `1-D array-like` : Numpy 1-dimensional array for the normalized weights P(x).
`correction_ff` : `1-D array-like` : Numpy 1-dimensional array for the reweighting potential V(x).

Global variable: hyper_intermediate, in order to follow steps of minimization.

temperature : float : The value of temperature T, in measure units consistently with V(x), namely, such that V(x)/T is adimensional.

logZ_P : float : The value of log Z.

Function compute_DeltaDeltaG_terms

{#MDRefine.loss_and_minimizer.compute_DeltaDeltaG_terms}

> def compute_DeltaDeltaG_terms(
> data, > logZ_P >
)

This tool computes the chi2 for Delta Delta G (free-energy differences from thermodynamic cycles), contributing to the loss function with alchemical calculations.

Parameters

Global variable: hyper_intermediate, in order to follow steps of minimization.

`data` : class instance : Object data; here, data.properties has the attribute cycle_names (list of names of the thermodynamic cycles); for s in data.properties.cycle_names: data.cycle[s] has attributes temperature (of the cycle) and gexp_DDG; for s in my_list (where my_list is the list of system names associated to a thermodynamic cycle `my_list = [x2 for x in list(data.properties.cycle_names.values()) for x2 in x]`): data.mol[s] has attributes temperature (of the system) and logZ.

Global variable: hyper_intermediate, in order to follow steps of minimization.

logZ_P : dict : Dictionary for logarithm of the partition function Z_P, namely, average value of `exp(-V_phi(x)/temperature)` ###### over the original ensemble; its keys are the selected system_names. ######

Returns

new_av_DG : dict : Dictionary of reweighted averages of Delta G.

chi2 : dict : Dictionary of chi2 (one for each thermodynamic cycle).

Global variable: hyper_intermediate, in order to follow steps of minimization.

`loss` : float : Total contribution to the loss function from free-energy differences Delta Delta G, given by 1/2 of the total chi2.

Function `compute_chi2` {#MDRefine.loss_and_minimizer.compute_chi2}

> def compute_chi2(> ref, > weights, > g, > gexp, > if_separate=False >)

This tool computes the chi2 (for a given molecular system: the input dictionaries are structured as the attributes of data.mol[mol_name]).

Parameters

Global variable: hyper_intermediate, in order to follow steps of minimization.

ref : dict : Dictionary for references (=, >, <, ><) used to compute the appropriate chi2.

weights : 1-D array-like : Numpy 1-dimensional array of weights.

g : dict : Dictionary of observables specific for the given molecular system.

gexp : dict : Dictionary of experimental values specific for the given molecular system (coherently with g).

Global variable: hyper_intermediate, in order to follow steps of minimization.

`if_separate` : bool : Boolean variable, True if you are distinguishing between LOWER and UPPER bounds (`name_type + 'LOWER'` ###### or `name_type + 'UPPER'`), needed for minimizations with double bounds.

######

Returns `This tool returns 4 variables:` 3 `dictionaries (with keys running over different kinds of observables)` `and 1 float:` : `av_g` : dict : Dictionary of average values of the observables g.

Global variable: hyper_intermediate, in order to follow steps of minimization.

`chi2` : dict : Dictionary of chi2.

`rel_diffs` : dict : Dicionary of relative differences.

`tot_chi2` : float : Total chi2 for the given molecular system.

Function `compute_details_ER`

{#MDRefine.loss_and_minimizer.compute_details_ER} > def compute_details_ER(> weights_P, > g, > data, > lambdas, > alpha >)

Global variable: hyper_intermediate, in order to follow steps of minimization.
This is an internal tool of loss_function() which computes explicitely the contribution to the loss function due to Ensemble Refinement (namely, 1/2 chi2 + alpha D_KL) and compare this value with -alpha*Gamma (they are equal in the minimum: check). It cycles over different systems. It acts after the minimization of the loss function inside loss_function() (not for the minimization itself, since we exploit the Gamma function).

Global variable: hyper_intermediate, in order to follow steps of minimization.

Be careful to use either: normalized values for lambdas and g (if `hasattr(data.mol[name_mol],'normg_mean')`) or non-normalized ones (if `not hasattr(data.mol[name_mol],'normg_mean')`).
######
Parameters
`weights_P` : dict : Dictionary of Numpy arrays, namely, the weights on which Ensemble Refinement acts (those with force-field correction in the fully combined refinement).
`g` : dict : Dictionary of dictionaries, like for data.mol[name_mol].g, corresponding to the observables (computed with updated forward-model coefficients).

Global variable: hyper_intermediate, in order to follow steps of minimization.

`data` : dict : The original data object.
`lambdas` : dict : Dictionary of Numpy arrays, corresponding to the coefficients for Ensemble Refinement.
`alpha` : float : The alpha hyperparameter, for Ensemble Refinement.
###
Function `compute_js`
{#MDRefine.loss_and_minimizer.compute_js}
> def compute_js(>
n_experiments
>)

Global variable: hyper_intermediate, in order to follow steps of minimization.

This tool computes the indices js (defined by cumulative sums) for lambdas corresponding to different molecular systems and types of observables. Be careful to follow always the same order: let's choose it as that of data.n_experiments, which is a dictionary n_experiments[name_mol][name].

###

Function `compute_new_weights` {#MDRefine.loss_and_minimizer.compute_new_weights}

> def compute_new_weights(
>
weights: numpy.ndarray,
> correction: numpy.ndarray
>)

Global variable: hyper_intermediate, in order to follow steps of minimization.

This tool computes the new weights as weights*exp(-correction). It modifies Parameters weights are normalized and correction is shifted by `correction -= shift`, where `shift = np.min(correction)`. It returns two variables: a Numpy array new_weights and a float logZ.

###

Function `deconvolve_lambdas` {#MDRefine.loss_and_minimizer.deconvolve_lambdas}

> def deconvolve_lambdas(
> data, > lambdas: numpy.ndarray,
> if_denormalize: bool = True
>)

Global variable: hyper_intermediate, in order to follow steps of minimization.

This tool deconvolves lambdas from Numpy array to dictionary of dictionaries (corresponding to data.mol[name_mol].g); if if_denormalize, then lambdas has been computed with normalized data, so use data.mol[name_mol].normg_std and data.mol[name_mol].normg_mean in order to go back to corresponding lambdas for non-normalized data. The order of lambdas is the one described in compute_js().
###
Function `gamma_function` {#MDRefine.loss_and_minimizer.gamma_function}

Global
variable:
hy-
per_intermediate,
in order to
follow steps
of mini-
mization.

> def
gamma_function(
> lamb-
das: numpy.ndarray,
>
g: numpy.ndarray,
>
gexp: numpy.ndarray,
>
weights: numpy.ndarray,
>
alpha: float,
>
if_gradient: bool = False
>)
This tool
computes
gamma
function
and (if
if_gradient)
its
derivatives
and the
average
values of
the
observables
av_g.
Make sure
that
lambdas
follow the
same order
as g, gexp
(let's use
that of
data.n_experiments).
######
Parameters

Global variable: hyper_intermediate, in order to follow steps of minimization.

lambdas : 1-D array-like : Numpy 1-dimensional array of length N, where lambdas[j] is the lambda value for the j-th observable.

g : 2-D array-like : Numpy 2-dimensional array (M x N); g[i,j] is the j-th observable computed in the i-th frame.

gexp : 2-D array-like : Numpy 2-dimensional array (N x 2); gexp[j,0] is the experimental value of the j-th observable, gexp[j,1] is the associated experimental uncertainty.

Global variable: hyper_intermediate, in order to follow steps of minimization.

`weights` : `1-D array-like` : Numpy 1-dimensional array of length M; w[i] is the weight of the i-th frame (possibly non-normalized).

`alpha` : float : The value of the alpha hyperparameter.

`if_gradient` : bool : If true, return also the gradient of the gamma function.

###

Function `l2_regularization`

{#MDRefine.loss_and_minimizer.l2_regularization}

> def l2_regularization(

>

pars: numpy.ndarray,

>

choice: str = 'plain l2' >)

Global variable: hyper_intermediate, in order to follow steps of minimization.
This tool computes the L2 regularization for the force-field correction coefficients pars as specified by choice. It includes: - `'plain l2'` (plain L2 regularization of pars);

Global variable: hyper_intermediate, in order to follow steps of minimization.

- L2 regularization for alchemical calculations with charges (as described by Valerio Piomponi et al., see main paper): `pars[:-1]` are the charges and `pars[-1]` is V_eta; there is the constraint on the total charge, and there are 3 pars[4] charges in the molecule; so, `'constraint 1'` is the L2 regularization on charges, while `'constraint 2'` is the L2 regularization on charges and on V_eta. Output values: lossf_reg and gradient (floats).

Global
variable:
hy-
per_intermediate,
in order to
follow steps
of mini-
mization.

###
Function
`loss_function`
{#MDRefine.loss_and_minimizer.loss_function}
> def
loss_function(
>
pars_ff_fm: numpy.ndarray,
>
data: dict,
>
regulariza-
tion: dict,
> al-
pha: float = inf,
>
beta: float = inf,
>
gamma: float = inf,
>
fixed_lambdas: numpy.ndarray = None,
>
gtol_inn: float = 0.001,
>
if_save: bool = False,
>
bounds: dict = None
>)
This tool
computes
the fully-
combined
loss
function
(to
minimize),
taking
advantage
of the inner
minimiza-
tion with
Gamma
function.
If not
np.isinf(alpha):

Global variable: hyper_intermediate, in order to follow steps of minimization.
- if `fixed_lambdas == None`, then do the inner minimization of Gamma (in this case, you have the global variable lambdas, corresponding to the starting point of the minimization; it is a Numpy array sorted as in compute_js()).

Global variable: hyper_intermediate, in order to follow steps of minimization.

- else: lambdas is fixed (fixed_lambdas is not None) and the Gamma function is evaluated at this value of lambda, which must correspond to its point of minimum, otherwise there is a mismatch between the Gamma function and the Ensemble Refinement loss. The order followed for lambdas is the one of compute_js(), which is not modified in any step.

Global variable: hyper_intermediate, in order to follow steps of minimization.

If if_save: loss_function() returns Details class instance with the detailed results; otherwise, it returns just the loss value. The input data are not modified by loss_function() (neither explicitely by loss_function() nor by its inner functions): for forward-model updating, loss_function() defines a new variable g (through copy.deepcopy). ###### Parameters

Global variable: hyper_intermediate, in order to follow steps of minimization.

pars_ff_fm : 1-D array-like : Numpy 1-dimensional array with parameters for force-field corrections and/or forward models. These parameters are sorted as: first force-field correction (ff), then forward model (fm); order for ff: `names_ff_pars = []; for k in system_names: [names_ff_pars.append(x) for x in data[k].f.keys() if x not in names_ff_pars];` order for fm: the same as data.forward_coeffs_0.

data : dict : Dictionary of class instances as organised in load_data, which constitutes the data object.

Global variable: hyper_intermediate, in order to follow steps of minimization.

regularization : dict : Dictionary for the force-field and forward-model correction regularizations (see MDRefinement).

alpha, **beta**, **gamma** : floats : The hyperparameters of the three refinements (respectively, to: the ensemble, the force-field, the forward-model); (`+np.inf` by default, namely no refinement in that direction).

Global variable: hyper_intermediate, in order to follow steps of minimization.

fixed_lambdas : 1-D array-like, optional : Numpy 1-dimensional array of fixed values of lambdas (coefficients for Ensemble Refinement, organized as in compute_js()). (None by default).

gtol_inn : float : Tolerance gtol for the inner minimization of Gamma function (1e-3 by default).

if_save : bool : Boolean variable (False by default).

bounds : dict : Dictionary of boundaries for the inner minimization (None by default).

Function loss_function_and_grad {#MDRefine.loss_and_minimizer.loss_function_and_grad}

Global variable: hyper_intermediate, in order to follow steps of minimization.

```
> def loss_function_and_grad(
>
pars: numpy.ndarray,
>
data: dict,
>
regularization: dict,
>
alpha: float,
>
beta: float,
>
gamma: float,
>
gtol_inn: float,
> boundaries: dict,
> gradient_fun, >
if_print: bool = False
> )
```

Global variable: hyper_intermediate, in order to follow steps of minimization.

This tool returns loss_function() and its gradient; the gradient function, which is going to be evaluated, is computed by Jax and passed as input variable gradient_fun. If not np.isinf(alpha), it appends also loss and lambdas to intermediates.loss and intermediates.lambdas, respectively. Global variable: intermediates (intermediate values during the minimization steps of loss_function()).
######
Parameters

Global variable: hyper_intermediate, in order to follow steps of minimization.

pars : 1-D array-like : Numpy array of parameters for force-field correction and forward model, respectively.

data, **regularization** : dicts : Dictionaries for data object and regularizations (see in MDRefinement).

alpha, **beta**, **gamma** : floats : Values of the hyperparameters.

gtol_inn : float : Tolerance gtol for the inner minimization in loss_function().

boundaries : dict : Dictionary of boundaries for the inner minimization in loss_function().

Global variable: hyper_intermediate, in order to follow steps of minimization.

---

gradient_fun : function : Gradient function of loss_function(), computed by Jax.

###

Function **minimizer** {#MDRefine.loss_and_minimizer.minimizer}

> def minimizer(

> original_data,

> *, >

regularization: dict = None,

> alpha: float = inf,

>

beta: float = inf,

>

gamma: float = inf,

>

gtol: float = 0.001,

>

gtol_inn: float = 0.001,

>

data_valid: dict = None,

> starting_pars: numpy.ndarray = None,

>

if_print_biblio: bool = True

>)

This tool minimizes loss_function on original_data and do validation() on data_valid (if not None), at given hyperparameters.

######

Parameters

Global variable: hyper_intermediate, in order to follow steps of minimization.

original_data : dict : Dictionary for data-like object employed for the minimization of loss_function().

regularization : dict : Dictionary for the regularizations (see in MDRefinement).

alpha, **beta**, **gamma** : floats : Values of the hyperparameters for combined refinement (`+np.inf` by default: no refinement in that direction).

gtol, **gtol_inn** : floats : Tolerances gtol for the minimizations of loss_function() and inner gamma_function(), respectively.

Global variable: hyper_intermediate, in order to follow steps of minimization.

data_valid : dict : Dictionary for data-like object employed as validation set (None by default, namely no validation, just minimization).

starting_pars : 1-D array-like : Numpy 1-dimensional array for pre-defined starting point of loss_function() minimization (None by default).

Function normalize_observables

{#MDRefine.loss_and_minimizer.normalize_observables}

> def normalize_observables(
> gexp, >
g, >
weights=None
>)

Global variable: hyper_intermediate, in order to follow steps of minimization.

This tool normalizes g and gexp. Since experimental observables have different units, it is better to normalize them, in order that varying any lambda coefficient by the same value epsilon would result in comparable effects to the ensemble. This results to be useful in the minimization of gamma_function(). ###### Parameters **gexp**, **g** : dicts : Dictionaries corresponding to data.mol[name_mol].gexp and data.mol[name_mol].g. **weights** : `1-D array-like` :

Global variable: hyper_intermediate, in order to follow steps of minimization.

Numpy 1-dimensional array, by default None (namely, equal weight for each frame).

Returns **norm_g**, **norm_gexp** : dict : Dictionaries for normalized g and gexp. **norm_gmean**, **norm_gstd** : dict : Dictionaries for the reference values for normalization (average and standard deviation).

Function `print_references` {#MDRefine.loss_and_minimizer.print_references} > def print_references(> alpha, > beta, > gamma, > if_ddg >)

Function `split_dataset` {#MDRefine.loss_and_minimizer.split_dataset}

Global variable: hyper_intermediate, in order to follow steps of minimization.

```
> def split_dataset(
> data, >
*, >
frames_fraction: float = 0.2,
>
obs_fraction: float = 0.2,
> random_state: int = None,
>
valid_frames: dict = None,
>
valid_obs: dict = None,
>
if_all_frames: bool = False,
>
replica_infos: dict = None,
>
if_verbose: bool = True
> )
```

This tool splits the data set into training and validation (or test) set. You can either randomly select the frames and/or the observables (accordingly to frames_fraction, obs_fraction, random_state) or pass the dictionaries valid_obs and/or valid_frames. They refer to validation / test set.

Global variable: hyper_intermediate, in order to follow steps of minimization.

######

Parameters

data : class instance : Class instance for the data object.

frames_fraction, **obs_fraction** : float : Values for the fractions of frames and observables for the validation / test set, respectively. Each of them is a number in (0,1) (same fraction for every system), by default 0.2.

random_state : int : The random state (or seed), used to make the same choice for different hyperparameters; if None, it is randomly taken.

Global variable: hyper_intermediate, in order to follow steps of minimization.	
valid_frames, **valid_obs**	: dicts : Dictionaries for the validation frames and observables.
if_all_frames	: bool : Boolean variable, False by default; if True, then use all the frames for the validation observables in the validation set, otherwise just the validation frames.

Global variable: hyper_intermediate, in order to follow steps of minimization.

replica_infos : dict : Dictionary of information used to select frames based on continuous trajectories ("demuxing"), by default None (just randomly select frames). It includes: n_temp_replica, path_directory, stride. If not None, split_dataset() will read replica_temp.npy files with shape (n_frames, n_replicas) containing numbers from 0 to `n_replicas - 1` which indicate corresponding temperatures (for each replica ###### index in `axis=1`). ###### Returns

Global variable: hyper_intermediate, in order to follow steps of minimization.

data_train, **data_valid** : class instances : Class instances for training and validation data; data_valid includes: trained observables and non-trained (validation) frames (where it is not specified new); non-trained (validation) observables and non-trained/all (accordingly to if_all_frames) frames (where specified new).

valid_obs, **valid_frames** : dicts : Dictionaries for the observables and frames selected for the validation set.

###
Function `validation`
{#MDRefine.loss_and_minimizer.validation}

```
Global
variable:
hy-
per_intermediate,
in order to
follow steps
of mini-
mization.
> def
validation(
>
pars_ff_fm,
> lambdas,
>
data_valid,
> *, >
regulariza-
tion=None,
>
alpha=inf,
> beta=inf,
>
gamma=inf,
>
data_train=None,
>
which_return='details'
> )
This tool
evaluates
loss_function()
in detail
over the
validation
set; then,
```

Global variable: hyper_intermediate, in order to follow steps of minimization.
- if `which_return == 'chi2 valid. frames'`, it returns the total chi2 on the `'valid. frames'` data set (training observables, validation frames); this is required to compute the derivatives of the chi2 in `'validation'` with Jax;

Global variable: hyper_intermediate, in order to follow steps of minimization.

- elif `which_return == 'chi2 validation'`, it returns the total chi2 on the `'validation'` data set (validation observables, validation frames or all frames if data_train is not None); this is required to compute the derivatives of the chi2 in `'validation'` with Jax;
- else, it returns Validation_values class instance, with all the computed values (both chi2 and regularizations).

Parameters

Global variable: hyper_intermediate, in order to follow steps of minimization.

pars_ff_fm : 1-D array-like : Numpy 1-dimensional array for the force-field and forward-model coefficients.

lambdas : 1-D array-like : Numpy 1-dimensional array of lambdas coefficients (those for ensemble refinement).

data_valid : dict : Dictionary for the validation data set, data-like object, as returned by split_dataset().

regularization : dict : Dictionary for the regularizations (see in MDRefinement), by default, None.

Global variable: hyper_intermediate, in order to follow steps of minimization.

alpha, **beta**, **gamma** : floats : Values for the hyperparameters (by default, `+np.inf`, namely, no refinement).

data_train : dict : Dictionary for the training data set, data-like object, as returned by split_dataset() (None by default, namely use only validation frames for new observables).

which_return : str : String described above (by default `'details'`).

Classes

Class `class_test` {#MDRefine.loss_and_minimizer.class_test}

> class class_test(
>
> data_mol,
>
> test_obs_mol
>)

Global variable: hyper_intermediate, in order to follow steps of minimization.

Class for test data set, with similar structure as data_class.
Class `class_train`
{#MDRefine.loss_and_minimizer.class_train}
> class class_train(
>
data_mol,
>
valid_frames_mol,
>
valid_obs_mol
>)
Class for training data set, with similar structure as data_class.
Class `class_validation`
{#MDRefine.loss_and_minimizer.class_validation}
> class class_validation(
>
data_mol,
>
valid_frames_mol,
>
valid_obs_mol,
>
if_all_frames,
>
data_train_mol
>)

Global variable: hyper_intermediate, in order to follow steps of minimization.
Class for validation data set, with similar structure as data_class. ### Class `intermediates_class` {#MDRefine.loss_and_minimizer.intermediates_class} > class intermediates_class(> alpha >) Class for the intermediate steps of the minimization of the loss function.

Generated by *pdoc* 0.11.6 (https://pdoc3.github.io).