idmtools$_{calibra}$

Institute for Disease Modeling

Mar 15, 2022

CONTENTS

idmtools_calibra is a collection of Python scripts and utilities to aid in model calibration. The idmtools package should be used for streamlining the process of running disease modeling simulations. Additional information can be found at Welcome to idmtools.

See Welcome to idmtools for a diagram showing how idmtools and each of the related packages are used in an end-to-end workflow using EMOD as the disease transmission model.

CHAPTER

ONE

IDMTOOLS_CALIBRA INSTALLATION

Follow the steps below to install idmtools_calibra.

1.1 Prerequisites

First, ensure the following prerequisites are met.

- Windows 10 Pro or Enterprise, Linux, or Mac
- Python 3.6.2 or 3.7 64-bit (https://www.python.org/downloads/release)
- A file that indicates the pip index-url:
  - Windows
  - Linux

  In C:\Users\Username\pip\pip.ini, containing the following:

```
[global]
index-url = https://packages.idmod.org/api/pypi/pypi-production/simple
```

  In $HOME/.config/pip/pip.conf, containing the following:

```
[global]
index-url = https://packages.idmod.org/api/pypi/pypi-production/simple
```

1.2 Installation instructions

1. Open a command prompt and create a virtual environment in any directory you choose. The command below names the environment "v-calibra", but you may use any desired name:

```
python -m venv v-calibra
```

2. Activate the virtual environment:
   - Windows
   - Linux

   Enter the following:

```
v-calibra\Scripts\activate
```

Enter the following:

```
source v-calibra/bin/activate
```

3. Install idmtools_calibra packages:

```
pip install idmtools_calibra
```

If you are on Python 3.6, also run:

```
pip install dataclasses
```

If you are on Linux, also run:

```
pip install keyrings.alt
```

4. When you are finished, deactivate the virtual environment by entering the following at a command prompt:

```
deactivate
```

CHAPTER

TWO

IDMTOOLS_CALIBRA OVERVIEW

Add overview of library here

CHAPTER

THREE

IDMTOOLS_CALIBRA

3.1 idmtools_calibra package

3.1.1 Subpackages

idmtools_calibra.algorithms package

Subpackages

idmtools_calibra.algorithms.pbnb package

Submodules

idmtools_calibra.algorithms.pbnb.c_sub_region module

Created on Fri Jun 23 14:21:27 2017

@author: TingYu Ho

class idmtools_calibra.algorithms.pbnb.c_sub_region.**cSubRegion**(*coordinate_lower*, *coordinate_upper*, *params*)

Bases: object

idmtools_calibra.algorithms.pbnb.fun_pbnb_support_functions module

idmtools_calibra.algorithms.pbnb.fun_pbnb_support_functions.**fun_sample_points_generator_det**

idmtools_calibra.algorithms.pbnb.fun_pbnb_support_functions.**fun_sample_points_generator_no**

idmtools_calibra.algorithms.pbnb.fun_pbnb_support_functions.**turn_to_power**(*list*, *power*)

idmtools_calibra.algorithms.pbnb.fun_pbnb_support_functions.**fun_results_organizer_determin**

idmtools_calibra.algorithms.pbnb.fun_pbnb_support_functions.**fun_results_organizer_noise**(*l_su*
*df_*
*i_n_*
*i_n_*
s_si
par

idmtools_calibra.algorithms.pbnb.fun_pbnb_support_functions.**fun_order_subregion**(*c_subr*)

idmtools_calibra.algorithms.pbnb.fun_pbnb_support_functions.**fun_replication_update**(*l_subr*, *i_n_rep*, *f_alpha*)

idmtools_calibra.algorithms.pbnb.fun_pbnb_support_functions.**fun_order_region**(*l_subr*)

idmtools_calibra.algorithms.pbnb.fun_pbnb_support_functions.**fun_pruning_indicator**(*l_subr*, *f_CI_u*)

idmtools_calibra.algorithms.pbnb.fun_pbnb_support_functions.**fun_maintaining_indicator**(*l_subr*
f_CI_l

idmtools_calibra.algorithms.pbnb.fun_pbnb_support_functions.**fun_elite_indicator**(*l_subr*, *f_CI_l*)

idmtools_calibra.algorithms.pbnb.fun_pbnb_support_functions.**fun_worst_indicator**(*l_subr*, *f_CI_u*)

idmtools_calibra.algorithms.pbnb.fun_pbnb_support_functions.**fun_quantile_update**(*l_subr*, *f_delta*)

idmtools_calibra.algorithms.pbnb.fun_pbnb_support_functions.**fun_ci_builder**(*l_subr*, *pd_order_z*, *f_delta_k*, *f_alpha_k*, *f_epsilon*)

idmtools_calibra.algorithms.pbnb.fun_pbnb_support_functions.**fun_pruning_labeler**(*l_subr*)

idmtools_calibra.algorithms.pbnb.fun_pbnb_support_functions.**fun_maintaining_labeler**(*l_subr*)

idmtools_calibra.algorithms.pbnb.fun_pbnb_support_functions.**fun_reg_branching**(*c_subr*, *i_n_branching*, *params*, *s_branching_dim*)

idmtools_calibra.algorithms.pbnb.fun_pbnb_support_functions.**fun_plot2D**(*l_subr*, *l_initial_coordinate_lower*, *l_initial_coordinate_upper*, *params*, *str_k*, *s_running_file_name*, *i_iteration*)

idmtools_calibra.algorithms.pbnb.m_intial_paramters_setting module

Created on Fri Jun 23 22:15:00 2017

@author: TingYu Ho

idmtools_calibra.algorithms.pbnb.optim_tool_pbnb module

class idmtools_calibra.algorithms.pbnb.optim_tool_pbnb.**OptimToolPBnB**(*params*, *s_running_file_name*, *s_problem_type*, *constrain_sample_fn=<function OptimToolPBnB.<lambda>>*, *f_delta=0.6*, *f_alpha=0.5*, *i_k_b=3*, *i_n_branching=4*, *i_c=120*, *i_replication=1*, *i_stopping_max_k=10*, *i_max_num_simulation_per_ru*
f_elite_worst_sampling_para=4
05)

Bases: *idmtools_calibra.algorithms.next_point_algorithm.NextPointAlgorithm*

get_samples_for_iteration(*iteration*)

fun_generate_samples_from_df(*df_samples*)

fun_probability_branching_and_bound(*iteration*)

logging_saver(*iteration*)

print_results_for_iteration()

set_results_for_iteration(*iteration*, *results*)

get_state()

set_state(*state*, *iteration*)

end_condition()

get_results_to_cache(*results*)

get_final_samples()

update_summary_table(*iteration_state*, *previous_results*)
copy from NExtPointAlgorithm, output: all_results, summary_table

get_param_names()

cleanup()

Submodules

idmtools_calibra.algorithms.fisher_inf_matrix module

idmtools_calibra.algorithms.fisher_inf_matrix.**perturbed_points**(*center*, *xmin*, *xmax*, *m=10*, *n=5*, *sample_size=1*, *resolution_raw=None*)

Having the estimated optimal point (*), the next step is generating a set of perturbed points (samples). At this step, you can specify the size of perturbation (resolution). In case you do not input the perturbation size (resolution_raw=None), the script will pick a perturbation using the given range for each parameter (Xmin-Xmax). You need to allocate per-realization (M) and cross-realization (N). Note that $M > p^2$, where p is the number of parameters (size of). The number of cross-realization (N) should be set depending on the uncertainty of the model at the given final point (*). To choose N, you can use Chebyshev's inequality.

You can allocate number of replicates of the model (n). The log-likelihood at each point obtains from n replicates of the model with different run numbers. Then these n replicates are used to compute the likelihood at that point. When we have a model with high uncertainty, the easiest way to compute the likelihood might be taking average of the multiple (n) replicates of the model run to compute the likelihood. Note that the algorithm only accepts n=1 now. But the Cramer Rao script has the potential to accept higher n, whenever a smoothing technique which requires multiple replicates of the model for computing the likelihood be added to the Analyzers.

Parameters

- **center** – center point 1xp nparray
- **xmin** – minimum of parameters 1xp nparray
- **xmax** – maximum of parameters 1xp nparray
- **m** – number of Hessian estimates scalar-positive integer
- **n** – number of pseudodata vectors scalar-positive integer
- **sample_size** – sample size scalar-positive integer
- **resolution_raw** – minimum meaningful perturbation for each parameter 1xp nparray

Returns perturbed points (4MNn x 4+p) nparray

Return type X_perturbed

idmtools_calibra.algorithms.fisher_inf_matrix.**compute_fisher_inf_matrix**(*center_point*, *df_ll_points*, *data_columns*)

Compute the Fisher Information matrix using the LL of perturbed points

Computation of the Fisher information matrix (covariance matrix) This step returns a p×p covariance matrix, called .

Atiye Alaeddini, 12/15/2017

Parameters

- **center_point** – center point (1 x p) nparray
- **df_ll_points** – Log Likelihood of points DataFrame
- **data_columns** – List of columns to filter points with

Returns Fisher Information matrix (p x p) np array

Return type Fisher

idmtools_calibra.algorithms.fisher_inf_matrix.**sample_cov_ellipse**(*cov*, *pos*, *num_of_pts=10*)

Sample 'num_of_pts' points from the specified covariance matrix (*cov*).

Parameters

- **cov** – 2-D array_like, The covariance matrix (inverse of fisher matrix). It must be symmetric and positive-semidefinite for proper sampling.
- **pos** – 1-D array_like, The location of the center of the ellipse, Mean of the multi variate distribution
- **num_of_pts** – The number of sample points.

Returns ndarray of the drawn samples, of shape (num_of_pts,).

idmtools_calibra.algorithms.fisher_inf_matrix.**trunc_gauss**(*mu*, *sigma*, *low_bound*, *high_bound*, *num_of_pts*, *batch_size=100*)

idmtools_calibra.algorithms.fisher_inf_matrix.**div0**(*a*, *b*)

ignore / 0, div0([-1, 0, 1], 0) -> [0, 0, 0]

idmtools_calibra.algorithms.fisher_inf_matrix.**near_pd**(*a*, *nit=10*)

idmtools_calibra.algorithms.generic_iterative_next_point module

class idmtools_calibra.algorithms.generic_iterative_next_point.**GenericIterativeNextPoint**(*in*

Bases: *idmtools_calibra.algorithms.next_point_algorithm.NextPointAlgorithm*

Represents a Generic Next Point allowing thew Calibtool to function as a more generic iterative process. Here a dictionary needs to be passed as the state. For example:

```
initial_state = [{
 'Run_Number': rn
 } for rn in range(2)]
```

Then the results of the analyzers are stored in the self.data associating iteration with results. Both the initial state and the results are stored there allowing to easily refer to it:

```
self.data = [
                {
                'Run_Number': 1
                'results':{
                    'what_comes_from_analyzers':{}
                },
                ...
            ]
```

Note that the results needs to be contained in a Dictionary. If you want to leverage pandas.DataFrame instead, you should use OptimTool.

set_state(*state*, *iteration*)

cleanup()

update_iteration(*iteration*)

Update the current iteration state of the algorithm.

get_param_names()

get_samples_for_iteration(*iteration*)

get_state()

prep_for_dict(*df*)
Utility function allowing to transform a DataFrame into a dict removing null values

set_results_for_iteration(*iteration*, *results*)

end_condition()

get_final_samples()

update_summary_table(*iteration_state*, *previous_results*)
Returns a summary table of the form: [result1 result2 results_total param1 param2 iteration simIds] index = sample Used by OptimTool and IMIS algorithm

get_results_to_cache(*results*)

idmtools_calibra.algorithms.gpc module

class idmtools_calibra.algorithms.gpc.**GPC**(*x_cols*, *y_col*, *training_data*, *param_info*, *kernel_mode='RBF'*, *kernel_params=None*, *verbose=False*, *debug=False*, ***kwargs*)

Bases: object

classmethod from_config(*config_fn*)

classmethod from_dict(*config*)

set_training_data(*new_training_data*)

save(*save_to=None*)

define_kernel(*params*)

kernel_xx(*x*, *theta*)

static kernel_xp(*x*, *p*, *theta*)

kxx_gpu_wrapper(*x*, *theta*, *deriv=- 1*)

kxp_gpu_wrapper(*x*, *p*, *theta*)

assign_rep(*sample*)

expectation_propagation(*theta*)

find_posterior_mode(*theta*, *f_guess=None*, *tol_grad=1e-06*, *max_iter=100*)

negative_log_marginal_likelihood(*theta*)

negative_log_marginal_likelihood_and_gradient(*theta*, *f_guess=None*)

static func_wrapper(*f*, *cache_size=100*)

laplace_predict(*theta*, *f_hat*, *p*)

ep_predict(*theta*, *p*)

optimize_hyperparameters(*x0*, *bounds=()*, *k=- 1*, *eps=0.01*, *disp=True*, *maxiter=15000*)

evaluate(*data*)

plot_data(*samples_to_circle=None*)

plot_histogram()

plot(*x_center*, *res=10*)

plot_errors(*train*, *test*, *mean_col*, *var_predictive_col*, *truth_col=None*, *figsize=16, 10*)

idmtools_calibra.algorithms.imis module

class idmtools_calibra.algorithms.imis.**IMIS**(*Incremental Mixture Importance Sampling*)
Bases: *idmtools_calibra.algorithms.next_point_algorithm.NextPointAlgorithm*

Algorithm ported from R code: http://cran.r-project.org/web/packages/IMIS

Full description from Adrian Raftery and Le Bao (2009): http://www.stat.washington.edu/research/reports/2009/tr560.pdf

The basic idea of IMIS is that points with high importance weights are in areas where the target density is underrepresented by the importance sampling distribution. At each iteration, a multivariate normal distribution centered at the point with the highest importance weight is added to the current importance sampling distribution, which thus becomes a mixture of such functions and of the prior. In this way underrepresented parts of the parameter space are successively identified and are given representation, ending up with an iteratively constructed importance sampling distribution that covers the target distribution well.

validate_parameters()
Ensure valid parameter ranges: 'samples_per_iteration' is used to select N-closest points to maximum-weight sample, so it can't exceed 'n_initial_samples'. It is also used to estimate weighted covariance, so it cannot be smaller than the dimensionality of the samples.

choose_initial_samples()

set_initial_samples()
Set the initial samples points for the algorithm. If initial_samples parameter is an array, use those values as initial samples. Otherwise, if initial_samples parameter is a number, draw the specified number randomly from the prior distribution.

add_samples(*samples*, *iteration*)

get_next_samples_for_iteration(*iteration*)

choose_next_point_samples(*iteration*)

generate_variables_from_data()
restore some properties from self.data

get_samples_for_iteration(*iteration*)

update_iteration(*iteration*)
Initial Stage (iteration: k = 0)

(a) Sample N inputs $\theta_1, \theta_2, ..., \theta_N$ from the prior distribution $p(\theta)$.

(b) For each θ_i, calculate the likelihood L_i, and form the importance weights: $w_i^{(0)} = L_i / sum(L_j)$.

Importance Sampling Stage (iteration: k > 0; samples_per_iteration: B)

(c) Calculate the likelihood of the new inputs and combine the new inputs with the previous ones. Form the importance weights: $w^{(k)_i} = c * L_i * p(\theta_i)/q^{(k)}(\theta_i)$, where c is chosen so that the weights add to 1, $q^{(k)}$ is the mixture sampling distribution: $q^{(k)} = (N_0/N_k) * p + (B/N_k) * sum(H_s)$ where H_s is the Sth multivariate normal distribution, and $N_k = N_0 + B_k$ is the total number of inputs up to iteration k.

update_state(*iteration*)
Update the next-point algorithm state and select next samples.

update_gaussian()
Importance Sampling Stage (iteration: k > 0; samples_per_iteration: B)

(a) Choose the current maximum weight input as the center $\theta^{(k)}$. Estimate $\Sigma^{(k)}$ from the weighted covariance of the B inputs with the smallest Mahalanobis distances to $\theta^{(k)}$, where the distances are calculated with respect to the covariance of the prior distribution and the weights are taken to be proportional to the average of the importance weights and $1/N_k$.

(b) Sample 'samples_per_iteration' new inputs from a multivariate Gaussian distribution H_k with covariance matrix $\Sigma^{(k)}$.

update_gaussian_center()
Choose the current maximum weight input as the center point for the next iteration of multivariate-normal sampling.

weighted_distances_from_center()
Calculate the covariance-weighted distances from the current maximum-weight sample. N.B. Using normalized Euclid instead of Mahalanobis if we're just going to diagonalize anyways.

update_gaussian_covariance(*distances*)
Calculate the covariance of the next-iteration of multivariate-normal sampling from the "samples_per_iteration" closest samples.

get_param_names()

verify_valid_samples(*next_samples*)
Resample from next-point function until all samples have non-zero prior.

next_point_fn()
IMIS next-point sampling from multivariate normal centered on the maximum weight.

update_gaussian_probabilities(*iteration*)
Calculate the probabilities of all sample points as estimated from the multivariate-normal probability distribution function centered on the maximum weight and with covariance fitted from the most recent iteration.

calculate_weighted_covariance(*samples*, *weights*, *center*)
A weighted covariance of sample points. N.B. The weights are normalized as in the R function "cov.wt"

end_condition()

Stopping Criterion: The algorithm ends when the importance sampling weights are reasonably uniform. Specifically, we end the algorithm when the expected fraction of unique points in the resample is at least (1 - 1/e) = 0.632. This is the expected fraction when the importance sampling weights are all equal, which is the case when the importance sampling function is the same as the target distribution.

get_final_samples()

get_state()

set_state(*state*, *iteration*)

set_results_for_iteration(*iteration*, *results*)

cleanup()

restore(*iteration_state*)

idmtools_calibra.algorithms.next_point_algorithm module

class idmtools_calibra.algorithms.next_point_algorithm.**NextPointAlgorithm**
Bases: object

abstract set_state(*state*, *iteration*)

restore(*iteration_state*)

abstract cleanup()

abstract get_param_names()

abstract get_samples_for_iteration(*iteration*)

abstract get_state()

abstract set_results_for_iteration(*iteration*, *results*)

abstract end_condition()

abstract get_final_samples()

update_iteration(*iteration*)
Update the current iteration state of the algorithm.

prep_for_dict(*df*)
Utility function allowing to transform a DataFrame into a dict removing null values

static sample_from_function(*function*, *N*)

update_summary_table(*iteration_state*, *previous_results*)

Returns a summary table of the form: [result1 result2 results_total param1 param2 iteration simIds] index = sample Used by OptimTool and IMIS algorithm

get_results_to_cache(*results*)

generate_samples_from_df(*dfsamples*)

idmtools_calibra.algorithms.optim_tool module

class idmtools_calibra.algorithms.optim_tool.**OptimTool**(*params*, *constrain_sample_fn=<function OptimTool.<lambda>>*, *mu_r=0.1*, *sigma_r=0.02*, *center_repeats=2*, *samples_per_iteration=100.0*, *rsquared_thresh=0.5*)
Bases: *idmtools_calibra.algorithms.next_point_algorithm.NextPointAlgorithm*

The basic idea of OptimTool is

cleanup()

verify_param()

resolve_args(*iteration*)

add_samples(*samples*, *iteration*)

get_samples_for_iteration(*iteration*)

clamp(*X*)

set_results_for_iteration(*iteration*, *results*)

choose_initial_samples()

choose_samples_via_gradient_ascent(*iteration*)

choose_and_clamp_hypersphere_samples_for_iteration(*iteration*)

sample_hypersphere(*N*, *state*)

end_condition()

get_final_samples()
Resample Stage:

prep_for_dict(*df*)
Utility function allowing to transform a DataFrame into a dict removing null values

get_state()

set_state(*state*, *iteration*)

get_param_names()

static get_r(*num_params*, *volume_fraction*)

idmtools_calibra.algorithms.optim_tools_pspo module

class idmtools_calibra.algorithms.optim_tools_pspo.**OptimToolPSPO**(*params*, *constrain_sample_fn*, *comps_per_iteration=10*)

Bases: *idmtools_calibra.algorithms.next_point_algorithm.NextPointAlgorithm*

OptimTool

The basic idea of OptimTool is

cleanup()

resolve_args(*iteration*)

add_samples(*samples*, *iteration*)

get_samples_for_iteration(*iteration*)

clamp(*X*)

set_results_for_iteration(*iteration*, *results*)

choose_initial_samples()

stochastic_newton_raphson(*iteration*)

choose_and_clamp_samples_for_iteration(*iteration*)

sample_simultaneous_perturbation(*M*, *iteration*, *state*)

end_condition()

get_final_samples()

prep_for_dict(*df*)
Utility function allowing to transform a DataFrame into a dict removing null values

get_state()

set_state(*state*, *iteration*)

get_param_names()

idmtools_calibra.algorithms.optim_tools_spsa module

class idmtools_calibra.algorithms.optim_tools_spsa.**OptimToolSPSA**(*params*, *constrain_sample_fn*, *comps_per_iteration=10*)

Bases: *idmtools_calibra.algorithms.next_point_algorithm.NextPointAlgorithm*

The basic idea of OptimToolSPSA is

cleanup()

resolve_args(*iteration*)

add_samples(*samples*, *iteration*)

get_samples_for_iteration(*iteration*)

clamp(*X*)

set_results_for_iteration(*iteration*, *results*)

choose_initial_samples()

stochastic_newton_raphson(*iteration*)

choose_and_clamp_samples_for_iteration(*iteration*)

sample_simultaneous_perturbation(*M*, *iteration*, *state*, *resolution=None*)

end_condition()

get_final_samples()
Resample Stage:

prep_for_dict(*df*)
Utility function allowing to transform a DataFrame into a dict removing null values

get_state()

set_state(*state*, *iteration*)

get_param_names()

idmtools_calibra.algorithms.separatrix_bhm module

class idmtools_calibra.algorithms.separatrix_bhm.**SeparatrixBHM**(*params*, *constrain_sample_fn=<function SeparatrixBHM.<lambda>>*, *implausibility_threshold=3*, *target_success_probability=0.7*, *num_past_iterations_to_include_in_meta*, *samples_per_iteration=32*, *samples_final_iteration=128*, *max_iterations=10*, *training_frac=0.8*)

Bases: *idmtools_calibra.algorithms.next_point_algorithm.NextPointAlgorithm*

Separatrix using Bayesian History Matching

The basic idea of Separatrix is that each simulation results in a success (+1) or a failure (-1), and the success probability varies as a function of the input parameters. We seek an isocline of the the latent success probability function.

cleanup()

resolve_args(*iteration*)

add_samples(*samples*, *iteration*)

get_samples_for_iteration(*iteration*)

set_results_for_iteration(*iteration*, *results*)

set_gpc_vec(*iteration*, *gpc*)

choose_initial_samples()

choose_samples_via_history_matching(*iteration*)

end_condition()

get_final_samples()
Return some number of samples from the residual non-implausible area:

prep_for_dict(*df*)
Utility function allowing to transform a DataFrame into a dict removing null values

get_state()

set_state(*state*, *iteration*)

get_param_names()

idmtools_calibra.analyzers package

Submodules

idmtools_calibra.analyzers.base_calibration_analyzer module

class idmtools_calibra.analyzers.base_calibration_analyzer.**BaseCalibrationAnalyzer**(*uid=None*, *working_dir=N*
parse=Tru
*need_dir_*
filenames=No
reference_data=
weight=1)

Bases: idmtools.entities.ianalyzer.IAnalyzer

cache()

idmtools_calibra.cli package

Submodules

idmtools_calibra.cli.commands module

idmtools_calibra.cli.utils module

idmtools_calibra.cli.utils.**load_config_module**(*config_name*)

idmtools_calibra.cli.utils.**get_calib_manager**(*config_name*, *calib_callback='get_manager'*)

idmtools_calibra.cli.utils.**read_calib_data**(*calib_path*, *force=False*)

idmtools_calibra.output package

Submodules

idmtools_calibra.output.output_parser module

idmtools_calibra.output.spatial_output module

class idmtools_calibra.output.spatial_output.**SpatialOutput**

Bases: object

classmethod from_bytes(*bytes*, *filtered=False*)

to_dict()

idmtools_calibra.plotters package

Submodules

idmtools_calibra.plotters.base_plotter module

class idmtools_calibra.plotters.base_plotter.**BasePlotter**(*combine_sites=True*)
Bases: object

get_iteration_directory()

get_plot_directory()

property all_results

abstract visualize(*iteration_state*)

static combine_by_site(*site_name*, *analyzer_names*, *results*)
Sum the result values over analyzers performed on the specified site and add SITE_total to results :param site_name: The name of the CalibSite :param analyzer_names: A list of CalibAnalyzer names corresponding to the site :param results: A pandas.DataFrame of results into which to add a combined site-analyzer-result column :return: None

idmtools_calibra.plotters.likelihood_plotter module

class idmtools_calibra.plotters.likelihood_plotter.**LikelihoodPlotter**(*combine_sites=True*)
Bases: *idmtools_calibra.plotters.base_plotter.BasePlotter*

property param_names

property prior_fn

property directory

visualize(*iteration_state*)

plot_by_parameter_and_site()

plot_by_parameter(*site=''*, ***kwargs*)

static plot1d_by_iteration(*results*, *param*, *total*, ***kwargs*)

idmtools_calibra.plotters.optim_tool_pbnb_plotter module

class idmtools_calibra.plotters.optim_tool_pbnb_plotter.**OptimToolPBnBPlotter**
Bases: *idmtools_calibra.plotters.base_plotter.BasePlotter*

cleanup()

static plot_state_evolution(***kwargs*)

visualize(*iteration_state*)

visualize_results()

idmtools_calibra.plotters.optim_tool_plotter module

class idmtools_calibra.plotters.optim_tool_plotter.**OptimToolPlotter**
Bases: *idmtools_calibra.plotters.base_plotter.BasePlotter*

cleanup()

plot_state_evolution(***kwargs*)

visualize(*iteration_state*)

visualize_results()

visualize_optimtool_diagnoistics()

cleanup_plot(*calib_manager*)

idmtools_calibra.plotters.optim_tool_spsa_plotter module

class idmtools_calibra.plotters.optim_tool_spsa_plotter.**OptimToolSPSAPlotter**
Bases: *idmtools_calibra.plotters.base_plotter.BasePlotter*

cleanup()

plot_state_evolution(***kwargs*)

visualize(*iteration_state*)

visualize_results()

visualize_optimtool_diagnoistics()

idmtools_calibra.plotters.separatrix_bhm_plotter module

class idmtools_calibra.plotters.separatrix_bhm_plotter.**SeparatrixBHMPlotter**
Bases: *idmtools_calibra.plotters.base_plotter.BasePlotter*

make_prediction_grid()

cleanup()

visualize(*iteration_state*)

visualize_at_end_of_iteration()

visualize_at_start_of_iteration()

cleanup_plot(*calib_manager*)

idmtools_calibra.plotters.site_data_plotter module

class idmtools_calibra.plotters.site_data_plotter.**SiteDataPlotter**(*combine_sites=True*, *num_to_plot=5*, *ll_all_name: str = 'LL_all.csv'*)
Bases: *idmtools_calibra.plotters.base_plotter.BasePlotter*

property directory

get_site_analyzer(*site_name*, *analyzer_name*)

get_analyzer_data(*iteration*, *site_name*, *analyzer_name*)

visualize(*iteration_state*)

plot_analyzers(*site_name*, *analyzer_names*, *samples*)

plot_best(*site_name*, *analyzer_name*, *samples*)

plot_all(*site_name*, *analyzer_name*, *samples*, *clim*)

cleanup()
cleanup the existing plots :param calib_manager: :return:

cleanup_plot_by_analyzers(*site*, *analyzers*, *samples*)
cleanup the existing plots :param site: :param analyzers: :param samples: :return:

cleanup_plot_for_best(*site_analyzer*, *samples*)
cleanup the existing plots :param site_analyzer: :param samples: :return:

cleanup_plot_for_all(*site_analyzer*)
cleanup the existing plots :param site_analyzer: :return:

write_LL_csv(*ll_all_name: str = None*)
Write the LL_summary.csv with what is in the CalibManager

idmtools_calibra.resamplers package

Submodules

idmtools_calibra.resamplers.base_resampler module

class idmtools_calibra.resamplers.base_resampler.**BaseResampler**(*calib_manager=None*)
Bases: object

abstract resample(*calibrated_points*, *selection_values*, *initial_calibration_points*)

post_analysis(*resampled_points*, *analyzer_results*, *from_resample=None*)

set_calibration_manager(*calib_manager:* idmtools_calibra.calib_manager.CalibManager)

resample_and_run(*calibrated_points*, *resample_step*, *selection_values*, *initial_calibration_points*)
Canonical entry method for using the resampler.

Parameters

- **calibrated_points** – Calibrated Points
- **resample_step** –
- **selection_values** –
- **initial_calibration_points** –

Returns:

idmtools_calibra.resamplers.calibration_point module

class idmtools_calibra.resamplers.calibration_point.**CalibrationPoint**(*parameters=None*, *likelihood=None*)

Bases: object

DYNAMIC = 'dynamic'

STATIC = 'static'

ALL = 'all'

PARAMETER_TYPES = ['dynamic', 'static', 'all']

LIST = 'list'

SERIES = 'series'

NUMPY = 'numpy'

to_value_dict(*parameter_type=None*, *include_likelihood=False*)
Return the dict of dict containing {parameter_name:value} for this CalibrationPoint

get_attribute(*key*, *parameter_type=None*, *as_type=None*)
Returns the specified attribute of each CalibrationParameter as a list, ordered by parameter

Parameters

- **key** –
- **parameter_type** –
- **as_type** –

Returns:

property parameter_names

get_parameter(*name*)
Relies on their being exactly one parameter with the given name.

Parameters **name** –

Returns:

to_dict()
Converts CalibrationPoint objects to a dictionary. Useful e.g. for dumping to a json file.

Returns a dict containing all needed information for recreating a CalibrationPoint object via from_dict()

classmethod from_dict(*src_dict*)
Inverse method of to_dict. Builds point from a Dictionary :param src_dict: a dictionary equivalent to one returned by to_dict()

Returns a CalibrationPoint object

to_dataframe(*parameter_type=None*)

write_point(*filename*)

class idmtools_calibra.resamplers.calibration_point.**CalibrationParameter**(*name*, *min*, *max*, *value*, *mapTo*, *dynamic*, *guess*)

Bases: object

classmethod from_dict(*parameters*)

classmethod from_calibration_parameter(*parameter*, *value*)
Create a new CalibrationParameter object from another one but with a new value. :param parameter: input CalibrationParameter to copy :param value: the value to override the copy with :return: a CalibrationParameter object

to_dict()

to_dataframe()

idmtools_calibra.resamplers.calibration_points module

class idmtools_calibra.resamplers.calibration_points.**CalibrationPoints**(*points*)

Bases: object

write(*filename*)

classmethod read(*filename*)

idmtools_calibra.resamplers.cramer_rao_resampler module

class idmtools_calibra.resamplers.cramer_rao_resampler.**CramerRaoResampler**(*calib_manager=None*, ***kwargs*)

Bases: *idmtools_calibra.resamplers.base_resampler.BaseResampler*

resample(*calibrated_points*, *selection_values*, *initial_calibration_points*)
Takes in a list of 1+ Point objects and returns method-specific resampled points as a list of Point objects The resultant Point objects should be copies of the input Points BUT with Value overridden on each, e.g.:

```
new_point = Point.copy(one_of_the_input_calibrated_points)
for param in new_point.list_params():
  new_point.set_param_value(param, value=SOME_NEW_VALUE)
```

Parameters **calibrated_points** – input points for this resampling method

Returns A list of resampled Point objects

post_analysis(*resampled_points*, *analyzer_results*, *from_resample*)

idmtools_calibra.resamplers.random_perturbation_resampler module

class idmtools_calibra.resamplers.random_perturbation_resampler.**RandomPerturbationResampler**

Bases: *idmtools_calibra.resamplers.base_resampler.BaseResampler*

resample(*calibrated_points*, *selection_values*, *initial_calibration_points*)
Takes in a list of 1+ Point objects and returns method-specific resampled points as a list of Point objects
The resultant Point objects should be copies of the input Points BUT with Value overridden on each, e.g.:

```
new_point = Point.copy(one_of_the_input_calibrated_points)
for param in new_point.list_params():
  new_point.set_param_value(param, value=SOME_NEW_VALUE)
```

Parameters

- **calibrated_points** – input points for this resampling method
- **selection_values** –
- **initial_calibration_points** –

Returns A list of resampled Point objects

post_analysis(*resampled_points*, *analyzer_results*, *from_resample=None*)

Parameters

- **resampled_points** –
- **analyzer_results** –
- **from_resample** –

Returns:

generate_perturbed_points(*center_point*) → pandas.core.frame.DataFrame
given center and generate perturbed points

Parameters **center_point** – center point

Returns:

idmtools_calibra.utilities package

Subpackages

idmtools_calibra.utilities.parsers package

Submodules

idmtools_calibra.utilities.parsers.JSON module

idmtools_calibra.utilities.parsers.bin2np module

idmtools_calibra.utilities.parsers.dict2pd module

3.1.2 Submodules

idmtools_calibra.calib_manager module

idmtools_calibra.calib_manager.**set_parameter_sweep_callback**(*simulation: idmtools.entities.simulation.Simulation*, *param: str*, *value: Any*) → Dict[str, Any]

Convenience callback for sweeps

Parameters

- **simulation** – Simulation we are updating
- **param** – Parameter
- **value** – Value

Returns Tags to set on simulation

class idmtools_calibra.calib_manager.**SampleIndexWrapper**(*map_sample_to_model_input_fn*)
Bases: object

Wrapper for a SimConfigBuilder-modifying function to add metadata on sample-point index when called in a iteration over sample points

class idmtools_calibra.calib_manager.**CalibManager**(*task: idmtools.entities.itask.ITask*, *map_sample_to_model_input_fn*, *sites: List[*idmtools_calibra.calib_site.CalibSite*]*, *next_point:* idmtools_calibra.algorithms.next_point_algorithm.NextPointAlg *platform: Optional[idmtools.entities.iplatform.IPlatform] = None*, *name: str = 'calib_test'*, *sim_runs_per_param_set: int = 1*, *max_iterations: int = 5*, *plotters: List[*idmtools_calibra.plotters.base_plotter.BasePlotter*] = None*, *map_replicates_callback=None*, *directory='.'*)
Bases: object

Manages the creation, execution, and resumption of multi-iteration a calibration suite. Each iteration spawns a new ExperimentManager to configure and commission either local or HPC simulations for a set of random seeds, sample points, and site configurations.

classmethod open_for_reading(*calibration_directory*)

property suite_id

property iteration

run_calibration(***kwargs*)
Create and run a complete multi-iteration calibration suite. kwargs supports the following optional parameters: :param resume: bool, default=False, flag required for calibration resume :param iteration: int, default=None, from which iteration to resume :param iter_step: str, default=None, from which calibration step to resume. support values: 'commission', 'analyze', plot' and 'next_point' :param loop: bool, default=True, if like to continue to next iteration :param max_iterations, int, default=None, user can override the max_iterations defined in calib_manager: :param backup: bool, default=False, if like to backup Calibration.json :param dry_run: bool, default=False, if like to really execute resume action :param directory: str, default=None, calibration directory

Returns: None

run_iterations(*iteration: int = 0*, *max_iterations: int = None*, *loop: bool = True*)
Run iterations in a loop :param iteration: the # of iterations :param max_iterations: max iterations :param loop: if or not continue iteration loop

Returns: None

post_iteration()

set_experiment_builder_function(*exp_builder_function*)
Set the experiment builder to a predefined one, such that exp_builder_func will return it :param exp_builder_function: an experiment builder object

Returns: None

set_map_replicates_callback(*map_replicates_callback*)
This sets the maps replicate callback. This callback should take a value parameter that is the value of the

replicate.

Normally you would want to map this to a value in the config :param map_replicates_callback:

Returns:

default_experiment_builder_function(*next_params*, *n_replicates: Optional[int] = None*) → idmtools.builders.simulation_builder.SimulationBuilder

Defines the function that builds the experiment for each iteration of a calibration run

Parameters

- **next_params** – The next parameters to run
- **n_replicates** – Number of replicates

Returns Simulation Builder

create_iteration_state(*iteration*)

Create iteration state :param iteration: the # of the iteration

Returns: created IterationState

create_calibration()

Create the working directory for a new calibration. Cache the relevant suite-level information to allow re-initializing this instance.

finalize_calibration()

Get the final samples from the next point algorithm.

cache_calibration(***kwargs*)

Cache information about the CalibManager that is needed to resume after an interruption. N.B. This is not currently the complete state, some of which relies on nested and frozen functions. As such, the 'resume' logic relies on the existence of the original configuration script. :param **kwargs: extra info

Returns: None

backup_calibration()

Backup CalibManager.json for resume action

serialize_results()

resume_calibration(***kwargs*)

kill()

cleanup()

Cleanup the current calibration - Delete the result directory - If LOCAL -> also delete the simulations

read_calib_data(*force=False*)

property calibration_path

property analyzer_list

property required_components

iteration_directory()

state_for_iteration(*iteration*)

param_names()

site_analyzer_names()

get_last_iteration()
Determines the last (most recent) completed iteration number. Returns: the last completed iteration number as an int

get_parameter_sets_with_likelihoods()
Returns: a list of ParameterSet objects.

idmtools_calibra.calib_site module

class idmtools_calibra.calib_site.**SiteFunctions**(*name*, *setup_functions*, *verbose=False*)
Bases: object

A helper to take a set of bare SimConfigBuilder-modifying functions and combine them into a single function of the same format

set_calibration_site(*task*)
N.B. The name of this function is chosen to ensure it is applied first by ModBuilder.set_mods and other aspects can be over-ridden as needed by sample-point modifications.

class idmtools_calibra.calib_site.**CalibSite**(*name*)
Bases: object

A class to represent the base behavior of a calibration site

abstract get_reference_data(*reference_type*)
Callback function for derived classes to pass site-specific reference data that is requested by analyzers by the relevant reference_type.

abstract get_analyzers()
Derived classes return a list of BaseComparisonAnalyzer instances that have been passed a reference to the CalibSite for site-specific analyzer setup.

abstract get_setup_functions()
Derived classes return a list of functions to apply site-specific modifications to the base configuration. These are combined into a single function using the SiteFunctions helper class in the CalibSite constructor.

idmtools_calibra.iteration_state module

class idmtools_calibra.iteration_state.**IterationState**(***kwargs*)
Bases: object

Holds the settings, parameters, simulation state, analysis results, etc. for one calibtool iteration.

Allows for the resumption or extension of existing CalibManager instances from an arbitrary point in the iterative process.

property status

update(***kwargs*)

restore_results(*iteration*)
Restore summary results from serialized state. :param iteration: the # of iteration

Returns:

run()

starting_step()

commission_step()

analyze_step()

plotting_step()

next_point_step()

commission_iteration(*next_params*)
Commission an experiment of simulations constructed from a list of combinations of random seeds, calibration sites, and the next sample points. Cache the relevant experiment and simulation information to the IterationState. :param next_params: the next sample

Returns: None

plot_iteration()

analyze_iteration()
Analyze the output of completed simulations by using the relevant analyzers by site. Cache the results that are returned by those analyzers.

wait_for_finished(*init_sleep=1.0*, *sleep_time=30*)

kill()

property iteration_directory

property iteration_file

property param_names

finished()
The next-point algorithm has reached its termination condition.

classmethod from_file(*filepath*)

to_file()

classmethod restore_state(*iteration*)
Restore IterationState

set_samples_for_iteration(*samples*, *next_point*)

save()
Cache information about the IterationState that is needed to resume after an interruption. If resuming from an existing iteration, also copy to backup the initial cached state.

get_parameter_sets_with_likelihoods()

idmtools_calibra.process_state module

class idmtools_calibra.process_state.**StatusPoint**(*value*)
Bases: enum.Enum

An enumeration.

iteration_start = 0

commission = 1

running = 2

analyze = 3

plot = 4

next_point = 5

done = 6

idmtools_calibra.resample_manager module

class idmtools_calibra.resample_manager.**ResampleManager**(*steps*, *calibration_manager: idmtools_calibra.calib_manager.CalibManager*, *restart_at_step=None*)

Bases: object

resample_and_run()

write_restart(*step*, *selection_values*)

load_restart(*step*)

get_calibrated_points()

Retrieve information about the most recent (final completed) iteration's calibrated point, merging from the final IterationState.json and CalibManager.json.

Returns:

PYTHON MODULE INDEX

i

INDEX

D

E

F

G

I

N

O

P

R

S

T

U

V

W