mud Documentation

Release 0.1rc1.post1.dev1+g7732468

Mathematical Michael

Jul 05, 2022

CONTENTS

This is the documentation of the **mud** library.

Warning: This website is under active construction. Please report incomplete documentation. Last edited: Jul 05, 2022

CHAPTER

ONE

EXAMPLE USAGE

```
from mud.funs import mud_sol
mud_sol()
```

CHAPTER

TWO

CONTENTS

2.1 Project Description

2.1.1 MUD

Analytical solutions and some associated utility functions for computing Maximal Updated Density (MUD) parameter estimates for Data-Consistent Inversion.

Description

Maximal Updated Density Points are the values which maximize an updated density, analogous to how a MAP (Maximum A-Posteriori) point maximizes a posterior density from Bayesian inversion. Updated densities differ from posteriors in that they are the solution to a different problem which seeks to match the push-forward of the updated density to a specified observed distribution.

2.2 mud

2.2.1 mud package

Submodules

mud.base module

class mud.base.**BayesProblem**(*X: Union[ndarray, List]*, *y: Union[ndarray, List]*, *domain: Optional[Union[ndarray, List]] = None*)

Bases: object

Sets up Bayesian Inverse Problem for parameter identification

Parameters

- **X** (*ndarray*) – 2D array containing parameter samples from an initial distribution. Rows represent each sample while columns represent parameter values.
- **y** (*ndarray*) – array containing push-forward values of paramters samples through the forward model. These samples will form the data-likelihood distribution.
- **domain** (*array_like, optional*) – 2D Array containing ranges of each paramter value in the parameter space. Note that the number of rows must equal the number of parameters, and the number of columns must always be two, for min/max range.

Examples

```
>>> from mud.base import BayesProblem
>>> import numpy as np
>>> from scipy.stats import distributions as ds
>>> X = np.random.rand(100,1)
>>> num_obs = 50
>>> Y = np.repeat(X, num_obs, 1)
>>> y = np.ones(num_obs)*0.5 + np.random.randn(num_obs)*0.05
>>> B = BayesProblem(X, Y, np.array([[0,1]]))
>>> B.set_likelihood(ds.norm(loc=y, scale=0.05))
>>> np.round(B.map_point()[0],1)
0.5
```

estimate()

fit()

map_point()

property n_features

property n_params

property n_samples

plot_obs_space(*obs_idx=0*, *ax=None*, *y_range=None*, *aff=1000*, *ll_opts={'color': 'r', 'label': 'Data-Likelihood', 'linestyle': '-', 'linewidth': 4}*, *pf_opts={'color': 'g', 'label': 'PF of Posterior', 'linestyle': ':', 'linewidth': 4}*)

Plot probability distributions defined over observable space.

plot_param_space(*param_idx=0*, *ax=None*, *x_range=None*, *aff=1000*, *pr_opts={'color': 'b', 'label': 'Prior', 'linestyle': '--', 'linewidth': 4}*, *ps_opts={'color': 'g', 'label': 'Posterior', 'linestyle': ':', 'linewidth': 4}*)

Plot probability distributions over parameter space

set_likelihood(*distribution*, *log=False*)

set_prior(*distribution=None*)

class mud.base.**DensityProblem**(*X: ndarray*, *y: ndarray*, *domain: Optional[Union[ndarray, List[float]]] = None*, *weights: Optional[Union[ndarray, List[float]]] = None*)

Bases: object

Sets up Data-Consistent Inverse Problem for parameter identification

Data-Consistent inversion is a way to infer most likely model paremeters using observed data and predicted data from the model.

X

Array containing parameter samples from an initial distribution. Rows represent each sample while columns represent parameter values. If 1 dimensional input is passed, assumed that it represents repeated samples of a 1-dimensional parameter.

Type
np.ndarray

y

Array containing push-forward values of paramters samples through the forward model. These samples will form the *predicted distribution*.

Type
np.ndarray

domain

Array containing ranges of each paramter value in the parameter space. Note that the number of rows must equal the number of parameters, and the number of columns must always be two, for min/max range.

Type
np.ndarray

weights

Weights to apply to each parameter sample. Either a 1D array of the same length as number of samples or a 2D array if more than one set of weights is to be incorporated. If so the weights will be multiplied and normalized row-wise, so the number of columns must match the number of samples.

Type
np.ndarray, optional

Examples

Generate test 1-D parameter estimation problem. Model to produce predicted data is the identity map and observed signal comes from true value plus some random gaussian nose.

See *mud.examples.identity_uniform_1D_density_prob()* for more details

```
>>> from mud.examples import identity_uniform_1D_density_prob as I1D
```

First we set up a well-posed problem. Note the domain we are looking over contains our true value. We take 1000 samples, use 50 observations, assuming a true value of 0.5 populated with gaussian noise $\mathcal{N}(0, 0.5)$. Or initial uniform distribution is taken from a $[0, 1]$ range.

```
>>> D = I1D(1000, 50, 0.5, 0.05, domain=[0,1])
```

Estimate mud_point -> Note since WME map used, observed implied to be the standard normal distribution and does not have to be set explicitly from observed data set.

```
>>> np.round(D.mud_point()[0],1)
0.5
```

Expecation value of r, ratio of observed and predicted distribution, should be near 1 if predictabiltiy assumption is satisfied.

```
>>> np.round(D.expected_ratio(),0)
1.0
```

Set up ill-posed problem -> Searching out of range of true value

```
>>> D = I1D(1000, 50, 0.5, 0.05, domain=[0.6,1])
```

Mud point will be close as we can get within the range we are searching for

```
>>> np.round(D.mud_point()[0],1)
0.6
```

Expectation of r is close to zero since predictability assumption violated.

```
>>> np.round(D.expected_ratio(),1)
0.0
```

estimate()

Estimate

Returns the best estimate for most likely paramter values for the given model data using the data-consistent framework.

Returns
mud_point – Maximal Updated Density (MUD) point.

Return type
ndarray

expected_ratio()

Expectation Value of R

Returns the expectation value of the R, the ratio of the observed to the predicted density values.

$$R = \frac{\pi_{ob}(\lambda)}{\pi_{pred}(\lambda)} \tag{2.1}$$

If the predictability assumption for the data-consistent framework is satisfied, then $E[R] \approx 1$.

Returns
expected_ratio – Value of the E(r). Should be close to 1.0.

Return type
float

fit(***kwargs*)

Update Initial Distribution

Constructs the updated distribution by fiting osberved data to predicted data with:

$$\pi_{up}(\lambda) = \pi_{in}(\lambda)\frac{\pi_{ob}(Q(\lambda))}{\pi_{pred}(Q(\lambda))} \tag{2.2}$$

Note that if initial, predicted, and observed distributions have not been set before running this method, they will be run with default values. To set specific predicted, observed, or initial distributions use the `set_` methods.

Parameters
****kwargs** (*dict, optional*) – If specified, optional arguments are passed to the *set_predicted()* call in the case that the predicted distribution has not been set yet.

mud_point()

Maximal Updated Density (MUD) Point

Returns the Maximal Updated Density or MUD point as the parameter sample from the initial distribution with the highest update density value:

$$\lambda^{MUD} := \text{argmax}\pi_{up}(\lambda) \tag{2.3}$$

Note if the updated distribution has not been computed yet, this function will call *fit()* to compute it.

Returns
mud_point – Maximal Updated Density (MUD) point.

Return type
np.ndarray

property n_features

property n_params

property n_samples

plot_obs_space(*obs_idx: int = 0*, *ax: Optional[Axes] = None*, *y_range: Optional[ndarray] = None*, *aff=1000*, *ob_opts={'color': 'r', 'label': 'Observed', 'linestyle': '-', 'linewidth': 4}*, *pr_opts={'color': 'b', 'label': 'PF of Initial', 'linestyle': '--', 'linewidth': 4}*, *pf_opts={'color': 'k', 'label': 'PF of Updated', 'linestyle': '-.', 'linewidth': 4}*)

Plot probability distributions over parameter space

Observed distribution is plotted using the distribution function passed to *set_observed()* (or defaul). The predicted distribution is plotted using the stored predicted distribution function set in *set_predicted()*. The push-forward of the updated distribution is computed as a gkde on the predicted samples *y* as well, but using the product of the update ratio (2.1) and the initial weights as weights.

Parameters

- **obs_idx** (*int, default=0*) – Index of observable value to plot.
- **ax** (matplotlib.axes.Axes, optional) – Axes to plot distributions on. If non specified, a figure will be initialized to plot on.
- **y_range** (*list or np.ndarray, optional*) – Range over parameter value to plot over.
- **aff** (*int, default=100*) – Number of points to plot within x_range, evenly spaced.
- **ob_opts** (*dict, optional*) – Plotting option for observed distribution line. Defaults to `{'color':'r', 'linestyle':'-','linewidth':4, 'label':'Observed'}`. To supress plotting, pass in `None`.
- **pr_opts** (*dict, optional*) – Plotting option for predicted distribution line. Defaults to `{'color':'b', 'linestyle':'--','linewidth':4, 'label':'PF of Initial'}`. To supress plotting, pass in `None`.
- **pf_opts** (*dict, optional*) – Plotting option for push-forward of updated destribution line. Defaults to `{'color':'k', 'linestyle':'-.','linewidth':4, 'label':'PF of Updated'}`. To supress plotting, pass in `None`.

plot_param_space(*param_idx: int = 0*, *ax: Optional[Axes] = None*, *x_range: Optional[Union[ndarray, List[float]]] = None*, *aff: int = 1000*, *in_opts={'color': 'b', 'label': 'Initial', 'linestyle': '--', 'linewidth': 4}*, *up_opts={'color': 'k', 'label': 'Updated', 'linestyle': '-.', 'linewidth': 4}*, *win_opts={'color': 'g', 'label': 'Weighted Initial', 'linestyle': '--', 'linewidth': 4}*)

Plot probability distributions over parameter space

Initial distribution is plotted using the distribution function passed to *set_initial()*. The updated distribution is plotted using a weighted gaussian kernel density estimate (gkde) on the initial samples, using the product of the update ratio (2.1) value times the initial weights as weights for the gkde. The weighted initial is built using a weighted gkde on the initial samples, but only using the initial weights.

Parameters

- **param_idx** (*int, default=0*) – Index of parameter value to plot.
- **ax** (matplotlib.axes.Axes, optional) – Axes to plot distributions on. If non specified, a figure will be initialized to plot on.
- **x_range** (*list or np.ndarray, optional*) – Range over parameter value to plot over.
- **aff** (*int, default=100*) – Number of points to plot within x_range, evenly spaced.
- **in_opts** (*dict, optional*) – Plotting option for initial distribution line. Defaults to `{'color':'b', 'linestyle':'--','linewidth':4, 'label':'Initial'}`. To supress plotting, pass in `None` explicitly.
- **up_opts** (*dict, optional*) – Plotting option for updated distribution line. Defaults to `{'color':'k', 'linestyle':'-.','linewidth':4, 'label':'Updated'}`. To supress plotting, pass in `None` explicitly.
- **win_opts** (*dict, optional*) – Plotting option for weighted initial distribution line. Defaults to `{'color':'g', 'linestyle':'--','linewidth':4, 'label':'Weighted Initial'}`. To supress plotting, pass in `None` explicitly.

set_initial(*distribution: Optional[rv_continuous] = None*)

Set initial probability distribution of model parameter values $\pi_{in}(\lambda)$.

Parameters

distribution (*scipy.stats.rv_continuous, optional*) – scipy.stats continuous distribution object from where initial parameter samples were drawn from. If none provided, then a uniform distribution over domain of the density problem is assumed. If no domain is specified for density, then a standard normal distribution $N(0, 1)$ is assumed.

> **Warning:** Setting initial distribution resets the predicted and updated distributions, so make sure to set the initial first.

set_observed(*distribution: ~scipy.stats._distn_infrastructure.rv_continuous = <scipy.stats._distn_infrastructure.rv_frozen object>*)

Set distribution for the observed data.

The observed distribution is determined from assumptions on the collected data. In the case of using a weighted mean error map on sequential data from a single output, the distribution is stationary with respect to the number data points collected and will always be the standard normal d distribution $N(0,1)$.

Parameters

distribution (*scipy.stats.rv_continuous, default=scipy.stats.norm()*) – scipy.stats continuous distribution like object representing the likelihood of observed data. Defaults to a standard normal distribution N(0,1).

set_predicted(*distribution: Optional[rv_continuous] = None*, *bw_method: Optional[Union[str, Callable, generic]] = None*, *weights: Optional[ndarray] = None*, ***kwargs*)

Set Predicted Distribution

The predicted distribution over the observable space is equal to the push-forward of the initial through the model $\pi_{pr}(Q(\lambda)$. If no distribution is passed, `scipy.stats.gaussian_kde` is used over the predicted values `y` to estimate the predicted distribution.

Parameters

- **distribution** (`scipy.stats.rv_continuous`, optional) – If specified, used as the predicted distribution instead of the default of using gaussian kernel density estimation on observed values y. This should be a frozen distribution if using *scipy*, and otherwise be a class containing a *pdf()* method return the probability density value for an array of values.
- **bw_method** (*str, scalar, or Callable, optional*) – Method to use to calculate estimator bandwidth. Only used if distribution is not specified, See documentation for `scipy.stats.gaussian_kde` for more information.
- **weights** (*np.ndarray, optional*) – Weights to use on predicted samples. Note that if specified, *set_weights()* will be run first to calculate new weights. Otherwise, whatever was previously set as the weights is used. Note this defaults to a weights vector of all 1s for every sample in the case that no weights were passed on upon initialization.
- ****kwargs** (*dict, optional*) – If specified, any exra keyword arguments will be passed along to the passed `distribution.pdf()` function for computing values of predicted samples.
- **Note** (*distribution* should be a frozen distribution if using *scipy*.) –

> **Warning:** If passing a *distribution* argument, make sure that the initial distribution has been set first, either by having run *set_initial()* or *fit()* first.

set_weights(*weights: Union[ndarray, List[float]]*, *normalize: bool = False*)

Set Sample Weights

Sets the weights to use for each sample. Note weights can be one or two dimensional. If weights are two dimensional the weights are combined by multiplying them row wise and normalizing, to give one weight per sample. This combining of weights allows incorporating multiple sets of weights from different sources of prior belief.

Parameters

- **weights** (*np.ndarray, List[float]*) – Numpy array or list of same length as the *n_samples* or if two dimensional, number of columns should match *n_samples*
- **normalise** (*bool, default=False*) – Whether to normalize the weights vector.

> **Warning:** Resetting weights will delete the predicted and updated distribution values in the class, requiring a re-run of adequate *set_* methods and/or *fit()* to reproduce with new weights.

class mud.base.**IterativeLinearProblem**(*A*, *b*, *y=None*, *mu_i=None*, *cov=None*, *data_cov=None*, *idx_order=None*)

Bases: *LinearGaussianProblem*

get_errors(*ref_param*)

Get errors with resepct to a reference parameter

plot_chain(*ref_param*, *ax=None*, *color='k'*, *s=100*, ***kwargs*)

Plot chain of solutions and contours

plot_chain_error(*ref_param*, *ax=None*, *alpha=1.0*, *color='k'*, *label=None*, *s=100*, *fontsize=12*)

Plot error over iterations

solve(*num_epochs=1*, *method='mud'*)

Iterative Solutions Performs num_epochs iterations of estimates

class mud.base.**LinearGaussianProblem**(*A=array([[1], [1]])*, *b=None*, *y=None*, *mean_i=None*, *cov_i=None*, *cov_o=None*, *alpha=1.0*)

Bases: object

Sets up inverse problems with Linear/Affine Maps

Class provides solutions using MAP, MUD, and least squares solutions to the linear (or affine) problem from p parameters to d observables.

$$M(\mathbf{x}) = A\mathbf{x} + \mathbf{b}, A \in \mathbb{R}^{d\times p}, \mathbf{x}, \in \mathbb{R}^p, \mathbf{b}, \in \mathbb{R}^d, \tag{2.4}$$

A

2D Array defining kinear transformation from model parameter space to model output space.

Type
np.ndarray

y

1D Array containing observed values of Q(lambda) Array containing push-forward values of paramters samples through the forward model. These samples will form the *predicted distribution*.

Type
np.ndarray

domain

Array containing ranges of each paramter value in the parameter space. Note that the number of rows must equal the number of parameters, and the number of columns must always be two, for min/max range.

Type
np.ndarray

weights

Weights to apply to each parameter sample. Either a 1D array of the same length as number of samples or a 2D array if more than one set of weights is to be incorporated. If so the weights will be multiplied and normalized row-wise, so the number of columns must match the number of samples.

Type
np.ndarray, optional

Examples

Problem set-up:

$$A = \begin{bmatrix}1 & 1\end{bmatrix}, b = 0, y = 1\lambda_0 = \begin{bmatrix}0.25 & 0.25\end{bmatrix}^T, \Sigma_{init} = \begin{bmatrix}1 & -0.25\\ -0.25 & 0.5\end{bmatrix}, \Sigma_{obs} = \begin{bmatrix}0.25\end{bmatrix}$$

```
>>> from mud.base import LinearGaussianProblem as LGP
>>> lg1 = LGP(A=np.array([[1, 1]]),
...         b=np.array([[0]]),
...         y=np.array([[1]]),
...         mean_i=np.array([[0.25, 0.25]]).T,
...         cov_i=np.array([[1, -0.25], [-0.25, 0.5]]),
...         cov_o=np.array([[1]]))
>>> lg1.solve('mud')
array([[0.625],
       [0.375]])
```

compute_functionals(*X*, *terms='all'*)

For a given input and observed data, compute functionals or individual terms in functionals that are minimized to solve the linear gaussian problem.

property n_features

property n_params

property n_samples

plot_contours(*ref=None*, *subset=None*, *ax=None*, *annotate=False*, *note_loc=None*, *w=1*, *label='{i}'*, *plot_opts={'color': 'k', 'fs': 20, 'ls': ':', 'lw': 1}*, *annotate_opts={'fontsize': 20}*)

Plot Linear Map Solution Contours

plot_fun_contours(*mesh=None*, *terms='dc'*, *ax=None*, *N=250*, *r=1*, ***kwargs*)

Plot contour map offunctionals being minimized over input space

plot_sol(*point='mud'*, *ax=None*, *label=None*, *note_loc=None*, *pt_opts={'color': 'k', 'marker': 'o', 's': 100}*, *ln_opts={'color': 'xkcd:blue', 'lw': 1, 'marker': 'd', 'zorder': 10}*, *annotate_opts={'backgroundcolor': 'w', 'fontsize': 14}*)

Plot solution points

solve(*method='mud'*, *output_dim=None*)

Explicitly solve linear problem using given method.

mud.examples module

mud.examples.**identity_uniform_1D_density_prob**(*num_samples=2000*, *num_obs=20*, *y_true=0.5*, *noise=0.05*, *weights=None*, *domain=[0, 1]*, *wme_map=True*, *analytical_pred=True*)

1D Density Problem using WME on identity map with uniform initial

Sets up a Density Problem using a given domain (unit by default) and a uniform initial distribution under an identity map and the Weighted Mean Error map to . This function is used as a set-up for tejjsts to the DensityProblem class.

num_obs observations are collected from an initial distribution and used as the true signal, with noise being added to each observation. Sets up an inverse problem using the unit domain and uniform distribution under an

identity map. This is equivalent to studying a "steady state" signal over time, or taking repeated measurements of the same quantity to reduce variance in the uncertainty.

mud.examples.**rotation_map**(*qnum=10*, *tol=0.1*, *b=None*, *ref_param=None*, *seed=None*)

Generate test data linear rotation map

mud.examples.**rotation_map_trials**(*numQoI=10*, *method='ordered'*, *num_trials=100*, *model_eval_budget=100*, *ax=None*, *color='r'*, *label='Ordered QoI $(10\\times 10D)$'*, *seed=None*)

Run a set of trials for linear rotation map problems

mud.funs module

Python console script for *mud*, installed with *pip install .* or *python setup.py install*

mud.funs.**check_args**(*A*, *b*, *y*, *mean*, *cov*, *data_cov*)

mud.funs.**iterate**(*A*, *b*, *y*, *initial_mean*, *initial_cov*, *data_cov=None*, *num_epochs=1*, *idx=None*)

mud.funs.**main**(*args*)

Main entry point allowing external calls

Parameters
args (*[str]*) – command line parameter list

mud.funs.**makeRi**(*A*, *initial_cov*)

mud.funs.**map_problem**(*lam*, *qoi*, *qoi_true*, *domain*, *sd=0.05*, *num_obs=None*, *log=False*)

Wrapper around map problem, takes in raw qoi + synthetic data and instantiates solver object

mud.funs.**map_sol**(*A*, *b*, *y=None*, *mean=None*, *cov=None*, *data_cov=None*, *w=1*)

mud.funs.**map_sol_with_cov**(*A*, *b*, *y=None*, *mean=None*, *cov=None*, *data_cov=None*, *w=1*)

mud.funs.**mud_problem**(*lam*, *qoi*, *qoi_true*, *domain*, *sd=0.05*, *num_obs=None*, *split=None*, *weights=None*)

Wrapper around mud problem, takes in raw qoi + synthetic data and performs WME transformation, instantiates solver object.

mud.funs.**mud_sol**(*A*, *b*, *y=None*, *mean=None*, *cov=None*, *data_cov=None*)

For SWE problem, we are inverting N(0,1). This is the default value for *data_cov*.

mud.funs.**mud_sol_with_cov**(*A*, *b*, *y=None*, *mean=None*, *cov=None*, *data_cov=None*)

Doesn't use R directly, uses new equations. This presents the equation as a rank-k update to the error of the initial estimate.

mud.funs.**parse_args**(*args*)

Parse command line parameters

Parameters
args (*[str]*) – command line parameters as list of strings

Returns
command line parameters namespace

Return type
argparse.Namespace

mud.funs.**performEpoch**(*A*, *b*, *y*, *initial_mean*, *initial_cov*, *data_cov=None*, *idx=None*)

mud.funs.**run**()

Entry point for console_scripts

mud.funs.**setup_logging**(*loglevel*)

Setup basic logging

Parameters
loglevel (*int*) – minimum loglevel for emitting messages

mud.funs.**updated_cov**(*X*, *init_cov=None*, *data_cov=None*)

We start with the posterior covariance from ridge regression Our matrix R = init_cov^(-1) - X.T @ pred_cov^(-1) @ X replaces the init_cov from the posterior covariance equation. Simplifying, this is given as the following, which is not used due to issues of numerical stability (a lot of inverse operations).

up_cov = (X.T @ np.linalg.inv(data_cov) @ X + R)^(-1) up_cov = np.linalg.inv(X.T@(np.linalg.inv(data_cov) - inv_pred_cov)@X + np.linalg.inv(init_cov))

We return the updated covariance using a form of it derived which applies Hua's identity in order to use Woodbury's identity.

```
>>> updated_cov(np.eye(2))
array([[1., 0.],
       [0., 1.]])
>>> updated_cov(np.eye(2)*2)
array([[0.25, 0.  ],
       [0.  , 0.25]])
>>> updated_cov(np.eye(3)[:, :2]*2, data_cov=np.eye(3))
array([[0.25, 0.  ],
       [0.  , 0.25]])
>>> updated_cov(np.eye(3)[:, :2]*2, init_cov=np.eye(2))
array([[0.25, 0.  ],
       [0.  , 0.25]])
```

mud.funs.**wme**(*predictions*, *data*, *sd=None*)

Calculates Weighted Mean Error (WME) functional.

Parameters

- **predictions** (*numpy.ndarray of shape (n_samples, n_features)*) – Predicted values against which data is compared.
- **data** (*list or numpy.ndarray of shape (n_features, 1)*) – Collected (noisy) data
- **sd** (*float, optional*) – Standard deviation

Return type
numpy.ndarray of shape (n_samples, 1)

mud.norm module

mud.norm.**full_functional**(*operator*, *inputs*, *data*, *initial_mean*, *initial_cov*, *observed_mean=0*, *observed_cov=1*)

mud.norm.**inner_product**(*X*, *mat*)

Inner-product induced vector norm implementation.

Returns square of norm defined by the inner product `(x,x)_C := x^T C^-1 x`

Parameters

- **X** (*(M, N) array_like*) – Input array. N = number of samples, M = dimension
- **mat** (*(M, M) array_like*) – Positive-definite operator which induces the inner product

Returns
Z – inner-product of each column in `X` with respect to `mat`

Return type
(N, 1) ndarray

mud.norm.**norm_data**(*operator*, *inputs*, *data*, *observed_mean*, *observed_cov*)

mud.norm.**norm_input**(*inputs*, *initial_mean*, *initial_cov*)

mud.norm.**norm_predicted**(*operator*, *inputs*, *initial_mean*, *initial_cov*)

mud.plot module

mud.plot.**make_2d_normal_mesh**(*N=50*, *window=1*)

mud.plot.**make_2d_unit_mesh**(*N=50*, *window=1*)

mud.plot.**plotChain**(*mud_chain*, *ref_param*, *color='k'*, *s=100*)

mud.plot.**plot_contours**(*A*, *ref_param*, *subset=None*, *color='k'*, *ls=':'*, *lw=1*, *fs=20*, *w=1*, *s=100*, ***kwds*)

mud.util module

mud.util.**make_2d_normal_mesh**(*N: int = 50*, *window: int = 1*)

Constructs mesh based on normal distribution to discretize each axis. >>> from mud.util import make_2d_normal_mesh >>> x, y, XX = make_2d_normal_mesh(3) >>> print(XX) [[-1. -1.]

[0. -1.] [1. -1.] [-1. 0.] [0. 0.] [1. 0.] [-1. 1.] [0. 1.] [1. 1.]]

mud.util.**make_2d_unit_mesh**(*N: int = 50*, *window: int = 1*)

Make 2D Unit Mesh

Constructs mesh based on uniform distribution to discretize each axis.

Parameters

- **N** (*int, default=50*) – Size of unit mesh. *N* points will be generated in each x,y direction.
- **window** (*int, defalut=1*) – Upper bound of mesh. Lower bound fixed at 0 always.

Returns

- **grid** (*tuple of np.ndarray*) – Tuple of *(X, Y, XX)*, the grid *X* and *Y* and 2D mesh *XX*

- *Example Usage*
- ————-
- >>> *from mud.util import make_2d_unit_mesh*
- >>> *x, y, XX = make_2d_unit_mesh(3)*
- >>> *print(XX)*
- *[[0. 0.]* – [0.5 0.] [1. 0.] [0. 0.5] [0.5 0.5] [1. 0.5] [0. 1.] [0.5 1.] [1. 1.]]

mud.util.**null_space**(*A: ndarray*, *rcond: Optional[float] = None*)

Construct an orthonormal basis for the null space of A using SVD

Method is slight modification of `scipy.linalg`

Parameters

- **A** (*(M, N) array_like*) – Input array
- **rcond** (*float, optional*) – Relative condition number. Singular values `s` smaller than `rcond * max(s)` are considered zero. Default: floating point eps * max(M,N).

Returns

Z – Orthonormal basis for the null space of A. K = dimension of effective null space, as determined by rcond

Return type

(N, K) ndarray

Examples

One-dimensional null space:

```
>>> import numpy as np
>>> from mud.util import null_space
>>> A = np.array([[1, 1], [1, 1]])
>>> ns = null_space(A)
>>> ns * np.sign(ns[0,0])  # Remove the sign ambiguity of the vector
array([[ 0.70710678],
       [-0.70710678]])
```

Two-dimensional null space:

```
>>> B = np.random.rand(3, 5)
>>> Z = null_space(B)
>>> Z.shape
(5, 2)
>>> np.allclose(B.dot(Z), 0)
True
```

The basis vectors are orthonormal (up to rounding error):

```
>>> np.allclose(Z.T.dot(Z), np.eye(2))
True
```

mud.util.**set_shape**(*array: ndarray*, *shape: Union[List, Tuple] = (1, - 1)*) → ndarray

Resizes inputs if they are one-dimensional.

mud.util.**std_from_equipment**(*tolerance=0.1*, *probability=0.95*)

Converts tolerance *tolerance* for precision of measurement equipment to a standard deviation, scaling so that (100`probability`) percent of measurements are within *tolerance*. A mean of zero is assumed. *erfinv* is imported from *scipy.special*

mud.util.**transform_linear_map**(*operator: ndarray*, *data: Union[ndarray, List[float], Tuple[float]]*, *std: Union[ndarray, float, List[float], Tuple[float]]*)

Takes a linear map *operator* of size (len(data), dim_input) or (1, dim_input) for repeated observations, along with a vector *data* representing observations. It is assumed that *data* is formed with *M@truth + sigma* where *sigma ~ N(0, std)*

This then transforms it to the MWE form expected by the DCI framework. It returns a matrix *A* of shape (1, dim_input) and np.float *b* and transforms it to the MWE form expected by the DCI framework.

```
>>> X = np.ones((10, 2))
>>> x = np.array([0.5, 0.5]).reshape(-1, 1)
>>> std = 1
>>> d = X @ x
>>> A, b = transform_linear_map(X, d, std)
>>> np.linalg.norm(A @ x + b)
0.0
>>> A, b = transform_linear_map(X, d, [std]*10)
>>> np.linalg.norm(A @ x + b)
0.0
>>> A, b = transform_linear_map(np.array([[1, 1]]), d, std)
>>> np.linalg.norm(A @ x + b)
0.0
>>> A, b = transform_linear_map(np.array([[1, 1]]), d, [std]*10)
Traceback (most recent call last):
...
ValueError: For repeated measurements, pass a float for std
```

mud.util.**transform_linear_setup**(*operator_list: List[ndarray]*, *data_list: Union[List[ndarray], Tuple[ndarray]]*, *std_list: Union[float, ndarray, List[float], Tuple[float], Tuple[Tuple[float]], List[List[float]]]*)

Module contents

2.3 License

HOLDERS BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER LIABILITY, WHETHER IN AN ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING FROM, OUT OF OR IN CONNECTION WITH THE SOFTWARE OR THE USE OR OTHER DEALINGS IN THE SOFTWARE.

2.4 Contributors

- Mathematical Michael <consistentbayes@gmail.com>

2.5 Changelog

2.5.1 Versions 0.0.x

- Setting up initial repository, configuring CI/CD
- Migration of code from CU-Denver-UQ/mud-paper repo
- Revisions of architecture, moving modules around
- Rapid iteration, not sticking to semantic versioning
- Possible breaking versions between patches (some functions moved to *mud-examples*)
- Defines basic functionality, classes, helpful functions

2.5.2 Version 0.0.25

- Updated packaging to comply with PEP 517/518 using *pyscaffold `v4.0.2*
- Removes pyerf in favor of erfinv from *scipy.special* (available since *v0.2*)
- Renames *testing* to *dev* for optional dependency installation
- Adds *black* as a *dev* dependency
- Run *black* + *flake8* on whole project
- clean up *setup.cfg* file
- adds file for readthedocs

2.5.3 Version 0.0.26

- Read the Docs set up, documentation infrastructure.

2.5.4 Version 0.0.27

- Adding docstrings
- Removing *plot* module. *mud-examples* already has it.
- Fixing CHANGELOG typos with version numbers.
- Update README
- Update project description + metadata in *setup.cfg*
- *sphinx_copybutton* extension added

2.5.5 Version 0.1

- Basic functionality and repo complete with information
- Beginning of adherence to semantic versioning rules
- i.e., breaking changes in major revision, contract changes in minor, bugfixes/features in patch.

CHAPTER

THREE

INDICES AND TABLES

- genindex
- modindex
- search

PYTHON MODULE INDEX

m

INDEX

P

R

S

T

U

W

X

Y