mud Documentation

Release 0.1.1

Mathematical Michael

Jan 22, 2024

CONTENTS

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

Warning: This website is under active construction. Please report incomplete documentation. Last edited: Jan 22, 2024

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

Subpackages

mud.examples package

Submodules

mud.examples.adcirc module

mud.examples.comparison module

MUD vs MAP Comparison Example

Functions for running 1-dimensional polynomial inversion problem.

mud.examples.comparison.**comparison_plot**(*d_prob: DensityProblem*, *b_prob: BayesProblem*, *space: str = 'param'*, *ax: Axes | None = None*)

Generate plot comparing MUD vs MAP solution

Parameters

- **d_prob** (mud.base.DensityProblem) – DensityProblem object that has been solved already with d_prob.estimate() or another such method.
- **b_prob** (mud.base.BayesProblem) – BayesProblem object that has been solved already with b_prob.estimate() or another such method.
- **space** (*str, default="param"*) – What space to plot. Default is "param" to plot the parameter, or input, space, and thus the updated parameter distributions and associated MUD/MAP solutions. Any other value will plot the observable space, which includes the predicted distribution for the DensityProblem and the data-likelihood distribution for the BayesProblem.
- **ax** (*matplotlib.pyplot.Axes, optional*) – Existing matplotlib Axes object to plot onto. If none provided (default), then a figure is initialized.

Returns

ax – Axes object that was plotted onto or created.

Return type

matplotlib.pyplot.Axes

mud.examples.comparison.**run_comparison_example**(*p: int = 5*, *num_samples: int = 1000*, *mu: float = 0.25*, *sigma: float = 0.1*, *domain: List[int] = [-1, 1]*, *N_vals: List[int] = [1, 5, 10, 20]*, *latex_labels: bool = True*, *save_path: str | None = None*, *dpi: int = 500*, *close_fig: bool = False*)

Run MUD vs MAP Comparison Example

Entry-point function for running simple polynomial example comparing Bayesian and Data-Consistent solutions. Inverse problem involves inverting the polynomial QoI map Q(lam) = lam^5.

Parameters

- **p** (*int, default=5*) – Power of polynomial in *QoI map*.
- **num_samples** (*int, default=100*) – Number of λ samples to generate from a uniform distribution over `domain` for solving inverse problem.
- **mu** (*float, default=0.25*) – True mean value of observed data.
- **sigma** (*float, default=0.1*) – Standard deviation of observed data.
- **domain** (numpy.typing.ArrayLike, default=[[-1, 1]]) – Domain to draw lambda samples from.
- **N_vals** (*List[int], default=[1, 5, 10, 20]*) – Values for N, the number of data-points to use to solve inverse problems, to use. Each N value will produce two separate plots.

- **save_path** (*str, optional*) – If provided, path to save the resulting figure to.
- **dpi** (*int, default=500*) – Resolution of images saved
- **close_fig** (*bool , default=False*) – Set to True to close figure and only save it. Useful when running in notebook environments.

Returns

res – matplotlib.axes.Axes]] List of Tuples of `(d_prob, b_prob, ax)` containing the resulting Density and Bayesian problem objects in `d_prob` and `b_prob`, resp., and the matplotlib axis to which the results were plotted, for each N case run.

Return type

List[Tuple[*mud.base.DensityProblem*, *mud.base.BayesProblem*,

mud.examples.examples module

mud.examples.exp_decay module

Exponential Decay Example

mud.examples.exp_decay.**exp_decay_1D**(*u_0=0.75*, *time_range=[0, 4.0]*, *domain=[0, 1]*, *num_samples=10000*, *lambda_true=0.5*, *t_start=0.0*, *sampling_freq=100.0*, *std_dev=0.05*)

mud.examples.exp_decay.**exp_decay_2D**(*time_range=[0, 3.0]*, *domain=array([[0.7, 0.8], [0.25, 0.75]])*, *num_samples=100*, *lambda_true=[0.75, 0.5]*, *N=100*, *t_start=0.0*, *sampling_freq=10.0*, *std_dev=0.05*)

mud.examples.fenics module

mud.examples.linear module

MUD Linear Examples

Functions for examples for linear problems.

mud.examples.linear.**call_comparison**(*lin_prob:* LinearGaussianProblem, ***kwargs*)

mud.examples.linear.**call_consistent**(*lin_prob:* LinearGaussianProblem, ***kwargs*)

mud.examples.linear.**call_map**(*lin_prob:* LinearGaussianProblem, ***kwargs*)

mud.examples.linear.**call_mismatch**(*lin_prob:* LinearGaussianProblem, ***kwargs*)

mud.examples.linear.**call_mud**(*lin_prob:* LinearGaussianProblem, ***kwargs*)

mud.examples.linear.**call_tikhonov**(*lin_prob:* LinearGaussianProblem, ***kwargs*)

mud.examples.linear.**noisy_linear_data**(*M*, *reference_point*, *std*, *num_data=None*)

Creates data produced by model assumed to be of the form:

Q(lam) = M * lam + odj,i =Mj(†)+i, i N(0,j2), 1iNj

mud.examples.linear.**random_linear_problem**(*dim_input: int = 10*, *dim_output: int = 10*, *mean_i: _SupportsArray[dtype] | _NestedSequence[_SupportsArray[dtype]] | bool | int | float | complex | str | bytes | _NestedSequence[bool | int | float | complex | str | bytes] | None = None*, *cov_i: _SupportsArray[dtype] | _NestedSequence[_SupportsArray[dtype]] | bool | int | float | complex | str | bytes | _NestedSequence[bool | int | float | complex | str | bytes] | None = None*, *seed: int | None = None*)

Construct a random linear Gaussian Problem

mud.examples.linear.**random_linear_wme_problem**(*reference_point*, *std_dev*, *num_qoi=1*, *num_observations=10*, *dist='normal'*, *repeated=False*)

Create a random linear WME problem

Parameters

- **reference_point** (*np.ndarray*) – Reference true parameter value.
- **dist** (*str, default='normal'*) – Distribution to draw random linear map from. 'normal' or 'uniform' supported at the moment.
- **num_qoi** (*int, default = 1*) – Number of QoI
- **num_observations** (*int, default = 10*) – Number of observation data points.
- **std_dev** (*np.ndarray, optional*) – Standard deviation of normal distribution from where observed data points are drawn from. If none specified, noise-less data is created.

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

Generate test data linear rotation map

mud.examples.linear.**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.examples.linear.**run_contours**(*plot_fig: List[str] | None = None*, *save_path: str | None = None*, *dpi: int = 500*, *close_fig: bool = False*, ***kwargs*)

Run Contours

Produces contour plots of 2-D parameter space for 2-to-1 linear map found in Figures 3 and 5 of [ref]. These contour plots show the different regularization terms between Bayesian and Data-Consistent solutions that lead to a different optimization problem and therefore a different solution to the inverse problem.

Parameters

- **plot_fig** (*str, default='all'*) – Which figures to produce. Possible options are data_mismatch,
- **save_path** (*str, optional*) – If provided, path to save the resulting figure to.
- **dpi** (*int, default=500*) – Resolution of images saved
- **close_fig** (*bool, default=False*) – Set to True to close figure and only save it.
- **kwargs** (*dict, optional*) – kwargs to overwrite default arguments used to build linear problem. Possible values include and their expected types, and default values:
  - A : 2D array, default=[[1, 1]]
  - b - 1D array, default = [0]

- y - 1D array, default =[1]
- mean_i - 1D array, default = [0.25, 0.25]
- cov_i - 2D array, default = [[1, -0.25], [-0.25, 0.5]]
- cov_o - 1D array, default = [1]

Returns
lin_prob – LinearGaussianProblem object with solved linear inverse problem and associated data within.

Return type
mud.base.LinearGaussianProblem

mud.examples.linear.**run_high_dim_linear**(*dim_input: int = 100*, *dim_output: int = 100*, *seed: int = 21*, *save_path: str | None = None*, *dpi: int = 500*, *close_fig: bool = True*)

Run High Dimension Linear Example

Reproduces Figure 6 from [ref], showing the relative error between the true parameter value and the MUD, MAP and least squares solutions to linear gaussian inversion problems for increasing dimension and rank of a randomly generated linear map A.

Parameters

- **dim_input** (*int, default=20*) – Input dimension of linear map (number of rows in A).
- **dim_output** (*int, default=5*) – Output dimension of linear map (number of columns in A).
- **seed** (*int, default = 21*) – To fix results for reproducibility. Set to None to randomize results.
- **save_path** (*str, optional*) – If provided, path to save the resulting figure to.
- **dpi** (*int, default=500*) – Resolution of images saved
- **close_fig** (*bool, default=False*) – Set to True to close figure and only save it.

Returns
rank_errs, dim_errs – Tuple containing the error between the true solution and each of the (mud, map, least_squares) solutions for increasing dimension and rank from 1 to dim_output. These arrays are used to produce the plots given.

Return type
Tuple[np.array, np.array]

mud.examples.linear.**run_wme_covariance**(*dim_input: int = 20*, *dim_output: int = 5*, *sigma: float = 0.1*, *Ns: List[int] = [10, 100, 1000, 10000]*, *seed: int | None = None*, *save_path: str | None = None*, *dpi: int = 500*, *close_fig: bool = False*)

Weighted Mean Error Map Updated Covariance

Reproduces figure 4 from [ref], showing the spectral properties of the updated covriance for a the Weighted Mean Error map on a randomly generated linear operator as more data from repeated measurements is used to constructthe QoI map.

Parameters

- **dim_input** (*int, default=20*) – Input dimension of linear map (number of rows in A).
- **dim_output** (*int, default=5*) – Output dimension of linear map (number of columns in A).

- **sigma** (*float, default=1e-1*) – N(0, sigma) error added to produce "measurements" from linear operator.
- **Ns** (*List[str]. default = [10, 100, 1000, 10000]*) – List of number of data points to collect in constructing Q_WME map to view how the spectral properties of the updated covariance change as more data is included in the Q_WME map.
- **seed** (*int, default = 21*) – To fix results for reproducibility. Set to None to randomize results.
- **save_path** (*str, optional*) – If provided, path to save the resulting figure to.
- **dpi** (*int, default=500*) – Resolution of images saved
- **close_fig** (*bool, default=False*) – Set to True to close figure and only save it.

Returns

linear_wme_prob, ax – Tuple containing solved linear WME problems for each Ns value, and axes containing the plot of the first 20 eigenvalues of the updated covariances for each Q_WME map.

Return type

Tuple[*mud.base.LinearWMEProblem*, matplotlib.pyplot.Axes]

mud.examples.poisson module

mud.examples.simple module

Simple Example

mud.examples.simple.**identity_1D_bayes_prob**(*num_samples=1000*, *num_obs=50*, *mu=0.5*, *sigma=0.05*, *init_dist=<scipy.stats._distn_infrastructure.rv_continuous_frozen object>*, *domain=None*)

Identity 1D Bayes Problem

Solving 1d identity map parameter estimation problem using the BayesProlem class and the map point estimate.

mud.examples.simple.**identity_1D_density_prob**(*num_samples=2000*, *num_obs=20*, *mu=0.5*, *sigma=0.05*, *weights=None*, *init_dist='uniform'*, *normalize=False*, *domain=[0, 1]*, *analytical_pred=True*)

Identity 1D Density Problem

Solving 1d identity map parameter estimation problem using the DensityProblem class and the mud point estimate.

mud.examples.simple.**identity_1D_temporal_prob**(*num_samples=2000*, *num_obs=20*, *y_true=0.5*, *noise=0.05*, *weights=None*, *init_dist=<scipy.stats._distn_infrastructure.rv_continuous_frozen object>*, *normalize=False*, *domain=None*, *wme_map=True*, *analytical_pred=True*)

Identity 1D Temporal Problem

Solving 1d identity map parameter estimation problem using the SpatioTemporalProblem class to construct DensityProblem class using the WME map to aggregate data.

mud.examples.simple.**polynomial_1D_data**(*p: int = 5, num_samples: int = 1000, domain: ~numpy._typing._array_like._SupportsArray[~numpy.dtype] | ~numpy._typing._nested_sequence._NestedSequence[~numpy._typing._array_like._S | bool | int | float | complex | str | bytes | ~numpy._typing._nested_sequence._NestedSequence[bool | int | float | complex | str | bytes] = [[-1, 1]], init_dist=<scipy.stats._distn_infrastructure.rv_continuous_frozen object>, mu: float = 0.25, sigma: float = 0.1, N: int = 1*)

Polynomial 1D QoI Map

Generates test data for an inverse problem involving the polynomial QoI map

$$Q_p(lambda) = lambda^p \tag{2.1}$$

Where the uncertain parameter to be determined is λ. `num_samples` samples from a uniform distribution over `domain` are generated using `numpy.random.uniform()` and pushed through the *forward model*. `N` observed data points are generated from a normal distribution centered at `mu` with standard deviation `sigma` using `scipy.stats.norm`.

Parameters

- **p** (*int, default=5*) – Power of polynomial in *QoI map*.
- **num_samples** (*int, default=100*) – Number of λ samples to generate from a uniform distribution over `domain` for solving inverse problem.
- **domain** (`numpy.typing.ArrayLike`, default=[[-1, 1]]) – Domain to draw lambda samples from.
- **mu** (*float, default=0.25*) – True mean value of observed data.
- **sigma** (*float, default=0.1*) – Standard deviation of observed data.
- **N** (*int, default=1*) – Number of data points to generate from observed distribution. Note if 1, the default value, then the singular drawn value will always be `mu`.

Returns

data – Tuple of `(lam, q_lam, data)` where `lam` is contains the λ samples, `q_lam` the value of $Q_p(\lambda)$, and `data` the observed data values from the $\mathcal{N}(\mu, \sigma)$ distribution.

Return type

Tuple[`numpy.ndarray`,]

Examples

Note when N=1, data point drawn is always equal to mean.

```
>>> import numpy as np
>>> from mud.examples.comparison import polynomial_1D_data
>>> lam, q_lam, data = polynomial_1D_data(num_samples=10, N=1)
>>> data[0]
0.25
>>> len(lam)
10
```

For higher values of N, values are drawn from N(mu, sigma) distribution.

```
>>> lam, q_lam, data = polynomial_1D_data(N=10, mu=0.5, sigma=0.01)
>>> len(data)
10
>>> np.mean(data) < 0.6
True
```

Module contents

Submodules

mud.base module

class mud.base.**BayesProblem**(*X: ndarray | List*, *y: ndarray | List*, *domain: ndarray | List | None = 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 parameters samples through the forward model. These samples will form the data-likelihood distribution.
- **domain** (*array_like, optional*) – 2D Array containing ranges of each parameter 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}*, *map_opts={'color': 'g', 'label': '$\\lambda^\\mathrm{MAP}$'}*, *true_opts={'color': 'r', 'label': '$\\lambda^{\\dagger}$', 'linestyle': '-.'}*, *true_val=None*)

Plot probability distributions over parameter space

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

set_prior(*distribution=None*)

class mud.base.**DensityProblem**(*X: _SupportsArray[dtype] | _NestedSequence[_SupportsArray[dtype]] | bool | int | float | complex | str | bytes | _NestedSequence[bool | int | float | complex | str | bytes]*, *y: _SupportsArray[dtype] | _NestedSequence[_SupportsArray[dtype]] | bool | int | float | complex | str | bytes | _NestedSequence[bool | int | float | complex | str | bytes]*, *domain: _SupportsArray[dtype] | _NestedSequence[_SupportsArray[dtype]] | bool | int | float | complex | str | bytes | _NestedSequence[bool | int | float | complex | str | bytes] | None = None*, *weights: _SupportsArray[dtype] | _NestedSequence[_SupportsArray[dtype]] | bool | int | float | complex | str | bytes | _NestedSequence[bool | int | float | complex | str | bytes] | None = None*, *normalize: bool = False*, *pad_domain: float = 0.1*)

Bases: `object`

Sets up Data-Consistent Inverse Problem for parameter identification

Data-Consistent inversion is a way to infer most likely model parameters 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
ArrayLike

y

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

Type
ArrayLike

domain

Array containing ranges of each parameter 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
ArrayLike

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
ArrayLike, 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.simple import identity_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
```

Expectation value of r, ratio of observed and predicted distribution, should be near 1 if predictability 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 parameter values for the given model data using the data-consistent framework.

Returns
mud_point – Maximal Updated Density (MUD) point.

Return type
np.ndarray

`expected_ratio()`

Expectation Value of R

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

$$R = \frac{\pi_{ob}(\lambda)}{\pi_{pred}(\lambda)} \tag{2.2}$$

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 fitting observed data to predicted data with:

$$\pi_{up}(\lambda) = \pi_{in}(\lambda)\frac{\pi_{ob}(Q(\lambda))}{\pi_{pred}(Q(\lambda))} \tag{2.3}$$

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:

$$\lambda^{MUD} := \text{argmax}\pi_{up}(\lambda) \tag{2.4}$$

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: Axes | None = None*, *y_range: _SupportsArray[dtype] | _NestedSequence[_SupportsArray[dtype]] | bool | int | float | complex | str | bytes | _NestedSequence[bool | int | float | complex | str | bytes] | None = None*, *aff=100*, *ob_opts={'color': 'r', 'label': '$\\pi_\\mathrm{obs}$', 'linestyle': '-'}*, *pr_opts={'color': 'b', 'label': '$\\pi_\\mathrm{pred}$', 'linestyle': '-'}*, *pf_opts={'color': 'k', 'label': '$\\pi_\\mathrm{pf-pr}$', 'linestyle': '-.'}*)

Plot probability distributions over parameter space

Observed distribution is plotted using the distribution function passed to *set_observed()* (or default). 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 (2.2) 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 suppress 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 suppress plotting, pass in `None`.
- **pf_opts** (*dict, optional*) – Plotting option for push-forward of updated distribution line. Defaults to `{'color':'k', 'linestyle':'-.','linewidth':4, 'label':'PF of Updated'}`. To suppress plotting, pass in `None`.

plot_param_space(*param_idx: int = 0*, *true_val: _SupportsArray[dtype] | _NestedSequence[_SupportsArray[dtype]] | bool | int | float | complex | str | bytes | _NestedSequence[bool | int | float | complex | str | bytes] | None = None*, *ax: Axes | None = None*, *x_range: list | ndarray | None = None*, *ylim: float | None = None*, *pad_ratio: float = 0.05*, *aff: int = 100*, *in_opts={'color': 'b', 'label': '$\\pi_\\mathrm{init}$', 'linestyle': '-'}*, *up_opts={'color': 'k', 'label': '$\\pi_\\mathrm{update}$', 'linestyle': '-.'}*, *win_opts=None*, *mud_opts={'color': 'g', 'label': '$\\lambda^\\mathrm{MUD}$'}*, *true_opts={'color': 'r', 'label': '$\\lambda^{\\dagger}$'}*)

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 (2.2) 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 suppress 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 suppress 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 suppress plotting, pass in `None` explicitly.

plot_params_2d(*x_1: int = 0, x_2: int = 1, y: int = 0, contours: bool = False, colorbar: bool = True, ax: Axes | None = None, label=True, **kwargs*)

2D plot over two indices of param space, optionally contoured by a y value.

Parameters

- **x_1** (*int, default=0*) – Index of param value (column of X) to use as x value in plot.
- **x_2** (*int, default=1*) – Index of param value (column of X) to use as y value in plot.
- **y** (*int, optional*) – Index of observable (column of y) to use as z value in contour plot.
- **ax** (`matplotlib.axes.Axes`, optional) – Axes to plot distributions on. If non specified, a figure will be initialized to plot on.
- **kwargs** (*dict, optional*) – Additional keyword arguments will be passed to *matplotlib.pyplot.scatter*.

plot_qoi(*idx_x: int = 0, idx_y: int = 1, ax: Axes | None = None, **kwargs*)

Plot 2D plot over two indices of y space.

Parameters

- **idx_x** (*int, default=0*) – Index of observable value (column of y) to use as x value in plot.
- **idx_y** (*int, default=1*) – Index of observable value (column of y) to use as y value in plot.
- **ax** (`matplotlib.axes.Axes`, optional) – Axes to plot distributions on. If non specified, a figure will be initialized to plot on.
- **kwargs** (*dict, optional*) – Additional keyword arguments will be passed to *matplotlib.pyplot.scatter*.

set_initial(*distribution: rv_continuous | None = 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_continuous_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: rv_continuous | None = None, bw_method: str | Callable | generic | None = None, weights: _SupportsArray[dtype] | _NestedSequence[_SupportsArray[dtype]] | bool | int | float | complex | str | bytes | _NestedSequence[bool | int | float | complex | str | bytes] | None = 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** (*ArrayLike, 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 extra 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: _SupportsArray[dtype] | _NestedSequence[_SupportsArray[dtype]] | bool | int | float | complex | str | bytes | _NestedSequence[bool | int | float | complex | str | bytes] | None = None, 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*) – Numpy array or list of same length as the *n_samples* or if two dimensional, number of columns should match *n_samples*
- **normalize** (*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 respect 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.

$$M(\mathbf{x}) = A\mathbf{x} + \mathbf{b}, A \in \mathbb{R}^{d\times p}, \mathbf{x}, \in \mathbb{R}^p, \mathbf{b}, \in \mathbb{R}^d, \tag{2.5}$$

A

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

Type
ArrayLike

y

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

Type
ArrayLike

domain

Array containing ranges of each parameter 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
ArrayLike

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
ArrayLike, 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 of functionals 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.

updated_cov(*A=None*, *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.

Check using alternate for updated_covariance.

```
>>> from mud.base import LinearGaussianProblem as LGP
>>> lg2 = LGP(A=np.eye(2))
>>> lg2.updated_cov()
array([[1., 0.],
       [0., 1.]])
>>> lg3 = LGP(A=np.eye(2)*2)
>>> lg3.updated_cov()
array([[0.25, 0.  ],
       [0.  , 0.25]])
>>> lg3 = LGP(A=np.eye(3)[:, :2]*2)
>>> lg3.updated_cov()
array([[0.25, 0.  ],
       [0.  , 0.25]])
>>> lg3 = LGP(A=np.eye(3)[:, :2]*2, cov_i=np.eye(2))
>>> lg3.updated_cov()
array([[0.25, 0.  ],
       [0.  , 0.25]])
```

class mud.base.**LinearWMEProblem**(*operators*, *data*, *sigma*, *y=None*, *mean_i=None*, *cov_i=None*, *cov_o=None*, *alpha=1.0*)

Bases: *LinearGaussianProblem*

Linear Inverse Problems using the Weighted Mean Error Map

class mud.base.**SpatioTemporalProblem**(*df=None*)

Bases: object

Class for parameter estimation problems related to spatio-temporal problems. equation models of real world systems. Uses a QoI map of weighted residuals between simulated data and measurements to do inversion

TODO

Type
Finish

TODO: Finish

property data

property domain

get_closest_to_measurements(*samples_mask=None*, *times_mask=None*, *sensors_mask=None*)

Get closest simulated data point to measured data in l^2-norm.

get_closest_to_true_vals()

Get closest simulated data point to noiseless true values in l^2-norm.

Note for now no sub-sampling implemented here.

property lam

property lam_ref

load(*df*, *lam='lam'*, *data='data'*, ***kwargs*)

Load data from a file on disk for a PDE parameter estimation problem.

Parameters
fname (*str*) – Name of file on disk. If ends in '.nc' then assumed to be netcdf file and the xarray library is used to load it. Otherwise the data is assumed to be pickled data.

Returns
ds – Dictionary containing data from file for PDE problem class

Return type
dict,

property measurements

measurements_from_reference(*ref=None*, *std_dev=None*, *seed=None*)

Add noise to a reference solution.

mud_problem(*method='pca'*, *data_weights=None*, *sample_weights=None*, *num_components=2*, *samples_mask=None*, *times_mask=None*, *sensors_mask=None*)

Build QoI Map Using Data and Measurements

property n_params: int

property n_qoi: int

property n_samples: int

property n_sensors: int

property n_ts: int

plot_ts(*ax=None*, *samples=None*, *times=None*, *sensor_idx=0*, *max_plot=100*, *meas_kwargs={}*, *samples_kwargs={}*, *alpha=0.1*)

Plot time series data

sample_data(*samples_mask=None*, *times_mask=None*, *sensors_mask=None*)

property sample_dist

sensor_contour_plot(*idx=0*, *c_vals=None*, *ax=None*, *mask=None*, *fill=True*, *colorbar=True*, ***kwargs*)

Plot locations of sensors in space

sensor_scatter_plot(*ax=None*, *mask=None*, *colorbar=None*, ***kwargs*)

Plot locations of sensors in space

property true_vals

validate(*check_meas=True*, *check_true=False*)

Validates if class has been set-up appropriately for inversion

mud.cli module

mud.funs module

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

mud.funs.**data_prob**(*lam*, *qoi*, *qoi_true=None*, *measurements=None*, *std_dev=None*, *sample_dist='uniform'*, *domain=None*, *lam_ref=None*, *times=None*, *sensors=None*, *idxs=None*, *method='wme'*, *init_dist=<scipy.stats._distn_infrastructure.rv_continuous_frozen object>*)

Data-Constructed Map Solve

Wrapper around SpatioTemporalProblem class to create and solve a MUD problem by first aggregating observed and siumalated data in a data-constructed qoi map.

mud.funs.**iter_lin_solve**(*A*, *b*, *y=None*, *mean=None*, *cov=None*, *data_cov=None*, *method='mud'*, *num_epochs=1*, *idx_order=None*)

Iterative Linear Gaussian Problem Solver Entrypoint

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

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

Linear Gaussian Problem Solver Entrypoint

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*)

MAP Linear Gaussian Problem Solve

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

MAP Linear Gaussian Problem Solve

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

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 Plotting Module

Plotting utility functions for visualizing data-sets and distributions related to algorithm implemented within the MUD library.

Functions

mud.plot.**build_nd_mesh_grid**(*domain*, *aff=100*)

Build n-dimensional mesh grid

Parameters

- **domain** (*List[List[int]]*) – List of *[min, max]* ranges for each parameter to construct grid over.
- **aff** (*int, default=100*) – Number of points to use in each dimension.

Returns

(mesh, grid_points) – Tuple consistent of n-dimensional grid of points *mesh* and the list of coordinates $(x_1, x_2, …, x_n)$ for each grid point along.

Return type

Tuple[numpy.ndarray, numpy.ndarray]

Examples

Building 1d ``mesh”

```
>>> mesh_1d, pts_1d = build_nd_mesh_grid([[0,1]], aff=5)
>>> mesh_1d
[array([0.  , 0.25, 0.5 , 0.75])]
>>> pts_1d
array([[0.  , 0.25, 0.5 , 0.75]])
```

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_1D_vecs**(*vecs*, *markers=None*, *ax=None*, *label=True*, ***kwargs*)

Plot components of 1D vectors

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

mud.plot.**plot_dist**(*dist*, *domain*, *ax=None*, *idx=0*, *source='kde'*, *aff=100*, ***kwargs*)

Plot a probability distribution over a given domain.

mud.plot.**plot_vert_line**(*ax*, *x_loc*, *ylim=None*, ***kwargs*)

Plot a vertical line on an existing axis at *x_loc*

mud.plot.**save_figure**(*fname: str*, *save_path: str = 'figures'*, *close_fig: bool = True*, ***kwargs*)

Save Figure Utility

Utility to save figure to a given folder path if specified.

Parameters

- **fname** (*str*) – Name of image, with extension.
- **save_path** (*str*) – Directory to save figure to. Assumed to exist. Default: figures/
- **close_fig** (*bool, default=True*) – Whether to close the figure after saving it.
- **kwargs** (*dict, optional*) – Arguments to pass to savefig()

mud.preprocessing module

MUD Pre-Processing Module

All functions for pre-processing QoI data-sets before applying inversion algorithms can be found in this module.

Functions

pca - Applly Principle Component Analysis transformation to QoI data.

mud.preprocessing.**pca**(*data: _SupportsArray[dtype] | _NestedSequence[_SupportsArray[dtype]] | bool | int | float | complex | str | bytes | _NestedSequence[bool | int | float | complex | str | bytes]*, *n_components: int = 2*, ***kwargs*) → Tuple[PCA, ndarray]

Apply Principal Component Analysis

Uses `sklearn.decomposition.PCA` class to perform a truncated PCA transformation on input `data` using the first `n_components` principle components. Note `sklearn.preprocessing.StandardScaler` transformation is applied to the data first.

Parameters

- **ds** (`numpy.typing.ArrayLike`) – Data to apply PCA transformation to. Must be 2 dimensional.
- **n_components** (*int, default=2*) – Number of principal components to use.
- **kwargs** (*dict, optional*) – Additional keyword arguments will be passed to `sklearn.decomposition.PCA` class constructor. See sklearn's documentation for more information on how PCA is performed.

Returns

pca_res – Tuple of `(pca, X_train)` where `pca` is the `sklearn.decomposition.PCA` class with principle component vectors accessible at `pca.components_` and `X_train` being the transformed data-set, which should have same number of rows as original `data`, but now only `n_components` columns.

Return type

Tuple[`sklearn.decomposition.PCA`, `numpy.ndarray`]

Examples

For a simple example lets apply the PCA transformation to the identity matrix in 2 dimensions, using first 1 principle component.

```
>>> data = np.eye(2)
>>> pca_1, X_train_1 = pca(data, n_components=1)
>>> np.around(X_train_1, decimals=1)
array([[-1.4],
       [ 1.4]])
>>> np.around(pca_1.components_, decimals=1)
array([[-0.7,  0.7]])
```

Now lets try using two components

```
>>> pca_2, X_train_2 = pca(data, n_components=2)
>>> np.around(X_train_2, decimals=1)
array([[-1.4,  0. ],
       [ 1.4,  0. ]])
>>> np.abs(np.around(pca_2.components_, decimals=1))
array([[0.7, 0.7],
       [0.7, 0.7]])
```

Note that if we have three dimensional data we must flatten it before sending using `pca()`

```
>>> data = np.random.rand(2,2,2)
>>> pca, X_train = pca(data)
Traceback (most recent call last):
    ...
ValueError: Data is 3 dimensional. Must be 2D
```

Assuming the first dimension indicates each sample, and each sample contains 2D data within the 2nd and 3rd dimensions of the of the data set, then we can flatten this 2D data into a vector and then perform the PCA transformation.

```
>>> data = np.reshape(data, (2,-1))
>>> pca, X_train = pca(data)
>>> X_train.shape
(2, 2)
```

mud.preprocessing.**svd**(*data: _SupportsArray[dtype] | _NestedSequence[_SupportsArray[dtype]] | bool | int | float | complex | str | bytes | _NestedSequence[bool | int | float | complex | str | bytes]*, ***kwargs*) → Tuple[ndarray, ndarray, ndarray]

Apply Singular Value Decomposition

Uses `np.linalg.svd` class to perform an SVD transformation on input `data`. Note `sklearn.preprocessing.StandardScaler` transformation is applied to the data first.

Parameters

- **ds** (`numpy.typing.ArrayLike`) – Data to apply SVD transformation to. Must be 2 dimensional.
- **kwargs** (*dict, optional*) – Additional keyword arguments will be passed to `np.linalg.svd` method.

Returns
svd_res – Tuple of `(U, singular_values, singular_vectors)` corresponding to the X = Sigma UV^T decomposition elements.

Return type
Tuple[numpy.ndarray,]

Examples

For a simple example lets apply the PCA transformation to the identity matrix in 2 dimensions, using first 1 principle component.

```
>>> data = np.eye(2)
>>> U, S, V = svd(data)
>>> np.around(U, decimals=1)
array([[-0.7,  0.7],
       [ 0.7,  0.7]])
>>> np.around(S, decimals=1)
array([2., 0.])
```

Note that if we have three dimensional data we must flatten it before sending using `pca()`

```
>>> data = np.random.rand(2,2,2)
>>> U, S, V = svd(data)
Traceback (most recent call last):
    ...
ValueError: Data is 3 dimensional. Must be 2D
```

Assuming the first dimension indicates each sample, and each sample contains 2D data within the 2nd and 3rd dimensions of the of the data set, then we can flatten this 2D data into a vector and then perform `svd()`.

```
>>> data = np.reshape(data, (2,-1))
>>> U, S, V = svd(data)
>>> U.shape
(2, 2)
```

mud.util module

mud.util.**add_noise**(*signal: _SupportsArray[dtype] | _NestedSequence[_SupportsArray[dtype]] | bool | int | float | complex | str | bytes | _NestedSequence[bool | int | float | complex | str | bytes]*, *sd: float = 0.05*, *seed: int | None = None*)

Add Noise

Add noise to synthetic signal to model a real measurement device. Noise is assumed to be from a standard normal distribution std deviation *sd*:

$mathcal{N}(0,sigma)$

Paramaters

signal
[numpy.typing.ArrayLike] Signal to add noise to.

sd
[float, default = 0.05] Standard deviation of error to add.

seed
[int, optional] Seed to use for numpy random number generator.

returns

- **noisy_signal** (*numpy.typing.ArrayLike*) – Signal with noise added to it.
- *Example Usage*
- ————-
- *Generate test signal, add noise, check average distance*
- *>>> seed = 21*
- *>>> test_signal = np.ones(5)*
- *>>> noisy_signal = add_noise(test_signal, sd=0.05, seed=21)*
- *>>> np.round(1000*np.mean(noisy_signal-test_signal))*
- *4.0*

mud.util.**fit_domain**(*x: ndarray | None = None*, *min_max_bounds: ndarray | None = None*, *pad_ratio: float = 0.1*) → ndarray

Fit domain bounding box to array x

Parameters

- **x** (*ArrayLike*) – 2D array to calculate min, max values along columns.
- **pad_ratio** (*float, default=0.1*) – What ratio of total range=max-min to subtract/add to min/max values to construct final domain range. Padding is done per x column dimension.

Returns
min_max_bounds – Domain fitted to values found in 2D array x, with padding added.

Return type
ArrayLike

Examples

Input must be 2D. Set pad_ratio = 0 to get explicit min/max bounds >>> fit_domain(np.array([[1, 10], [0, -10]]), pad_ratio=0.0) array([[0, 1],

[-10, 10]])

Can extend domain around the array values using the pad_ratio argument.

```
>>> fit_domain(np.array([[1, 10], [0, -10]]), pad_ratio=1)
array([[ -1,   2],
       [-30,  30]])
```

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, default=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*
- ————-
- *>>> 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*, *rcond=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
>>> 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)
```

(continues on next page)

(continued from previous page)

```
>>> 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.**rank_decomposition**(*A: _SupportsArray[dtype] | _NestedSequence[_SupportsArray[dtype]] | bool | int | float | complex | str | bytes | _NestedSequence[bool | int | float | complex | str | bytes]*) → List[ndarray]

Build list of rank k updates of A

mud.util.**set_shape**(*array: ndarray*, *shape: 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*, *data*, *std*)

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*, *data_list*, *std_list*)

Module contents

2.3 License

2.4 Contributors

- Mathematical Michael <consistentbayes@gmail.com>
- Carlos del-Castillo-Negrete <cdelcastillo21@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

N

P

R

S

T

U

V

W

X

Y