# mud Documentation

***Release 0.0.26***

**Mathematical Michael**

**Jul 12, 2021**

# CONTENTS

This is the documentation of **mud**.

CHAPTER

# ONE

# CONTENTS

## 1.1 readme

### 1.1.1 MUD

Analytical solutions and some associated utility functions for computing maximal updated density points for Data-Consistent Inversion.

### 1.1.2 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.

More about the differences here…

What does this package include?

## 1.2 License

## 1.3 Contributors

- Mathematical Michael <consistentbayes@gmail.com>

## 1.4 mud

### 1.4.1 mud package

#### Submodules

#### mud.base module

**class** mud.base.**BayesProblem**(*X*, *y*, *domain=None*)
Bases: object

Sets up Bayesian Inverse Problem for parameter identification

```
>>> 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], [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**()

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

**set_prior**(*distribution=None*)

**class** mud.base.**DensityProblem**(*X*, *y*, *domain=None*)
Bases: object

Sets up Data-Consistent Inverse Problem for parameter identification

```
>>> from mud.base import DensityProblem
>>> from mud.funs import wme
>>> import numpy as np
>>> 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
>>> W = wme(Y, y)
>>> B = DensityProblem(X, W, np.array([[0,1], [0,1]]))
>>> np.round(B.mud_point()[0],1)
0.5
```

**estimate**()

**fit**(***kwargs*)

**mud_point**()

**set_initial**(*distribution=None*)

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

**set_predicted**(*distribution=None*, ***kwargs*)

### 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*, *return_pred=False*)

mud.funs.**mud_problem**(*lam*, *qoi*, *qoi_true*, *domain*, *sd=0.05*, *num_obs=None*, *split=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*, *return_pred=False*)
For SWE problem, we are inverting N(0,1). This is the default value for *data_cov*.

mud.funs.**mud_sol_alt**(*A*, *b*, *y=None*, *mean=None*, *cov=None*, *data_cov=None*, *return_pred=False*)
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**(*X*, *data*, *sd=None*)

### 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.**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
>>> 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.**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

## 1.5 Changelog

### 1.5.1 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

### 1.5.2 Version 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

### 1.5.3 Version 0.1

- Basic functionality
- Testing for all modules
- Beginning of adherence to semantic versioning rules
- i.e., breaking changes in major revision, contract changes in minor, bugfixes/features in patch.

CHAPTER

TWO

# INDICES AND TABLES

- genindex
- modindex
- search

# PYTHON MODULE INDEX

## m

# INDEX

## U



## W