KLIFF Documentation

Release 0.4.1

Mingjian Wen

Oct 07, 2022

CONTENTS

KLIFF is an interatomic potential fitting package that can be used to fit both physics-motivated potentials (e.g. the Stillinger-Weber potential) and machine learning potentials (e.g. neural network potential). The trained potential can be deployed with the KIM-API, which is supported by major simulation codes such as LAMMPS, ASE, DL_POLY, and GULP among others.

CHAPTER

ONE

INSTALLATION

KLIFF requires:

- Python 3.6 or newer.
- A C++ compiler that supports C++11.

1.1 KLIFF

The easiest way to install KLIFF is using a package manager, do either

```
$ conda intall -c conda-forge kliff
```

or

```
$ pip install kliff
```

Alternatively, you can install from source:

```
$ git clone https://github.com/openkim/kliff
$ pip install ./kliff
```

1.2 Other dependencies

1.2.1 KIM Models

KLIFF is built on top of KIM to fit physics-motivated potentials archived on OpenKIM. To get KLIFF work with OpenKIM models, kim-api and kimpy, and openkim-models are needed.

The easiest way to install them is via conda:

```
$ conda install -c conda-forge kim-api kimpy openkim-models
```

Note: After installation, you can do `$ kim-api-collections-management list`. If you see a list of directories where the KIM model drivers and models are placed, then you are good to go. Otherwise, you may forget to set up the `PATH` and bash completions, which can be achieved by (assuming you are using Bash): `$ source path/to/the/kim/library/bin/kim-api-activate`. See the kim-api documentation for more information.

Warning: The conda approach should work for most systems, but not all (e.g. Mac with Apple Chip). Refer to https://openkim.org/doc/usage/obtaining-models for other installing instructions (e.g. from source).

1.2.2 PyTorch

For machine learning potentials, KLIFF takes advantage of PyTorch to build neural network models and conduct the training. So if you want to train neural network potentials, PyTorch needs to be installed. Please follow the instructions given on the official PyTorch website to install it.

CHAPTER

TWO

TUTORIALS

To learn how to use KLIFF, begin with the tutorials.

Train a Stillinger-Weber potential: a good entry point to see the basics of training a physics-motivated potential.

Parameter transformation for the Stillinger-Weber potential: it is similar to *Train a Stillinger-Weber potential*, except that some parameters are transformed to the log space for optimization.

Train a neural network potential: walks through the steps to train a machine-learning neural network potential.

Train a neural network potential for SiC: similar to *Train a neural network potential*, but train for a system of multiple species.

Train a Lennard-Jones potential: similar to *Train a Stillinger-Weber potential* (where a KIM model is used), here the Lennard-Jones model built in KLIFF is used.

More examples can be found at https://github.com/openkim/kliff/tree/master/examples.

2.1 Train a linear regression potential

In this tutorial, we train a linear regression model on the descriptors obtained using the symmetry functions.

```
from kliff.calculators import CalculatorTorch
from kliff.dataset import Dataset
from kliff.descriptors import SymmetryFunction
from kliff.models import LinearRegression
from kliff.utils import download_dataset

descriptor = SymmetryFunction(
    cut_name="cos", cut_dists={"Si-Si": 5.0}, hyperparams="set30", normalize=True
)


model = LinearRegression(descriptor)

# training set
dataset_path = download_dataset(dataset_name="Si_training_set")
dataset_path = dataset_path.joinpath("varying_alat")
tset = Dataset(dataset_path)
configs = tset.get_configs()

# calculator
```

(continues on next page)

(continued from previous page)

```
calc = CalculatorTorch(model)
calc.create(configs, reuse=False)
```

Out:

```
2022-04-28 10:49:35.846 | INFO     | kliff.dataset.dataset:_read:397 - 400␣
↪configurations read from /Users/mjwen/Applications/kliff/examples/Si_training_set/
↪varying_alat
2022-04-28 10:49:35.848 | INFO     | kliff.calculators.calculator_torch:_get_device:417 -
↪ Training on cpu
2022-04-28 10:49:35.849 | INFO     | kliff.descriptors.descriptor:generate_
↪fingerprints:104 - Start computing mean and stdev of fingerprints.
2022-04-28 10:49:56.708 | INFO     | kliff.descriptors.descriptor:generate_
↪fingerprints:121 - Finish computing mean and stdev of fingerprints.
2022-04-28 10:49:56.712 | INFO     | kliff.descriptors.descriptor:generate_
↪fingerprints:129 - Fingerprints mean and stdev saved to `fingerprints_mean_and_stdev.
↪pkl`.
2022-04-28 10:49:56.712 | INFO     | kliff.descriptors.descriptor:_dump_fingerprints:164␣
↪- Pickling fingerprints to `fingerprints.pkl`
2022-04-28 10:49:56.734 | INFO     | kliff.descriptors.descriptor:_dump_fingerprints:176␣
↪- Processing configuration: 0.
2022-04-28 10:49:57.294 | INFO     | kliff.descriptors.descriptor:_dump_fingerprints:176␣
↪- Processing configuration: 100.
2022-04-28 10:49:57.906 | INFO     | kliff.descriptors.descriptor:_dump_fingerprints:176␣
↪- Processing configuration: 200.
2022-04-28 10:49:58.640 | INFO     | kliff.descriptors.descriptor:_dump_fingerprints:176␣
↪- Processing configuration: 300.
2022-04-28 10:49:59.157 | INFO     | kliff.descriptors.descriptor:_dump_fingerprints:219␣
↪- Pickle 400 configurations finished.
```

We can train a linear regression model by minimizing a loss function as discussed in *Train a neural network potential*. But linear regression model has analytic solutions, and thus we can train the model directly by using this feature. This can be achieved by calling the `fit()` function of its calculator.

```
# fit the model
calc.fit()


# save model
model.save("linear_model.pkl")
```

Out:

```
2022-04-28 10:49:59.693 | INFO     | kliff.models.linear_regression:fit:39 - fit model
↪"LinearRegression" finished.
fit model "LinearRegression" finished.
```

Total running time of the script: (0 minutes 25.892 seconds)

2.2 Train a Lennard-Jones potential

In this tutorial, we train a Lennard-Jones potential that is build in KLIFF (i.e. not models archived on OpenKIM). From a user's perspective, a KLIFF built-in model is not different from a KIM model.

Compare this with *Train a Stillinger-Weber potential*.

Out:

```
2022-04-28 11:07:49.718 | INFO     | kliff.dataset.dataset:_read:397 - 4 configurations␣
↪read from /Users/mjwen/Applications/kliff/examples/Si_training_set_4_configs
#=============================================================================
# Available parameters to optimize.
# Parameters in `original` space.
# Model: LJ6-12
#=============================================================================

name: epsilon
value: [1.]
size: 1

name: sigma
value: [2.]
size: 1

name: cutoff
value: [5.]
size: 1


#=============================================================================
# Model parameters that are optimized.
# Note that the parameters are in the transformed space if
# `params_transform` is provided when instantiating the model.
#=============================================================================

sigma 1
  2.000000000000000e+00

epsilon 1
  1.000000000000000e+00


2022-04-28 11:07:49.726 | INFO     | kliff.calculators.calculator:create:107 - Create␣
↪calculator for 4 configurations.
2022-04-28 11:07:49.726 | INFO     | kliff.loss:minimize:290 - Start minimization using␣
↪method: L-BFGS-B.
2022-04-28 11:07:49.727 | INFO     | kliff.loss:_scipy_optimize:404 - Running in serial␣
↪mode.
2022-04-28 11:07:53.172 | INFO     | kliff.loss:minimize:292 - Finish minimization using␣
↪method: L-BFGS-B.
#=============================================================================
# Model parameters that are optimized.
# Note that the parameters are in the transformed space if
```

(continues on next page)

(continued from previous page)

```
# `params_transform` is provided when instantiating the model.
#=============================================================================

sigma 1
  2.0629043239028659e+00

epsilon 1
  1.5614870430532530e+00
```

```
from kliff.calculators import Calculator
from kliff.dataset import Dataset
from kliff.loss import Loss
from kliff.models import LennardJones
from kliff.utils import download_dataset

# training set
dataset_path = download_dataset(dataset_name="Si_training_set_4_configs")
tset = Dataset(dataset_path)
configs = tset.get_configs()

# calculator
model = LennardJones()
model.echo_model_params()

# fitting parameters
model.set_opt_params(sigma=[["default"]], epsilon=[["default"]])
model.echo_opt_params()

calc = Calculator(model)
calc.create(configs)

# loss
loss = Loss(calc, nprocs=1)
result = loss.minimize(method="L-BFGS-B", options={"disp": True, "maxiter": 10})


# print optimized parameters
model.echo_opt_params()
model.save("kliff_model.yaml")
```

Total running time of the script: (0 minutes 5.538 seconds)

2.3 Train a Stillinger-Weber potential

In this tutorial, we train a Stillinger-Weber (SW) potential for silicon that is archived on OpenKIM.

Before getting started to train the SW model, let's first make sure it is installed.

If you haven't already, follow *Installation* to install `kim-api` and `kimpy`, and `openkim-models`.

Then do `$ kim-api-collections-management list`, and make sure `SW_StillingerWeber_1985_Si__MO_405512056662_006` is listed in one of the collections.

Note: If you see `SW_StillingerWeber_1985_Si__MO_405512056662_005` (note the last three digits), you need to change `model = KIMModel(model_name="SW_StillingerWeber_1985_Si__MO_405512056662_006")` to the corresponding model name in your installation.

We are going to create potentials for diamond silicon, and fit the potentials to a training set of energies and forces consisting of compressed and stretched diamond silicon structures, as well as configurations drawn from molecular dynamics trajectories at different temperatures. Download the training set `Si_training_set.tar.gz`. (It will be automatically downloaded if not present.) The data is stored in # **extended xyz** format, and see *Dataset* for more information of this format.

> **Warning:** The `Si_training_set` is just a toy data set for the purpose to demonstrate how to use KLIFF to train potentials. It should not be used to train any potential for real simulations.

Let's first import the modules that will be used in this example.

```
from kliff.calculators import Calculator
from kliff.dataset import Dataset
from kliff.dataset.weight import Weight
from kliff.loss import Loss
from kliff.models import KIMModel
from kliff.utils import download_dataset
```

2.3.1 Model

We first create a KIM model for the SW potential, and print out all the available parameters that can be optimized (we call this `model parameters`).

```
model = KIMModel(model_name="SW_StillingerWeber_1985_Si__MO_405512056662_006")
model.echo_model_params()
```

```
#=============================================================================
# Available parameters to optimize.
# Parameters in `original` space.
# Model: SW_StillingerWeber_1985_Si__MO_405512056662_006
#=============================================================================

name: A
value: [15.28484792]
size: 1
```

(continues on next page)

(continued from previous page)

```
name: B
value: [0.60222456]
size: 1

name: p
value: [4.]
size: 1

name: q
value: [0.]
size: 1

name: sigma
value: [2.0951]
size: 1

name: gamma
value: [2.51412]
size: 1

name: cutoff
value: [3.77118]
size: 1

name: lambda
value: [45.5322]
size: 1

name: costheta0
value: [-0.33333333]
size: 1



'#=============================================================================\n#␣
↪Available parameters to optimize.\n# Parameters in `original` space.\n# Model: SW_
↪StillingerWeber_1985_Si__MO_405512056662_006\n
↪#=============================================================================\n\
↪nname: A\nvalue: [15.28484792]\nsize: 1\n\nname: B\nvalue: [0.60222456]\nsize: 1\n\
↪nname: p\nvalue: [4.]\nsize: 1\n\nname: q\nvalue: [0.]\nsize: 1\n\nname: sigma\nvalue:␣
↪[2.0951]\nsize: 1\n\nname: gamma\nvalue: [2.51412]\nsize: 1\n\nname: cutoff\nvalue: [3.
↪77118]\nsize: 1\n\nname: lambda\nvalue: [45.5322]\nsize: 1\n\nname: costheta0\nvalue:␣
↪[-0.33333333]\nsize: 1\n\n'
```

The output is generated by the last line, and it tells us the `name`, `value`, `size`, `data type` and a `description` of each parameter.

Note: You can provide a `path` argument to the method `echo_model_params(path)` to write the available parameters information to a file indicated by `path`.

Note: The available parameters information can also by obtained using the **kliff** *Command Line Tool*: `$ kliff model --echo-params SW_StillingerWeber_1985_Si__MO_405512056662_006`

Now that we know what parameters are available for fitting, we can optimize all or a subset of them to reproduce the training set.

```
model.set_opt_params(
    A=[[5.0, 1.0, 20]], B=[["default"]], sigma=[[2.0951, "fix"]], gamma=[[1.5]]
)
model.echo_opt_params()
```

```
#=============================================================================
# Model parameters that are optimized.
# Note that the parameters are in the transformed space if
# `params_transform` is provided when instantiating the model.
#=============================================================================

A 1
  5.0000000000000000e+00   1.0000000000000000e+00   2.0000000000000000e+01

B 1
  6.0222455840000000e-01

sigma 1
  2.0951000000000000e+00 fix

gamma 1
  1.5000000000000000e+00



'#=============================================================================\n#␣
↪Model parameters that are optimized.\n# Note that the parameters are in the␣
↪transformed space if \n# `params_transform` is provided when instantiating the model.\n
↪#=============================================================================\n\nA␣
↪1\n  5.0000000000000000e+00   1.0000000000000000e+00   2.0000000000000000e+01 \n\nB 1\
↪n  6.0222455840000000e-01 \n\nsigma 1\n  2.0951000000000000e+00 fix \n\ngamma 1\n  1.
↪5000000000000000e+00 \n\n'
```

Here, we tell KLIFF to fit four parameters `B`, `gamma`, `sigma`, and `A` of the SW model. The information for each fitting parameter should be provided as a list of list, where the size of the outer list should be equal to the `size` of the parameter given by `model.echo_model_params()`. For each inner list, you can provide either one, two, or three items.

- One item. You can use a numerical value (e.g. `gamma`) to provide an initial guess of the parameter. Alternatively, the string `'default'` can be provided to use the default value in the model (e.g. `B`).
- Two items. The first item should be a numerical value and the second item should be the string `'fix'` (e.g. `sigma`), which tells KLIFF to use the value for the parameter, but do not optimize it.
- Three items. The first item can be a numerical value or the string `'default'`, having the same meanings as the one item case. In the second and third items, you can list the lower and upper bounds for the parameters, respectively. A bound could be provided as a numerical values or `None`. The latter indicates no bound is applied.

The call of `model.echo_opt_params()` prints out the fitting parameters that we require KLIFF to optimize. The

number 1 after the name of each parameter indicates the size of the parameter.

Note: The parameters that are not included as a fitting parameter are fixed to the default values in the model during the optimization.

2.3.2 Training set

KLIFF has a *Dataset* to deal with the training data (and possibly test data). Additionally, we define the `energy_weight` and `forces_weight` corresponding to each configuration using `Weight`. In this example, we set `energy_weight` to `1.0` and `forces_weight` to `0.1`. For the silicon training set, we can read and process the files by:

```
dataset_path = download_dataset(dataset_name="Si_training_set")
weight = Weight(energy_weight=1.0, forces_weight=0.1)
tset = Dataset(dataset_path, weight)
configs = tset.get_configs()
```

```
2022-10-06 23:44:01.093 | INFO     | kliff.dataset.dataset:_read:398 - 1000␣
↪configurations read from /Users/mjwen.admin/Packages/kliff/examples/Si_training_set
```

The `configs` in the last line is a list of *Configuration*. Each configuration is an internal representation of a processed **extended xyz** file, hosting the species, coordinates, energy, forces, and other related information of a system of atoms.

2.3.3 Calculator

`Calculator` is the central agent that exchanges information and orchestrate the operation of the fitting process. It calls the model to compute the energy and forces and provide this information to the *Loss function* (discussed below) to compute the loss. It also grabs the parameters from the optimizer and update the parameters stored in the model so that the up-to-date parameters are used the next time the model is evaluated to compute the energy and forces. The calculator can be created by:

```
calc = Calculator(model)
_ = calc.create(configs)
```

```
2022-10-06 23:44:01.499 | INFO     | kliff.calculators.calculator:create:107 - Create␣
↪calculator for 1000 configurations.
```

where `calc.create(configs)` does some initializations for each configuration in the training set, such as creating the neighbor list.

2.3.4 Loss function

KLIFF uses a loss function to quantify the difference between the training set data and potential predictions and uses minimization algorithms to reduce the loss as much as possible. KLIFF provides a large number of minimization algorithms by interacting with SciPy. For physics-motivated potentials, any algorithm listed on scipy.optimize.minimize and scipy.optimize.least_squares can be used. In the following code snippet, we create a loss of energy and forces and use 2 processors to calculate the loss. The `L-BFGS-B` minimization algorithm is applied to minimize the loss, and the minimization is allowed to run for a max number of 100 iterations.

```
steps = 100
loss = Loss(calc, nprocs=2)
loss.minimize(method="L-BFGS-B", options={"disp": True, "maxiter": steps})
```

```
2022-10-06 23:44:01.500 | INFO     | kliff.loss:minimize:290 - Start minimization using␣
↪method: L-BFGS-B.
2022-10-06 23:44:01.501 | INFO     | kliff.loss:_scipy_optimize:406 - Running in␣
↪multiprocessing mode with 2 processes.
2022-10-06 23:44:36.663 | INFO     | kliff.loss:minimize:292 - Finish minimization using␣
↪method: L-BFGS-B.

      fun: 0.6940780132865667
 hess_inv: <3x3 LbfgsInvHessProduct with dtype=float64>
      jac: array([ 8.88178346e-07, -3.50830474e-06,  7.77156122e-08])
  message: 'CONVERGENCE: NORM_OF_PROJECTED_GRADIENT_<=_PGTOL'
     nfev: 184
      nit: 37
     njev: 46
   status: 0
  success: True
        x: array([14.93863457,  0.58740273,  2.20146129])
```

The minimization stops after running for 27 steps. After the minimization, we'd better save the model, which can be loaded later for the purpose to do a retraining or evaluations. If satisfied with the fitted model, you can also write it as a KIM model that can be used with LAMMPS, GULP, ASE, etc. via the kim-api.

```
model.echo_opt_params()
model.save("kliff_model.yaml")
model.write_kim_model()
# model.load("kliff_model.yaml")
```

```
#=========================================================================
# Model parameters that are optimized.
# Note that the parameters are in the transformed space if
# `params_transform` is provided when instantiating the model.
#=========================================================================

A 1
  1.4938634567965085e+01   1.0000000000000000e+00   2.0000000000000000e+01

B 1
  5.8740272891468026e-01

sigma 1
  2.095100000000000e+00 fix

gamma 1
  2.2014612879744848e+00


2022-10-06 23:44:36.670 | INFO     | kliff.models.kim:write_kim_model:695 - KLIFF␣
↪trained model write to `/Users/mjwen.admin/Packages/kliff/examples/SW_StillingerWeber_
↪1985_Si__MO_405512056662_006_kliff_trained`
```

The first line of the above code generates the output. A comparison with the original parameters before carrying out the minimization shows that we recover the original parameters quite reasonably. The second line saves the fitted model to a file named `kliff_model.pkl` on the disk, and the third line writes out a KIM potential named `SW_StillingerWeber_1985_Si__MO_405512056662_006_kliff_trained`.

See also:

For information about how to load a saved model, see *Frequently Used Modules*.

Total running time of the script: (0 minutes 36.524 seconds)

2.4 Parameter transformation for the Stillinger-Weber potential

Parameters in the empirical interatomic potential are often restricted by some physical constraints. As an example, in the Stillinger-Weber (SW) potential, the energy scaling parameters (e.g., `A` and `B`) and the length scaling parameters (e.g., `sigma` and `gamma`) are constrained to be positive.

Due to these constraints, we might want to work with the log of the parameters, i.e., `log(A)`, `log(B)`, `log(sigma)`, and `log(gamma)` when doing the optimization. After the optimization, we can transform them back to the original parameter space using an exponential function, which will guarantee the positiveness of the parameters.

In this tutorial, we show how to apply parameter transformation to the SW potential for silicon that is archived on OpenKIM. Compare this with *Train a Stillinger-Weber potential*.

To start, let's first install the SW model:

```
$ kim-api-collections-management install user SW_StillingerWeber_1985_Si__MO_
↪405512056662_006
```

See also:

This installs the model and its driver into the `User Collection`. See *Install a model* for more information about installing KIM models.

This is

```
import numpy as np

from kliff.calculators import Calculator
from kliff.dataset import Dataset
from kliff.dataset.weight import Weight
from kliff.loss import Loss
from kliff.models import KIMModel
from kliff.models.parameter_transform import LogParameterTransform
from kliff.utils import download_dataset
```

Before creating a KIM model for the SW potential, we first instantiate the parameter transformation class that we want to use. `kliff` has a built-in log-transformation; however, extending it to other parameter transformation can be done by creating a subclass of `ParameterTransform`.

To make a direct comparison to *Train a Stillinger-Weber potential*, in this tutorial we will apply log-transformation to parameters `A`, `B`, `sigma`, and `gamma`, which correspond to energy and length scales.

```
transform = LogParameterTransform(param_names=["A", "B", "sigma", "gamma"])
model = KIMModel(
    model_name="SW_StillingerWeber_1985_Si__MO_405512056662_006",
    params_transform=transform,
```

(continues on next page)

(continued from previous page)

```
)
model.echo_model_params(params_space="original")
```

Out:

```
#=============================================================================
# Available parameters to optimize.
# Parameters in `original` space.
# Model: SW_StillingerWeber_1985_Si__MO_405512056662_006
#=============================================================================

name: A
value: [15.28484792]
size: 1

name: B
value: [0.60222456]
size: 1

name: p
value: [4.]
size: 1

name: q
value: [0.]
size: 1

name: sigma
value: [2.0951]
size: 1

name: gamma
value: [2.51412]
size: 1

name: cutoff
value: [3.77118]
size: 1

name: lambda
value: [45.5322]
size: 1

name: costheta0
value: [-0.33333333]
size: 1



'#=============================================================================\n#␣
↪Available parameters to optimize.\n# Parameters in `original` space.\n# Model: SW_
↪StillingerWeber_1985_Si__MO_405512056662_006\n
```

(continues on next page)

(continued from previous page)

```
↪#================================================================================\n\
↪nname: A\nvalue: [15.28484792]\nsize: 1\n\nname: B\nvalue: [0.60222456]\nsize: 1\n\
↪nname: p\nvalue: [4.]\nsize: 1\n\nname: q\nvalue: [0.]\nsize: 1\n\nname: sigma\nvalue:␣
↪[2.0951]\nsize: 1\n\nname: gamma\nvalue: [2.51412]\nsize: 1\n\nname: cutoff\nvalue: [3.
↪77118]\nsize: 1\n\nname: lambda\nvalue: [45.5322]\nsize: 1\n\nname: costheta0\nvalue:␣
↪[-0.33333333]\nsize: 1\n\n'
```

`model.echo_model_params(params_space="original")` above will print out parameter values in the original, untransformed space, i.e., the original parameterization of the model. If we supply the argument `params_space="transformed"`, then the printed parameter values are given in the transformed space, e.g., log space (below). The values of the other parameters are not changed.

```
model.echo_model_params(params_space="original")
```

Out:

```
#================================================================================
# Available parameters to optimize.
# Parameters in `original` space.
# Model: SW_StillingerWeber_1985_Si__MO_405512056662_006
#================================================================================

name: A
value: [15.28484792]
size: 1

name: B
value: [0.60222456]
size: 1

name: p
value: [4.]
size: 1

name: q
value: [0.]
size: 1

name: sigma
value: [2.0951]
size: 1

name: gamma
value: [2.51412]
size: 1

name: cutoff
value: [3.77118]
size: 1

name: lambda
value: [45.5322]
```

(continues on next page)

(continued from previous page)

```
size: 1

name: costheta0
value: [-0.33333333]
size: 1



'#=========================================================================\n#␣
↪Available parameters to optimize.\n# Parameters in `original` space.\n# Model: SW_
↪StillingerWeber_1985_Si__MO_405512056662_006\n
↪#=========================================================================\n\
↪nname: A\nvalue: [15.28484792]\nsize: 1\n\nname: B\nvalue: [0.60222456]\nsize: 1\n\
↪nname: p\nvalue: [4.]\nsize: 1\n\nname: q\nvalue: [0.]\nsize: 1\n\nname: sigma\nvalue:␣
↪[2.0951]\nsize: 1\n\nname: gamma\nvalue: [2.51412]\nsize: 1\n\nname: cutoff\nvalue: [3.
↪77118]\nsize: 1\n\nname: lambda\nvalue: [45.5322]\nsize: 1\n\nname: costheta0\nvalue:␣
↪[-0.33333333]\nsize: 1\n\n'
```

Compare the output of `params_space="transformed"` and `params_space="original"`, you can see that the values of `A`, `B`, `sigma`, and `gamma` are in the log space after the transformation.

Next, we will set up the initial guess of the parameters to optimize. A value of `"default"` means the initial guess will be directly taken from the value already in the model.

Note: The parameter values we initialize, as well as the lower and upper bounds, are in transformed space (i.e. log space here).

```
model.set_opt_params(
    A=[[np.log(5.0), np.log(1.0), np.log(20)]],
    B=[["default"]],
    sigma=[[np.log(2.0951), "fix"]],
    gamma=[[np.log(1.5)]],
)
model.echo_opt_params()
```

Out:

```
#=========================================================================
# Model parameters that are optimized.
# Note that the parameters are in the transformed space if
# `params_transform` is provided when instantiating the model.
#=========================================================================

A 1
  1.6094379124341003e+00   0.0000000000000000e+00   2.9957322735539909e+00

B 1
 -5.0712488263019628e-01

sigma 1
  7.3960128493182953e-01 fix
```

(continues on next page)

(continued from previous page)

```
gamma 1
  4.0546510810816438e-01



'#=========================================================================\n#␣
↪Model parameters that are optimized.\n# Note that the parameters are in the␣
↪transformed space if \n# `params_transform` is provided when instantiating the model.\n
↪#=========================================================================\n\nA␣
↪1\n  1.6094379124341003e+00   0.0000000000000000e+00   2.9957322735539909e+00 \n\nB 1\
↪n -5.0712488263019628e-01 \n\nsigma 1\n  7.3960128493182953e-01 fix \n\ngamma 1\n  4.
↪0546510810816438e-01 \n\n'
```

We can show the parameters we've just set by `model.echo_opt_params()`.

Note: `model.echo_opt_params()` always displays the parameter values in the transformed space. And it only shows all the parameters specified to optimize. To show all the parameters, do `model.echo_model_params(params_space="transformed")`.

Once we set the model and the parameter transformation scheme, then further calculations, e.g., training the model, will be performed using the transformed space and can be done in the same way as in *Train a Stillinger-Weber potential*.

```
# Training set
dataset_path = download_dataset(dataset_name="Si_training_set")
weight = Weight(energy_weight=1.0, forces_weight=0.1)
tset = Dataset(dataset_path, weight)
configs = tset.get_configs()

# Calculator
calc = Calculator(model)
_ = calc.create(configs)

# Loss function and model training
steps = 100
loss = Loss(calc, nprocs=2)
loss.minimize(method="L-BFGS-B", options={"disp": True, "maxiter": steps})

model.echo_model_params(params_space="original")
```

Out:

```
2022-04-28 11:08:19.996 | INFO     | kliff.dataset.dataset:_read:397 - 1000␣
↪configurations read from /Users/mjwen/Applications/kliff/examples/Si_training_set
2022-04-28 11:08:23.706 | INFO     | kliff.calculators.calculator:create:107 - Create␣
↪calculator for 1000 configurations.
2022-04-28 11:08:23.707 | INFO     | kliff.loss:minimize:290 - Start minimization using␣
↪method: L-BFGS-B.
2022-04-28 11:08:23.707 | INFO     | kliff.loss:_scipy_optimize:406 - Running in␣
↪multiprocessing mode with 2 processes.
2022-04-28 11:09:37.865 | INFO     | kliff.loss:minimize:292 - Finish minimization using␣
```

(continues on next page)

(continued from previous page)

```
↪method: L-BFGS-B.
#=========================================================================================
# Available parameters to optimize.
# Parameters in `original` space.
# Model: SW_StillingerWeber_1985_Si__MO_405512056662_006
#=========================================================================================

name: A
value: [14.93863379]
size: 1

name: B
value: [0.58740269]
size: 1

name: p
value: [4.]
size: 1

name: q
value: [0.]
size: 1

name: sigma
value: [2.0951]
size: 1

name: gamma
value: [2.2014612]
size: 1

name: cutoff
value: [3.77118]
size: 1

name: lambda
value: [45.5322]
size: 1

name: costheta0
value: [-0.33333333]
size: 1



'#=========================================================================================\n#␣
↪Available parameters to optimize.\n# Parameters in `original` space.\n# Model: SW_
↪StillingerWeber_1985_Si__MO_405512056662_006\n
↪#=========================================================================================\n\
↪nname: A\nvalue: [14.93863379]\nsize: 1\n\nname: B\nvalue: [0.58740269]\nsize: 1\n\
↪nname: p\nvalue: [4.]\nsize: 1\n\nname: q\nvalue: [0.]\nsize: 1\n\nname: sigma\nvalue:␣
↪[2.0951]\nsize: 1\n\nname: gamma\nvalue: [2.2014612]\nsize: 1\n\nname: cutoff\nvalue:␣
```

(continues on next page)

(continued from previous page)

```
↪[3.77118]\nsize: 1\n\nname: lambda\nvalue: [45.5322]\nsize: 1\n\nname: costheta0\
↪nvalue: [-0.33333333]\nsize: 1\n\n'
```

The optimized parameter values from this model training are very close, if not the same, as in *Train a Stillinger-Weber potential*. This is expected for the simple tutorial example considered. But for more complex models, training in a transformed space can make it much easier for the optimizer to navigate the parameter space.

Total running time of the script: (1 minutes 20.550 seconds)

2.5 Train a neural network potential

In this tutorial, we train a neural network (NN) potential for silicon.

We are going to fit the NN potential to a training set of energies and forces from compressed and stretched diamond silicon structures (the same training set used in *Train a Stillinger-Weber potential*). Download the training set `Si_training_set.tar.gz` # (It will be automatically downloaded if it is not present.) The data is stored in **extended xyz** format, and see *Dataset* for more information of this format.

> **Warning:** The `Si_training_set` is just a toy data set for the purpose to demonstrate how to use KLIFF to train potentials. It should not be used to train any potential for real simulations.

Let's first import the modules that will be used in this example.

```
from kliff import nn
from kliff.calculators import CalculatorTorch
from kliff.dataset import Dataset
from kliff.dataset.weight import Weight
from kliff.descriptors import SymmetryFunction
from kliff.loss import Loss
from kliff.models import NeuralNetwork
from kliff.utils import download_dataset
```

2.5.1 Model

For a NN model, we need to specify the descriptor that transforms atomic environment information to the fingerprints, which the NN model uses as the input. Here, we use the symmetry functions proposed by Behler and coworkers.

```
descriptor = SymmetryFunction(
    cut_name="cos", cut_dists={"Si-Si": 5.0}, hyperparams="set51", normalize=True
)
```

The `cut_name` and `cut_dists` tell the descriptor what type of cutoff function to use and what the cutoff distances are. `hyperparams` specifies the set of hyperparameters used in the symmetry function descriptor. If you prefer, you can provide a dictionary of your own hyperparameters. And finally, `normalize` informs that the generated fingerprints should be normalized by first subtracting the mean and then dividing the standard deviation. This normalization typically makes it easier to optimize NN model.

We can then build the NN model on top of the descriptor.

```
N1 = 10
N2 = 10
model = NeuralNetwork(descriptor)
model.add_layers(
    # first hidden layer
    nn.Linear(descriptor.get_size(), N1),
    nn.Tanh(),
    # second hidden layer
    nn.Linear(N1, N2),
    nn.Tanh(),
    # output layer
    nn.Linear(N2, 1),
)
model.set_save_metadata(prefix="./kliff_saved_model", start=5, frequency=2)
```

In the above code, we build a NN model with an input layer, two hidden layer, and an output layer. The `descriptor` carries the information of the input layer, so it is not needed to be specified explicitly. For each hidden layer, we first do a linear transformation using `nn.Linear(size_in, size_out)` (essentially carrying out $y = xW + b$, where W is the weight matrix of size `size_in` by `size_out`, and b is a vector of size `size_out`. Then we apply the hyperbolic tangent activation function `nn.Tanh()` to the output of the Linear layer (i.e. y) so as to add the nonlinearity. We use a Linear layer for the output layer as well, but unlike the hidden layer, no activation function is applied here. The input size `size_in` of the first hidden layer must be the size of the descriptor, which is obtained using `descriptor.get_size()`. For all other layers (hidden or output), the input size must be equal to the output size of the previous layer. The `out_size` of the output layer must be 1 such that the output of the NN model gives the energy of the atom.

The `set_save_metadata` function call informs where to save intermediate models during the optimization (discussed below), and what the starting epoch and how often to save the model.

2.5.2 Training set and calculator

The training set and the calculator are the same as explained in *Train a Stillinger-Weber potential*. The only difference is that we need to use the *CalculatorTorch()*, which is targeted for the NN model. Also, its `create()` method takes an argument `reuse` to inform whether to reuse the fingerprints generated from the descriptor if it is present. To train on gpu, set `gpu=True` in `Calculator`.

```
# training set
dataset_path = download_dataset(dataset_name="Si_training_set")
dataset_path = dataset_path.joinpath("varying_alat")
weight = Weight(forces_weight=0.3)
tset = Dataset(dataset_path, weight)
configs = tset.get_configs()

# calculator
calc = CalculatorTorch(model, gpu=False)
_ = calc.create(configs, reuse=False)
```

2.5.3 Loss function

KLIFF uses a loss function to quantify the difference between the training data and potential predictions and uses minimization algorithms to reduce the loss as much as possible. In the following code snippet, we create a loss function that uses the `Adam` optimizer to minimize it. The Adam optimizer supports minimization using *mini-batches* of data, and here we use `100` configurations in each minimization step (the training set has a total of 400 configurations as can be seen above), and run through the training set for `10` epochs. The learning rate `lr` used here is `0.001`, and typically, one may need to play with this to find an acceptable one that drives the loss down in a reasonable time.

```
loss = Loss(calc)
result = loss.minimize(method="Adam", num_epochs=10, batch_size=100, lr=0.001)
```

Out:

```
Epoch = 0       loss = 7.3307514191e+01
Epoch = 1       loss = 7.2090656281e+01
Epoch = 2       loss = 7.1389846802e+01
Epoch = 3       loss = 7.0744289398e+01
Epoch = 4       loss = 7.0117309570e+01
Epoch = 5       loss = 6.9499519348e+01
Epoch = 6       loss = 6.8886824608e+01
Epoch = 7       loss = 6.8277158737e+01
Epoch = 8       loss = 6.7668614388e+01
Epoch = 9       loss = 6.7058616638e+01
Epoch = 10      loss = 6.6683934212e+01
```

We can save the trained model to disk, and later can load it back if we want. We can also write the trained model to a KIM model such that it can be used in other simulation codes such as LAMMPS via the KIM API.

```
model.save("final_model.pkl")
loss.save_optimizer_state("optimizer_stat.pkl")

model.write_kim_model()
```

Note: Now we have trained an NN for a single specie Si. If you have multiple species in your system and want to use different parameters for different species, take a look at the *Train a neural network potential for SiC* example.

Total running time of the script: (0 minutes 45.015 seconds)

2.6 Train a neural network potential for SiC

In this tutorial, we train a neural network (NN) potential for a system containing two species: Si and C. This is very similar to the training for systems containing a single specie (take a look at *Train a neural network potential* for Si if you haven't yet).

```
from kliff import nn
from kliff.calculators.calculator_torch import CalculatorTorchSeparateSpecies
from kliff.dataset import Dataset
from kliff.dataset.weight import Weight
from kliff.descriptors import SymmetryFunction
from kliff.loss import Loss
```

(continues on next page)

(continued from previous page)

```
from kliff.models import NeuralNetwork
from kliff.utils import download_dataset

descriptor = SymmetryFunction(
    cut_name="cos",
    cut_dists={"Si-Si": 5.0, "C-C": 5.0, "Si-C": 5.0},
    hyperparams="set51",
    normalize=True,
)
```

We will create two models, one for Si and the other for C. The purpose is to have a separate set of parameters for Si and C so that they can be differentiated.

```
N1 = 10
N2 = 10
model_si = NeuralNetwork(descriptor)
model_si.add_layers(
    # first hidden layer
    nn.Linear(descriptor.get_size(), N1),
    nn.Tanh(),
    # second hidden layer
    nn.Linear(N1, N2),
    nn.Tanh(),
    # output layer
    nn.Linear(N2, 1),
)
model_si.set_save_metadata(prefix="./kliff_saved_model_si", start=5, frequency=2)


N1 = 10
N2 = 10
model_c = NeuralNetwork(descriptor)
model_c.add_layers(
    # first hidden layer
    nn.Linear(descriptor.get_size(), N1),
    nn.Tanh(),
    # second hidden layer
    nn.Linear(N1, N2),
    nn.Tanh(),
    # output layer
    nn.Linear(N2, 1),
)
model_c.set_save_metadata(prefix="./kliff_saved_model_c", start=5, frequency=2)


# training set
dataset_path = download_dataset(dataset_name="SiC_training_set")
weight = Weight(forces_weight=0.3)
tset = Dataset(dataset_path, weight)
configs = tset.get_configs()

# calculator
```

(continues on next page)

(continued from previous page)

```
calc = CalculatorTorchSeparateSpecies({"Si": model_si, "C": model_c}, gpu=False)
_ = calc.create(configs, reuse=False)

# loss
loss = Loss(calc)
result = loss.minimize(method="Adam", num_epochs=10, batch_size=4, lr=0.001)
```

Out:

```
Epoch = 0       loss = 5.7247632980e+01
Epoch = 1       loss = 5.7215625763e+01
Epoch = 2       loss = 5.7186323166e+01
Epoch = 3       loss = 5.7158138275e+01
Epoch = 4       loss = 5.7130514145e+01
Epoch = 5       loss = 5.7103128433e+01
Epoch = 6       loss = 5.7075778961e+01
Epoch = 7       loss = 5.7048318863e+01
Epoch = 8       loss = 5.7020624161e+01
Epoch = 9       loss = 5.6992567062e+01
Epoch = 10      loss = 5.6973577499e+01
```

We can save the trained model to disk, and later can load it back if we want.

```
model_si.save("final_model_si.pkl")
model_c.save("final_model_c.pkl")
loss.save_optimizer_state("optimizer_stat.pkl")
```

Total running time of the script: (0 minutes 1.955 seconds)

2.7 MCMC sampling

In this example, we demonstrate how to perform uncertainty quantification (UQ) using parallel tempered MCMC (PTMCMC). We use a Stillinger-Weber (SW) potential for silicon that is archived in OpenKIM.

For simplicity, we only set the energy-scaling parameters, i.e., `A` and `lambda` as the tunable parameters. Furthermore, these parameters are physically constrained to be positive, thus we will work in log parameterization, i.e. `log(A)` and `log(lambda)`. These parameters will be calibrated to energies and forces of a small dataset, consisting of 4 compressed and stretched configurations of diamond silicon structure.

To start, let's first install the SW model:

```
$ kim-api-collections-management install user SW_StillingerWeber_1985_Si__MO_
↪405512056662_006
```

See also:

This installs the model and its driver into the `User Collection`. See *Install a model* for more information about installing KIM models.

```
from multiprocessing import Pool

import numpy as np
from corner import corner
```

(continues on next page)

(continued from previous page)

```
from kliff.calculators import Calculator
from kliff.dataset import Dataset
from kliff.dataset.weight import MagnitudeInverseWeight
from kliff.loss import Loss
from kliff.models import KIMModel
from kliff.models.parameter_transform import LogParameterTransform
from kliff.uq import MCMC, autocorr, mser, rhat
from kliff.utils import download_dataset
```

Before running MCMC, we need to define a loss function and train the model. More detail information about this step can be found in *Train a Stillinger-Weber potential* and *Parameter transformation for the Stillinger-Weber potential*.

```
# Instantiate a transformation class to do the log parameter transform
param_names = ["A", "lambda"]
params_transform = LogParameterTransform(param_names)

# Create the model
model = KIMModel(
    model_name="SW_StillingerWeber_1985_Si__MO_405512056662_006",
    params_transform=params_transform,
)

# Set the tunable parameters and the initial guess
opt_params = {
    "A": [["default", -8.0, 8.0]],
    "lambda": [["default", -8.0, 8.0]],
}

model.set_opt_params(**opt_params)
model.echo_opt_params()

# Get the dataset and set the weights
dataset_path = download_dataset(dataset_name="Si_training_set_4_configs")
# Instantiate the weight class
weight = MagnitudeInverseWeight(
    weight_params={
        "energy_weight_params": [0.0, 0.1],
        "forces_weight_params": [0.0, 0.1],
    }
)
# Read the dataset and compute the weight
tset = Dataset(dataset_path, weight=weight)
configs = tset.get_configs()

# Create calculator
calc = Calculator(model)
ca = calc.create(configs)

# Instantiate the loss function
residual_data = {"normalize_by_natoms": False}
loss = Loss(calc, residual_data=residual_data)
```

(continues on next page)

(continued from previous page)

```
# Train the model
loss.minimize(method="L-BFGS-B", options={"disp": True})
model.echo_opt_params()
```

Out:

```
#=========================================================================================
# Model parameters that are optimized.
# Note that the parameters are in the transformed space if
# `params_transform` is provided when instantiating the model.
#=========================================================================================

A 1
  2.7268620056558381e+00  -8.0000000000000000e+00   8.0000000000000000e+00

lambda 1
  3.8184197679684773e+00  -8.0000000000000000e+00   8.0000000000000000e+00


/home/yonatank/.local/lib/python3.8/site-packages/numpy/linalg/linalg.py:2500:␣
↪VisibleDeprecationWarning: Creating an ndarray from ragged nested sequences (which is␣
↪a list-or-tuple of lists-or-tuples-or ndarrays with different lengths or shapes) is␣
↪deprecated. If you meant to do this, you must specify 'dtype=object' when creating the␣
↪ndarray.
  x = asarray(x)
2022-08-10 17:00:23.489 | INFO     | kliff.dataset.dataset:_read:398 - 4 configurations␣
↪read from /home/yonatank/modules/kliff/examples/Si_training_set_4_configs
2022-08-10 17:00:23.493 | INFO     | kliff.calculators.calculator:create:107 - Create␣
↪calculator for 4 configurations.
2022-08-10 17:00:23.494 | INFO     | kliff.loss:minimize:290 - Start minimization using␣
↪method: L-BFGS-B.
2022-08-10 17:00:23.496 | INFO     | kliff.loss:_scipy_optimize:404 - Running in serial␣
↪mode.
2022-08-10 17:00:23.675 | INFO     | kliff.loss:minimize:292 - Finish minimization using␣
↪method: L-BFGS-B.
#=========================================================================================
# Model parameters that are optimized.
# Note that the parameters are in the transformed space if
# `params_transform` is provided when instantiating the model.
#=========================================================================================

A 1
  2.7269268430321811e+00  -8.0000000000000000e+00   8.0000000000000000e+00

lambda 1
  3.8183682461406869e+00  -8.0000000000000000e+00   8.0000000000000000e+00



'#=========================================================================================\n#␣
↪Model parameters that are optimized.\n# Note that the parameters are in the␣
```

(continues on next page)

(continued from previous page)

```
↪transformed space if \n# `params_transform` is provided when instantiating the model.\n
↪#==========================================================================\n\nA␣
↪1\n  2.7269268430321811e+00  -8.0000000000000000e+00   8.0000000000000000e+00 \n\
↪nlambda 1\n  3.8183682461406869e+00  -8.0000000000000000e+00   8.0000000000000000e+00 \
↪n\n'
```

To perform MCMC simulation, we use *MCMC*.This class interfaces with ptemcee Python package to run PTMCMC, which utilizes the affine invariance property of MCMC sampling. We simulate MCMC sampling at several different temperatures to explore the effect of the scale of bias and overall error bars.

```
# Define some variables that correspond to the dimensionality of the problem
ntemps = 4  # Number of temperatures to simulate
ndim = calc.get_num_opt_params()  # Number of parameters
nwalkers = 2 * ndim  # Number of parallel walkers to simulate
```

We start by instantiating *MCMC*. This requires *Loss* instance to construct the likelihood function. Additionally, we can specify the prior (or log-prior to be more precise) via the `logprior_fn` argument, with the default option be a uniform prior that is bounded over a finite range that we specify via the `logprior_args` argument.

Note: When user uses the default uniform prior but doesn't specify the bounds, then the sampler will retrieve the bounds from the model (see *set_opt_params()*). Note that an error will be raised when the uniform prior extends to infinity in any parameter direction.

To specify the sampling temperatures to use, we can use the arguments `ntemps` and `Tmax_ratio` to set how many temperatures to simulate and the ratio of the highest temperature to the natural temperature T_0, respectively. The default values of `ntemps` and `Tmax_ratio` are 10 and 1.0, respectively. Then, an internal function will create a list of logarithmically spaced points from $T = 1.0$ to $T = T_{\text{max\_ratio}} \times T_0$. Alternatively, we can also give a list of the temperatures via `Tladder` argument, which will overwrites `ntemps` and `Tmax_ratio`.

Note: It has been shown that including temperatures higher than T_0 helps the convergence of walkers sampled at T_0.

The sampling processes can be parallelized by specifying the pool. Note that the pool needs to be declared after instantiating *MCMC*, since the posterior function is defined during this process.

```
# Set the boundaries of the uniform prior
bounds = np.tile([-8.0, 8.0], (ndim, 1))

# It is a good practice to specify the random seed to use in the calculation to generate
# a reproducible simulation.
seed = 1717
np.random.seed(seed)

# Create a sampler
sampler = MCMC(
    loss,
    ntemps=ntemps,
    logprior_args=(bounds,),
    random=np.random.RandomState(seed),
)
# Declare a pool to use parallelization
sampler.pool = Pool(nwalkers)
```

Note: As a default, the algorithm will set the number of walkers for each sampling temperature to be twice the number of parameters, but we can also specify it via the `nwalkers` argument.

To run the MCMC sampling, we use `run_mcmc()`. This function requires us to provide initial states p_0 for each temperature and walker. We also need to specify the number of steps or iterations to take.

Note: The initial states p_0 need to be an array with shape `(K, L, N,)`, where `K`, `L`, and `N` are the number of temperatures, walkers, and parameters, respectively.

```
# Initial starting point. This should be provided by the user.
p0 = np.empty((ntemps, nwalkers, ndim))
for ii, bound in enumerate(bounds):
    p0[:, :, ii] = np.random.uniform(*bound, (4, 4))

# Run MCMC
sampler.run_mcmc(p0, 5000)
sampler.pool.close()

# Retrieve the chain
chain = sampler.chain
```

The resulting chains still need to be processed. First, we need to discard the first few iterations in the beginning of each chain as a burn-in time. This is similar to the equilibration time in a molecular dynamic simulation before we can start the measurement. KLIFF provides a function to estimate the burn-in time, based on the Marginal Standard Error Rule (MSER). This can be accessed via `mser()`.

```
# Estimate equilibration time using MSER for each temperature, walker, and dimension.
mser_array = np.empty((ntemps, nwalkers, ndim))
for tidx in range(ntemps):
    for widx in range(nwalkers):
        for pidx in range(ndim):
            mser_array[tidx, widx, pidx] = mser(
                chain[tidx, widx, :, pidx], dmin=0, dstep=10, dmax=-1
            )

burnin = int(np.max(mser_array))
print(f"Estimated burn-in time: {burnin}")
```

Out:

```
Estimated burn-in time: 480
```

Note: `mser()` only compute the estimation of the burn-in time for one single temperature, walker, and parameter. Thus, we need to calculate the burn-in time for each temperature, walker, and parameter separately.

After discarding the first few iterations as the burn-in time, we only want to keep every τ-th iteration from the remaining chain, where τ is the autocorrelation length, to ensure uncorrelated samples. This calculation can be done using `autocorr()`.

```
# Estimate the autocorrelation length for each temperature
chain_no_burnin = chain[:, :, burnin:]

acorr_array = np.empty((ntemps, nwalkers, ndim))
for tidx in range(ntemps):
    acorr_array[tidx] = autocorr(chain_no_burnin[tidx], c=1, quiet=True)

thin = int(np.ceil(np.max(acorr_array)))
print(f"Estimated autocorrelation length: {thin}")
```

Out:

```
Estimated autocorrelation length: 15
```

Note: `acorr()` is a wrapper for emcee.autocorr.integrated_time, As such, the shape of the input array for this function needs to be `(L, M, N,)`, where `L`, `M`, and `N` are the number of walkers, steps, and parameters, respectively. This also implies that we need to perform the calculation for each temperature separately.

Finally, after obtaining the independent samples, we need to assess whether the resulting samples have converged to a stationary distribution, and thus a good representation of the actual posterior. This is done by computing the potential scale reduction factor (PSRF), denoted by $\hat{R}^p$. The value of $\hat{R}^p$ declines to 1 as the number of iterations goes to infinity. A common threshold is about 1.1, but higher threshold has also been used.

```
# Assess the convergence for each temperature
samples = chain_no_burnin[:, :, ::thin]

threshold = 1.1  # Threshold for rhat
rhat_array = np.empty(ntemps)
for tidx in range(ntemps):
    rhat_array[tidx] = rhat(samples[tidx])

print(f"$\hat{{r}}^p$ values: {rhat_array}")
```

Out:

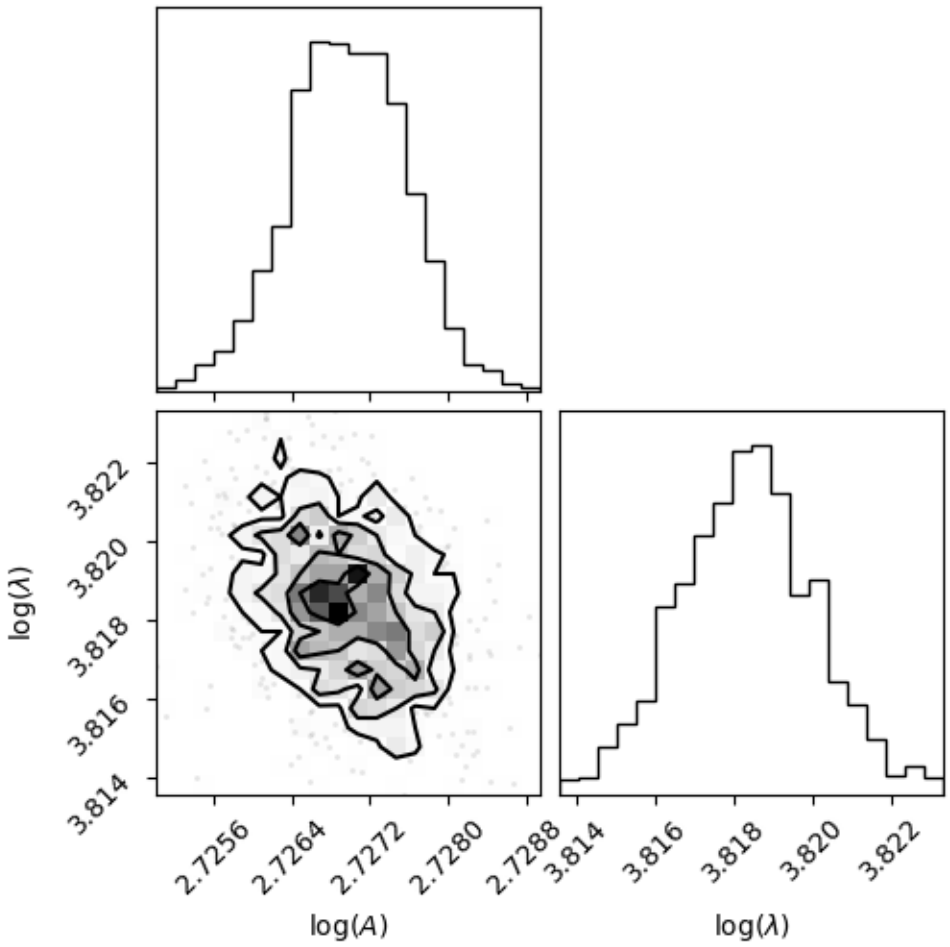
```
$\hat{r}^p$ values: [0.99832965 1.07774255 1.10594832 1.07248159]
```

Note: `rhat()` only computes the PSRF for one temperature, so that the calculation needs to be carried on for each temperature separately.

Notice that in this case, $\hat{R}^p < 1.1$ for all temperatures. When this criteria is not satisfied, then the sampling process should be continued. Note that some sampling temperatures might converge at slower rates compared to the others.

After obtaining the independent samples from the MCMC sampling, the uncertainty of the parameters can be obtained by observing the distribution of the samples. As an example, we will use corner Python package to present the MCMC result at sampling temperature 1.0 as a corner plot.

```
# Plot samples at T=1.0
corner(samples[0].reshape((-1, ndim)), labels=[r"$\log(A)$", r"$\log(\lambda)$"])
```



Out:

```
<Figure size 550x550 with 4 Axes>
```

Note: As an alternative, KLIFF also provides a wrapper to emcee. This can be accessed by setting `sampler="emcee"` when instantiating *MCMC*. For further documentation, see `EmceeSampler`.

Total running time of the script: (3 minutes 14.012 seconds)

CHAPTER

THREE

THEORY

A parametric potential typically takes the form

$$\mathcal{V} = \mathcal{V}(r_1, \ldots, r_{N_a}, Z_1, \ldots, Z_{N_a}; \theta)$$

where $r_1, \ldots, r_{N_a}$ and $Z_1, \ldots, Z_{N_a}$ are the coordinates and species of a system of N_a atoms, respectively, and θ denotes a set of fitting parameters. For notational simplicity, in the following discussion, we assume that the atomic species information is implicitly carried by the coordinates and thus we can exclude Z from the functional form, and use R to denote the coordinates of all atoms in the configuration. Then we have

$$\mathcal{V} = \mathcal{V}(R; \theta).$$

A potential parameterization process is typically formulated as a weighted least-squares minimization problem, where we adjust the potential parameters θ so as to reproduce a training set of reference data obtained from experiments and/or first-principles computations. Mathematically, we hope to minimize a loss function

$$\mathcal{L}(\theta) = \frac{1}{2} \sum_{i=1}^{N_p} \| w_i (p_i(\mathcal{V}(R_i; \theta)) - q_i) \|^2$$

with respect to θ, where $\{q_1, \ldots, q_{N_p}\}$ is a training set of N_p reference data, p_i is the corresponding prediction for q_i computed from the potential (as indicated by its argument), $\| \cdot \|$ denote the L_2 norm, and w_i is the weight for the i-th data point. We call

$$u = p(\mathcal{V}(R; \theta)) - q$$

the residual function that characterizes the difference between the potential predictions and the reference data for a set of properties.

Generally speaking, q can be a collection of any material properties considered important for a given application, such as the cohesive energy, equilibrium lattice constant, and elastic constants of a given crystal phase. These materials properties can be obtained from experiments and/or first-principles calculations. However, nowadays, most of the potentials are trained using the *force-matching* scheme, where the potential is trained to a large set of forces on atoms (and/or energies, stresses) obtained by first-principles calculations for a set of atomic configurations. This is extremely true for machine learning potentials, where a large set of training data is necessary, and it seems impossible to collect sufficient number of material properties for the training set.

The reference q and the prediction p are typically represented as vectors such that $q[m]$ is the m-th reference property and $p[m]$ is the corresponding m-th prediction obtained from the potential. Assuming we want to fit a potential to energy and forces, then q is a vector of size $1 + 3N_a$, in which N_a is the number of atoms in a configuration, with

$$\begin{aligned}
q[0] &= E_{\text{ref}} \\
q[1] &= f_{\text{ref}}^{0,x}, \quad q[2] = f_{\text{ref}}^{0,y}, \quad q[3] = f_{\text{ref}}^{0,z}, \\
q[4] &= f_{\text{ref}}^{1,x}, \quad q[5] = f_{\text{ref}}^{1,y}, \quad q[6] = f_{\text{ref}}^{1,z}, \\
&\ldots \\
q[3N_a - 2] &= f_{\text{ref}}^{N_a-1,x}, \quad q[3N_a - 1] = f_{\text{ref}}^{N_a-1,y}, \quad q[3N_a] = f_{\text{ref}}^{N_a-1,z},
\end{aligned}$$

where E_{ref} is the reference energy, and $f_{\text{ref}}^{i,x}$, $f_{\text{ref}}^{i,y}$, and $f_{\text{ref}}^{i,z}$ denote the x-, y-, and z-component of reference force on atom i, respectively. In other words, we put the energy as the 0th component of q, and then put the force on the first atom as the 1st to 3rd components of q, the force on the second atom the next three components till the forces on all atoms are placed in q. In the same fashion, we can construct the prediction vector p, and then to compute the residual vector.

Note: We use boldface with subscript to denote a data point (e.g. q_i means the i-th data point in the training set), and use normal text with square bracket to denote the component of a data point (e.g. : $q[m]$ indicates the m-th component of a general data point q.

If stress is used in the fitting, $q[3N_a]$ to $q[3N_a + 5]$ will store the reference Voigt stress $\sigma_{xx}, \sigma_{yy}, \sigma_{zz}, \sigma_{yz}, \sigma_{xy}, \sigma_{xz}$, and, of course, $p[3N_a]$ to $p[3N_a + 5]$ are the corresponding predictions computed from the potential.

The objective of the parameterization process is to find a set of parameters θ of potential that reproduce the reference data as well as possible.

CHAPTER

FOUR

FREQUENTLY USED MODULES

In this section, we introduce some of the most frequently used modules. See *Package Reference* for their APIs and other modules in KLIFF.

Quick links

Models	*Dataset*
Calculators	*Loss*

Note: See also *Package Reference*.

4.1 Dataset

This Module contains classes and functions to deal with dataset.

A dataset is comprised of a set of configurations, which provide the training data to optimize potential (parameters) or provide the test data to test the quality of potential.

A configuration should have three `lattice vectors` of the simulation cell, flags to indicate periodic boundary conditions (`PBC`), `species` and `coordinates` of all atoms. These collectively define a configuration and are, typically, considered as the input in terms of potential fitting. A configuration should also contain a set of output (target), which the optimization algorithm adjust the potential (parameters) to match. For example, if the force-matching scheme is used for the fitting, the output can be the forces on individual atoms. The currently supported outputs include `energy`, `forces`, and `stress`.

See also:

See *kliff.dataset.Configuration* for a complete list of the member functions of the *Configuration* class.

To create a data, do:

```
from kliff.dataset import Dataset
path = 'path_to_my_dataset_files'
dset = Dataset(path, format='extxyz')
```

where `path` is a file storing a configuration or a directory containing multiple files. If given a directory, all the files in this directory and its subdirectories with the extension corresponding to the specified format will be read. For example, if `format='extxyz'`, all the files with an extension `.xyz` in `path` and its subdirectories will be read.

The size of the dataset can be obtained by:

```
dset_size = dset.get_num_configs()
```

and a list of configurations constituting the dataset can be obtained by:

```
configs = dset.get_configs()
```

See also:

See *kliff.dataset.Dataset* for a complete list of the member functions of the *Dataset* class.

4.1.1 Inspect dataset

KLIFF provides a command line tool to get a statistics of a dataset of files. For example, for the `Si_training_set.tar.gz` (the tarball can be extracted by: `$ tar xzf Si_training_set.tar.gz`), running:

```
$ kliff dataset --count Si_training_set
```

prints out the below information:

```
================================================================================
                            KLIFF Dataset Count

Notation: "--dir_name (a/b)"
a: number of .xyz files in the directory "dir_name"
b: number of .xyz files in the directory "dir_name" and its subdirectories

Si_training_set (0/1000)
├──NVT_runs (600/600)
└──varying_alat (400/400)

================================================================================
```

4.1.2 Dataset Format

More than often, your dataset is generated from first-principles calculations using packages like *VASP*, *SIESTA*, and *Quantum Espresso* among others. Their output file format may not be support by KLIFF. You can use parse these output to get the necessary data, and then convert to the format supported by KLIFF using the functions `kliff.dataset.write_config()` and `kliff.dataset.read_config()`.

Currently supported dataset format include:

- extended XYZ (.xyz)

Extended XYZ

The Extended XYZ format is an enhanced version of the basic XYZ format that allows extra columns to be present in the file for additional per-atom properties as well as standardizing the format of the comment line to include the cell lattice and other per-frame parameters. It typically has the `.xyz` extension.

It would be easy to explain the format with an example. Below is an example of the extended XYZ format supported by KLIFF:

```
8
Lattice="4.8879 0 0 0 4.8879 0 0 0 4.8879"  PBC="1 1 1"  Energy=-29.3692121943 ␣
↪Properties=species:S:1:pos:R:3:force:R:3
Si     0.00000e+00   0.00000e+00   0.00000e+00  2.66454e-15  -8.32667e-17   4.02456e-16
Si     2.44395e+00   2.44395e+00   0.00000e+00  1.62370e-15   7.21645e-16   8.46653e-16
Si     0.00000e+00   2.44395e+00   2.44395e+00  0.00000e+00   3.60822e-16   2.01228e-16
Si     2.44395e+00   0.00000e+00   2.44395e+00  1.33227e-15  -4.44089e-16   8.74350e-16
Si     1.22198e+00   1.22198e+00   1.22198e+00  4.44089e-15   1.80411e-16   1.87350e-16
Si     3.66593e+00   3.66593e+00   1.22198e+00  9.29812e-16  -2.67841e-15  -3.22659e-16
Si     1.22198e+00   3.66593e+00   3.66593e+00  5.55112e-17   3.96905e-15   8.87786e-16
Si     3.66593e+00   1.22198e+00   3.66593e+00 -2.60902e-15  -9.43690e-16   6.37999e-16
```

- The first line list the number of atoms in the system.
- The second line follow the `key=value` structure. if a `value` contains any space (e.g. `Lattice`), it should be placed in the quotation marks `" "`. The supported keys are:
  - `Lattice` represents the three Cartesian lattice vectors: the first 3 numbers denote a_1, the next three numbers denote a_2, and the last 3 numbers denote a_3. Note that a_1, a_2, and a_3 should follow the right-hand rule such that the volume of the cell can be obtained by $(a_1 \times a_2) \cdot a_3$.
  - `PBC`. Three integers of `1` or `0` (or three characters of `T` or `F`) to indicate whether to use periodic boundary conditions along a_1, a_2, and a_3, respectively.
  - `Energy`. A real value of the total potential energy of the system.
  - `Properties` provides information of the names, size, and types of the data that are listed in the body part of the file. For example, the `Properties` in the above example means that the atomic species information (a string) is listed in the first column of the body, the next three columns list the atomic coordinates, and the last three columns list the forces on atoms.

Each line in the body lists the information, indicated by `Properties` in the second line, for one atom in the system, taking the form:

```
species  x  y  z  fx  fy  fz
```

The coordinates `x y z` should be given in Cartesian values, not fractional values. The forces `fx fy fz` can be skipped if you do not want to use them.

Note: An atomic configuration stored in the extended XYZ format can be visualized using the OVITO program.

4.1.3 Weight

As mentioned in *Theory*, the reference q can be any material properties, which can carry different physical units. The weight in the loss function can be used to put quantities with different units on a common scale. The weights also give us access to set which properties or configurations are more important, for example, in developing a potential for a certain application (see *Define your weight class*).

KLIFF uses weight class to compute and store the weight information for each configuration. The basic structure of the class is shown below.

```
class Weight():
    """A class to deal with weights for each configuration."""

    def __init__(self):
        #... Do necessary steps to initialize the class

    def compute_weight(self, config):
        #... Compute the weights for the given configutation

    @property
    def some_weight(self):
        #... Add properties to retrieve the weight values
```

Default weight class

KLIFF has several built-in weight classes. As a default, KLIFF uses `kliff.dataset.weight.Weight`, which put a single weight for each property.

```
from kliff.dataset import Dataset
from kliff.dataset.weight import  Weight

path = 'path_to_my_dataset_files'
weight = Weight()
dset = Dataset(path, weight=weight, format='extxyz')

# Retrieve the weights
config_weight = configs[0].config_weight
energy_weight = configs[0].energy_weight
forces_weight = configs[0].forces_weight
stress_weight = configs[0].stress_weight
```

`config_weight` is the weight for the configuration and `energy_weight`, `forces_weight`, and `stress_weigth` are the weights for energy, forces, and stress, respectively. The default value for each weight is 1.0.

One can also specify different values for these weights. For example, one might want to weigh the energy 10 times as the forces. It can be done by specifying the weight values while instantiating `kliff.dataset.weight.Weight`.

```
weight = Weight(
    config_weight=1.0, energy_weight=10.0, forces_weight=1.0, stress_weight=1.0
)
```

Note: Another use case is if one wants to, for example, exclude the energy in the loss function, which can be done by

setting `energy_weight=0.0`.

Magnitude-inverse weight

KLIFF also provides another weight class that computes the weight based on the magnitude of the data, applying different weight on each data point. The weight calculation is motivated by formulation suggested by Lenosky et al. [lenosky1997],

$$\frac{1}{w_i}^2 = c_1^2 + c_2^2 \|p_i\|^2$$

c_1 and c_2 are parameters to compute the weight. They can be thought as a padding and a fractional scaling terms. When p_i corresponds to energy, the norm is the absolute value of the energy. When p_i correspond to forces, the norm is a vector norm of the force vector acting on the corresponding atom. This also mean that each force component acting on the same atom will have the same weight. If p_i correspond to stress, then the norm is a Frobenius norm of the stress tensor, giving the same weight for each component in the stress tensor.

To use this weight, we instantiate `MagnitudeInverseWeight` weight class:

```
from kliff.dataset.weight import MagnitudeInverseWeight
weight = MagnitudeInverseWeight(
    config_weight=1.0,
    weight_params={
        "energy_weight_params": [c1e, c2e],
        "forces_weight_params": [c1f, c2f],
        "stress_weight_params": [c1s, c2s],
    }
)
```

`config_weight` specifies the weight for the entire configuration.

`weight_params` is a dictionary containing c_1 and c_2 for energy, forces, and stress. The default value is:

```
weight_params = {
    "energy_weight_params": [1.0, 0.0],
    "forces_weight_params": [1.0, 0.0],
    "stress_weight_params": [1.0, 0.0],
}
```

Additionally, for each key, we can pass in a `float`, which set the value of c_1 with $c_2 = 0.0$.

Define your weight class

We can also define a custom weight class to use in KLIFF. As an example, suppose we are developing a potential that will be used to investigate fracture properties. The training sets includes both configurations with and without cracks. For this application, we might want to put larger weights for the configurations with cracks. Below is an example of weight class that achieve this goal.

```
from kliff.dataset.weight import Weight

class WeightForCracks(Weight):
    """An example weight class that put larger weight on the configurations with
    cracks. This class inherit from ``kliff.dataset.weight.Weight``. We just need to
```

(continues on next page)

(continued from previous page)

```
    modify ``compute_weight`` method to put larger weight for the configurations with
    cracks. Other modifications might need to be done for different weight class.
    """

    def __init__(self, energy_weight, forces_weight):
        super().__init__(energy_weight=energy_weight, forces_weight=forces_weight)

    def compute_weight(self, config):
        identifier = config.identifier
        if 'with_cracks' in identifier:
            self._config_weight = 10.0
```

With this weight class, we can use the built-in `residual_fn` to achieve the same result as the implementation in *Use your own residual function*.

4.2 Models

4.2.1 KIM models

4.2.2 Neural network models

4.3 Descriptors

4.3.1 Symmetry functions

4.3.2 Bispectrum

4.4 Calculators

A calculator is the central agent that exchanges information between a model and the minimizer.

- It uses the computation methods provided by a model to calculate the energy, forces, etc. and pass these properties, together with the corresponding reference data, to `Loss` to construct a loss function to be minimized by the optimizer.
- It also inquires the model to get parameters that are going to be optimized, and provide these parameters to the optimizer, which will be used as the initial values by the optimizer to carry out the optimization.
- In the reverse direction, at each optimization step, the calculator grabs the new parameters from the optimizer and update the model parameters with the new ones. So, in the next minimization step, the loss function will be calculated using the new parameters.

A calculator for the physics-motivated potential can be created by:

```
from kliff.calculators import Calculator

model = ...  # create a model
configs = ...  # get a list of configurations
calc = Calculator(model)
calc.create(configs, use_energy=True, use_forces=True, use_stress=False)
```

It creates a calculator for a `model` (discussed in *Models*), and `configs` is a list of *Configuration* (discussed in *Dataset*), for which the calculator is going to make predictions. `use_energy`, `use_forces`, and `use_stress` inform the calculator whether *energy*, *forces*, and *stress* will be requested from the calculator. If the potential is to be trained on *energy* only, it would be better to set `use_forces` and `use_stress` to `False`, which turns off the calculations for `forces` and `stress` and thus can speed up the fitting process.

Other methods of the calculator include:

- *Initialization*: *get_compute_arguments()*.
- *Property calculation using a model*: *compute()*, *get_compute_arguments()*, *compute()*, *get_energy()*, *get_forces()*, *get_stress()*, *get_prediction()*, *get_reference()*.
- *Optimizing parameters*: *get_opt_params()*, *get_opt_params_bounds()*, *update_model_params()*.

See also:

See *kliff.calculators.Calculator* for a complete list of the member functions and their docs.

4.5 Loss

As discussed in *Theory*, we solve a minimization problem to fit the potential. For physics-motivated potentials, the geodesic Levenberg-Marquardt (`geodesicLM`) minimization method [transtrum2012geodesicLM] can be used, which has been shown to perform well for potentials in [wen2016potfit]. KLIFF also interacts with SciPy to utilize the zoo of optimization methods there. For machine learning potentials, KLIFF wraps the optimization methods in PyTorch.

KLIFF provides a uniform interface to use all the optimization methods. To carry out optimization, first create a loss object:

```
from kliff.loss import Loss

calculator = ...  # create a calculator
Loss(calculator, nprocs=1, residual_fn=None, residual_data=None)
```

`calculator` (discussed in *Calculators*) provides predictions calculated using a potential and the corresponding reference data via `get_prediction()` and `get_reference()`, respectively, which the optimizer can be used to construct the objective function.

`nprocs` informs KLIFF the number of cores that KLIFF can use to parallelize over the dataset to evaluate the objective function.

`residual_data` is a dictionary that will be used by `residual_fn` to compute the residual. `residual_data` is optional, and its default is:

```
residual_data = {'normalize_by_natoms': True}
```

The meaning of this value is made clear in the below discussion.

`residual_fn` is a function used to compute the residual. As discussed in *Theory*, the objective function is a sum of the square of the norm of the residual of each individual configuration, i.e.

$$\mathcal{L}(\theta) = \frac{1}{2}\sum_{i=1}^{N_p} \|w_i u_i\|^2$$

with the residual

$$u_i = p_i - q_i,$$

in which p_i is a vector of predictions computed using the potential for the i-th configuration, and q_i is a vector of the corresponding reference data. The residual is computed using the `residual_fn`, which should be of the form

```
def residual_fn(identifier, natoms, weight, prediction, reference, data):
    """A function to compute the residual for a configuration."""

    # ... compute u based on p (prediction) and q (reference)
    # and it should be a vector
    return u
```

In the above residual function,

- `identifier` is a (unique) `str` associated with the configuration, which is specified in *Configuration*. If it is not provided there, `identifier` is default to the path to the file that storing the configuration, e.g. `Si_training_set/NVT_runs/T300_step100.xyz`.
- `natoms` is an `int` denoting the number of atoms in the configuration.
- `weight` is a `Weight` instance that generates the weights from the configuration (see *Weight*).
- `prediction` is a vector of the prediction p computed from the potential.
- `reference` is a vector of the corresponding reference data q.
- `data` is `residual_data` provided at the initialization of `Loss`. `residual_data` is a dictionary, with which the user can provide extra information to `residual_fn`.

`residual_fn` is also optional, and it defaults to *energy_forces_residual()* discussed below.

4.5.1 Built-in residual function

KLIFF provides a number of residual functions readily to be plugged into `Loss` and let the wheel spin. For example, the *energy_forces_residual()* that constructs the residual using energy and forces is defined as (in a nutshell):

```
def energy_forces_residual(identifier, natoms, weight, prediction, reference, data):

    # extract up the weight information
    config_weight = weight.config_weight
    energy_weight = weight.energy_weight
    forces_weight = weight.forces_weight

    # obtain residual and properly normalize it
    residual = config_weight * (prediction - reference)
    residual[0] *= energy_weight
    residual[1:] *= forces_weight

    if data["normalize_by_natoms"]:
        residual /= natoms

    return residual
```

This residual function retrieves the weights for energy and forces f``weight`` instance and enables the normalization of the residual based on the number of atoms. Normalization by the number of atoms makes each individual configuration in the training set contributes equally to the loss function; otherwise, configurations with more atoms can dominate the loss, which (most of the times) is not what we prefer.

One can provide a `residual_data` instead of using the default one to tune the loss, for example, if one wants to ignore the normalization by the number of atoms.

```
from kliff.loss import Loss
from kliff.loss import energy_forces_residual

calculator = ...  # create a calculator

# provide my data
residual_data = {'normalize_by_natoms': False}
Loss(calculator, nprocs=1, residual_fn=energy_forces_residual, residual_data=residual_
↪data)
```

Warning: Even though `residual_fn` and `residual_data` is optional, we strongly recommend the users to explicitly provide them to reminder themselves what they are doing as done above.

See also:

See *kliff.loss* for other built-in residual functions.

4.5.2 Use your own residual function

The built-in residual function treats each configuration in the training set, and each atom in a configuration equally important. Sometimes, this may not be what you want. In these cases, you can define and use your own `residual_fn`.

For example, if you are creating a potential that is going to be used to investigate fracture properties, and your training set include both configurations with cracks and configurations without cracks, then you may want to weigh more for the configurations with cracks.

```
from kliff.loss import Loss

# define my own residual function
def residual_fn(identifier, natoms, weight, prediction, reference, data):

    # extract the weight information
    config_weight = weight.config_weight
    energy_weight = weight.energy_weight
    forces_weight = weight.forces_weight

    # larger weight for configuration with cracks
    if 'with_cracks' in identifer:
        config_weight *= 10

    normalize = data["normalize_by_natoms"]
    if normalize:
        energy_weight /= natoms
        forces_weight /= natoms

    # obtain residual and properly normalize it
    residual = config_weight * (prediction - reference)
    residual[0] *= energy_weight
    residual[1:] *= forces_weight

    return residual
```

(continues on next page)

(continued from previous page)

```
calculator = ...  # create a calculator
loss = Loss(
    calculator,
    nprocs=1,
    residual_fn=residual_fn,
    residual_data={"normalize_by_natoms": True}
)
```

The above code takes advantage of `identifier` to distinguish configurations with cracks and without cracks, and then weigh more for configurations with cracks.

For configurations with cracks, you may even want to weigh more for the atoms near the creak tip. Then you need to identify which atoms are near the crack tip and manipulate the corresponding components of `residual`.

Note: If you are using your own `residual_fn`, its `data` argument can be completely ignored since it can be directly provided in your own `residual_fn`.

See also:

See *Define your weight class* for an alternative implementation of this example.

Note: Handling the weight is preferably done using the weight class (see *Weight*) instead of in the residual function.

4.6 Uncertainty Quantification (UQ)

Uncertainty quantification (UQ) is an emerging field in applied mathematics that aims to quantify uncertainties in mathematical models as a result from error propagation in the modeling process. This is especially important since we use the model, i.e., the potential, to predict material properties that are not used in the training process. Thus, UQ process is especially important to assess the reliability of these out-of-sample predictions.

In UQ process, we first quantify the uncertainty of the model parameters. Having found the parametric uncertainty, then we can propagate the uncertainty of the parameters and get the uncertainty of the material properties of interest, e.g., by evaluating the ensemble that is obtained from sampling the distribution of the parameters. As the first uncertainty propagation is more involved, KLIFF implements tools to quantify the uncertainty of the parameters.

4.6.1 MCMC

The Bayesian Markov chain Monte Carlo (MCMC) is the UQ method of choice for interatomic potentials. The distribution of the parameters is given by the posterior $P(\theta|q)$. By Bayes' theorem

$$P(\theta|q) \propto L(\theta|q) \times \pi(\theta),$$

where $L(\theta|q)$ is the likelihood (which encodes the information learned from the data) and $\pi(\theta)$ is the prior distribution. Then, some MCMC algorithm is used to sample the posterior and the distribution of the parameters is inferred from the distribution of the resulting samples.

The likelihood function is given by

$$L(\theta|q) \propto \exp\left(-\frac{\mathcal{L}(\theta)}{T}\right).$$

The inclusion of the sampling temperature T is to account for model inadequacy, or bias, in the potential [Kurniawan2022]. Frederiksen et al. (2004) [Frederiksen2004] suggest to estimate the bias by setting the temperature to

$$T_0 = \frac{2\mathcal{L}_0}{N},$$

where $\mathcal{L}_0$ is the value of the loss function evaluated at the best fit parameters and N is the number of tunable parameters.

See also:

For more discussion about this formulation, see [KurniawanKLIFFUQ].

4.6.2 MCMC implementation

In KLIFF, the UQ tools are implemented in *uq*. In the currect version, only MCMC sampling is implemented, with the integration of other UQ methods will be added in the future.

For the MCMC sampling, KLIFF adopts parallel-tempered MCMC (PTMCMC) methods, via the ptemcee Python package, as a way to perform MCMC sampling with several different temperatures. Additionally, multiple parallel walkers are deployed for each sampling temperature. PTMCMC has been widely used to improve the mixing rate of the sampling. Furthermore, by sampling at several different temperatures, we can assess the effect of the size of the bias to any conclusion drawn from the samples.

We start the UQ process by instantiating *MCMC*,

```
from kliff.uq import MCMC

loss = ...  # define the loss function
sampler = MCMC(
    loss, nwalkers, logprior_fn, logprior_args, ntemps, Tmax_ratio, Tladder, **kwargs
)
```

As a default, *MCMC* inherits from ptemcee.Sampler. The arguments to instantiate the sampler are:

- `loss`, which is a *Loss* instance. This is a required argument to construct the untempered likelihood function ($T = 1$) and to compute T_0.
- `nwalkers` specifies the number of parallel walkers to run for each sampling temperature. As a default, this quantity is set to twice the number of parameters in the model.
- `logprior_fn` argument allows user to specify the prior distribution $\pi(\theta)$ to use. The function should accept an array of parameter values as an input and compute the logarithm of the prior distribution. Note that the prior distribution doesn't need to be normalized. The default prior is a uniform distribution over a finite range. See the next argument on how to set the boundaries of the uniform prior.
- `logprior_args` is a tuple that contains additional positional arguments needed by `logprior_fn`. If the default uniform prior is used, then the boundaries of the prior support (where $\pi(\theta) > 0$) need to be specified here as a $N \times 2$ array, where the first and second columns of the array contain the lower and upper bound for each parameter.
- `ntemps` specifies the number of temperatures to simulate.
- `Tmax_ratio` is used to set the highest temperature by $T_{\text{max}} = T_{\text{max\_ratio}} \times T_0$. An internal function is used to construct a list of logarithmically spaced `ntemps` points from 1.0 to T_{max}, inclusive.

- `Tladder` allows user to specify a list of temperatures to use. This argument will overwrites `ntemps` and `Tmax_ratio`.
- Other keyword arguments to be passed into ptemcee.Sampler needs to be specified in `kwargs`.

After the sampler is created, the MCMC run is done by calling `run_mcmc()`.

```
p0 = ...  # Define the initial position of each walker
sampler.run_mcmc(p0, iterations, *args, **kwargs)
```

The required arguments are:

- `p0`, which is a $K \times L \times N$ array containing the position of each walker for each temperature in parameter space, where K, L, and N are the number of temperatures, walkers, and parameters, respectively.
- `iterations` specifies the number of MCMC step to take. Since the position is step i in Markov chain only depends on step $(i-1)$, it is possible to break up the MCMC run into smaller batches, with the note that the initial positions of the current run needs to be set to the last positions of the previous run.

See also:

For other possible arguments, see also ptemcee.Sampler.run_mcmc.

The resulting chain can be retrieved from via `sampler.chain` as a $K \times L \times M \times N$ array, where M is the total number of iteratiions.

Parallelization

In principle, parallelization for the MCMC run can be done in 2 places: in the likelihood (or loss function) evaluation for each parameter set (see *Run in parallel mode*) and in the likelihood evaluation across different walkers. In the current implementation we supports OpenMP-style parallelization in the loss evaluation and both OpenMP and MPI for the sampling for different walkers when running MCMC sampling.

In general, parallelization in the sampling process is done by declaring a pool and set it to `sampler.pool` prior to running MCMC, for example:

```
from multiprocessing import Pool

sampler.pool = Pool(nprocs)  # nprocs is the number of parallel process to use
sampler.run_mcmc(p0, iterations, *args, **kwargs)
```

To do parallelization with MPI, we can utilize `MPIPool` from schwimmbad:

```
from schwimmbad import MPIPool

sampler.pool = MPIPool()
sampler.run_mcmc(p0, iterations, *args, **kwargs)
```

and run the Python script with `mpiexec` bash command.

If enough compute resources are available, we can also employ a hybrid parallelization, for example, using `multiprocessing` in the loss evaluation (by specifying argument `nprocs` > 1) and MPI in the likelihood evaluation across different walkers. Then, we can run the Python script as follows.

```
$ export MPIEXEC_OPTIONS="--bind-to core --map-by slot:PE=<num_openmp_processes> port-
↪bindings"
$ mpiexec -np <num_mpi_workers> ${MPIEXEC_OPTIONS} python script.py
```

4.6.3 MCMC analysis

The chains from the MCMC simulation needs to be processed. In a nutshell, the steps to take are

- Estimate the burn-in time and discard it from the beginning of the chain,
- Estimate the autocorrelation length, τ, and only take every τ step from the remaining chain,
- Assess convergence of the samples, i.e., the remaining chain after the two steps above.

Burn-in time

First we need to discard the first few iterations in the beginning of each chain as a burn-in time. This is similar to the equilibration time in a molecular dynamics simulation before the measurement. This action also ensure that the result is independent of the initial positions of the walkers.

KLIFF provides a function to estimate the burn-in time, based on the Marginal Standard Error Rule (MSER). This can calculation can be done using the function `mser()`. However, note that this calculation needs to be performed for each temperature, walker, and parameter dimension separately.

Autocorrelation length

In Markov chain, the position at step i is not independent from the previous step. However, after several iterations (denote this number by τ, which is the autocorrelation length), the walker will "forget" where it started, i.e., the position at step i is independent from step $(i + \tau)$. Thus, we need to only keep every τ-th step to obtain the independent, unceorrelated samples.

The estimation of the autocorrelation length in KLIFF is done via the function `autocorr()`, which wraps over `emcee.autocorr.integrated_time`. This calculation needs to be done for each temperature independently. The required input argument is a $L \times \tilde{M} \times N$ array, where L and N are the number of walkers and parameters, respectively, and $\tilde{M}$ is the remaining number of iterations after discarding the burn-in time.

Convergence

Finally, after a sufficient number of iterations, the distribution of the MCMC samples will converge to the posterior. For multi-chain MCMC simulation, the convergence can be assessed by calculating the multivariate potential scale reduction factor, denoted by $\hat{R}^p$. This quantity compares the variance between and within independent chains. The value of $\hat{R}^p$ declines to 1 as the number of iterations goes to infinity, with a common threshold is about 1.1.

In KLIFF, the function *rhat()* computes $\hat{R}^p$ for one temperature. The required input argument is a $L \times \tilde{M}^* \times N$ array of independent samples ($\tilde{M}^*$ is the number of independent samples in each walker). When the resulting $\hat{R}^p$ values are larger than the threshold (e.g., 1.1), then the MCMC simulation should be continued until this criteria is satisfied.

Note: Some sampling temperatures might converge at slower rates compared to the others. So, user can terminate the MCMC simulation as long as the samples at the target temperatures, e.g., T_0, have converged.

See also:

See the tutorial for running MCMC in *MCMC sampling*.

CHAPTER

FIVE

HOW TO

Some (incomplete) examples demonstrating how to use KLIFF.

5.1 Save and load a model

Once you've trained a model, you can save it disk and load it back later for the purpose of retraining, evaluation, etc.

5.1.1 Save a model

The `save()` method of a model can be used to save it. Suppose you've trained the Stillinger-Weber (SW) potential discussed in *Train a Stillinger-Weber potential*, you can save the model by:

```
path = "./kliff_model.pkl"
model.save(path)
```

which creates a pickled file named `kliff_model.pkl` in the current working directory. All the information related to the model are saved to the file, including the final values of the parameters, the constraints on the parameters (such as the bounds on parameters set via *set_one_opt_param()* or *set_opt_params()*), and others.

5.1.2 Load a model

A model can be loaded using `load()` after the instantiation. For the same SW potential discussed in *Train a Stillinger-Weber potential*, it can be loaded by:

```
model = KIMModel(model_name="Three_Body_Stillinger_Weber_Si__MO_405512056662_004")
path = "./kliff_model.pkl"
model.load(path)
```

If you want to retrain the loaded model, you can attach it to a calculator and then proceed as what discussed in *Train a Stillinger-Weber potential* and *Train a neural network potential*.

5.2 Install a model

5.2.1 Install a KIM model

The `kim-api-collections-management` command line tool from the kim-api makes it easy to install a model archived on the OpenKIM website. You can do:

```
$ kim-api-collections-management install user <model_name>
```

to automatically download a model from the OpenKIM website, and install it to the `User Collection`. Just replace `<model_name>` with the `Extended KIM ID` of the model you want to install as listed on OpenKIM. To see that the model has been successfully installed, do:

```
$ kim-api-collections-management list
```

See also:

A model can be installed into a different collection other than the `User Collection` specified by `user`. You can use `kim-api-collections-management` to remove and reinstall models. See the kim-api documentation for more information.

5.2.2 Install a KLIFF-trained model

As discussed in *Train a Stillinger-Weber potential* and *Train a neural network potential*, you can write a trained model to a KIM model that is compatible with the kim-api by:

```
path = "./kliff_trained_model"
model.write_kim_model(path)
```

Which writes to the current working directory a directory named `kliff_trained_model`.

Note: The `path` argument is optional, and KLIFF automatically generates a `path` if it is `None`.

To install the local `kliff_trained_model`, do:

```
$ kim-api-collections-management install user kliff_trained_model
```

which installs the model into the *User Collection* of the kim-api, and, of course, you can see the installed model by:

```
$ kim-api-collections-management list
```

The installed model can then be used with simulation codes like LAMMPS, ASE, and GULP via the kim-api.

5.3 Implement a new model

5.4 Run in parallel mode

KLIFF supports parallelization over data. It can be run on shared-memory multicore desktop machines as well as HPCs composed of multiple standalone machines connected by a network.

5.4.1 Physics-based models

We implement two parallelization schemes for physics-based models. The first is suitable to be used on a desktop machine and the second is targed for HPCs.

multiprocessing

This scheme uses the `multiprocessing` module of Python and it can only be used on shared-memory desktop (laptop). It's straightforward to use: simply set `nprocs` to the number of processes you want to use when instantiate *`Loss`*. For example,

```
calc = ...  # create calculator
loss = Loss(calc, ..., nprocs=2)
loss.minimize(method="L-BFGS-B")
```

See also:

See *Train a Stillinger-Weber potential* for a full example.

MPI

The MPI scheme is targeted for HPCs (of course, it can be used on desktops) and we use the mpi4py Python wrapper of MPI. mpi4py supports `OpenMPI` and `MPICH`. Once you have one of the two working, mpi4py can be installed by:

```
$ pip install mpi4py
```

See the mpi4py package documentation for more information on how to install it. Once it is successfully installed, we can run KLIFF in parallel. For example, for the tutorial example *Train a Stillinger-Weber potential*, we can do:

```
$ mpiexec  -np 2  python example_kim_SW_Si.py
```

Note: When using this MPI scheme, the `nprocs` argument passed to *`Loss`* is ignored.

Note: We only parallelize the evaluation of the loss during the minimization. As a result, the other parts will be executed multiple times. For example, if *`kliff.models.Model.echo_model_params()`* is used, the information of model parameters will be repeated multiple times. If this annoys you, you can let only of process (say the rank 0 process) to do it by doing something like:

```
from mpi4py import MPI

rank = MPI.COMM_WORLD.Get_rank()
if rank == 0:
    model.echo_model_params()
```

5.4.2 Machine learning models

CHAPTER

SIX

COMMAND LINE TOOL

KLIFF has a command line tool called **kliff** that can be invoked directly from the terminal. The command line tool **kliff** has the following sub-commands:

sub-command	description
test	Test installation
dataset	Count, split, and other operations on dataset.

For all command line tools, you can do:

```
$ kliff --help
$ kliff sub-command --help
```

to get help (or `-h` for short).

CHAPTER

SEVEN

CONTRIBUTING GUIDE

7.1 Code style

- KLIFF uses **isort_** and black to format the code. To format the code, install `pre-commit` and then do:

```
pre-commit run pre-commit run --all-files --show-diff-on-failure
```

- The docstring of **KLIFF** follows the *Google* style, which can be found at googledoc.

7.2 Docs

- We use sphinx-gallery to generate the tutorials. The source file should be placed in *kliff/examples* and the file name should start with `example_`.
- To generate the docs (including the tutorials), do:

```
$ cd kliff/docs
$ make html
```

  The generated docs will be at `kliff/docs/build/html/index.html`.

- The above commands will not only parse the docstring in the tutorials, but also run the codes in the tutorials. Running the codes may take a long time. So, if you just want to generate the docs, do:

```
$ cd kliff/docs
$ make html-notutorial
```

  This will not run the code in the tutorials.

Below is Mingjian's personal notes on how to generate API docs. Typically, you will not need it.

To generate the API docs for a specific module, do:

```
sphinx-apidoc -f -o <TARGET> <SOURCE>
```

where *<TARGET>* should be a directory where you want to place the generated *.rst* file, and *<SOURCE>* is path to your Python modules (should also be a directory). For example, to generate docs for all the modules in *kliff*, you can run (from the *kliff/docs* directory)

```
sphinx-apidoc -f -o tmp ../kliff
```

After generating the docs for a module, make necessary modifications and then move the *.rst* files in *tmp* to *kliff/docs/apidoc*.

Note: The *kliff/docs/apidoc/kliff.rst* is referenced in *index.rst*, serving as the entry for all docs.

CHAPTER

EIGHT

CHANGE LOG

8.1 v0.4.1 (2022/10/06)

8.1.1 Added

- Uncertainty quantification via MCMC (@yonatank93). New tutorial and explanation of the functionality provided in the doc.
- Issue and PR template

8.1.2 Fixed

- Linear regression model parameter shape
- NN multispecies calculator to use parameters of all models

8.1.3 Updated

- Documentation on installing KLIFF and dependences

8.2 v0.4.0 (2022/04/27)

8.2.1 Added

- Add ParameterTransform class to transform parameters into a different space (e.g. log space) @yonatank93
- Add Weight class to set weight for energy/forces/stress. This is not backward compatible, which changes the signature of the residual function. Previously, in a residual function, the weights are passed in via the *data* argument, but now, its passed in via an instance of the Weight class. @yonatank93

8.2.2 Fixed

- Fix checking cutoff entry @adityakavalur
- Fix energy_residual_fn and forces_residual_fn to weigh different component

8.2.3 Updated

- Change to use precommit GH action to check code format

8.3 v0.3.3 (2022/03/25)

8.3.1 Fixed

- Fix neighlist (even after v0.3.2, the problem can still happen). Now neighlist is the same as kimpy

8.4 v0.3.2 (2022/03/01)

8.4.1 Added

- Enable params_relation_callback() for KIM model

8.4.2 Fixed

- Fix neighbor list segfault due to numerical error for 1D and 2D cases

8.5 v0.3.1 (2021/11/20)

- add gpu training for NN model; set the `gpu` parameter of a calculator (e.g. `CalculatorTorch(model, gpu=True)`) to use it
- add pyproject.toml, requirements.txt, dependabot.yml to config repo
- switch to `furo` doc theme
- changed: compute grad of energy wrt desc in batch mode (NN calculator)
- fix: set *fingerprints_filename* and load descriptor state dict when reuse fingerprints (NN calculator)

8.6 v0.3.0 (2021/08/03)

- change license to LGPL
- set default optimizer
- put `kimpy` code in `try except` block
- add `state_dict` for descriptors and save it together with model
- change to use `loguru` for logging and allows user to set log level

8.7 v0.2.2 (2021/05/24)

- update to be compatible with `kimpy v2.0.0`

8.8 v0.2.1 (2021/05/24)

- update to be compatible with `kimpy v2.0.0`
- use entry `entry_points` to handle command line tool
- rename `utils` to `devtool`

8.9 v0.2.0 (2021/01/19)

- add type hint for all codes
- reorganize model and parameters to make it more robust
- add more docstring for many undocumented class and functions

8.10 v0.1.7 (2020/12/20)

- add GitHub actions to automatically deploy to PyPI
- add a simple example to README

8.11 v0.1.5 (2020/2/13)

- add neighborlist utility, making NN model independent on kimpy
- add calculator to deal with multiple species for NN model
- update dropout layer to be compatible with the pytorch 1.3

8.12 v0.1.4 (2019/8/24)

- add support for the geodesic Levenberg-Marquardt minimization algorithm
- add command line tool `model` to inquire available parameters of KIM model

8.13 v0.1.3 (2019/8/19)

- add RMSE and Fisher information analyzers
- allow configuration weight for ML models
- add write optimizer state dictionary for ML models
- combine functions `generate_training_fingerprints()` and `generate_test_fingerprints()` of descriptor to `generate_fingerprints()` (supporting passing mean and stdev file)
- rewrite symmetry descriptors to share with KIM driver

8.14 v0.1.2 (2019/6/27)

- MPI parallelization for physics-based models
- reorganize machine learning related files
- various bug fixes
- API changes * class `DataSet` renamed to `Dataset` * class `Calculator` moved to module `calculators` from module `calculator`

8.15 v0.1.1 (2019/5/13)

- KLIFF available from PyPI now. Using `$pip install kliff` to install.
- Use SW model from the KIM website in tutorial.
- Format code with `black`.

8.16 v0.1.0 (2019/3/29)

First official release, but API is not guaranteed to be stable.

- Add more docs to *Package Reference*.

8.17 v0.0.1 (2019/1/1)

Pre-release.

CHAPTER

NINE

FREQUENTLY ASKED QUESTIONS

9.1 I am using a KIM model, but it fails. What should I do?

- Check you have the model installed. You can use `$ kim-api-collections-management list` to see what KIM models are installed.
- Check `kim.log` to see what the error is. Note that `kim.log` stores all the log info chronologically, so you may want to delete it and run you fitting code to get a fresh one.
- Make sure that parameters like `cutoff`, `rhocutoff` is not used as fitting parameters. See *What does "error * * Simulator supplied GetNeighborList() routine returned error" in kim.log mean?* for more.

9.2 What does "error * * Simulator supplied GetNeighborList() routine returned error" in `kim.log` mean?

Probably you use parameters related to cutoff distance (e.g. `cutoff` and `rhocutoff`) as fitting parameters. KLIFF build neighbor list only once at the beginning, and reuse it during the optimization process. If the cutoff changes, the neighbor list could be invalid any more. Typically, in the training of potentials, we treat cutoffs as predefined hyperparameters and do not optimize them. So simply remove them from your fitting parameters.

9.3 I am using `mpirun` (`mpiexec`), but why the output shows it is *Running in multiprocessing mode with x processes*?

If you are running something like `mpiexec -np 2 python example_kim_SW_Si.py` and see each minimization step executed twice, you may forget to install `mpi4py`. See *Run in parallel mode* for more one how to run in parallel.

CHAPTER

TEN

PACKAGE REFERENCE

10.1 kliff.analyzers

class kliff.analyzers.**EnergyForcesRMSE**(*calculator*, *energy=True*, *forces=True*)

Analyzer to compute the root-mean-square error (RMSE) for energy and forces.

The *energy difference norm* for a configuration is defined as:

$$e_{\text{norm}} = |e_{\text{pred}} - e_{\text{ref}}|/N,$$

where e_{pred} is the prediction of the total energy from the model, e_{ref} is the corresponding reference energy, and N is the number of atoms in the configuration. The division by N is applied only when `normalize = True` in `run`. Similarly, the *forces difference norm* for a configuration is defined as:

$$f_{\text{norm}} = ||f_{\text{pred}} - f_{\text{ref}}||/N,$$

where f_{pred} is the prediction of the forces on atoms from the model and f_{ref} is the corresponding reference forces, N is the number of atoms in the configuration. The division by N is applied only when `normalize = True` in `run`.

The RMSEs for energy and forces are defined as:

$$e_{\text{RMSE}} = \sqrt{\frac{\sum_{m=1}^{M} e_{\text{norm}}^2}{M}}$$

and

$$f_{\text{RMSE}} = \sqrt{\frac{\sum_{m=1}^{M} f_{\text{norm}}^2}{M}},$$

in which M is the total number of configurations in the dataset.

run(*normalize=True*, *sort=None*, *path=None*, *verbose=1*)

Run the RMSE analyzer.

Parameters

- **normalize** (*bool*) – Whether to normalize the energy (forces) by the number of atoms in a configuration.
- **sort** (*str (optional)*) – Sort per configuration information according to *energy* or *forces*. If *None*, no sort. This works only when per configuration information is requested, i.e. `verbose > 0`.

- **path** (*str (optional)*) – Path to write out the results. If *None*, write to stdout, otherwise, write to the file specified by *path*. Note, if `verbose==3`, the difference of energy and forces will be written to a directory named *energy_forces_RMSE-difference*.
- **verbose** (*int (optional)*) – Verbose level of the output info. Available values are: 0, 1, 2. If `verbose=0`, only output the energy and forces RMSEs for the dataset. If `verbose==1`, output the norms of the energy and forces for each configuration additionally. If `verbose==2`, output the difference of the energy and forces for each atom, and the information is written to extended XYZ files with the location specified by `path`.

class kliff.analyzers.**Fisher**(*calculator*)

Fisher information matrix.

Compute the Fisher information according to

..math::
I_{ij} = sum_m frac{partial bm f_m}{partial theta_i} cdot frac{partial bm f_m}{partial theta_j}

where f_m are the forces on atoms in configuration m, θ_i is the ith model parameter. Derivatives are computed numerically using Ridders' algorithm: https://en.wikipedia.org/wiki/Ridders%27_method

Parameters
calculator – A calculator object.

run(*verbose=1*)

Compute the Fisher information matrix and the standard deviation.

Parameters
verbose (*int*) – If `0`, do not write out to file; if `1`, write to a file named `analysis_Fisher_info_matrix.txt`.

Returns

- **I** (*2D array, shape(N, N)*) – Fisher information matrix, where N is the number of parameters.
- **I_stdev** (*2D array, shape(N, N)*) – Standard deviation of Fisher information matrix, where N is the number of parameters.

10.2 kliff.atomic_data

10.3 kliff.calculators

class kliff.calculators.**Calculator**(*model*)

Calculator to compute properties (e.g. energy and forces) using a potential model for each configuration in the dataset.

The properties, together with the corresponding reference data stored in the configurations, are provided to *Loss* to construct a loss function for the optimizer.

In the reverse direction, a calculator grabs the new parameters from the optimizer and update the model with the new parameters.

Parameters
model (*Model*) – An instance of *Model*.

create(*configs*, *use_energy=True*, *use_forces=True*, *use_stress=False*)

Create compute arguments for a collection of configurations.

By compute arguments, we mean the information needed by a model to carry on a calculation, such as the coordinates, species, cutoff distance, neighbor list, etc. Each configuration has its own compute arguments, and this function creates the compute arguments for all the configurations in `configs`.

Parameters

- **configs** (List[*Configuration*]) – atomic configurations. instances.
- **use_energy** (Union[List[bool], bool]) – Whether to require the calculator to compute energy. If a list of bool is provided, each component is for one configuration in *configs*. If a bool is provided, it is applied to all configurations.
- **use_forces** (Union[List[bool], bool]) – Whether to require the calculator to compute forces. If a list of bool is provided, each component is for one configuration in *configs*. If a bool is provided, it is applied to all configurations.
- **use_stress** (Union[List[bool], bool]) – Whether to require the calculator to compute stress. If a list of bool is provided, each component is for one configuration in *configs*. If a bool is provided, it is applied to all configurations.

get_compute_arguments()

Return a list of compute arguments, each associated with a configuration.

Return type
List[Any]

compute(*compute_arguments*)

Compute the properties given the compute arguments associated with a configuration.

Parameters
compute_arguments – A compute arguments instance for a configuration.

Return type
Dict[str, Any]

Returns

A dictionary of properties, with keys of *energy*, *forces* and *stress*,
values of float or np.array.

get_energy(*compute_arguments*)

Get the energy of a configuration.

Parameters
compute_arguments – A compute arguments instance for a configuration.

Return type
float

Returns
The energy of the configuration associated with the compute arguments.

get_forces(*compute_arguments*)

Get the forces of a configuration.

Parameters
compute_arguments – A compute arguments instance for a configuration.

Return type
array

Returns

Forces on atoms. 2D array of shape (N, 3), where N is the number of atoms
in the configuration.

`get_stress`(*compute_arguments*)

Get the stress of a configuration.

Parameters
compute_arguments – A compute arguments instance for a configuration.

Return type
array

Returns
Virial stress of the configuration associated with the compute arguments, 1D array with 6 component. The returned stress is in Voigt notation, i.e. this function returns: $[\sigma_{xx}, \sigma_{yy}, \sigma_{zz}, \sigma_{yz}, \sigma_{xy}, \sigma_{xz}]$

`get_prediction`(*compute_arguments*)

Get the prediction of all properties that are requested to compute.

The *energy*, *forces*, and *stress* are each flattened to a 1D array, and then concatenated (in the order of *energy*, *forces*, and *stress*) to form the prediction. Depending on the values of `use_energy`, `use_forces`, and `use_stress` that are provided in *create()*, one or more of *energy*, *forces* and *stress* may not be included in the prediction.

Parameters
compute_arguments – A compute arguments instance for a configuration.

Returns

- If `use_energy`, `use_forces`, and `use_stress` are all `True`, the size of prediction is *1+3N+6*, with the 1st component the *energy*, the 2nd to 1+3N components the flattened *forces* on N atoms, and the remaining 6 components the Voigt *stress*.
- If one or more of `use_energy`, `use_forces`, and `use_stress` is `False`, its corresponding value is removed from prediction, and the size of prediction shrinks accordingly. For example, if both `use_energy` and `use_stress` are `True` but `use_forces` is `False`, then the size of prediction is *1+6*, with the 1st component the *energy*, and the 2nd to 7th components the Voigt stress.

Return type
1D array of predictions. For a configuration of N atoms

`get_reference`(*compute_arguments*)

Get the reference data of all properties that are requested to compute.

Same as *get_prediction()*, the *energy*, *forces*, and *stress* are each flattened to a 1D array, and then concatenated (in the order of *energy*, *forces*, and *stress*) to form the reference. Depending on the values of `use_energy`, `use_forces`, and `use_stress` that are provided in *create()*, one or more of *energy*, *forces* and *stress* may not be included in the reference.

Parameters
compute_arguments – A compute arguments instance for a configuration.

Returns

- If `use_energy`, `use_forces`, and `use_stress` are all `True`, the size of reference is *1+3N+6*, with the 1st component the *energy*, the 2nd to 1+3N components the flattened *forces* on N atoms, and the remaining 6 components the Voigt *stress*.

- If one or more of `use_energy`, `use_forces`, and `use_stress` is `False`, its corresponding value is removed from reference, and the size of reference shrinks accordingly. For example, if both `use_energy` and `use_stress` are `True` but `use_forces` is `False`, then the size of reference is *1+6*, with the 1st component the *energy*, and the 2nd to 7th components the Voigt stress.

Return type
1D array of reference values. For a configuration of N atoms

get_opt_params()

Return a list of optimizing parameters.

The optimizing parameters is a list consisting of the values of the model parameters that is set to fit via *kliff.models.Model.set_opt_params()* or *kliff.models.Model.set_one_opt_param()*. The returned value is typically passed to an optimizer as the initial values to carry out the optimization.

Return type
`array`

get_num_opt_params()

Return the number of optimizing parameters.

Return type
`int`

has_opt_params_bounds()

Return a bool to indicate whether there are parameters whose bounds are provided.

get_opt_params_bounds()

Return the lower and upper bounds for the optimizing parameters.

The returned value is a list of (lower, upper) tuples. Each tuple contains the lower and upper bounds for the corresponding parameter obtained from *get_opt_params()*. `None` for `lower` or `upper` means that no bound should be applied.

update_model_params(*opt_params*)

Update model parameters from a list of opt_params.

This is the reverse of *get_opt_params()*.

Parameters
opt_params (`List[float]`) – Optimizing parameters.

class kliff.calculators.**CalculatorTorch**(*model*, *gpu=None*)

A calculator for torch based models.

Parameters

- **model** (*ModelTorch*) – torch models, e.g. `NeuralNetwork`.
- **gpu** (`Union[bool, int, None]`) – whether to use gpu for training. If *int* (e.g. 0), will trained on this gpu device. If *True* will always train on gpu *0*.

implemented_property = ['energy', 'forces', 'stress']

create(*configs*, *use_energy=True*, *use_forces=True*, *use_stress=False*, *fingerprints_filename='fingerprints.pkl'*, *fingerprints_mean_stdev_filename=None*, *reuse=False*, *use_welford_method=False*, *nprocs=1*)

Process configs to generate fingerprints.

Parameters

- **configs** (List[*Configuration*]) – atomic configurations
- **use_energy** (bool) – Whether to require the calculator to compute energy.
- **use_forces** (bool) – Whether to require the calculator to compute forces.
- **use_stress** (bool) – Whether to require the calculator to compute stress.
- **fingerprints_filename** (Union[Path, str]) – Path to save the generated fingerprints. If *reuse=True*, Will not generate the fingerprints, but directly use the one provided via this file.
- **fingerprints_mean_stdev_filename** (Union[str, Path, None]) – Path to save the mean and standard deviation of the fingerprints. If *reuse=True*, Will not generate new fingerprints mean and stdev, but directly use the one provided via this file. If *normalize* is not required by a descriptor, this is ignored.
- **reuse** (bool) – Whether to reuse provided fingerprints.
- **use_welford_method** (bool) – Whether to compute mean and standard deviation using the Welford method, which is memory efficient. See https://en.wikipedia.org/wiki/Algorithms_for_calculating_variance
- **nprocs** (int) – Number of processes used to generate the fingerprints. If *1*, run in serial mode, otherwise *nprocs* processes will be forked via multiprocessing to do the work.

get_compute_arguments(*batch_size=1*)

Return the dataloader with batch size set to *batch_size*.

fit()

compute(*batch*)

property model

Get the underlying torch model

save_model(*epoch*, *force_save=False*)

Save the model to disk.

When to save a model is dependent on *epoch* and a model's metadata for save.

Parameters

- **epoch** (int) – current optimization epoch.
- **force_save** (bool) – save the model, ignoring *epoch* and save metadata.

get_energy(*batch*)

get_forces(*batch*)

get_stress(*batch*)

10.4 kliff.dataset

class kliff.dataset.**Configuration**(*cell*, *species*, *coords*, *PBC*, *energy=None*, *forces=None*, *stress=None*, *weight=None*, *identifier=None*)

Class of atomic configuration. This is used to store the information of an atomic configuration, e.g. supercell, species, coords, energy, and forces.

Parameters

- **cell** (ndarray) – A 3x3 matrix of the lattice vectors. The first, second, and third rows are a_1, a_2, and a_3, respectively.
- **species** (List[str]) – A list of N strings giving the species of the atoms, where N is the number of atoms.
- **coords** (ndarray) – A Nx3 matrix of the coordinates of the atoms, where N is the number of atoms.
- **PBC** (List[bool]) – A list with 3 components indicating whether periodic boundary condition is used along the directions of the first, second, and third lattice vectors.
- **energy** (Optional[float]) – energy of the configuration.
- **forces** (Optional[ndarray]) – A Nx3 matrix of the forces on atoms, where N is the number of atoms.
- **stress** (Optional[List[float]]) – A list with 6 components in Voigt notation, i.e. it returns $\sigma = [\sigma_{xx}, \sigma_{yy}, \sigma_{zz}, \sigma_{yz}, \sigma_{xz}, \sigma_{xy}]$. See: https://en.wikipedia.org/wiki/Voigt_notation
- **weight** (Optional[Weight]) – an instance that computes the weight of the configuration in the loss function.
- **identifier** (Union[str, Path, None]) – a (unique) identifier of the configuration

classmethod from_file(*filename*, *weight=None*, *file_format='xyz'*)

Read configuration from file.

Parameters

- **filename** (Path) – Path to the file that stores the configuration.
- **file_format** (str) – Format of the file that stores the configuration (e.g. *xyz*).

to_file(*filename*, *file_format='xyz'*)

Write the configuration to file.

Parameters

- **filename** (Path) – Path to the file that stores the configuration.
- **file_format** (str) – Format of the file that stores the configuration (e.g. *xyz*).

property cell: ndarray

3x3 matrix of the lattice vectors of the configurations.

Return type
ndarray

property PBC: List[bool]

A list with 3 components indicating whether periodic boundary condition is used along the directions of the first, second, and third lattice vectors.

Return type
List[bool]

property species: List[str]

Species string of all atoms.

Return type
List[str]

property coords: ndarray

A Nx3 matrix of the Cartesian coordinates of all atoms.

Return type
ndarray

property energy: Optional[float]

Potential energy of the configuration.

Return type
Optional[float]

property forces: ndarray

Return a *Nx3* matrix of the forces on each atoms.

Return type
ndarray

property stress: List[float]

Stress of the configuration. The stress is given in Voigt notation i.e $\sigma = [\sigma_{xx}, \sigma_{yy}, \sigma_{zz}, \sigma_{yz}, \sigma_{xz}, \sigma_{xy}]$.

Return type
List[float]

property weight

Get the weight class of the loss function.

property identifier: str

Return identifier of the configuration.

Return type
str

property path: Optional[Path]

Return the path of the file containing the configuration. If the configuration is not read from a file, return None.

Return type
Optional[Path]

get_num_atoms()

Return the total number of atoms in the configuration.

Return type
int

get_num_atoms_by_species()

Return a dictionary of the number of atoms with each species.

Return type
Dict[str, int]

`get_volume()`

Return volume of the configuration.

Return type
`float`

`count_atoms_by_species(symbols=None)`

Count the number of atoms by species.

Parameters
symbols (`Optional[List[str]]`) – species to count the occurrence. If *None*, all species present in the configuration are used.

Returns

with *key* the species string, and *value* the number of
atoms with each species.

Return type
{specie, count}

`order_by_species()`

Order the atoms according to the species such that atoms with the same species have contiguous indices.

`class kliff.dataset.Dataset(path=None, weight=None, file_format='xyz')`

A dataset of multiple configurations (*Configuration*).

Parameters

- **path** (`Optional[Path]`) – Path of a file storing a configuration or filename to a directory containing multiple files. If given a directory, all the files in this directory and its subdirectories with the extension corresponding to the specified file_format will be read.
- **weight** (`Optional[Weight]`) – an instance that computes the weight of the configuration in the loss function.
- **file_format** – Format of the file that stores the configuration, e.g. *xyz*.

`add_configs(path, weight=None)`

Read configurations from filename and added them to the existing set of configurations. This is a convenience function to read configurations from different directory on disk.

Parameters

- **path** (`Path`) – Path the directory (or filename) storing the configurations.
- **weight** (`Optional[Weight]`) – an instance that computes the weight of the configuration in the loss function.

`get_configs()`

Get the configurations.

Return type
`List[Configuration]`

`get_num_configs()`

Return the number of configurations in the dataset.

Return type
`int`

kliff.dataset.**read_extxyz**(*filename*)

Read atomic configuration stored in extended xyz file_format.

Parameters
filename (Path) – filename to the extended xyz file

Returns

3x3 array, supercell lattice vectors species: species of atoms coords: Nx3 array, coordinates of atoms PBC: periodic boundary conditions energy: potential energy of the configuration; *None* if not provided in file forces: Nx3 array, forces on atoms; *None* if not provided in file stress: 1D array of size 6, stress on the cell in Voigt notation; *None* if not

provided in file

Return type
cell

kliff.dataset.**write_extxyz**(*filename*, *cell*, *species*, *coords*, *PBC*, *energy=None*, *forces=None*, *stress=None*)

Write configuration info to a file in extended xyz file_format.

Parameters

- **filename** (Path) – filename to the extended xyz file
- **cell** (ndarray) – 3x3 array, supercell lattice vectors
- **species** (List[str]) – species of atoms
- **coords** (ndarray) – Nx3 array, coordinates of atoms
- **PBC** (List[bool]) – periodic boundary conditions
- **energy** (Optional[float]) – potential energy of the configuration; If *None*, not write to file
- **forces** (Optional[ndarray]) – Nx3 array, forces on atoms; If *None*, not write to file
- **stress** (Optional[List[float]]) – 1D array of size 6, stress on the cell in Voigt notation; If *None*, not write to file

10.5 kliff.descriptors

class kliff.descriptors.**Descriptor**(*cut_dists*, *cut_name*, *hyperparams*, *normalize=True*, *dtype=<class 'numpy.float32'>*)

Base class of atomic environment descriptors.

Process dataset to generate fingerprints. This is the base class for all descriptors, so it should not be used directly. Instead, descriptors built on top of this such as *SymmetryFunction* and *Bispectrum* can be used to transform the atomic environment information into fingerprints.

Parameters

- **cut_dists** (Dict[str, float]) – Cutoff distances, with key of the form *A-B* where *A* and *B* are species string, and value should be a float. Example: *cut_dists = {'C-C': 5.0}*
- **cut_name** (str) – Name of the cutoff function, such as *cos*, *P3*, and *P7*.
- **hyperparams** (Union[Dict, str]) – A dictionary of the hyperparams of the descriptor or a string to select the predefined hyperparams.

- **normalize** (bool) – If *True*, the fingerprints is centered and normalized: *zeta = (zeta - mean(zeta)) / stdev(zeta)*
- **dtype** – np.dtype Data type of the generated fingerprints, such as *np.float32* and *np.float64*.

size

int Length of the fingerprint vector.

mean

list Mean of the fingerprints.

stdev

list Standard deviation of the fingerprints.

generate_fingerprints(*configs*, *fit_forces=False*, *fit_stress=False*, *fingerprints_filename='fingerprints.pkl'*, *fingerprints_mean_stdev_filename=None*, *use_welford_method=False*, *nprocs=1*)

Convert all configurations to their fingerprints.

Parameters

- **configs** (List[*Configuration*]) – Dataset configurations
- **fit_forces** (bool) – Whether to compute the gradient of fingerprints w.r.t. atomic coordinates so as to compute forces.
- **fit_stress** (bool) – Whether to compute the gradient of fingerprints w.r.t. atomic coordinates so as to compute stress.
- **use_welford_method** (bool) – Whether to compute mean and standard deviation using the Welford method, which is memory efficient. See https://en.wikipedia.org/wiki/Algorithms_for_calculating_variance
- **fingerprints_filename** (Union[Path, str]) – Path to dump fingerprints to a pickle file.
- **fingerprints_mean_stdev_filename** (Union[str, Path, None]) – Path to dump the mean and standard deviation of the fingerprints as a pickle file. If *normalize=False* for the descriptor, this is ignored.
- **nprocs** (int) – Number of processes used to generate the fingerprints. If *1*, run in serial mode, otherwise *nprocs* processes will be forked via multiprocessing to do the work.

transform(*conf*, *fit_forces=False*, *fit_stress=False*)

Transform atomic coords to atomic environment descriptor values.

Parameters

- **conf** (*Configuration*) – atomic configuration
- **fit_forces** (bool) – Whether to fit forces, so as to compute gradients of fingerprints w.r.t. coords
- **fit_stress** (bool) – Whether to fit stress, so as to compute gradients of fingerprints w.r.t. coords

Returns

Descriptor values. 2D array with shape (num_atoms, num_descriptors),
where num_atoms is the number of atoms in the configuration, and num_descriptors is the size of the descriptor vector (depending on the choice of the hyperparameters).

dzeta_dr: Gradient of the descriptor w.r.t. atomic coordinates. 4D array if grad is *True*, otherwise *None*. Shape: (num_atoms, num_descriptors, num_atoms, 3), where num_atoms and num_descriptors has the same meanings as described in zeta, and 3 denotes the 3D space for the Cartesian coordinates.

dzeta_ds: Gradient of the descriptor w.r.t. virial stress component. 2D array of shape (num_atoms, num_descriptors, 6), where num_atoms and num_descriptors has the same meanings as described in zeta, and 6 denote the virial stress component in Voigt notation, see https://en.wikipedia.org/wiki/Voigt_notation

Return type
zeta

`write_kim_params`(*path*, *fname='descriptor.params'*)

Write descriptor info for KIM model.

Parameters

- **path** (Union[Path, str]) – Directory Path to write the file.
- **fname** (str) – Name of the file.

`get_size`()

Return the size of the descriptor vector.

`get_mean`()

Return a list of the mean of the fingerprints.

`get_stdev`()

Return a list of the standard deviation of the fingerprints.

`get_dtype`()

Return the data type of the fingerprints.

`get_cutoff`()

Return the name and values of cutoff.

`get_hyperparams`()

Return the hyperparameters of descriptors.

`state_dict`()

Return the state dict of the descriptor.

Return type
Dict[str, Any]

`load_state_dict`(*data*)

Load state dict of a descriptor.

Parameters
data (Dict[str, Any]) – state dict to load.

class kliff.descriptors.**SymmetryFunction**(*cut_dists*, *cut_name*, *hyperparams*, *normalize=True*, *dtype=<class 'numpy.float32'>*)

Atom-centered symmetry functions descriptor as discussed in [Behler2011].

Parameters

- **cut_dists** (*dict*) – Cutoff distances, with key of the form A-B where A and B are atomic species string, and value should be a float.
- **cut_name** (*str*) – Name of the cutoff function.

- **hyperparams** (*dict or str*) – A dictionary of the hyper parameters of that define the descriptor. We provide two sets of hyperparams that can be used by setting `hyperparams='set51'` or `hyperparams='set30'`, which are taken from [Artrith2012] and [Artrith2013], respectively. To see what they are, one can do:

```
>>> cut_name = 'cos'  # just for init purpose
>>> cut_dists = {'C-C': 5.}  # just for init purpose
>>> hyperparams = 'set51'
>>> desc = SymmetryFunction(cut_dists, cut_name, hyperparams)
>>> desc.get_hyperparams()
```

- **normalize** (*bool (optional)*) – If `True`, the fingerprints is centered and normalized according to: `zeta = (zeta - mean(zeta)) / stdev(zeta)`
- **dtype** (*np.dtype (optional)*) – Data type for the generated fingerprints, such as `np.float32` and `np.float64`.

Example

If `set51` or `set30` hyperparams are used, the cutoff distances should be given in `Angstrom`.

```
>>> cut_name = 'cos'
>>> cut_dists = {'C-C': 5., 'C-H': 4.5, 'H-H': 4.0}
>>> hyperparams = 'set51'
>>> desc = SymmetryFunction(cut_dists, cut_name, hyperparams)
```

You can provide your own hyperparams as a dictionary:

```
>>> cut_name = 'cos'
>>> cut_dists = {'C-C': 5., 'C-H': 4.5, 'H-H': 4.0}
>>> hyperparams = {'g1': None,
>>>                'g2': [{'eta':0.1, 'Rs':0.2}, {'eta':0.3, 'Rs':0.4}],
>>>                'g3': [{'kappa':0.1}, {'kappa':0.2}, {'kappa':0.3}]}
>>> desc = SymmetryFunction(cut_dists, cut_name, hyperparams)
```

References

transform(*conf*, *fit_forces=False*, *fit_stress=False*)

Transform atomic coords to atomic environment descriptor values.

Parameters

conf (*Configuration* object) – A configuration of atoms.

fit_forces: bool (optional)

Whether to compute the gradient of descriptor values w.r.t. atomic coordinates so as to compute forces.

fit_stress: bool (optional)

Whether to compute the gradient of descriptor values w.r.t. atomic coordinates so as to compute stress.

Returns

- **zeta** (*2D array*) – Descriptor values, each row for one atom. zeta has shape (num_atoms, num_descriptors), where num_atoms is the number of atoms in the configuration, and num_descriptors is the size of the descriptor vector (depending on the the choice of hyperparameters).

- **dzetadr_forces** (3D array if fit_forces is `True`, otherwise `None`) – Gradient of descriptor values w.r.t. atomic coordinates for forces computation. dzetadr_forces has shape (num_atoms, num_descriptors, num_atoms*DIM), where num_atoms and num_descriptors has the same meanings as described in zeta. DIM = 3 denotes three Cartesian coordinates.
- **dzetadr_stress** (3D array if fit_stress is `True`, otherwise `None`) – Gradient of descriptor values w.r.t. atomic coordinates for stress computation. dzetadr_stress has shape (num_atoms, num_descriptors, 6), where num_atoms and num_descriptors has the same meanings as described in zeta. The last dimension is the 6 component associated with virial stress in the order of 11, 22, 33, 23, 31, 12.

write_kim_params(*path*, *fname='descriptor.params'*)

Write descriptor info for KIM model.

Parameters

- **path** – Directory Path to write the file.
- **fname** – Name of the file.

get_size()

Return the size of the descriptor vector.

get_hyperparams()

Return the hyperparameters of descriptors.

generate_fingerprints(*configs*, *fit_forces=False*, *fit_stress=False*, *fingerprints_filename='fingerprints.pkl'*, *fingerprints_mean_stdev_filename=None*, *use_welford_method=False*, *nprocs=1*)

Convert all configurations to their fingerprints.

Parameters

- **configs** (List[*Configuration*]) – Dataset configurations
- **fit_forces** (bool) – Whether to compute the gradient of fingerprints w.r.t. atomic coordinates so as to compute forces.
- **fit_stress** (bool) – Whether to compute the gradient of fingerprints w.r.t. atomic coordinates so as to compute stress.
- **use_welford_method** (bool) – Whether to compute mean and standard deviation using the Welford method, which is memory efficient. See https://en.wikipedia.org/wiki/Algorithms_for_calculating_variance
- **fingerprints_filename** (Union[Path, str]) – Path to dump fingerprints to a pickle file.
- **fingerprints_mean_stdev_filename** (Union[str, Path, None]) – Path to dump the mean and standard deviation of the fingerprints as a pickle file. If *normalize=False* for the descriptor, this is ignored.
- **nprocs** (int) – Number of processes used to generate the fingerprints. If *1*, run in serial mode, otherwise *nprocs* processes will be forked via multiprocessing to do the work.

get_cutoff()

Return the name and values of cutoff.

get_dtype()

Return the data type of the fingerprints.

get_mean()

Return a list of the mean of the fingerprints.

get_stdev()

Return a list of the standard deviation of the fingerprints.

load_state_dict(*data*)

Load state dict of a descriptor.

Parameters
data (Dict[str, Any]) – state dict to load.

state_dict()

Return the state dict of the descriptor.

Return type
Dict[str, Any]

class kliff.descriptors.**Bispectrum**(*cut_dists*, *cut_name=None*, *hyperparams=None*, *normalize=True*, *dtype=<class 'numpy.float32'>*)

Bispectrum descriptor.

Process dataset to generate fingerprints using the Bispectrum descriptor as discussed in [Bartok2010] and [Thompson2015].

Parameters

- **cut_dists** (*dict*) – Cutoff distances, with key of the form A-B where A and B are atomic species string, and value should be a float.
- **cut_name** (*str*) – Name of the cutoff function.
- **hyperparams** (*dict*) – A dictionary of the hyperparams of the descriptor.
- **normalize** (*bool (optional)*) – If True, the fingerprints is centered and normalized according to: zeta = (zeta - mean(zeta)) / stdev(zeta)
- **dtype** (*np.dtype*) – Data type for the generated fingerprints, such as np.float32 and np.float64.

Example

```
>>> cut_name = 'cos'
>>> cut_dists = {'C-C': 5.0, 'C-H': 4.5, 'H-H': 4.0}
>>> hyperparams = {'jmax': 4, 'weight': {'C':1.0, 'H':1.0}}
>>> desc = Bispectrum(cut_dists, cut_name, hyperparams)
```

References

transform(*conf*, *grad=False*)

Transform atomic coords to atomic environment descriptor values.

Parameters

- **conf** – atomic configuration
- **fit_forces** – Whether to fit forces, so as to compute gradients of fingerprints w.r.t. coords
- **fit_stress** – Whether to fit stress, so as to compute gradients of fingerprints w.r.t. coords

Returns

Descriptor values. 2D array with shape (num_atoms, num_descriptors),
where num_atoms is the number of atoms in the configuration, and num_descriptors is the size of the descriptor vector (depending on the choice of the hyperparameters).

dzeta_dr: Gradient of the descriptor w.r.t. atomic coordinates. 4D array if
grad is *True*, otherwise *None*. Shape: (num_atoms, num_descriptors, num_atoms, 3), where num_atoms and num_descriptors has the same meanings as described in zeta, and 3 denotes the 3D space for the Cartesian coordinates.

dzeta_ds: Gradient of the descriptor w.r.t. virial stress component. 2D
array of shape (num_atoms, num_descriptors, 6), where num_atoms and num_descriptors has the same meanings as described in zeta, and 6 denote the virial stress component in Voigt notation, see https://en.wikipedia.org/wiki/Voigt_notation

Return type
zeta

update_hyperparams(*params*)

Update the hyperparameters based on the input at initialization.

get_size()

Return the size of descriptor.

generate_fingerprints(*configs*, *fit_forces=False*, *fit_stress=False*, *fingerprints_filename='fingerprints.pkl'*, *fingerprints_mean_stdev_filename=None*, *use_welford_method=False*, *nprocs=1*)

Convert all configurations to their fingerprints.

Parameters

- **configs** (List[*Configuration*]) – Dataset configurations
- **fit_forces** (bool) – Whether to compute the gradient of fingerprints w.r.t. atomic coordinates so as to compute forces.
- **fit_stress** (bool) – Whether to compute the gradient of fingerprints w.r.t. atomic coordinates so as to compute stress.
- **use_welford_method** (bool) – Whether to compute mean and standard deviation using the Welford method, which is memory efficient. See https://en.wikipedia.org/wiki/Algorithms_for_calculating_variance
- **fingerprints_filename** (Union[Path, str]) – Path to dump fingerprints to a pickle file.
- **fingerprints_mean_stdev_filename** (Union[str, Path, None]) – Path to dump the mean and standard deviation of the fingerprints as a pickle file. If *normalize=False* for the descriptor, this is ignored.

- **nprocs** (int) – Number of processes used to generate the fingerprints. If *1*, run in serial mode, otherwise *nprocs* processes will be forked via multiprocessing to do the work.

get_cutoff()

Return the name and values of cutoff.

get_dtype()

Return the data type of the fingerprints.

get_hyperparams()

Return the hyperparameters of descriptors.

get_mean()

Return a list of the mean of the fingerprints.

get_stdev()

Return a list of the standard deviation of the fingerprints.

load_state_dict(*data*)

Load state dict of a descriptor.

Parameters
data (Dict[str, Any]) – state dict to load.

state_dict()

Return the state dict of the descriptor.

Return type
Dict[str, Any]

write_kim_params(*path*, *fname='descriptor.params'*)

Write descriptor info for KIM model.

Parameters

- **path** (Union[Path, str]) – Directory Path to write the file.
- **fname** (str) – Name of the file.

10.6 kliff.error

exception kliff.error.**InputError**(*msg*)

args

with_traceback()

Exception.with_traceback(tb) – set self.__traceback__ to tb and return self.

exception kliff.error.**KeyNotFoundError**(*msg*)

args

with_traceback()

Exception.with_traceback(tb) – set self.__traceback__ to tb and return self.

kliff.error.**report_import_error**(*package*, *classname=None*)

10.7 kliff.log

kliff.log.**set_logger**(*level='INFO'*, *stderr=True*)

Set up loguru loggers.

The default logger has a log level of *INFO* and logs to stderr. Call this function at the beginning of your script to override it.

Parameters

- **level** (str) – log level, e.g. DEBUG, INFO, WARNING, ERROR, and CRITICAL.
- **stderr** (bool) – whether to log to stderr.

kliff.log.**get_log_level**()

Get the current log level.

Return type
Optional[str]

Returns
Log level in str, one of DEBUG, INFO, WARNING, ERROR, or CRITICAL.

10.8 kliff.loss

kliff.loss.**energy_forces_residual**(*identifier*, *natoms*, *weight*, *prediction*, *reference*, *data*)

A residual function using both energy and forces.

The residual is computed as

```
weight.config_weight * wi * (prediction - reference)
```

where wi can be weight.energy_weight or weight.forces_weight, depending on the property.

Parameters

- **identifier** (str) – (unique) identifier of the configuration for which to compute the residual. This is useful when you want to weigh some configuration differently.
- **natoms** (int) – number of atoms in the configuration
- **weight** (Weight) – an instance that computes the weight of the configuration in the loss function.
- **prediction** (array) – prediction computed by calculator, 1D array
- **reference** (array) – references data for the prediction, 1D array
- **data** (Dict[str, Any]) – additional data for calculating the residual. Supported key value pairs are: - normalize_by_atoms: bool (default: True) If normalize_by_atoms is True, the residual is divided by the number of atoms in the configuration.

Returns
1D array of the residual

Note: The length of *prediction* and *reference* (call it *S*) are the same, and it depends on *use_energy* and *use_forces* in Calculator. Assume the configuration contains of *N* atoms.

1. If *use_energy == True* and *use_forces == False*, then *S = 1*. *prediction[0]* is the potential energy computed by the calculator, and *reference[0]* is the reference energy.

2. If *use_energy == False* and *use_forces == True*, then *S = 3N*. *prediction[3*i+0]*, *prediction[3*i+1]*, and *prediction[3*i+2]* are the x, y, and z component of the forces on atom i in the configuration, respectively. Correspondingly, *reference* is the 3N concatenated reference forces.

3. If *use_energy == True* and *use_forces == True*, then *S = 3N + 1*. *prediction[0]* is the potential energy computed by the calculator, and *reference[0]* is the reference energy. *prediction[3*i+1]*, *prediction[3*i+2]*, and *prediction[3*i+3]* are the x, y, and z component of the forces on atom i in the configuration, respectively. Correspondingly, *reference* is the 3N concatenated reference forces.

---

kliff.loss.**energy_residual**(*identifier*, *natoms*, *weight*, *prediction*, *reference*, *data*)

A residual function using just the energy.

See the documentation of *energy_forces_residual()* for the meaning of the arguments.

kliff.loss.**forces_residual**(*identifier*, *natoms*, *weight*, *prediction*, *reference*, *data*)

A residual function using just the forces.

See the documentation of *energy_forces_residual()* for the meaning of the arguments.

class kliff.loss.**Loss**(*calculator*, *nprocs: int = 1*, *residual_fn: Optional[Callable] = None*, *residual_data: Optional[Dict[str, Any]] = None*)

Loss function class to optimize the potential parameters.

This is a wrapper over *LossPhysicsMotivatedModel* and *LossNeuralNetworkModel* to provide a united interface. You can use the two classes directly.

Parameters

- **calculator** – Calculator to compute prediction from atomic configuration using a potential model.
- **nprocs** – Number of processes to use..
- **residual_fn** – function to compute residual, e.g. *energy_forces_residual()*, *energy_residual()*, and *forces_residual()*. See the documentation of *energy_forces_residual()* for the signature of the function. Default to *energy_forces_residual()*.
- **residual_data** – data passed to `residual_fn`; can be used to fine tune the residual function. Default to {

  "normalize_by_natoms": True,

  } See the documentation of *energy_forces_residual()* for more.

class kliff.loss.**LossPhysicsMotivatedModel**(*calculator*, *nprocs=1*, *residual_fn=None*, *residual_data=None*)

Loss function class to optimize the physics-based potential parameters.

Parameters

- **calculator** (*Calculator*) – Calculator to compute prediction from atomic configuration using a potential model.
- **nprocs** (`int`) – Number of processes to use..
- **residual_fn** (`Optional[Callable]`) – function to compute residual, e.g. *energy_forces_residual()*, *energy_residual()*, and *forces_residual()*.

See the documentation of *energy_forces_residual()* for the signature of the function. Default to *energy_forces_residual()*.

- **residual_data** (Optional[Dict[str, Any]]) – data passed to residual_fn; can be used to fine tune the residual function. Default to {

  "normalize_by_natoms": True,

  } See the documentation of *energy_forces_residual()* for more.

scipy_minimize_methods = ['Nelder-Mead', 'Powell', 'CG', 'BFGS', 'Newton-CG', 'L-BFGS-B', 'TNC', 'COBYLA', 'SLSQP', 'trust-constr', 'dogleg', 'trust-ncg', 'trust-exact', 'trust-krylov']

scipy_minimize_methods_not_supported_args = ['bounds']

scipy_least_squares_methods = ['trf', 'dogbox', 'lm', 'geodesiclm']

scipy_least_squares_methods_not_supported_args = ['bounds']

minimize(*method='L-BFGS-B'*, ***kwargs*)

Minimize the loss.

Parameters

- **method** (str) – minimization methods as specified at: https://docs.scipy.org/doc/scipy/reference/generated/scipy.optimize.minimize.html https://docs.scipy.org/doc/scipy/reference/generated/scipy.optimize.least_squares.html
- **kwargs** – extra keyword arguments that can be used by the scipy optimizer

class kliff.loss.**LossNeuralNetworkModel**(*calculator*, *nprocs=1*, *residual_fn=None*, *residual_data=None*)

Loss function class to optimize the ML potential parameters.

This is a wrapper over *LossPhysicsMotivatedModel* and *LossNeuralNetworkModel* to provide a united interface. You can use the two classes directly.

Parameters

- **calculator** – Calculator to compute prediction from atomic configuration using a potential model.
- **nprocs** (int) – Number of processes to use..
- **residual_fn** (Optional[Callable]) – function to compute residual, e.g. *energy_forces_residual()*, *energy_residual()*, and *forces_residual()*. See the documentation of *energy_forces_residual()* for the signature of the function. Default to *energy_forces_residual()*.
- **residual_data** (Optional[Dict[str, Any]]) – data passed to residual_fn; can be used to fine tune the residual function. Default to {

  "normalize_by_natoms": True,

  } See the documentation of *energy_forces_residual()* for more.

torch_minimize_methods = ['Adadelta', 'Adagrad', 'Adam', 'SparseAdam', 'Adamax', 'ASGD', 'LBFGS', 'RMSprop', 'Rprop', 'SGD']

minimize(*method='Adam'*, *batch_size=100*, *num_epochs=1000*, *start_epoch=0*, ***kwargs*)

Minimize the loss.

Parameters

- **method** (str) – PyTorch optimization methods, and available ones are: [*Adadelta*, *Adagrad*, *Adam*, *SparseAdam*, *Adamax*, *ASGD*, *LBFGS*, *RMSprop*, *Rprop*, *SGD*] See also: https://pytorch.org/docs/stable/optim.html
- **batch_size** (int) – Number of configurations used in each minimization step.
- **num_epochs** (int) – Number of epochs to carry out the minimization.
- **start_epoch** (int) – The starting epoch number. This is typically 0, but if continuing a training, it is useful to set this to the last epoch number of the previous training.
- **kwargs** – Extra keyword arguments that can be used by the PyTorch optimizer.

save_optimizer_state(*path='optimizer_state.pkl'*)

Save the state dict of optimizer to disk.

load_optimizer_state(*path='optimizer_state.pkl'*)

Load the state dict of optimizer from file.

exception kliff.loss.**LossError**(*msg*)

args

with_traceback()

Exception.with_traceback(tb) – set self.__traceback__ to tb and return self.

10.9 kliff.models

class kliff.models.**Parameter**(*value*, *fixed=None*, *lower_bound=None*, *upper_bound=None*, *name=None*, *index=None*)

Potential parameter.

To follow the KIM model paradigm, a parameter is a list of floats that can be adjusted to fit the potential to some parameters.

This class is mainly by physically-motivated potentials.

Parameters

- **value** (Union[Sequence[float], ndarray]) – parameter values, should be a list of float or a 1D array.
- **fixed** (Optional[Sequence[bool]]) – whether parameter component should be fixed (i.e. not used for fitting). Should have the same length of *value*. Default to not fixed for all components.
- **lower_bound** (Optional[Sequence[float]]) – lower bound of the allowed value for the parameter. Default to *None*, i.e. no lower bound is applied.
- **upper_bound** (Optional[Sequence[float]]) – upper bound of the allowed value for the parameter. Default to *None*, i.e. no lower bound is applied.
- **name** (Optional[str]) – name of the parameter.
- **index** (Optional[int]) – integer index of the parameter.

property value: List[float]

Parameter value.

Return type
List[float]

set_value(*index*, *v*)

Set the value of a component.

property fixed: List[bool]

Whether each parameter component is fixed or not (i.e. allow to fitting or not).

Return type
List[bool]

set_fixed(*index*, *v*)

Set the fixed status of a component of the parameter.

Parameters

- **index** (int) – index of the component
- **v** (bool) – fix status

property lower_bound: List[float]

Lower bound of parameter.

Return type
List[float]

set_lower_bound(*index*, *v*)

Set the lower bound of a component of the parameter.

Parameters

- **index** (int) – index of the component
- **v** (float) – lower bound value

property upper_bound: List[float]

Upper bound of parameter.

Return type
List[float]

set_upper_bound(*index*, *v*)

Set the upper bound of a component of the parameter.

Parameters

- **index** (int) – index of the component
- **v** (float) – upper bound value

property name: str

Name of the parameter.

Return type
str

property index: int

Integer index of the parameter.

Return type
int

as_dict()

A JSON serializable dict representation of an object.

classmethod from_dict(*d*)

Parameters
d – Dict representation.

Returns
MSONable class.

REDIRECT = {}

to_json()
Returns a json string representation of the MSONable object.

Return type
str

unsafe_hash()
Returns an hash of the current object. This uses a generic but low performance method of converting the object to a dictionary, flattening any nested keys, and then performing a hash on the resulting object

classmethod validate_monty(*v*)
pydantic Validator for MSONable pattern

class kliff.models.**OptimizingParameters**(*model_params*)
A collection of parameters that will be optimized.

This can be all the parameters of a model or a subset of the parameters of a model. The behavior of individual component of a parameter can also be controlled. For example, keep the 2nd component of a parameters fixed, while optimizing the other components.

It interacts with optimizer to provide initial guesses of parameter values; it also receives updated parameters from the optimizer and update model parameters.

Parameters
model_params (Dict[str, *Parameter*]) – {name, parameter} all the parameters of a model. The attributes of these parameters will be modified to reflect whether it is optimized.

read(*filename*)
Read the parameters that will be optimized from a file.

Each parameter is a 1D array, and each component of the parameter array should be listed in a new line. Each line can contains 1, 2, or 3 elements, described in details below:

1st element: float or *DEFAULT*
Initial guess of the parameter component. If *DEFAULT* (case insensitive), the value from the calculator is used as the initial guess.

The 2nd and 3rd elements are optional.

If 2 elements are provided:

2nd element: *FIX* (case insensitive)
If *FIX*, the corresponding component will not be optimized.

If 3 elements are provided:

2nd element: float or *INF* (case insensitive)
Lower bound of the parameter. *INF* indicates that the lower bound is negative infinite, i.e. no lower bound is applied.

3rd element: float or *INF* (case insensitive)
Upper bound of the parameter. *INF* indicates that the upper bound is positive infinite, i.e. no upper bound is applied.

Instead of reading fitting parameters from a file, you can also setting them using a dictionary by calling the *set()* method.

Parameters

filename (Path) – path to file that includes the fitting parameters.

Example

put the below in a file, say *model_params.txt* and you can read the fitting # parameters by this_class.read(filename="model_params.txt")

A DEFAULT 1.1

B DEFAULT FIX 1.1 FIX

C DEFAULT 0.1 INF 1.0 INF 2.1 2.0 FIX

set(***kwargs*)

Set the parameters that will be optimized.

One or more parameters can be set. Each argument is for one parameter, where the argument name is the parameter name, the value of the argument is the settings(including initial value, fix flag, lower bound, and upper bound).

The value of the argument should be a list of list, where each inner list is for one component of the parameter, which can contain 1, 2, or 3 elements. See *~kliff.model.parameter.OptimizingParameters.read()* for the options of the elements.

Example

instance.set(A=[['DEFAULT'], [2.0, 1.0, 3.0]], B=[[1.0, 'FIX'], [2.0, 'INF', 3.0]])

set_one(*name*, *settings*)

Set one parameter that will be optimized.

The name of the parameter should be given as the first entry of a list (or tuple), and then each data line should be given in in a list.

Parameters

- **name** (str) – name of a fitting parameter
- **settings** (List[List[Any]]) – initial value, flag to fix a parameter, lower and upper bounds of a parameter.

Example

name = 'param_A' settings = [['default', 0, 20], [2.0, 'fix'], [2.2, 'inf', 3.3]] instance.set_one(name, settings)

echo_opt_params(*filename=<_io.TextIOWrapper name='<stdout>' mode='w' encoding='utf-8'>*, *echo_size=True*)

Get the optimizing parameters as a string and/or print to file (stdout).

Parameters

- **filename** (Union[Path, TextIO, None]) – Path to the file to output the optimizing parameters. If *None*, print to stdout.

- **echo_size** (bool) – Whether to print the size of parameters. (Each parameter may have one or more components).

Return type
str

Returns
Optimizing parameters as a string.

`get_num_opt_params()`

Number of optimizing parameters.

This is the total number of model parameter components. For example, if the model has two parameters set to be optimized and each have two components, this will be four.

Return type
int

`get_opt_params()`

Nest all optimizing parameter values (except the fixed ones) to a 1D array.

The obtained values can be provided to the optimizer as the starting parameters.

This is the opposite operation of update_model_params().

Returns
A 1D array of nested optimizing parameter values.

Return type
opt_params

`update_opt_params(params)`

Update the optimizing parameter values from a sequence of float.

This is the opposite operation of get_opt_params().

Parameters
params (Sequence[float]) – updated parameter values from the optimizer.

Return type
Dict[str, *Parameter*]

`get_opt_param_name_value_and_indices(index)`

Get the *name*, *value*, *parameter_index*, and *component_index* of optimizing parameter in slot *index*.

Parameters
index (int) – slot index of the optimizing parameter

Return type
Tuple[str, float, int, int]

`get_opt_params_bounds()`

Get the lower and upper bounds of optimizing parameters.

Return type
List[Tuple[int, int]]

`has_opt_params_bounds()`

Whether bounds are set for some parameters.

Return type
bool

as_dict()

A JSON serializable dict representation of an object.

classmethod from_dict(*d*)

Parameters
d – Dict representation.

Returns
MSONable class.

REDIRECT = {}

to_json()

Returns a json string representation of the MSONable object.

Return type
str

unsafe_hash()

Returns an hash of the current object. This uses a generic but low performance method of converting the object to a dictionary, flattening any nested keys, and then performing a hash on the resulting object

classmethod validate_monty(*v*)

pydantic Validator for MSONable pattern

class kliff.models.**ComputeArguments**(*conf*, *supported_species*, *influence_distance*, *compute_energy=True*, *compute_forces=True*, *compute_stress=False*)

Compute property (e.g. energy, forces, and stress) for a configuration.

This is the base class for other compute arguments. Typically, a user will not directly use this.

Parameters

- **conf** (*Configuration*) – atomic configurations
- **supported_species** (Dict[str, int]) – species supported by the potential model, with chemical symbol as key and integer code as value.
- **influence_distance** (float) – influence distance (aka cutoff distance) to calculate neighbors
- **compute_energy** (bool) – whether to compute energy
- **compute_forces** (bool) – whether to compute forces
- **compute_stress** (bool) – whether to compute stress

implemented_property = []

compute(*params*)

Compute the properties required by the compute flags, and store them in self.results.

Parameters
params (Dict[str, *Parameter*]) – the parameters of the model.

Example

energy = a_func_to_compute_energy() forces = a_func_to_compute_forces() stress = a_func_to_compute_stress() self.results['energy'] = energy self.results['forces'] = forces self.results['stress'] = stress

get_compute_flag(*name*)

Check whether the model is asked to compute property.

Parameters
name (str) – name of the property, e.g. energy, forces, and stresses

Return type
bool

get_property(*name*)

Get a property by name.

Parameters
name (str) – name of the property, e.g. energy, forces, and stresses

Return type
Any

get_energy()

Potential energy.

Return type
float

get_forces()

2D array of shape (N,3) of the forces on atoms, where N is the number of atoms in the configuration.

Return type
ndarray

get_stress()

1D array of the virial stress, in Voigt notation.

Return type
ndarray

get_prediction()

1D array of prediction from the model for the configuration.

Return type
ndarray

get_reference()

1D array of reference values for the configuration.

Return type
ndarray

class kliff.models.**Model**(*model_name=None*, *params_transform=None*)

Base class for all physics-motivated models.

Typically, a user will not directly use this.

Parameters

- **model_name** (Optional[str]) – name of the model.

- **params_transform** (Optional[ParameterTransform]) – optional transformation of parameters. Let's call the parameters initialized in *init_model_params()* as original parameter space. Sometimes, it's easier to work in another parameter (e.g. optimizers can perform better in the log space). Then we can use this parameter transformation class to transform between the original parameter space and the new easy-to-work space. Typically, a model only knows how to work in its original space to compute, e.g. energy and forces, so we need to inverse transform parameters back to original space (after an optimizer update its value in the log space). A *params_transform* instance should implement both a *transform* and an *inverse_transform* method to accomplish the above tasks. Note, all the parameters of this (the *Model*) class (e.g. *self.model_params*, and *self.opt_params*) are in the transformed easy-to-work space.

init_model_params(**args*, ***kwargs*)

Initialize the parameters of the model.

Should return a dictionary of parameters.

Example

model_params = {"sigma": Parameter([0.5]
"epsilon": Parameter([0.4])

return model_params

Return type
Dict[str, *Parameter*]

init_influence_distance(**args*, ***kwargs*)

Initialize the influence distance (aka cutoff distance) of the model.

This is used to compute the neighbor list of each atom.

Example

return 5.0

Return type
float

init_supported_species(**args*, ***kwargs*)

Initialize the supported species of the model.

Should return a dict with chemical symbol as key and integer code as value.

Example

return {"C:0, "O":1}

Return type
Dict[str, int]

get_compute_argument_class()

write_kim_model(*path=None*)

`get_influence_distance`()

Return influence distance (aka cutoff distance) of the model.

Return type
`float`

`get_supported_species`()

Return supported species of the model, a dict with chemical symbol as key and integer code as value.

Return type
`Dict[str, int]`

`get_model_params`()

Return all parameters of the model.

Return type
`Dict[str, Parameter]`

`echo_model_params`(*filename=<_io.TextIOWrapper name='<stdout>' mode='w' encoding='utf-8'>*, *params_space='original'*)

Echo the model parameters.

Parameters

- **filename** (`Union[Path, TextIO, None]`) – Path to write the model parameter info (e.g. sys.stdout). If *None*, do not write.
- **params_space** (`str`) – In which space to show the parameters, options are *original* and *transformed*.

Return type
`str`

Returns
model parameter info in a string

`read_opt_params`(*filename*)

Read optimizing parameters from a file.

Each parameter is a 1D array, and each component of the parameter array should be listed in a new line. Each line can contain 1, 2, or 3 elements, described in details below:

1st element: float or *DEFAULT*
Initial guess of the parameter component. If *DEFAULT* (case insensitive), the value from the calculator is used as the initial guess.

The 2nd and 3rd elements are optional.

If 2 elements are provided:

2nd element: *FIX* (case insensitive)
If *FIX*, the corresponding component will not be optimized.

If 3 elements are provided:

2nd element: float or *INF* (case insensitive)
Lower bound of the parameter. *INF* indicates that the lower bound is negative infinite, i.e. no lower bound is applied.

3rd element: float or *INF* (case insensitive)
Upper bound of the parameter. *INF* indicates that the upper bound is positive infinite, i.e. no upper bound is applied.

Instead of reading fitting parameters from a file, you can also setting them using a dictionary by calling the *set_opt_params()* or *set_one_opt_params()* method.

Parameters

filename (Path) – path to file that includes the fitting parameters.

Example

put the below in a file, say *model_params.txt* and you can read the fitting # parameters by this_class.read(filename="model_params.txt")

A DEFAULT 1.1

B DEFAULT FIX 1.1 FIX

C DEFAULT 0.1 INF 1.0 INF 2.1 2.0 FIX

set_opt_params(***kwargs*)

Set the parameters that will be optimized.

One or more parameters can be set. Each argument is for one parameter, where the argument name is the parameter name, the value of the argument is the settings(including initial value, fix flag, lower bound, and upper bound).

The value of the argument should be a list of list, where each inner list is for one component of the parameter, which can contain 1, 2, or 3 elements.

See *~kliff.model.model.Model.read_opt_params()* for the options of the elements.

Example

instance.set(A=[['DEFAULT'], [2.0, 1.0, 3.0]], B=[[1.0, 'FIX'], [2.0, 'INF', 3.0]])

set_one_opt_param(*name*, *settings*)

Set one parameter that will be optimized.

The name of the parameter should be given as the first entry of a list (or tuple), and then each data line should be given in in a list.

Parameters

- **name** (str) – name of a fitting parameter
- **settings** (List[List[Any]]) – initial value, flag to fix a parameter, lower and upper bounds of a parameter.

Example

name = 'param_A' settings = [['default', 0, 20], [2.0, 'fix'], [2.2, 'inf', 3.3]] instance.set_one(name, settings)

echo_opt_params(*filename=<_io.TextIOWrapper name='<stdout>' mode='w' encoding='utf-8'>*)

Echo the optimizing parameter to a file.

get_num_opt_params()

Number of optimizing parameters.

This is the total number of model parameter components. For example, if the model has two parameters set to be optimized and each have two components, this will be four.

Return type
int

get_opt_params()

Nest all optimizing parameter values (except the fixed ones) to a 1D array.

The obtained values can be provided to the optimizer as the starting parameters.

This is the opposite operation of update_model_params().

Returns
A 1D array of nested optimizing parameter values.

Return type
opt_params

update_model_params(*params*)

Update the optimizing parameter values from a sequence of float.

This is the opposite operation of get_opt_params().

Note, self.model_params will be updated as well, since self.opt_params is initialized from self.model_params without copying the *Parameter* instance.

Parameters
params (Sequence[float]) – updated parameter values from the optimizer.

get_opt_param_name_value_and_indices(*index*)

Get the *name*, *value*, *parameter_index*, and *component_index* of optimizing parameter in slot *index*.

Parameters
index (int) – slot index of the optimizing parameter

Return type
Tuple[str, float, int, int]

get_opt_params_bounds()

Get the lower and upper bounds of optimizing parameters.

Return type
List[Tuple[int, int]]

has_opt_params_bounds()

Whether bounds are set for some parameters.

Return type
bool

save(*filename='trained_model.yaml'*)

Save a model to disk.

Parameters
filename (Path) – Path where to store the model.

load(*filename='trained_model.yaml'*)

Load a model on disk into memory.

Parameters
filename (Path) – Path where the model is stored.

class kliff.models.**LennardJones**(*model_name='LJ6-12'*, *params_transform=None*)

KLIFF built-in Lennard-Jones 6-12 potential model.

`init_model_params()`

Initialize the parameters of the model.

Should return a dictionary of parameters.

Example

model_params = {"sigma": Parameter([0.5]
"epsilon": Parameter([0.4])

return model_params

`init_influence_distance()`

Initialize the influence distance (aka cutoff distance) of the model.

This is used to compute the neighbor list of each atom.

Example

return 5.0

`init_supported_species()`

Initialize the supported species of the model.

Should return a dict with chemical symbol as key and integer code as value.

Example

return {"C:0, "O":1}

`get_compute_argument_class()`

`echo_model_params`(*filename=<_io.TextIOWrapper name='<stdout>' mode='w' encoding='utf-8'>*, *params_space='original'*)

Echo the model parameters.

Parameters

- **filename** (Union[Path, TextIO, None]) – Path to write the model parameter info (e.g. sys.stdout). If *None*, do not write.
- **params_space** (str) – In which space to show the parameters, options are *original* and *transformed*.

Return type
str

Returns
model parameter info in a string

`echo_opt_params`(*filename=<_io.TextIOWrapper name='<stdout>' mode='w' encoding='utf-8'>*)

Echo the optimizing parameter to a file.

`get_influence_distance()`

Return influence distance (aka cutoff distance) of the model.

Return type
float

get_model_params()

Return all parameters of the model.

Return type
Dict[str, *Parameter*]

get_num_opt_params()

Number of optimizing parameters.

This is the total number of model parameter components. For example, if the model has two parameters set to be optimized and each have two components, this will be four.

Return type
int

get_opt_param_name_value_and_indices(*index*)

Get the *name*, *value*, *parameter_index*, and *component_index* of optimizing parameter in slot *index*.

Parameters
index (int) – slot index of the optimizing parameter

Return type
Tuple[str, float, int, int]

get_opt_params()

Nest all optimizing parameter values (except the fixed ones) to a 1D array.

The obtained values can be provided to the optimizer as the starting parameters.

This is the opposite operation of update_model_params().

Returns
A 1D array of nested optimizing parameter values.

Return type
opt_params

get_opt_params_bounds()

Get the lower and upper bounds of optimizing parameters.

Return type
List[Tuple[int, int]]

get_supported_species()

Return supported species of the model, a dict with chemical symbol as key and integer code as value.

Return type
Dict[str, int]

has_opt_params_bounds()

Whether bounds are set for some parameters.

Return type
bool

load(*filename='trained_model.yaml'*)

Load a model on disk into memory.

Parameters
filename (Path) – Path where the model is stored.

read_opt_params(*filename*)

Read optimizing parameters from a file.

Each parameter is a 1D array, and each component of the parameter array should be listed in a new line. Each line can contain 1, 2, or 3 elements, described in details below:

1st element: float or *DEFAULT*
Initial guess of the parameter component. If *DEFAULT* (case insensitive), the value from the calculator is used as the initial guess.

The 2nd and 3rd elements are optional.

If 2 elements are provided:

2nd element: *FIX* (case insensitive)
If *FIX*, the corresponding component will not be optimized.

If 3 elements are provided:

2nd element: float or *INF* (case insensitive)
Lower bound of the parameter. *INF* indicates that the lower bound is negative infinite, i.e. no lower bound is applied.

3rd element: float or *INF* (case insensitive)
Upper bound of the parameter. *INF* indicates that the upper bound is positive infinite, i.e. no upper bound is applied.

Instead of reading fitting parameters from a file, you can also setting them using a dictionary by calling the *set_opt_params()* or *set_one_opt_params()* method.

Parameters
filename (Path) – path to file that includes the fitting parameters.

Example

put the below in a file, say *model_params.txt* and you can read the fitting # parameters by this_class.read(filename="model_params.txt")

A DEFAULT 1.1

B DEFAULT FIX 1.1 FIX

C DEFAULT 0.1 INF 1.0 INF 2.1 2.0 FIX

save(*filename='trained_model.yaml'*)

Save a model to disk.

Parameters
filename (Path) – Path where to store the model.

set_one_opt_param(*name*, *settings*)

Set one parameter that will be optimized.

The name of the parameter should be given as the first entry of a list (or tuple), and then each data line should be given in in a list.

Parameters

- **name** (str) – name of a fitting parameter
- **settings** (List[List[Any]]) – initial value, flag to fix a parameter, lower and upper bounds of a parameter.

Example

name = 'param_A' settings = [['default', 0, 20], [2.0, 'fix'], [2.2, 'inf', 3.3]] instance.set_one(name, settings)

set_opt_params(***kwargs*)

Set the parameters that will be optimized.

One or more parameters can be set. Each argument is for one parameter, where the argument name is the parameter name, the value of the argument is the settings(including initial value, fix flag, lower bound, and upper bound).

The value of the argument should be a list of list, where each inner list is for one component of the parameter, which can contain 1, 2, or 3 elements.

See *~kliff.model.model.Model.read_opt_params()* for the options of the elements.

Example

instance.set(A=[['DEFAULT'], [2.0, 1.0, 3.0]], B=[[1.0, 'FIX'], [2.0, 'INF', 3.0]])

update_model_params(*params*)

Update the optimizing parameter values from a sequence of float.

This is the opposite operation of get_opt_params().

Note, self.model_params will be updated as well, since self.opt_params is initialized from self.model_params without copying the *Parameter* instance.

Parameters

params (Sequence[float]) – updated parameter values from the optimizer.

write_kim_model(*path=None*)

class kliff.models.**KIMModel**(*model_name*, *params_transform=None*)

A general interface to any KIM model.

Parameters

- **model_name** (str) – name of a KIM model. Available models can be found at: https://openkim.org. For example *SW_StillingerWeber_1985_Si__MO_405512056662_006*.
- **params_transform** (Optional[ParameterTransform]) – optional transformation of parameters. Let's call the parameters initialized in *init_model_params()* as original parameter space. Sometimes, it's easier to work in another parameter (e.g. optimizers can perform better in the log space). Then we can use this parameter transformation class to transform between the original parameter space and the new easy-to-work space. Typically, a model only knows how to work in its original space to compute, e.g. energy and forces, so we need to inverse transform parameters back to original space (after an optimizer update its value in the log space). A *params_transform* instance should implement both a *transform* and an *inverse_transform* method to accomplish the above tasks. Note, all the parameters of this (the *Model*) class (e.g. *self.model_params*, and *self.opt_params*) are in the transformed easy-to-work space.

init_model_params()

Initialize the parameters of the model.

Should return a dictionary of parameters.

Example

model_params = {“sigma”: Parameter([0.5]
“epsilon”: Parameter([0.4])

return model_params

Return type
Dict[str, *Parameter*]

init_influence_distance()

Initialize the influence distance (aka cutoff distance) of the model.

This is used to compute the neighbor list of each atom.

Example

return 5.0

Return type
float

init_supported_species()

Initialize the supported species of the model.

Should return a dict with chemical symbol as key and integer code as value.

Example

return {“C:0, “O”:1}

Return type
Dict[str, int]

get_compute_argument_class()

get_kim_model_params()

Inquire the KIM model to get all the parameters.

Return type
Dict[str, *Parameter*]

Returns
{name, parameter}, all parameters in a kim model.

create_a_kim_compute_argument()

Create a compute argument for the KIM model.

set_opt_params(***kwargs*)

Set the parameters that will be optimized.

One or more parameters can be set. Each argument is for one parameter, where the argument name is the parameter name, the value of the argument is the settings(including initial value, fix flag, lower bound, and upper bound).

The value of the argument should be a list of list, where each inner list is for one component of the parameter, which can contain 1, 2, or 3 elements.

See *~kliff.model.model.Model.read_opt_params()* for the options of the elements.

Example

instance.set(A=[['DEFAULT'], [2.0, 1.0, 3.0]], B=[[1.0, 'FIX'], [2.0, 'INF', 3.0]])

set_one_opt_param(*name*, *settings*)

Set one parameter that will be optimized.

The name of the parameter should be given as the first entry of a list (or tuple), and then each data line should be given in in a list.

Parameters

- **name** (str) – name of a fitting parameter
- **settings** (List[List[Any]]) – initial value, flag to fix a parameter, lower and upper bounds of a parameter.

Example

name = 'param_A' settings = [['default', 0, 20], [2.0, 'fix'], [2.2, 'inf', 3.3]] instance.set_one(name, settings)

update_model_params(*params*)

Update optimizing parameters (a sequence used by the optimizer) to the kim model.

write_kim_model(*path=None*)

Write out a KIM model that can be used directly with the kim-api.

This function typically write two files to *path*: (1) CMakeLists.txt, and (2) a parameter file like A.model_params. *path* will be created if it does not exist.

Parameters

path (Optional[Path]) – Path to the a directory to store the model. If *None*, it is set to *./MODEL_NAME_kliff_trained*, where *MODEL_NAME* is the *model_name* that provided at the initialization of this class.

Note: This only works for parameterized KIMModel models that support the writing of parameters.

---

echo_model_params(*filename=<_io.TextIOWrapper name='<stdout>' mode='w' encoding='utf-8'>*, *params_space='original'*)

Echo the model parameters.

Parameters

- **filename** (Union[Path, TextIO, None]) – Path to write the model parameter info (e.g. sys.stdout). If *None*, do not write.
- **params_space** (str) – In which space to show the parameters, options are *original* and *transformed*.

Return type

str

Returns

model parameter info in a string

echo_opt_params(*filename=<_io.TextIOWrapper name='<stdout>' mode='w' encoding='utf-8'>*)

Echo the optimizing parameter to a file.

get_influence_distance()

Return influence distance (aka cutoff distance) of the model.

Return type
float

get_model_params()

Return all parameters of the model.

Return type
Dict[str, *Parameter*]

get_num_opt_params()

Number of optimizing parameters.

This is the total number of model parameter components. For example, if the model has two parameters set to be optimized and each have two components, this will be four.

Return type
int

get_opt_param_name_value_and_indices(*index*)

Get the *name*, *value*, *parameter_index*, and *component_index* of optimizing parameter in slot *index*.

Parameters
index (int) – slot index of the optimizing parameter

Return type
Tuple[str, float, int, int]

get_opt_params()

Nest all optimizing parameter values (except the fixed ones) to a 1D array.

The obtained values can be provided to the optimizer as the starting parameters.

This is the opposite operation of update_model_params().

Returns
A 1D array of nested optimizing parameter values.

Return type
opt_params

get_opt_params_bounds()

Get the lower and upper bounds of optimizing parameters.

Return type
List[Tuple[int, int]]

get_supported_species()

Return supported species of the model, a dict with chemical symbol as key and integer code as value.

Return type
Dict[str, int]

has_opt_params_bounds()

Whether bounds are set for some parameters.

Return type
bool

`load`(*filename='trained_model.yaml'*)

Load a model on disk into memory.

Parameters
filename (Path) – Path where the model is stored.

`read_opt_params`(*filename*)

Read optimizing parameters from a file.

Each parameter is a 1D array, and each component of the parameter array should be listed in a new line. Each line can contain 1, 2, or 3 elements, described in details below:

1st element: float or *DEFAULT*
Initial guess of the parameter component. If *DEFAULT* (case insensitive), the value from the calculator is used as the initial guess.

The 2nd and 3rd elements are optional.

If 2 elements are provided:

2nd element: *FIX* (case insensitive)
If *FIX*, the corresponding component will not be optimized.

If 3 elements are provided:

2nd element: float or *INF* (case insensitive)
Lower bound of the parameter. *INF* indicates that the lower bound is negative infinite, i.e. no lower bound is applied.

3rd element: float or *INF* (case insensitive)
Upper bound of the parameter. *INF* indicates that the upper bound is positive infinite, i.e. no upper bound is applied.

Instead of reading fitting parameters from a file, you can also setting them using a dictionary by calling the *set_opt_params()* or *set_one_opt_params()* method.

Parameters
filename (Path) – path to file that includes the fitting parameters.

Example

put the below in a file, say *model_params.txt* and you can read the fitting # parameters by this_class.read(filename="model_params.txt")

A DEFAULT 1.1

B DEFAULT FIX 1.1 FIX

C DEFAULT 0.1 INF 1.0 INF 2.1 2.0 FIX

`save`(*filename='trained_model.yaml'*)

Save a model to disk.

Parameters
filename (Path) – Path where to store the model.

`class kliff.models.ModelTorch`(*descriptor*, *seed=35*)

Base class for machine learning models.

Typically, a user will not directly use this.

Parameters

- **descriptor** (*Descriptor*) – atomic environment descriptor for computing configuration fingerprints. See *SymmetryFunction()* and *Bispectrum()*.
- **seed** (int) – random seed.

forward(*x*)

Use the model to perform computation.

Parameters
x (Any) – input to the model

write_kim_model(*path=None*)

Write the model out as a KIM-API compatible one.

Parameters
path (Optional[Path]) – path to write the model

fit(*path*)

Fit the model using analytic solution.

Parameters
path (Path) – path to the fingerprints generated by the descriptor.

save(*filename*)

Save a model to disk.

Parameters
filename (Path) – Path to store the model.

load(*filename*, *mode='train'*)

Load a save model.

Parameters

- **filename** (Path) – Path where the model is stored, e.g. kliff_model.pkl
- **mode** (str) – Purpose of the loaded model. Should be either *train* or *eval*.

set_save_metadata(*prefix*, *start*, *frequency=1*)

Set metadata that controls how the model are saved during training.

If this function is called before minimization starts, the model will be saved to the directory specified by *prefix* every *frequency* epochs, beginning at the *start* epoch.

Parameters

- **prefix** (Path) – Path to the directory where the models are saved. Model will be named as *{prefix}/model_epoch{ep}.pt*, where *ep* is the epoch number.
- **start** (int) – Epoch number at which begins to save the model.
- **frequency** (int) – Save the model every *frequency* epochs.

property descriptor

property device

property dtype

property save_prefix

property save_start

property save_frequency

class kliff.models.NeuralNetwork(*descriptor*, *seed=35*)

Neural Network model.

A feed-forward neural network model.

Parameters

- **descriptor** (*Descriptor*) – A descriptor that transforms atomic environment information to the fingerprints, which are used as the input for the neural network.
- **seed** – Global seed for random numbers.

add_layers(**layers*)

Add layers to the sequential model.

Parameters

layers – torch.nn layers `torch.nn` layers that are used to build a sequential model. Available ones including: torch.nn.Linear, torch.nn.Dropout, and torch.nn.Sigmoid among others. See https://pytorch.org/docs/stable/nn.html for a full list.

forward(*x*)

Forward pass through the neural network.

Parameters

x – input descriptor to the neural network.

Returns

The output of the neural network.

write_kim_model(*path=None*, *driver_name='DUNN__MD_292677547454_000'*, *dropout_ensemble_size=None*)

Write out a model that is compatible with the KIM API.

Parameters

- **path** (Optional[Path]) – Path to write the model. If *None*, defaults to *./NeuralNetwork_KLIFF__MO_000000111111_000*.
- **driver_name** (str) – Name of the model driver.
- **dropout_ensemble_size** (Optional[int]) – Size of the dropout ensemble. Ignored if not fitting a dropout NN. Otherwise, defaults to 100 if *None*.

property descriptor

property device

property dtype

fit(*path*)

Fit the model using analytic solution.

Parameters

path (Path) – path to the fingerprints generated by the descriptor.

load(*filename*, *mode='train'*)

Load a save model.

Parameters

- **filename** (Path) – Path where the model is stored, e.g. kliff_model.pkl

- **mode** (str) – Purpose of the loaded model. Should be either *train* or *eval*.

save(*filename*)

Save a model to disk.

Parameters
filename (Path) – Path to store the model.

property save_frequency

property save_prefix

property save_start

set_save_metadata(*prefix*, *start*, *frequency=1*)

Set metadata that controls how the model are saved during training.

If this function is called before minimization starts, the model will be saved to the directory specified by *prefix* every *frequency* epochs, beginning at the *start* epoch.

Parameters

- **prefix** (Path) – Path to the directory where the models are saved. Model will be named as *{prefix}/model_epoch{ep}.pt*, where *ep* is the epoch number.
- **start** (int) – Epoch number at which begins to save the model.
- **frequency** (int) – Save the model every *frequency* epochs.

class kliff.models.LinearRegression(**args: Any*, ***kwargs: Any*)

Linear regression model.

forward(*x*)

Parameters
x (Tensor) – Descriptors of shape (N, M), where N is the number of descriptors and M is the descriptor size.

fit(*path*)

Fit the model using analytic solution.

property descriptor

property device

property dtype

load(*filename*, *mode='train'*)

Load a save model.

Parameters

- **filename** (Path) – Path where the model is stored, e.g. kliff_model.pkl
- **mode** (str) – Purpose of the loaded model. Should be either *train* or *eval*.

save(*filename*)

Save a model to disk.

Parameters
filename (Path) – Path to store the model.

property save_frequency

property save_prefix

property save_start

set_save_metadata(*prefix*, *start*, *frequency=1*)

Set metadata that controls how the model are saved during training.

If this function is called before minimization starts, the model will be saved to the directory specified by *prefix* every *frequency* epochs, beginning at the *start* epoch.

Parameters

- **prefix** (Path) – Path to the directory where the models are saved. Model will be named as *{prefix}/model_epoch{ep}.pt*, where *ep* is the epoch number.
- **start** (int) – Epoch number at which begins to save the model.
- **frequency** (int) – Save the model every *frequency* epochs.

write_kim_model(*path=None*)

Write the model out as a KIM-API compatible one.

Parameters
path (Optional[Path]) – path to write the model

10.10 kliff.neighbor

class kliff.neighbor.**NeighborList**(*conf*, *infl_dist*, *padding_need_neigh=False*)

Neighbor list class.

This uses the same approach that *LAMMPS* and *KIM* adopt: The atoms in the configuration (assuming a total of N atoms) are named contributing atoms, and padding atoms are created to satisfy the boundary conditions. The contributing atoms are numbered as 1, 2, … N-1, and the padding atoms are numbered as N, N+1, N+2… Neighbors of atom can include both contributing atoms and padding atoms.

Parameters

- **conf** (*Configuration*) – atomic configuration.
- **infl_dist** (float) – Influence distance, within which atoms are interacting with each other. In literatures, this is usually referred as cutoff.
- **padding_need_neigh** (bool) – Whether to generate neighbors for padding atoms.

coords

2D array Coordinates of contributing and padding atoms.

species

list Species string of contributing and padding atoms.

image

1D array Atom index, of which an atom is an image. The image of a contributing atom is itself.

padding_coords

2D array Coordinates of padding atoms.

padding_species

list Species string and padding atoms.

padding_image

1D array Atom index, of which a padding atom is an image.

Note: To get the total force on a contributing atom, the forces on all padding atoms that are images of the contributing atom should be added back to the contributing atom.

---

create_neigh()

get_neigh(*index*)

Get the indices, coordinates, and species string of a given atom.

Parameters
index (int) – Atom number whose neighbor info is requested.

Returns

Indices of neighbor atoms in self.coords and self.species. neigh_coords: 2D array of shape (N, 3), where N is the number of neighbors.

Coordinates of neighbor atoms.

neigh_species: Species symbol of neighbor atoms.

Return type
neigh_indices

get_numneigh_and_neighlist_1D(*request_padding=False*)

Get the number of neighbors and neighbor list for all atoms.

Parameters
request_padding (bool) – If True, the returned number of neighbors and neighbor list include those for padding atoms; If False, only return these for contributing atoms.

Returns

1D array; number of neighbors for all atoms. neighlist: 1D array; indices of the neighbors for all atoms stacked into a

1D array. Its total length is sum(numneigh), and the first numneigh[0] components are the neighbors of atom *0*, the next numneigh[1] components are the neighbors of atom *1* ....

Return type
numneigh

get_coords()

Return coords of both contributing and padding atoms. Shape (N,3).

Return type
array

get_species()

Return species of both contributing and padding atoms.

Return type
List[str]

get_species_code(*mapping*)

Integer species code of both contributing and padding atoms.

Parameters
mapping (Dict[str, int]) – A mapping between species string and its code.

Return type
array

Returns
1D array of integer species code.

get_image()

Return image of both contributing and padding atoms.

It is a 1D array of atom index, of which an atom is an image.

Note: The image of a contributing atom is itself.

Return type
array

get_padding_coords()

Return coords of padding atoms, 2D array of shape (N,3), where N is the number of padding atoms.

Return type
array

get_padding_speices()

Return species string of padding atoms.

Return type
List[str]

get_padding_species_code(*mapping*)

Integer species code of padding atoms.

Parameters
mapping (Dict[str, int]) – A mapping between species string and its code.

Return type
array

Returns
1D array of integer species code for padding atoms.

get_padding_image()

Return image of padding atoms.

It is a 1D array of atom index, of which a padding atom is an image.

Return type
array

kliff.neighbor.**assemble_forces**(*forces*, *n*, *padding_image*)

Assemble forces on padding atoms back to contributing atoms.

Parameters

- **forces** (array) – Partial forces on both contributing and padding atoms. 2D array of shape (Nc+Np, 3). where Nc is the number of contributing atoms, and Np is the number of padding atoms. The first Nc rows are the forces for contributing atoms.
- **n** (int) – Number of contributing atoms, i.e. Nc.
- **padding_image** (array) – atom index, of which the padding atom is an image. 1D int array of shape (Np,).

Return type
array

Returns
Total forces on contributing atoms. 2D array of shape (Nc, 3), where Nc is the number of contributing atoms.

kliff.neighbor.**assemble_stress**(*coords*, *forces*, *volume*)

Calculate the virial stress using the negative f dot r method.

Parameters

- **coords** (array) – Coordinates of both contributing and padding atoms. 2D array of shape (Nc+Np, 3), where Nc is the number of contributing atoms, and Np is the number of padding atoms. The first Nc rows are the coords of contributing atoms.
- **forces** (array) – Partial forces on both contributing and padding atoms. 2D array of shape (Nc+Np, 3), where Nc is the number of contributing atoms, and Np is the number of padding atoms. The first Nc rows are the forces on contributing atoms.
- **volume** (float) – Volume of the configuration.

Return type
array

Returns
Virial stress in Voigt notation. 1D array of shape (6,).

10.11 kliff.nn

class kliff.nn.**Dropout**(**args*, ***kwargs*)

A Dropout layer that zeros the same element of descriptor values for all atoms.

Note *torch.nn.Dropout* dropout each component independently.

Parameters

- **p** – float probability of an element to be zeroed. Default: 0.5
- **inplace** – bool If set to *True*, will do this operation in-place. Default: *False*

Shapes:
Input: [N, D] or [1, N, D] Output: [N, D] or [1, N, D] (same as Input) The first dimension 1 is because the dataloader provides only sample each iteration.

forward(*input*)

10.12 kliff.parallel

kliff.parallel.**parmap1**(*f*, *X*, **args*, *tuple_X=False*, *nprocs=2*)

Parallelism over data.

This function mimics `multiprocessing.Pool.map` to allow extra arguments to be used for the function `f`.

Parameters

- **f** (*function*) – The function that operates on the data.
- **X** (*list*) – Data to be parallelized.
- **args** (*args*) – Extra positional arguments needed by the function `f`.
- **tuple_X** (*bool*) – This depends on `X`. It should be set to `True` if multiple arguments are parallelized and set to `False` if only one argument is parallelized. See `Example` below.
- **nprocs** (*int*) – Number of processors to use.

Returns

A list of results, corresponding to `X`.

Return type

list

Note: The data is put into a job queue, a worker process gets a piece of the data to work on, the worker pushes the result back to the manager through another queue, and then get another piece of data until the job queue is empty. So, in principle, there will not be idle worker and it should be faster than *kliff.parallel.parmap2()*.

Warning: This is implemented using `multiprocessing.Queue`, which requires the function``f`` to be picklable. If it is not the case (e.g. use KIM library functions), use *kliff.parallel.parmap2()* that is based on `multiprocessing.Pipe`.

Example

```
>>> def func(x, y, z=1):
>>>     return x+y+z
>>> X = range(3)
>>> Y = range(3)
>>> parmap1(func, X, 1, nprocs=2)  # [2,3,4]
>>> parmap1(func, X, 1, 1, nprocs=2)  # [2,3,4]
>>> parmap1(func, zip(X, Y), tuple_X=True, nprocs=2)  # [1,3,5]
>>> parmap1(func, zip(X, Y), 1, tuple_X=True, nprocs=2)  # [1,3,5]
```

kliff.parallel.**parmap2**(*f*, *X*, **args*, *tuple_X=False*, *nprocs=2*)

Parallelism over data.

This is to mimic `multiprocessing.Pool.map`, which requires the function `f` to be picklable. This function does not have this restriction and allows extra arguments to be used for the function `f`.

Parameters

- **f** (*function*) – The function that operates on the data.

- **X** (*list*) – Data to be parallelized.
- **args** (*args*) – Extra positional arguments needed by the function `f`.
- **tuple_X** (*bool*) – This depends on `X`. It should be set to `True` if multiple arguments are parallelized and set to `False` if only one argument is parallelized. See `Example` below.
- **nprocs** (*int*) – Number of processors to use.

Returns
A list of results, corresponding to `X`.

Return type
list

Note: This function is implemented using `multiprocessing.Pipe`. The data is subdivided into `nprocs` groups and then each group of data is distributed to a process. The results from each group are then assembled together. The data is shuffled to balance the load in each process. See *kliff.parallel.parmap1()* for another implementation that uses `multiprocessing.Queue`.

Example

```
>>> def func(x, y, z=1):
>>>     return x+y+z
>>> X = range(3)
>>> Y = range(3)
>>> parmap2(func, X, 1, nprocs=2)  # [2,3,4]
>>> parmap2(func, X, 1, 1, nprocs=2)  # [2,3,4]
>>> parmap2(func, zip(X, Y), tuple_X=True, nprocs=2)  # [1,3,5]
>>> parmap2(func, zip(X, Y), 1, tuple_X=True, nprocs=2)  # [1,3,5]
```

kliff.parallel.**get_MPI_world_size**()

kliff.parallel.**get_context**()

10.13 kliff.uq

class kliff.uq.**MCMC**(*loss: Loss*, *nwalkers: Optional[int] = None*, *logprior_fn: Optional[Callable] = None*, *logprior_args: Optional[tuple] = None*, *sampler: Optional[str] = 'ptemcee'*, ***kwargs*)

MCMC sampler class for Bayesian uncertainty quantification.

This is a wrapper over `PtemceeSampler` and `EmceeSampler`. Currently, only these 2 samplers implemented.

Parameters

- **loss** (*Loss*) – Loss function class from *Loss*.
- **nwalkers** (*Optional[int]*) – Number of walkers to simulate. The minimum number of walkers is twice the number of parameters. It defaults to this minimum value.
- **logprior_fn** (*Optional[Callable]*) – A function that evaluate logarithm of the prior distribution. The prior doesn't need to be normalized. It defaults to a uniform prior over a finite range.

- **logprior_args** (*Optional[tuple]*) – Additional positional arguments of the logprior_fn. If the default logprior_fn is used, then the boundaries of the uniform prior can be specified here.
- **sampler** (*Optional[str] or sampler instance*) – An argument that specifies the MCMC sampler to use. The value can be one of the strings "ptemcee" (the default value) or "emcee", or a sampler class instance. If "ptemcee" or "emcee" is given, a respective internal sampler class will be uses.
- ****kwargs** (*Optional[dict]*) – Additional keyword arguments for ptemcee.Sampler or emcee.EnsembleSampler.

builtin_samplers = ['ptemcee', 'emcee']

kliff.uq.**get_T0**(*loss*)

Compute the natural temperature. The minimum loss is the loss value at the optimal parameters.

Parameters
loss (Loss) – Loss function class from *Loss*.

Returns
Value of the natural temperature.

Return type
float

kliff.uq.**mser**(*chain*, *dmin=1*, *dstep=10*, *dmax=-1*, *full_output=False*)

Estimate the equilibration time using marginal standard error rule (MSER). This is done by calculating the standard error (square) of chain_d, where chain_d contains the last $n - d$ element of the chain (n is the total number of iterations for each chain), for progresively larger d values, starting from dmin upto dmax, incremented by dstep. The SE values are stored in a list. Then we search the minimum element in the list and return the index of that element.

Parameters

- **chain** (*1D np.ndarray*) – Array containing the time series.
- **dmin** (*int*) – Index where to start the search in the time series.
- **dstep** (*int*) – How much to increment the search is done.
- **dmax** (*int*) – Index where to stop the search in the time series.
- **full_output** (*bool*) – A flag to return the list of squared standard error.

Returns
dstar – Estimate of the equilibration time using MSER. If full_output=True, then a dictionary containing the estimated equilibration time and the list of squared standard errors will be returned.

Return type
int or dict

kliff.uq.**autocorr**(*chain*, **args*, ***kwargs*)

Use emcee package to estimate the autocorrelation length.

Parameters

- **chain** (*np.ndarray (nwalkers, nsteps, ndim,)*) – Chains from the MCMC simulation. The shape of the chains needs to be (nsteps, nwalkers, ndim). Note that the burn-in time needs to be discarded prior to this calculation
- **args** – Additional positional and keyword arguments of emcee.autocorr.integrated_time.

- **kwargs** – Additional positional and keyword arguments of `emcee.autocorr.integrated_time`.

Returns
Estimate of the autocorrelation length for each parameter.

Return type
float or array

`kliff.uq.rhat(chain, time_axis=1, return_WB=False)`

Compute the value of $\hat{r}$ proposed by Brooks and Gelman [BrooksGelman1998]. If the samples come from PTMCMC simulation, then the chain needs to be from one of the temperature only.

Parameters

- **chain** (*ndarray*) – The MCMC chain as a ndarray, preferrably with the shape (nwalkers, nsteps, ndims). However, the shape can also be (nsteps, nwalkers, ndims), but the argument time_axis needs to be set to 0.
- **time_axis** (*int (optional)*) – Axis in which the time series is stored (0 or 1). For emcee results, the time series is stored in axis 0, but for ptemcee for a given temperature, the time axis is 1.
- **return_WB** (*bool (optional)*) – A flag to return covariance matrices within and between chains.

Returns

- **r** (*float*) – The value of rhat.
- **W, B** (*2d ndarray*) – Matrices of covariance within and between the chains.

References

10.14 kliff.utils

`kliff.utils.length_equal(a, b)`

`kliff.utils.torch_available()`

`kliff.utils.split_string(string, length=80, starter=None)`

Insert *n* into long string such that each line has size no more than *length*.

Parameters

- **string** (`str`) – The string to split.
- **length** – Targeted length of the each line.
- **starter** (`Optional[str]`) – String to insert at the beginning of each line.

`kliff.utils.seed_all(seed=35, cudnn_benchmark=False, cudnn_deterministic=False)`

`kliff.utils.to_path(path)`

Convert str (or filename) to pathlib.Path.

Return type
`Path`

kliff.utils.**download_dataset**(*dataset_name*)

Download dataset and untar it.

Parameters
dataset_name (str) – name of the dataset

Return type
Path

Returns
Path to the dataset

kliff.utils.**create_directory**(*path*, *is_directory=False*)

kliff.utils.**yaml_dump**(*data*, *filename*)

Dump data to a yaml file.

kliff.utils.**yaml_load**(*filename*)

Load data from a yaml file.

kliff.utils.**pickle_dump**(*data*, *filename*)

Dump data to a pickle file.

kliff.utils.**pickle_load**(*filename*)

Load data from a pikel file.

If you find KLIFF useful in your research, please cite:

```
@Article{wen2022kliff,
  title   = {{KLIFF}: A framework to develop physics-based and machine learning␣
↪interatomic potentials},
  author  = {Mingjian Wen and Yaser Afshar and Ryan S. Elliott and Ellad B. Tadmor},
  journal = {Computer Physics Communications},
  volume  = {272},
  pages   = {108218},
  year    = {2022},
  doi     = {10.1016/j.cpc.2021.108218},
 }
```

CHAPTER

ELEVEN

INDICES AND TABLES

- genindex
- modindex
- search

BIBLIOGRAPHY

[lenosky1997] Lenosky, T.J., Kress, J.D., Kwon, I., Voter, A.F., Edwards, B., Richards, D.F., Yang, S., Adams, J.B., 1997. Highly optimized tight-binding model of silicon. Phys. Rev. B 55, 15281544. https://doi.org/10.1103/PhysRevB.55.1528

[wen2016potfit] Wen, M., Li, J., Brommer, P., Elliott, R.S., Sethna, J.P. and Tadmor, E.B., 2016. A KIM-compliant potfit for fitting sloppy interatomic potentials: application to the EDIP model for silicon. Modelling and Simulation in Materials Science and Engineering, 25(1), p.014001.

[transtrum2012geodesicLM] Transtrum, M.K., Sethna, J.P., 2012. Improvements to the Levenberg-Marquardt algorithm for nonlinear least-squares minimization. arXiv:1201.5885 [physics].

[Kurniawan2022] Kurniawan, Y., Petrie, C.L., Williams Jr., K.J., Transtrum, M.K., Tadmor, E.B., Elliott, R.S., Karls, D.S., Wen, M., 2022. Bayesian, frequentist, and information geometric approaches to parametric uncertainty quantification of classical empirical interatomic potentials. J. Chem. Phys. https://doi.org/10.1063/5.0084988

[Frederiksen2004] S. L. Frederiksen, K. W. Jacobsen, K. S. Brown, and J. P. Sethna, "Bayesian Ensemble Approach to Error Estimation of Interatomic Potentials," Phys. Rev. Lett., vol. 93, no. 16, p. 165501, Oct. 2004, doi: 10.1103/PhysRevLett.93.165501.

[KurniawanKLIFFUQ] Kurniawan, Y., Petrie, C.L., Transtrum, M.K., Tadmor, E.B., Elliott, R.S., Karls, D.S., Wen, M., 2022. Extending OpenKIM with an Uncertainty Quantification Toolkit for Molecular Modeling. arXiv:2206.00578 [physics.comp-ph]

[Behler2011] J. Behler, "Atom-centered symmetry functions for constructing high-dimensional neural network potentials," J. Chem. Phys. 134, 074106 (2011).

[Artrith2012] N. Artrith and J. Behler. "High-dimensional neural network potentials for metal surfaces: A prototype study for copper." Physical Review B 85, no. 4 (2012): 045439.

[Artrith2013] N. Artrith, B. Hiller, and J. Behler. "Neural network potentials for metals and oxides–First applications to copper clusters at zinc oxide." physica status solidi (b) 250, no. 6 (2013): 1191-1203.

[Bartok2010] Bartók, Albert P., Mike C. Payne, Risi Kondor, and Gábor Csányi. "Gaussian approximation potentials: The accuracy of quantum mechanics, without the electrons." Physical review letters 104, no. 13 (2010): 136403.

[Thompson2015] Thompson, Aidan P., Laura P. Swiler, Christian R. Trott, Stephen M. Foiles, and Garritt J. Tucker. "Spectral neighbor analysis method for automated generation of quantum-accurate interatomic potentials." Journal of Computational Physics 285 (2015): 316-330.

[BrooksGelman1998] Brooks, S.P., Gelman, A., 1998. General Methods for Monitoring Convergence of Iterative Simulations. Journal of Computational and Graphical Statistics 7, 434455. https://doi.org/10.1080/10618600.1998.10474787

PYTHON MODULE INDEX

k

INDEX

G

H

I

K

L

M

N

O

P

R

S

T

U

V

W

Y