KLIFF Documentation

Release 0.4.3

Mingjian Wen

Dec 25, 2023

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 can be installed via package managers (conda or pip) or from source.

1.1 Installing KLIFF

This recommended way to install KLIFF is via conda. You can install it by:

```
$ conda create --name kliff_env
$ conda activate kliff_env
$ conda install -c conda-forge kliff
```

Alternatively, you can install using pip:

```
$ pip install kliff
```

or from source:

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

1.2 Other dependencies

1.2.1 KIM API and kimpy

KLIFF requires kim-api and kimpy to be installed. If you install KLIFF via conda as described above, these two packages are installed automatically, and you are good to go. Otherwise, you will need to install kim-api and kimpy before installing KLIFF. Of course, you can first install them using conda `$ conda install -c conda-forge kim-api kimpy` and then install KLIFF using pip or from source. Alternatively, you can install them from source as well, and see their documentation for more information.

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.

1.2.3 KIM Models

If you are interested in training physics-based models that are avaialbe from OpenKIM, you will need to install the KIM models that you want to use. After kim-api is installed, you can do `$ kim-api-collections-management list` to see the list of installed KIM models. You can also install the models you want by `$ kim-api-collections-management install <model-name>`. See the kim-api documentation for more information.

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.

CHAPTER

TWO

TUTORIALS

Note: We are transition the tutorials from sphinx-gallery to jupyter notebooks. Some links might be broken and we are working on fixing them.

2.1 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.

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 :download:`Si_training_set.tar.gz` <`https://raw.githubusercontent.com/openkim/kliff/master/examples/Si_training_set.tar.gz`>. (It will be automatically downloaded if not present.) The data is stored in # **extended xyz** format, and see `doc.dataset` for more information of this format.

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
```

```
---------------------------------------------------------------------------
ModuleNotFoundError                       Traceback (most recent call last)
Cell In[1], line 1
----> 1 from kliff.calculators import Calculator
      2 from kliff.dataset import Dataset
      3 from kliff.dataset.weight import Weight

ModuleNotFoundError: No module named 'kliff'
```

2.1.1 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

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'
```

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

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.

2.1.2 Training set

KLIFF has a :class:~`kliff.dataset.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 :class:~`kliff.dataset.weight.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()
```

```
2023-08-01 22:27:08.799 | INFO     | kliff.dataset.dataset:_read:398 - 1000␣
↪configurations read from /Users/mjwen.admin/Packages/kliff/docs/source/tutorials/Si_
↪training_set
```

The `configs` in the last line is a list of :class:~`kliff.dataset.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.

2.1.3 Calculator

:class:~`kliff.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)
```

```
2023-08-01 22:27:09.207 | 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.

2.1.4 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})
```

```
2023-08-01 22:27:09.210 | INFO     | kliff.loss:minimize:310 - Start minimization using␣
↪method: L-BFGS-B.
2023-08-01 22:27:09.212 | INFO     | kliff.loss:_scipy_optimize:429 - Running in␣
↪multiprocessing mode with 2 processes.
```

```
RUNNING THE L-BFGS-B CODE

           * * *

Machine precision = 2.220D-16
 N =            3     M =           10

At X0         0 variables are exactly at the bounds

At iterate    0    f=  4.47164D+03    |proj g|=  4.47898D+03

At iterate    1    f=  1.20212D+03    |proj g|=  2.13266D+03
At iterate    1    f=  1.20212D+03    |proj g|=  2.13266D+03

At iterate    2    f=  2.16532D+02    |proj g|=  1.90519D+02
At iterate    2    f=  2.16532D+02    |proj g|=  1.90519D+02

At iterate    3    f=  2.07552D+02    |proj g|=  1.06071D+02
At iterate    3    f=  2.07552D+02    |proj g|=  1.06071D+02

At iterate    4    f=  1.70033D+02    |proj g|=  3.48082D+02

At iterate    5    f=  1.64800D+02    |proj g|=  3.74180D+02
At iterate    5    f=  1.64800D+02    |proj g|=  3.74180D+02

At iterate    6    f=  1.38087D+02    |proj g|=  1.31340D+02
At iterate    6    f=  1.38087D+02    |proj g|=  1.31340D+02

At iterate    7    f=  1.34855D+02    |proj g|=  1.45391D+01
At iterate    7    f=  1.34855D+02    |proj g|=  1.45391D+01

At iterate    8    f=  1.34599D+02    |proj g|=  1.58968D+01

At iterate    9    f=  1.32261D+02    |proj g|=  8.46707D+01
```

(continues on next page)

(continued from previous page)

```
At iterate   10    f=  1.26954D+02    |proj g|=  2.36049D+02
At iterate   10    f=  1.26954D+02    |proj g|=  2.36049D+02

At iterate   11    f=  1.20788D+02    |proj g|=  2.42511D+02
At iterate   11    f=  1.20788D+02    |proj g|=  2.42511D+02

At iterate   12    f=  9.84653D+01    |proj g|=  2.90333D+02
At iterate   12    f=  9.84653D+01    |proj g|=  2.90333D+02

At iterate   13    f=  7.92970D+01    |proj g|=  1.27395D+02
At iterate   13    f=  7.92970D+01    |proj g|=  1.27395D+02

At iterate   14    f=  6.33426D+01    |proj g|=  1.12669D+02
At iterate   14    f=  6.33426D+01    |proj g|=  1.12669D+02

At iterate   15    f=  5.95658D+01    |proj g|=  2.50284D+02
At iterate   15    f=  5.95658D+01    |proj g|=  2.50284D+02

At iterate   16    f=  5.19898D+01    |proj g|=  2.97639D+02
At iterate   16    f=  5.19898D+01    |proj g|=  2.97639D+02

At iterate   17    f=  3.31620D+01    |proj g|=  2.39904D+02

At iterate   18    f=  2.00817D+01    |proj g|=  2.43105D+01

At iterate   19    f=  1.58825D+01    |proj g|=  1.94992D+02

At iterate   20    f=  1.00645D+01    |proj g|=  3.25943D+02
At iterate   20    f=  1.00645D+01    |proj g|=  3.25943D+02

At iterate   21    f=  4.82724D+00    |proj g|=  2.33796D+01
At iterate   21    f=  4.82724D+00    |proj g|=  2.33796D+01

At iterate   22    f=  3.26863D+00    |proj g|=  7.48010D+01
At iterate   22    f=  3.26863D+00    |proj g|=  7.48010D+01

At iterate   23    f=  2.81339D+00    |proj g|=  2.37520D+01
At iterate   23    f=  2.81339D+00    |proj g|=  2.37520D+01

At iterate   24    f=  2.53369D+00    |proj g|=  2.24782D+01

At iterate   25    f=  2.31427D+00    |proj g|=  4.19973D+01
At iterate   25    f=  2.31427D+00    |proj g|=  4.19973D+01

At iterate   26    f=  1.82162D+00    |proj g|=  5.03854D+01
At iterate   26    f=  1.82162D+00    |proj g|=  5.03854D+01

At iterate   27    f=  1.04312D+00    |proj g|=  2.46183D+01

At iterate   28    f=  7.95851D-01    |proj g|=  1.50873D+01
At iterate   28    f=  7.95851D-01    |proj g|=  1.50873D+01
```

(continues on next page)

(continued from previous page)

```
At iterate   29    f=  7.40878D-01    |proj g|=  1.52873D+00
At iterate   29    f=  7.40878D-01    |proj g|=  1.52873D+00

At iterate   30    f=  7.05900D-01    |proj g|=  1.50051D+01
At iterate   30    f=  7.05900D-01    |proj g|=  1.50051D+01

At iterate   31    f=  6.95221D-01    |proj g|=  4.45629D+00
At iterate   31    f=  6.95221D-01    |proj g|=  4.45629D+00

At iterate   32    f=  6.94089D-01    |proj g|=  1.64352D-01
At iterate   32    f=  6.94089D-01    |proj g|=  1.64352D-01

At iterate   33    f=  6.94079D-01    |proj g|=  2.10362D-02
At iterate   33    f=  6.94079D-01    |proj g|=  2.10362D-02

At iterate   34    f=  6.94078D-01    |proj g|=  8.86005D-03

At iterate   35    f=  6.94078D-01    |proj g|=  8.83015D-03
At iterate   35    f=  6.94078D-01    |proj g|=  8.83015D-03
```

```
2023-08-01 22:27:43.444 | INFO     | kliff.loss:minimize:312 - Finish minimization using␣
↪method: L-BFGS-B.
```

```
At iterate   36    f=  6.94078D-01    |proj g|=  5.10514D-04

           * * *

Tit   = total number of iterations
Tnf   = total number of function evaluations
Tnint = total number of segments explored during Cauchy searches
Skip  = number of BFGS updates skipped
Nact  = number of active bounds at final generalized Cauchy point
Projg = norm of the final projected gradient
F     = final function value

           * * *

   N    Tit     Tnf  Tnint  Skip  Nact     Projg        F
    3     36     44     37     0     0   5.105D-04   6.941D-01
  F =   0.69407801330347585

CONVERGENCE: REL_REDUCTION_OF_F_<=_FACTR*EPSMCH
```

```
      fun: 0.6940780133034758
 hess_inv: <3x3 LbfgsInvHessProduct with dtype=float64>
      jac: array([ 2.62567724e-05, -5.10513851e-04,  1.01474385e-05])
  message: 'CONVERGENCE: REL_REDUCTION_OF_F_<=_FACTR*EPSMCH'
     nfev: 176
      nit: 36
     njev: 44
   status: 0
```

(continues on next page)

(continued from previous page)

```
  success: True
        x: array([14.93863362,  0.58740265,  2.20146126])
```

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")
```

```
2023-08-01 22:27:43.455 | INFO     | kliff.models.kim:write_kim_model:692 - KLIFF␣
↪trained model write to `/Users/mjwen.admin/Packages/kliff/docs/source/tutorials/SW_
↪StillingerWeber_1985_Si__MO_405512056662_006_kliff_trained`
```

```
#================================================================================
# 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
  1.4938633615724747e+01   1.0000000000000000e+00   2.0000000000000000e+01

B 1
  5.8740264694219135e-01

sigma 1
  2.0951000000000000e+00 fix

gamma 1
  2.2014612645628717e+00
```

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`.

.. seealso:: For information about how to load a saved model, see `doc.modules`.

```
%matplotlib inline
```

2.2 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 `tut_kim_sw`). Download the training set :download:`Si_training_set.tar.gz # <https://raw.githubusercontent.com/openkim/kliff/master/examples/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 `doc.dataset` for more information of this format.

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
```

```
---------------------------------------------------------------------------
ModuleNotFoundError                       Traceback (most recent call last)
Cell In[2], line 1
----> 1 from kliff import nn
      2 from kliff.calculators import CalculatorTorch
      3 from kliff.dataset import Dataset

ModuleNotFoundError: No module named 'kliff'
```

2.2.1 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),
```

(continues on next page)

(continued from previous page)

```
    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.

2.2.2 Training set and calculator

The training set and the calculator are the same as explained in `tut_kim_sw`. The only difference is that we need to use the :mod:~`kliff.calculators.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)
```

2.2.3 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)
```

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()
```

```
%matplotlib inline
```

2.3 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 `tut_nn` 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
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,
)
```

```
---------------------------------------------------------------------------
ModuleNotFoundError                       Traceback (most recent call last)
Cell In[2], line 1
----> 1 from kliff import nn
      2 from kliff.calculators.calculator_torch import CalculatorTorchSeparateSpecies
      3 from kliff.dataset import Dataset

ModuleNotFoundError: No module named 'kliff'
```

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
```

(continues on next page)

(continued from previous page)

```
    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
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)
```

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")
```

```
%matplotlib inline
```

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 `tut_kim_sw`.

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

$ kim-api-collections-management install user SW_StillingerWeber_1985_Si__MO_405512056662_006

.. seealso:: This installs the model and its driver into the `User Collection`. See `install_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
```

```
---------------------------------------------------------------------------
ModuleNotFoundError                       Traceback (most recent call last)
Cell In[2], line 3
      1 import numpy as np
----> 3 from kliff.calculators import Calculator
      4 from kliff.dataset import Dataset
      5 from kliff.dataset.weight import Weight

ModuleNotFoundError: No module named 'kliff'
```

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 :class:~`kliff.models.parameter_transform.ParameterTransform`.

To make a direct comparison to `tut_kim_sw`, 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,
)
model.echo_model_params(params_space="original")
```

`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")
```

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.

```
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()
```

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

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 `tut_kim_sw`.

```
# 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")
```

The optimized parameter values from this model training are very close, if not the same, as in `tut_kim_sw`. 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.

```
%matplotlib inline
```

2.5 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

.. seealso:: This installs the model and its driver into the `User Collection`. See `install_model` for more information about installing KIM models.

```python
from multiprocessing import Pool

import numpy as np
from corner import corner

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
```

```
---------------------------------------------------------------------------
ModuleNotFoundError                       Traceback (most recent call last)
Cell In[2], line 4
      1 from multiprocessing import Pool
      3 import numpy as np
----> 4 from corner import corner
      6 from kliff.calculators import Calculator
      7 from kliff.dataset import Dataset

ModuleNotFoundError: No module named 'corner'
```

Before running MCMC, we need to define a loss function and train the model. More detail information about this step can be found in `tut_kim_sw` and `tut_params_transform`.

```python
# 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,
)

```

(continues on next page)

(continued from previous page)

```
# 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)

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

To perform MCMC simulation, we use :class:~`kliff.uq.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 :class:~`kliff.uq.MCMC`. This requires :class:~`kliff.loss.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.

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`.

The sampling processes can be parallelized by specifying the pool. Note that the pool needs to be declared after instantiating :class:~`kliff.uq.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)
```

To run the MCMC sampling, we use :meth:~`kliff.uq.MCMC.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.

```
# 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 :func:~`kliff.uq.mcmc_utils.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}")
```

After discarding the first few iterations as the burn-in time, we only want to keep every $\tau$-th iteration from the remaining chain, where $\tau$ is the autocorrelation length, to ensure uncorrelated samples. This calculation can be done using :func:~`kliff.uq.mcmc_utils.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}")
```

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}")
```

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"$\log(\lambda)$"])
```

2.6 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 `tut_kim_sw`.

```
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")
```

(continues on next page)

(continued from previous page)

```
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")
```

```
2023-08-01 21:59:15.496 | INFO     | kliff.dataset.dataset:_read:398 - 4 configurations␣
↪read from /Users/mjwen.admin/Packages/kliff/docs/source/tutorials/Si_training_set_4_
↪configs
2023-08-01 21:59:15.499 | INFO     | kliff.calculators.calculator:create:107 - Create␣
↪calculator for 4 configurations.
2023-08-01 21:59:15.499 | INFO     | kliff.loss:minimize:310 - Start minimization using␣
↪method: L-BFGS-B.
2023-08-01 21:59:15.500 | INFO     | kliff.loss:_scipy_optimize:427 - Running in serial␣
↪mode.
 This problem is unconstrained.
```

```
#================================================================================
# 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
```

(continues on next page)

(continued from previous page)

```


#==============================================================================
# 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
  2.0000000000000000e+00

epsilon 1
  1.0000000000000000e+00


RUNNING THE L-BFGS-B CODE

           * * *

Machine precision = 2.220D-16
 N =            2     M =           10

At X0         0 variables are exactly at the bounds

At iterate    0    f=  6.40974D+00    |proj g|=  2.92791D+01

At iterate    1    f=  2.98676D+00    |proj g|=  3.18782D+01

At iterate    2    f=  1.56102D+00    |proj g|=  1.02614D+01

At iterate    3    f=  9.61567D-01    |proj g|=  8.00167D+00

At iterate    4    f=  3.20489D-02    |proj g|=  7.63379D-01

At iterate    5    f=  2.42400D-02    |proj g|=  5.96998D-01

At iterate    6    f=  1.49914D-02    |proj g|=  6.87782D-01

At iterate    7    f=  9.48615D-03    |proj g|=  1.59376D-01

At iterate    8    f=  6.69609D-03    |proj g|=  1.14378D-01
```

```
2023-08-01 21:59:16.968 | INFO     | kliff.loss:minimize:312 - Finish minimization using␣
↪method: L-BFGS-B.
```

```
At iterate    9    f=  4.11024D-03    |proj g|=  3.20712D-01

At iterate   10    f=  2.97209D-03    |proj g|=  7.03411D-02

           * * *

```

(continues on next page)

(continued from previous page)

```
Tit   = total number of iterations
Tnf   = total number of function evaluations
Tnint = total number of segments explored during Cauchy searches
Skip  = number of BFGS updates skipped
Nact  = number of active bounds at final generalized Cauchy point
Projg = norm of the final projected gradient
F     = final function value

           * * *

   N    Tit     Tnf  Tnint  Skip  Nact     Projg        F
    2     10     13      1     0     0   7.034D-02   2.972D-03
  F =   2.9720927488600178E-003

STOP: TOTAL NO. of ITERATIONS REACHED LIMIT
#=================================================================================
# 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
  2.0629054951532582e+00

epsilon 1
  1.5614850326987884e+00
```

2.7 Train a linear regression potential

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

```python
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()

```

(continues on next page)

(continued from previous page)

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

```
2023-08-01 21:59:01.754 | INFO     | kliff.dataset.dataset:_read:398 - 400␣
↪configurations read from /Users/mjwen.admin/Packages/kliff/docs/source/tutorials/Si_
↪training_set/varying_alat
2023-08-01 21:59:01.755 | INFO     | kliff.calculators.calculator_torch:_get_device:592 -
↪ Training on cpu
2023-08-01 21:59:01.756 | INFO     | kliff.descriptors.descriptor:generate_
↪fingerprints:103 - Start computing mean and stdev of fingerprints.
2023-08-01 21:59:11.127 | INFO     | kliff.descriptors.descriptor:generate_
↪fingerprints:120 - Finish computing mean and stdev of fingerprints.
2023-08-01 21:59:11.129 | INFO     | kliff.descriptors.descriptor:generate_
↪fingerprints:128 - Fingerprints mean and stdev saved to `fingerprints_mean_and_stdev.
↪pkl`.
2023-08-01 21:59:11.129 | INFO     | kliff.descriptors.descriptor:_dump_fingerprints:163␣
↪- Pickling fingerprints to `fingerprints.pkl`
2023-08-01 21:59:11.131 | INFO     | kliff.descriptors.descriptor:_dump_fingerprints:175␣
↪- Processing configuration: 0.
2023-08-01 21:59:11.199 | INFO     | kliff.descriptors.descriptor:_dump_fingerprints:175␣
↪- Processing configuration: 100.
2023-08-01 21:59:11.261 | INFO     | kliff.descriptors.descriptor:_dump_fingerprints:175␣
↪- Processing configuration: 200.
2023-08-01 21:59:11.325 | INFO     | kliff.descriptors.descriptor:_dump_fingerprints:175␣
↪- Processing configuration: 300.
2023-08-01 21:59:11.386 | INFO     | kliff.descriptors.descriptor:_dump_fingerprints:218␣
↪- Pickle 400 configurations finished.
```

We can train a linear regression model by minimizing a loss function as discussed in `tut_nn`. 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")
```

```
2023-08-01 21:59:11.626 | INFO     | kliff.models.linear_regression:fit:42 - Finished␣
↪fitting model "LinearRegression"
```

```
Finished fitting model "LinearRegression"
```

CHAPTER

THREE

THEORY

A parametric potential typically takes the form

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

where $r_1, \dots, r_{N_a}$ and $Z_1, \dots, Z_{N_a}$ 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 $\{q_1, \dots, q_{N_p}\}$ 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

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

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:

```python
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:

```python
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:

```python
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.

```python
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:

```python
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:

```python
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`.

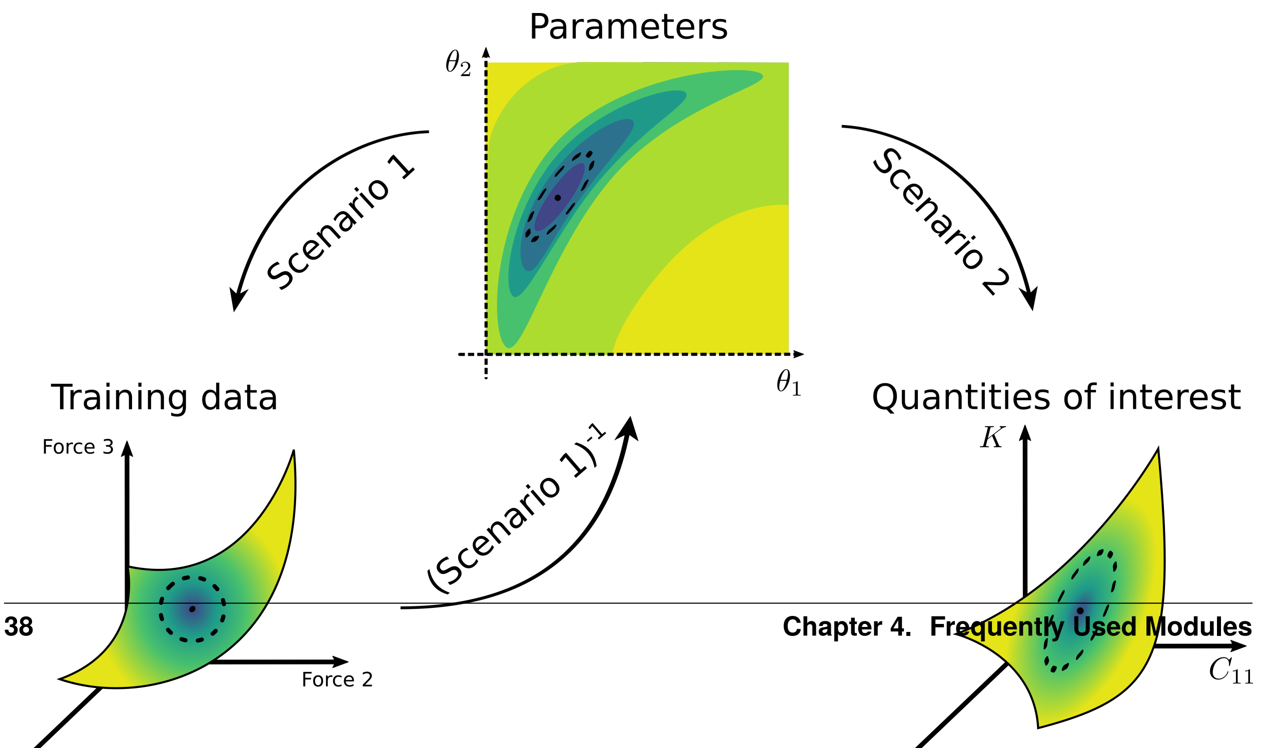
See also:

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 of 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 (represented by the dashed ellipse on the middle plot of the figure above). 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.

In KLIFF, the UQ tools are implemented in `uq`. In the current version, there are 2 methods implemented: Bayesian MCMC sampling and bootstrapping, with the integration of other UQ methods will be added in the future.

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 estimating 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].

Implementation

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 the user to specify the prior distribution $\pi(\theta)$ to use. The function should accept an array of parameter values as 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 steps to take. Since the position of 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 via `sampler.chain` as a $K \times L \times M \times N$ array, where M is the total number of iterations.

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 setting it to `sampler.pool` prior to running MCMC, for example:

```python
from multiprocessing import Pool

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

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

```python
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 the argument `nprocs` > 1) and MPI in the likelihood evaluation across different walkers. Then, we can run the Python script as follows.

```bash
$ 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
```

MCMC analysis

The chains from the MCMC simulation need 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 at 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 ensures 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 the Markov chain, the position at step i is not independent of 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, uncorrelated 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 criterion is satisfied.

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

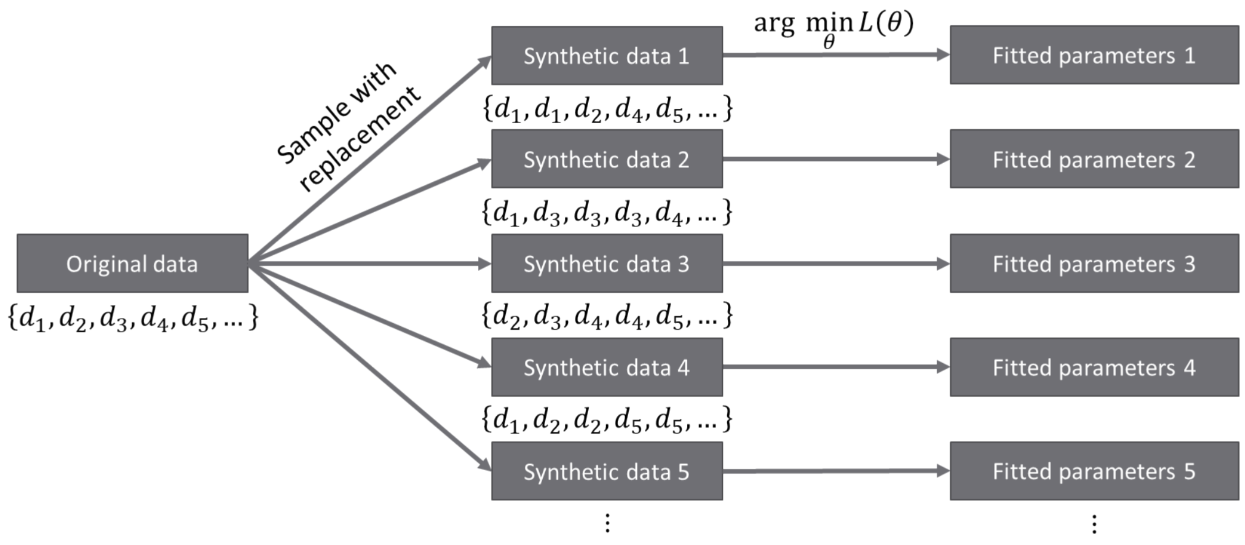
See also:

See the tutorial for running MCMC in tut_mcmc.

4.6.2 Bootstrap

In general, the training dataset contains some random noise. When the data collection process is repeated, we will not get the exact same values, but instead, we will get (slightly) different values, where the deviation comes from the random noise. If we train the model to fit different realizations of the training dataset, we will get a distribution of the parameters. The uncertainty of the parameters from this distribution gives how the error in the training data is propagated to the uncertainty of the parameters. However, oftentimes we don't have the luxury to repeat the data collection. A suggestion, in this case, is to generate artificial datasets and train the model to fit these artificial datasets.

Bootstrapping is a way to generate artificial datasets. We assume that the original dataset contains N *independent and identically distributed (iid)* data points. An artificial, bootstrap dataset is generated by sample M points from the original dataset with replacement. Note that this means that there are some data points that are repeated, while some



other data points are not sampled, thus the bootstrap dataset is not the same as the original dataset. The difference between the datasets gives a sense of probability in data.

Note: Although usually M is set to be the same as N, in principle it doesn't need to be.

Implementation

Bootstrapping is implemented in `Bootstrap`. A general workflow for this calculation is

1. Instantiate `Bootstrap` class instance.

   This process is straightforward. The only required argument is the `Loss` instance.

   ```
   from kliff.uq import Bootstrap

   loss = ...  # define the loss function
   # Train the potential
   min_kwargs = ...  # Optimizer setting
   loss.minimize(**min_kwargs)

   bs = Bootstrap(loss, *args, **kwargs)
   ```

   When instantiating the parent class `Bootstrap`, it will return either an instance of `BootstrapEmpiricalModel` or `BootstrapNeuralNetworkModel`, depending on whether we have a physics-based (empirical) model or a neural network model, respectively. When a neural network model is used, the user can specify an additional argument *orig_state_filename*, which specified the name and path of the file to use to export the initial state of the model prior to running bootstrap. This is to reset the state of the model at the end of performing bootstrap UQ.

2. Generate bootstrap datasets.

   In this implementation, we assume that the training dataset consists of many atomic configurations and the corresponding quantities. Note that the quantities corresponding to a single atomic configuration are **not** independent of each other. Thus, the resampling process to generate a bootstrap dataset should not be done at the data point level. Instead, we should generate a bootstrap dataset by resampling the atomic configurations.

The built-in bootstrap dataset generator function was set up to perform this type of resampling. Note that atomic configurations here are referred to as compute arguments, which also contain the type of data and weights to use.

```
nsamples = ...  # Number of samples to generate
bs.generate_bootstrap_compute_arguments(nsamples)
```

When an empirical model with multiple calculators is used, the resampling is done to the combined list of the compute arguments across all calculators. Then, an internal function will automatically assign back the bootstrap compute arguments to their respective calculators. This means that the number of compute arguments in each calculator when we do bootstrapping is more likely to be different than the original number of compute arguments per calculator, although the total number of compute arguments is still the same.

Also, note that the built-in bootstrap compute arguments generator assumes that the configurations are independent of each other. In the case where this is not satisfied, then a more sophisticated resampling method should be used. This can be done by defining a custom bootstrap compute arguments generator function. The only required argument for this function is the requested number of samples.

3. Run the optimization for each bootstrap dataset.

   After a set of bootstrap compute arguments is generated, then we need to iterate over each of them, and train the potential to fit each bootstrap dataset.

   ```
   bs.run(min_kwargs=min_kwargs)
   ```

   There are 2 arguments that are the same to run the optimization stage of bootstrapping, regardless if we use an empirical or neural network model. These arguments are:

   - `min_kwargs`, which is a dictionary containing the keyword arguments that will be passed into the optimizer. This argument can be thought of as the optimizer setting.

     Note: Since the mapping from the bootstrap dataset to the inferred parameters contains optimization, then it is recommended to use the same optimizer setting when we iterate over each bootstrap compute argument and train the potential. Additionally, the optimizer setting should also be the same as the setting used in the initial training, when we use the original set of compute arguments to train the potential.

   - `callback`, which is an option to specify a function that will be called in each iteration. This can be used as a debugging tool or to monitor convergence.

   For other additional arguments, please refer to the respective function documentation, i.e., `run()` for an empirical model or `run()` for neural network model.

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 tut_kim_sw, 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 tut_kim_sw, 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 tut_kim_sw and tut_nn.

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 tut_kim_sw and tut_nn, 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 tut_kim_sw 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 tut_kim_sw, 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

We welcome all kinds of contributions to KLIFF – typo fix, bug reports, feature requests, and documentation improvements. If you are interested in contributing to KLIFF, please read this guide. If you have any questions, please feel free to open an issue or contact us.

7.1 Code style

KLIFF adopts the Google style for docstrings in Python code. Other docs like this file is written in MyST markdown, which is an extension of the standard markdown. If you are familiar with the standard markdown, it should be easy to write in MyST.

KLIFF uses isort, black a set of other tools to format and statically check the correctness of the code. These tools are already configured using `pre-commit`. To use it,

7.1.1 Install pre-commit

```
pip install pre-commit
```

7.1.2 Install pre-commit hooks for KLIFF

```
cd kliff
pre-commit install
```

7.1.3 Run pre-commit checks

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

This will run all the checks on all files. If there are warnings and errors reported, you can fix them and then commit the changes.

Note: After `pre-commit install` the checks will be run automatically before each commit. You can do `pre-commit uninstall` to disable the checks.

7.2 Testing

If you are contributing new codes, please add tests for them. All the tests are placed in the `kliff/tests` directory. We use pytest for testing. After adding a new test, you can run it locally to make sure it passes.

First install the dependencies for testing:

```
cd kliff
pip install -e ".[tests]"
``
```

Then run the tests:

```
pytest
```

This will run all the tests. If you want to run a specific test, you can do

```
pytest path/to/your/awesome/test.py
```

7.3 Build the docs locally

You can generate the docs (including the tutorials) locally. First, install the dependencies for building the docs:

```
cd kliff
pip install -e ".[docs]"
```

Then you can build the docs:

```
cd docs
make html
```

The generated docs will be at `kliff/docs/build/html/index.html`, and you can open it in a browser.

7.4 Tutorials

If you have great tutorials, please write it in Jupyter notebook and place them in the `kliff/docs/tutorials` directory. Then update the `kliff/docs/tutorials.rst` file to include the new tutorials. After this, the tutorials will be automatically built and included in the docs.

In your Jupyter notebook, you can use MyST markdown to write the text.

Warning: We are in the process of migrating some of the docs from RestructuredText to MyST markdown. So you may see some of the docs are written in RestructuredText, and some links may be broken. We will fix them soon.

CHAPTER EIGHT

CHANGE LOG

8.1 v0.4.3 (2023/12/17)

- Fix installing ptemcee

8.2 v0.4.2 (2023/12/16)

8.2.1 Enhancements

- Refactor test by @mjwen in https://github.com/openkim/kliff/pull/125
- Update the ptemcee dependency by @yonatank93 in https://github.com/openkim/kliff/pull/137
- Update GH actions to use latest conda-forge kim-api and test on macOS by @mjwen in https://github.com/openkim/kliff/pull/143

8.2.2 Documentation

- Recreate docs building codes by @mjwen in https://github.com/openkim/kliff/pull/129

8.2.3 Other Changes

- Fix neighbor list bug by @mjwen in https://github.com/openkim/kliff/pull/90
- Fix _WrapperCalculator by @mjwen in https://github.com/openkim/kliff/pull/95
- Remove requirements.txt, add info in setup.py by @mjwen in https://github.com/openkim/kliff/pull/108
- Add multiple species support of LJ by @mjwen in https://github.com/openkim/kliff/pull/112
- Update CI to fix cmake version by @mjwen in https://github.com/openkim/kliff/pull/117
- WIP: Implement bootstrap by @yonatank93 in https://github.com/openkim/kliff/pull/107

8.3 v0.4.1 (2022/10/06)

8.3.1 Added

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

8.3.2 Fixed

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

8.3.3 Updated

- Documentation on installing KLIFF and dependencies

8.4 v0.4.0 (2022/04/27)

8.4.1 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

8.4.2 Fixed

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

8.4.3 Updated

- Change to use precommit GH action to check code format

8.5 v0.3.3 (2022/03/25)

8.5.1 Fixed

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

8.6 v0.3.2 (2022/03/01)

8.6.1 Added

- Enable params_relation_callback() for KIM model

8.6.2 Fixed

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

8.7 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)

8.8 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

8.9 v0.2.2 (2021/05/24)

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

8.10 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`

8.11 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

8.12 v0.1.7 (2020/12/20)

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

8.13 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

8.14 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

8.15 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

8.16 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`

8.17 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`.

8.18 v0.1.0 (2019/3/29)

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

- Add more docs to *Package Reference*.

8.19 v0.0.1 (2019/1/1)

Pre-release.

CHAPTER NINE

CHANGE LOG

9.1 v0.4.1 (2022/10/06)

9.1.1 Added

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

9.1.2 Fixed

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

9.1.3 Updated

- Documentation on installing KLIFF and dependencies

9.2 v0.4.0 (2022/04/27)

9.2.1 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

9.2.2 Fixed

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

9.2.3 Updated

- Change to use precommit GH action to check code format

9.3 v0.3.3 (2022/03/25)

9.3.1 Fixed

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

9.4 v0.3.2 (2022/03/01)

9.4.1 Added

- Enable params_relation_callback() for KIM model

9.4.2 Fixed

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

9.5 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)

9.6 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

9.7 v0.2.2 (2021/05/24)

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

9.8 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`

9.9 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

9.10 v0.1.7 (2020/12/20)

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

9.11 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

9.12 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

9.13 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

9.14 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`

9.15 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`.

9.16 v0.1.0 (2019/3/29)

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

- Add more docs to *Package Reference*.

9.17 v0.0.1 (2019/1/1)

Pre-release.

CHAPTER

TEN

CHANGE LOG

10.1 Enhancements

- Refactor test by @mjwen in https://github.com/openkim/kliff/pull/125
- Update the ptemcee dependency by @yonatank93 in https://github.com/openkim/kliff/pull/137
- Update GH actions to use latest conda-forge kim-api and test on macOS by @mjwen in https://github.com/openkim/kliff/pull/143

10.2 Documentation

- Recreate docs building codes by @mjwen in https://github.com/openkim/kliff/pull/129

10.3 Other Changes

- Fix neighbor list bug by @mjwen in https://github.com/openkim/kliff/pull/90
- Fix _WrapperCalculator by @mjwen in https://github.com/openkim/kliff/pull/95
- Remove requirements.txt, add info in setup.py by @mjwen in https://github.com/openkim/kliff/pull/108
- Add multiple species support of LJ by @mjwen in https://github.com/openkim/kliff/pull/112
- Update CI to fix cmake version by @mjwen in https://github.com/openkim/kliff/pull/117
- WIP: Implement bootstrap by @yonatank93 in https://github.com/openkim/kliff/pull/107

10.4 v0.4.1 (2022/10/06)

10.4.1 Added

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

10.4.2 Fixed

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

10.4.3 Updated

- Documentation on installing KLIFF and dependencies

10.5 v0.4.0 (2022/04/27)

10.5.1 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

10.5.2 Fixed

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

10.5.3 Updated

- Change to use precommit GH action to check code format

10.6 v0.3.3 (2022/03/25)

10.6.1 Fixed

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

10.7 v0.3.2 (2022/03/01)

10.7.1 Added

- Enable params_relation_callback() for KIM model

10.7.2 Fixed

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

10.8 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)

10.9 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

10.10 v0.2.2 (2021/05/24)

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

10.11 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`

10.12 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

10.13 v0.1.7 (2020/12/20)

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

10.14 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

10.15 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

10.16 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

10.17 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`

10.18 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`.

10.19 v0.1.0 (2019/3/29)

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

- Add more docs to *Package Reference*.

10.20 v0.0.1 (2019/1/1)

Pre-release.

CHAPTER

ELEVEN

CHANGE LOG

11.1 v0.4.1 (2022/10/06)

11.1.1 Added

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

11.1.2 Fixed

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

11.1.3 Updated

- Documentation on installing KLIFF and dependencies

11.2 v0.4.0 (2022/04/27)

11.2.1 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

11.2.2 Fixed

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

11.2.3 Updated

- Change to use precommit GH action to check code format

11.3 v0.3.3 (2022/03/25)

11.3.1 Fixed

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

11.4 v0.3.2 (2022/03/01)

11.4.1 Added

- Enable params_relation_callback() for KIM model

11.4.2 Fixed

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

11.5 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)

11.6 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

11.7 v0.2.2 (2021/05/24)

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

11.8 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`

11.9 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

11.10 v0.1.7 (2020/12/20)

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

11.11 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

11.12 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

11.13 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

11.14 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`

11.15 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`.

11.16 v0.1.0 (2019/3/29)

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

- Add more docs to *Package Reference*.

11.17 v0.0.1 (2019/1/1)

Pre-release.

CHAPTER

TWELVE

FREQUENTLY ASKED QUESTIONS

12.1 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.

12.2 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.

12.3 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

THIRTEEN

PACKAGE REFERENCE

13.1 kliff.analyzers

13.2 kliff.atomic_data

13.3 kliff.calculators

13.4 kliff.dataset

13.5 kliff.descriptors

13.6 kliff.error

13.7 kliff.log

13.8 kliff.loss

13.9 kliff.models

13.10 kliff.neighbor

13.11 kliff.nn

13.12 kliff.parallel

13.13 kliff.uq

13.14 kliff.utils

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

FOURTEEN

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, in: 2022 IEEE 18th International Conference on E-Science (e-Science). Presented at the 2022 IEEE 18th International Conference on e-Science (e-Science), pp. 367–377. https://doi.org/10.1109/eScience55777.2022.00050