flare Documentation

Release 0.0.1

Jonathan Vandermause

Jul 09, 2020

Contents

1 Contents **3**

1.1 Installation . . . 3
1.1.1 Requirements . . . 3
1.1.2 Installation using pip . . . 3
1.1.3 Manual Installation with Git . . . 3
1.1.4 Acceleration with multiprocessing and MKL . . . 4
1.1.5 Environment variables (optional) . . . 4
1.2 Tutorials . . . 5
1.2.1 Prepare your data . . . 5
1.2.1.1 VASP data . . . 5
1.2.1.2 Data from Quantum Espresso, LAMMPS, etc. . . . 5
1.2.1.3 Try building GP from data . . . 5
1.2.2 Training a Gaussian Process from an AIMD Run . . . 6
1.2.2.1 Roadmap Figure . . . 6
1.2.2.2 Step 1: Setting up a Gaussian Process Object . . . 6
1.2.2.3 Step 2 (Optional): Extracting the Frames from a previous AIMD Run . . . 8
1.2.2.4 Step 3: Training your Gaussian Process . . . 9
1.2.2.5 Pre-Training arguments . . . 10
1.2.3 On-the-fly aluminum potential . . . 10
1.2.3.1 Step 1: Set up a GP Model . . . 10
1.2.3.2 Step 2: Set up DFT Calculator . . . 11
1.2.3.3 Step 3: Set up OTF MD Training Engine . . . 11
1.2.3.4 Step 4: Launch the OTF Training . . . 12
1.2.4 On-the-fly training using ASE . . . 12
1.2.4.1 Step 1: Set up supercell with ASE . . . 12
1.2.4.2 Step 2: Set up FLARE calculator . . . 13
1.2.4.3 Step 3: Set up DFT calculator . . . 14
1.2.4.4 Step 4: Set up On-The-Fly MD engine . . . 15
1.2.5 After Training . . . 19
1.2.5.1 Parse OTF log file . . . 20
1.2.5.2 Construct GP model from log file . . . 20
1.2.5.3 Map the GP force field & Dump LAMMPS coefficient file . . . 20
1.2.5.4 Run LAMMPS with MGP pair style . . . 21
1.2.6 Compile LAMMPS with MGP Pair Style . . . 22
1.2.6.1 MPI . . . 22
1.2.6.2 MPI+OpenMP through Kokkos . . . 23

1.2.6.3 MPI+CUDA through Kokkos . . . 23
1.2.6.4 Notes on Newton (only relevant with Kokkos) . . . 24
1.3 Code Documentation . . . 24
1.3.1 Gaussian Process Force Fields . . . 24
1.3.1.1 Structures . . . 24
1.3.1.2 Atomic Environments . . . 26
1.3.1.3 Kernels . . . 27
1.3.1.4 Predict . . . 88
1.3.1.5 Helper functions for GP . . . 91
1.3.1.6 Output . . . 94
1.3.2 On-the-Fly Training . . . 101
1.3.2.1 DFT Interface . . . 101
1.3.3 Mapped Gaussian Process . . . 107
1.3.3.1 Splines Methods . . . 107
1.3.4 ASE Interface . . . 110
1.3.4.1 FLARE ASE Calculator . . . 110
1.3.4.2 On-the-fly training . . . 110
1.3.5 GP From AIMD . . . 112
1.3.5.1 Seed frames . . . 112
1.3.6 Utility . . . 113
1.3.6.1 Conversion between atomic numbers and element symbols . . . 113
1.3.6.2 Conditions to add training data . . . 113
1.3.6.3 Advanced Hyperparameters Set Up . . . 115
1.3.6.4 Construct Atomic Environment . . . 121
1.3.6.5 Utilities for Molecular Dynamics . . . 123
1.3.6.6 I/O for trajectories . . . 124
1.4 C++ Extension . . . 125
1.5 Frequently Asked Questions . . . 125
1.5.1 Frequently Asked Questions . . . 125
1.5.1.1 Installation and Packages . . . 125
1.5.1.2 Gaussian Processes and OTF . . . 125
1.5.1.3 GPFA . . . 126
1.5.1.4 MGP . . . 126
1.6 Applications . . . 126
1.7 How To Contribute . . . 127
1.7.1 Git Workflow . . . 127
1.7.1.1 General workflow . . . 127
1.7.1.2 Master, development, and topic branches . . . 127
1.7.1.3 Pushing changes to the MIR repo directly . . . 127
1.7.1.4 Pushing changes from a forked repo . . . 127
1.7.2 Code Standards . . . 128
1.7.2.1 PEP 8 . . . 128
1.7.2.2 Docstrings . . . 128
1.7.2.3 Tests . . . 128
1.8 How to Cite . . . 128

Python Module Index **129**

Index **131**


FLARE

CHAPTER 1

Contents

1.1 Installation

1.1.1 Requirements

- Python
- NumPy
- SciPy
- Memory_profiler
- Numba

Optional:

- ASE
- Pymatgen

1.1.2 Installation using pip

Pip can automatically fetch the source code from PyPI and install.

```
$ pip install mir-flare
```

For non-admin users

```
$ pip install --upgrade --user mir-flare
```

1.1.3 Manual Installation with Git

First, copy the source code from https://github.com/mir-group/flare

```
$ git clone https://github.com/mir-group/flare.git
```

Then add the current path to PYTHONPATH

```
$ cd flare; export PYTHONPATH=$(pwd):$PYTHONPATH
```

1.1.4 Acceleration with multiprocessing and MKL

If users have access to high-performance computers, we recommend Multiprocessing and MKL library set up to accelerate the training and prediction. The acceleration can be significant when the GP training data is large. This can be done in the following steps.

First, make sure the Numpy library is linked with MKL or Openblas and Lapack.

```
$ python -c "import numpy as np; print(np.__config__.show())"
```

If no libraries are linked, Numpy should be reinstalled. Detailed steps can be found in Conda manual.

Second, in the initialization of the GP class and OTF class, turn on the GP parallelizatition and turn off the OTF par.

```
gp_model = GaussianProcess(..., parallel=True, per_atom_par=False, n_cpus=2)
otf_instance = OTF(..., par, n_cpus=2)
```

Third, set the number of threads for MKL before running your python script.

```
export OMP_NUM_THREAD=2
python training.py
```

Note: The "n_cpus" and OMP_NUM_THREAD should be equal or less than the number of CPUs available in the computer. If these numbers are larger than the actual CPUs number, it can lead to an overload of the machine.

Note: If gp_model.per_atom_par=True and NUM_OMP_THREAD>1, it is equivalent to run with NUM_OMP_THREAD*otf.n_cpus threads because the MKL calls are nested in the multiprocessing code.

The current version of FLARE can only support parallel calculations within one compute node. Interfaces with MPI using multiple nodes are still under development.

If users encounter unusually slow FLARE training and prediction, please file us a Github Issue.

1.1.5 Environment variables (optional)

Flare uses a couple environmental variables in its tests for DFT and MD interfaces. These variables are not needed in the run of active learning.

```
# the path and filename of Quantum Espresso executable
export PWSCF_COMMAND=$(which pw.x)
# the path and filename of CP2K executable
export CP2K_COMMAND=$(which cp2k.popt)
# the path and filename of LAMMPS executable
export lmp=$(which lmp_mpi)
```

1.2 Tutorials

1.2.1 Prepare your data

If you have collected data for training, including atomic positions, chemical species, cell etc., you need to convert it into a list of `Structure` objects. Below we provide a few examples.

1.2.1.1 VASP data

If you have AIMD data from VASP, you can follow the step 2 of this instruction to generate `Structure` with the `vasprun.xml` file.

1.2.1.2 Data from Quantum Espresso, LAMMPS, etc.

If you have collected data from any calculator that ASE supports, or have dumped data file of format that ASE supports, you can convert your data into ASE `Atoms`, then from `Atoms` to `Structure` via `Structure.from_ase_atoms`.

For example, if you have collected data from QE, and obtained the QE output file `.pwo`, you can parse it with ASE, and convert ASE `Atoms` into `Structure`.

```
from ase.io import read
from flare.struc import Structure

frames = read('data.pwo', index=':', format='espresso-out') # read the whole traj
trajectory = []
for atoms in frames:
    trajectory.append(Structure.from_ase_atoms(atoms))
```

If the data is from the LAMMPS dump file, use

```
# if it's text file
frames = read('data.dump', index=':', format='lammps-dump-text')

# if it's binary file
frames = read('data.dump', index=':', format='lammps-dump-binary')
```

Then the `trajectory` can be used to train GP from AIMD data.

1.2.1.3 Try building GP from data

To have a more complete and better monitored training process, please use our GPFA module.

Here we are not going to use this module, but only provide a simple example on how the GP is constructed from the data.

```
from flare.gp import GaussianProcess
from flare.utils.parameter_helper import ParameterHelper

# set up hyperparameters, cutoffs
kernels = ['twobody', 'threebody']
parameters = {'cutoff_twobody': 4.0, 'cutoff_threebody': 3.0}
pm = ParameterHelper(kernels=kernels,
```

(continues on next page)

(continued from previous page)

```
                    random=True,
                    parameters=parameters)
hm = pm.as_dict()
hyps = hm['hyps']
cutoffs = hm['cutoffs']
hl = hm['hyp_labels']

kernel_type = 'mc' # multi-component. use 'sc' for single component system

# build up GP model
gp_model = \
    GaussianProcess(kernels=kernels,
                    component=kernel_type,
                    hyps=hyps,
                    hyp_labels=hl,
                    cutoffs=cutoffs,
                    hyps_mask=hm,
                    parallel=False,
                    n_cpus=1)

# feed training data into GP
# use the "trajectory" as from above, a list of Structure objects
for train_struc in trajectory:
    gp_model.update_db(train_struc, forces)
gp_model.check_L_alpha() # build kernel matrix from training data

# make a prediction with gp, test on a training data
test_env = gp_model.training_data[0]
gp_pred = gp_model.predict(test_env, 1) # obtain the x-component
                                        # (force_x, var_x)
                                        # x: 1, y: 2, z: 3
print(gp_pred)
```

1.2.2 Training a Gaussian Process from an AIMD Run

Steven Torrisi (torrisi@g.harvard.edu), December 2019

In this tutorial, we'll demonstrate how a previously existing Ab-Initio Molecular Dynamics (AIMD) trajectory can be used to train a Gaussian Process model.

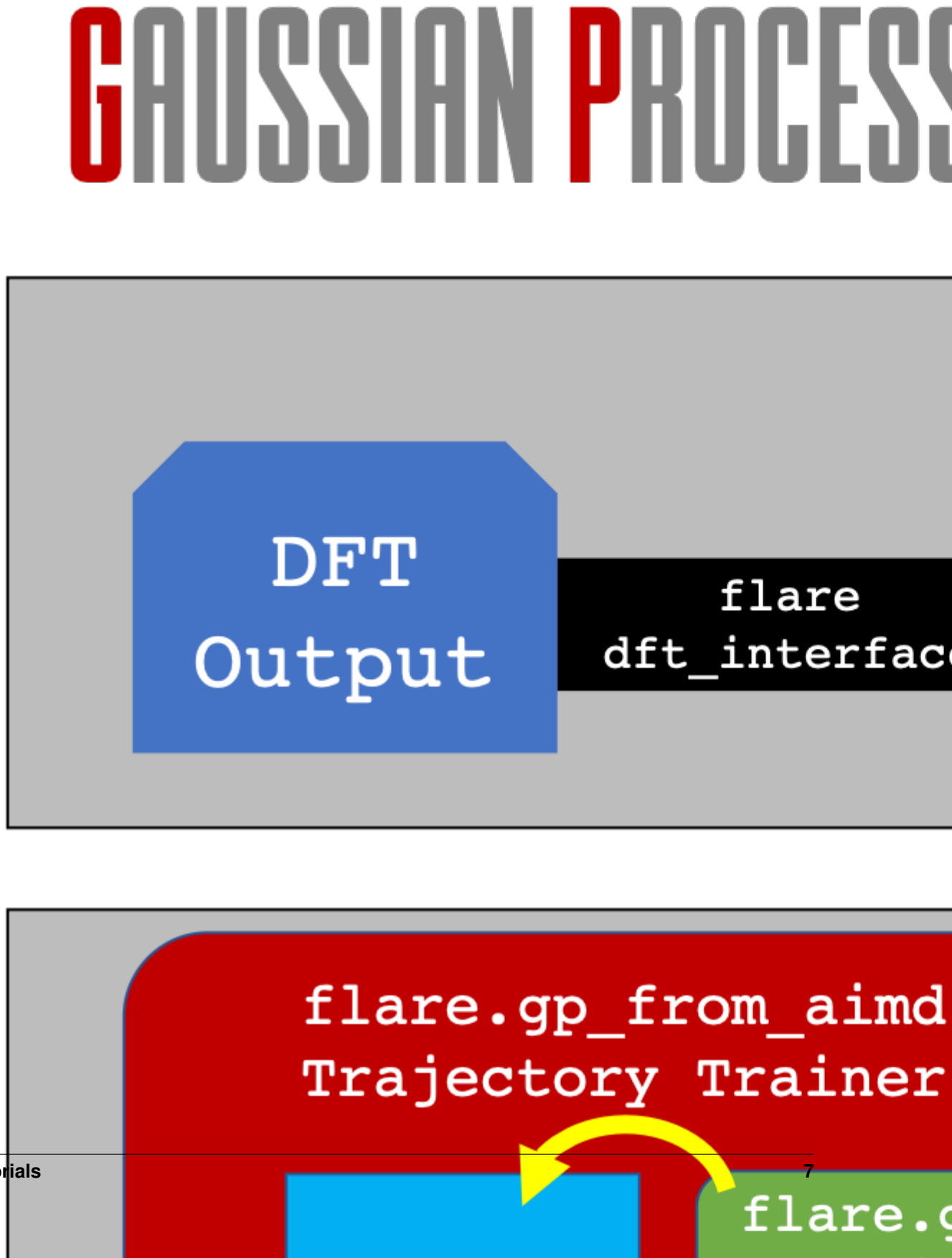
We can use a very short trajectory for a very simple molecule which is already included in the test files in order to demonstrate how to set up and run the code. The trajectory this tutorial focuses on involves a few frames of the molecule Methanol vibrating about its equilibrium configuration ran in VASP.

1.2.2.1 Roadmap Figure

In this tutorial, we will walk through the first two steps contained in the below figure. the GP from AIMD module is designed to give you the tools necessary to extract FLARE structures from a previously existing molecular dynamics run.

1.2.2.2 Step 1: Setting up a Gaussian Process Object

Our goal is to train a GP, which first must be instantiated with a set of parameters.


GAUSSIAN PROCESS
DFT
Output
flare
dft_interfac
flare.gp_from_aimd
Trajectory Trainer
flare.

For the sake of this example, which is a molecule, we will use a two-plus-three body kernel. We must provide the kernel name to the GP, "two-plus-three-mc" or "2+3mc'. Our initial guesses for the hyperparameters are not important. The hyperparameter labels are included below for later output. The system contains a small number of atoms, so we choose a relatively smaller 2-body cutoff (7 A) and a relatively large 3-body cutoff (7 A), both of which will completely contain the molecule.

At the header of a file, include the following imports:

```
from flare.gp import GaussianProcess
```

We will then set up the `GaussianProcess` object.

- The `GaussianProcess` object class contains the methods which, from an `AtomicEnvironment` object, predict the corresponding forces and uncertainties by comparing the atomic environment to each environment in the training set. The kernel we will use has 5 hyperparameters and requires two cutoffs.
- The first four hyperparameters correspond to the signal variance and length scale which parameterize the two- and three-body comparison functions. These hyperparameters will be optimized later once data has been fed into the `GaussianProcess` via likelihood maximization. The fifth and final hyperparameter is the noise variance. We provide simple initial guesses for each hyperparameter.
- The two cutoff values correspond to the functions which set up the two- and three-body Atomic Environments. Since Methanol is a small molecule, 7 Angstrom each will be sufficent.
- The kernel name must contain the terms you want to use.
- Here, we will use the `two_plus_three_body_mc` kernel, which uses two-body and three-body comparisons. `mc` means multi-component, indicating that it can handle multiple atomic species being present.

```
gp = GaussianProcess(kernels=['twobody', 'threebody'],
hyps=[0.01, 0.01, 0.01, 0.01, 0.01],
cutoffs = {'twobody':7, 'threebody':3},
hyp_labels=['Two-Body Signal Variance','Two-Body Length Scale','Three-Body Signal␣
↪Variance',
'Three-Body Length Scale', 'Noise Variance']
                )
```

1.2.2.3 Step 2 (Optional): Extracting the Frames from a previous AIMD Run

FLARE offers a variety of modules for converting DFT outputs into FLARE structures, which are then usable for model training and prediction tasks. For this example, we highlight the vasp_util module, which has a function called `md_trajectory_from_vasprun`, which can convert a `vasprun.xml` file into a list of FLARE `Structure` objects, using internal methods which call `pymatgen`'s IO functionality.

You can run it simply by calling the function on a file like so:

```
from flare.dft_interface.vasp_util import md_trajectory_from_vasprun
trajectory = md_trajectory_from_vasprun('path-to-vasprun')
```

1.2.2.4 Step 3: Training your Gaussian Process

If you don't have a previously existing Vasprun, you can also use the one available in the test_files directory, which is `methanol_frames.json`. You can open it via the command

```
from json import loads
from flare.struc import Structure
with open('path-to-methanol-frames','r') as f:
        loaded_dicts = [loads(line) for line in f.readlines()]
trajectory = [Structure.from_dict(d) for d in loaded_dicts]
```

Our trajectory is a list of FLARE structures, each of which is decorated with forces.

Once you have your trajectory and your `GaussianProcess` which has not seen any data yet, you are ready to begin your training!

We will next import the dedicated `TrajectoryTrainer` class, which has a variety of useful tools to help train your `GaussianProcess`.

The Trajectory Trainer has a large number of arguments which can be passed to it in order to give you a fine degree of control over how your model is trained. Here, we will pass in the following:

- `frames`: A list of FLARE ‘‘structure‘‘s decorated with forces. Ultimately, these structures will be iterated over and will be used to train the model.
- `gp`: Our `GaussianProcess` object. The process of training will involve populating the training set with representative atomic environments and optimizing the hyperparameters via likelihood maximization to best explain the data.

Input arguments for training include:

- `rel_std_tolerance`: The noise variance heuristically describes the amount of variance in force predictions which cannot be explained by the model. Once optimized, it provides a natural length scale for the degree of uncertainty expected in force predictions. A high uncertainty on a force prediction indicates that the `AtomicEnvironment` used is significantly different from all of the ‘‘AtomicEnvironment‘‘s in the training set. The criteria for adding atoms to the training set therefore be defined with respect to the noise variance: if we denote the noise variance of the model as sig_n, stored at gp.hyps[-1] by convention, then the the cutoff value used will be `rel_std_tolerance * sig_n`. Here, we will set it to 3.
- `abs_std_tolerance`: The above value describes a cutoff uncertainty which is defined with respect to the data set. In some cases it may be desirable to have a stringent cutoff which is invariant to the hyperparameters, in which case, if the uncertainty on any force prediction rises above `abs_std_tolerance` the associated atom will be added to the training set. Here, we will set it to 0. If both are defined, the lower of the two will be used.

1.2.2.5 Pre-Training arguments

When the training set contains a low diversity of atomic configurations relative to what you expect to see at test time, the hyperparameters may not be representative; furthermore, the training process when using `rel_std_tolerance` will depend on the hyperparameters, so it is desirable to have a training set with a baseline number of ‘‘AtomicEnvironment‘‘s before commencing training.

Therefore, we provide a variety of arguments to ‘seed’ the training set before commencing the full iteration over all of the frames passed into the function. By default, all of the atoms in the seed frames will be added to the training set. This is acceptable for small molecules, but you may want to use a more selective subset of atoms for large unit cells.

For now, we will only show one argument to seed frames for simplicity.

- `pre_train_on_skips`: Slice the input frames via `frames[::pre_train_on_skips]`; use those frames as seed frames. For instance, if we used `pre_train_on_skips=5` then we would use every fifth frame in the trajectory as a seed frame.

```
from flare.gp_from_aimd import TrajectoryTrainer
TT = TrajectoryTrainer(frames=trajectory,
                       gp = gp,
                       rel_std_tolerance = 3,
                       abs_std_tolerance=0,
                       pre_train_on_skips=5)
```

After this, all you need to do is call the run method!

```
TT.run()
print("Done!")
```

The results, by default, will be stored in `gp_from_aimd.out`, as well as a variety of other output files. The resultant model will be stored in a `.json` file format which can be later loaded using the `GaussianProcess.from_dict()` method.

Each frame will output the mae per species, which can be helpful for diagnosing if an individual species will be problematic (for example, you may find that an organic adsorbate on a metallic surface has a higher error, requiring more representative data for the dataset).

1.2.3 On-the-fly aluminum potential

Here we give an example of running OTF (on-the-fly) training with QE (Quantum Espresso) and NVE ensemble. We use our unit test file as illustration (`test_OTF_qe.py`)

1.2.3.1 Step 1: Set up a GP Model

Let’s start up with the GP model with three-body kernel function. (See *kernels.py* (single component) or *mc_simple.py* (multi-component) for more options.)

```
# make gp model
hyps = np.array([0.1, 1, 0.01])
hyp_labels = ['Signal Std', 'Length Scale', 'Noise Std']
cutoffs = {'threebody':3.9}

gp = \
    GaussianProcess(kernels=['threebody'],
```

(continues on next page)

(continued from previous page)

```
                    hyps=hyps,
                    cutoffs=cutoffs,
                    hyp_labels=hyp_labels,
                    maxiter=50)
```

Some Explanation about the parameters:

- `kernels`: set to be the name list of kernel functions to use
  - Currently we have the choices of `twobody`, `threebody` and `manybody` kernel functions.
  - If multiple kernels are listed, the resulted kernel is simply the summation of all listed kernels,
- `hyps`: the array of hyperparameters, whose names are shown in `hyp_labels`.
  - For two-body kernel function, an array of length 3 is needed, `hyps=[sigma_2, ls_2, sigma_n]`;
  - For three-body, `hyps=[sigma_3, ls_3, sigma_n]`;
  - For twobody plus threebody, `hyps=[sigma_2, ls_2, sigma_3, ls_3, sigma_n]`.
  - For twobody, threebody plus manybody, `hyps=[sigma_2, ls_2, sigma_3, ls_3, sigma_m, ls_m, sigma_n]`.
- `cutoffs`: a dictionary consists of corresponding cutoff values for each kernel. Usually we will set a larger one for two-body, and smaller one for threebody and manybody
- `maxiter`: set to constrain the number of steps in training hyperparameters.

Note:

1. See *GaussianProcess* for complete description of arguments of `GaussianProcess` class.
2. See *AdvancedHyperparametersSetUp* for more complicated hyper-parameters set up.

1.2.3.2 Step 2: Set up DFT Calculator

The next step is to set up DFT calculator, here we use QE (quantum espresso). Suppose we've prepared a QE input file in current directory `./pwscf.in`, and have set the environment variable `PWSCF_COMMAND` to the location of our QE's executable `pw.x`. Then we specify the input file and executable by `qe_input` and `dft_loc`.

```
# set up DFT calculator
qe_input = './pwscf.in' # quantum espresso input file
dft_loc = os.environ.get('PWSCF_COMMAND')
```

1.2.3.3 Step 3: Set up OTF MD Training Engine

Then we can set up our On-The-Fly (OTF) MD engine for training and simulation.

```
# set up OTF parameters
dt = 0.001                  # timestep (ps)
number_of_steps = 100       # number of steps
std_tolerance_factor = 1
max_atoms_added = 2
freeze_hyps = 3

otf = OTF(qe_input, dt, number_of_steps, gp, dft_loc,
          std_tolerance_factor, init_atoms=[0],
```

(continues on next page)

(continued from previous page)

```
            calculate_energy=True, output_name='al_otf_qe',
            freeze_hyps=freeze_hyps, skip=5,
            max_atoms_added=max_atoms_added)
```

Some Explanation about the parameters:

- `dt`: the time step in unit of *ps*
- `number_of_steps`: the number of steps that the MD is run
- `std_tolerance_factor`: the uncertainty threshold = std_tolerance_factor x hyps[-1]. In OTF training, when GP predicts uncertainty above the uncertainty threshold, it will call DFT
- `max_atoms_added`: constrain the number of atoms added to the training set after each DFT call
- `freeze_hyps`: stop training hyperparameters and fix them from the `freeze_hyps` th step. Usually set to a small number, because for large dataset the training will take long.
- `init_atoms`: list of atoms to be added in the first DFT call. Because there's no uncertainty predicted in the initial DFT call, so there's no selection rule to pick up "maximully uncertain" atoms into the training set, we have to specify which atoms to pick up by this variable.
- `calculate_energy`: if `True`, the local energy on each atom will be calculated
- `output_name`: the name of the logfile
- `skip`: record/dump the information every `skip` steps.

1.2.3.4 Step 4: Launch the OTF Training

Finally, let's run it!

```
# run OTF MD
otf.run()
```

After OTF training is finished, we can check log file `al_otf_qe.out` for all the information dumped. This output file can be parsed using our `otf_parser.py` module, which we will give an introduction later.

1.2.4 On-the-fly training using ASE

This is a quick introduction of how to set up our ASE-OTF interface to train a force field. We will train a force field model for diamond. To run the on-the-fly training, we will need to

1. Create a supercell with ASE Atoms object
2. Set up FLARE ASE calculator, including the kernel functions, hyperparameters, cutoffs for Gaussian process, and mapping parameters (if Mapped Gaussian Process is used)
3. Set up DFT ASE calculator. Here we will give an example of Quantum Espresso
4. Set up on-the-fly training with ASE MD engine

Please make sure you are using the LATEST FLARE code in our master branch.

1.2.4.1 Step 1: Set up supercell with ASE

Here we create a 2x1x1 supercell with lattice constant 3.855, and randomly perturb the positions of the atoms, so that they will start MD with non-zero forces.

[1]:
```
import numpy as np
from ase import units
from ase.spacegroup import crystal
from ase.build import bulk

np.random.seed(12345)

a = 3.52678
super_cell = bulk('C', 'diamond', a=a, cubic=True)
```

1.2.4.2 Step 2: Set up FLARE calculator

Now let's set up our Gaussian process model in the same way as introduced before

[2]:
```
from flare.gp import GaussianProcess
from flare.utils.parameter_helper import ParameterHelper

# set up GP hyperparameters
kernels = ['twobody', 'threebody'] # use 2+3 body kernel
parameters = {'cutoff_twobody': 5.0,
              'cutoff_threebody': 3.5}
pm = ParameterHelper(
    kernels = kernels,
    random = True,
    parameters=parameters
)

hm = pm.as_dict()
hyps = hm['hyps']
cut = hm['cutoffs']
print('hyps', hyps)

gp_model = GaussianProcess(
    kernels = kernels,
    component = 'sc', # single-component. For multi-comp, use 'mc'
    hyps = hyps,
    cutoffs = cut,
    hyp_labels = ['sig2','ls2','sig3','ls3','noise'],
    opt_algorithm = 'L-BFGS-B',
    n_cpus = 1
)
```

```
twobody0 cutoff is not define. it's going to use the universal cutoff.
threebody0 cutoff is not define. it's going to use the universal cutoff.
```

```
hyps [0.92961609 0.31637555 0.18391881 0.20456028 0.05      ]
```

Optional

If you want to use Mapped Gaussian Process (MGP), then set up MGP as follows

[3]:
```
from flare.mgp import MappedGaussianProcess

grid_params = {'twobody':   {'grid_num': [64]},
               'threebody': {'grid_num': [16, 16, 16]}}
```

(continues on next page)

(continued from previous page)

```
mgp_model = MappedGaussianProcess(grid_params,
                                  unique_species = [6],
                                  n_cpus = 1,
                                  map_force = False,
                                  mean_only = False)
```

Now let's set up FLARE's ASE calculator. If you want to use MGP model, then set `use_mapping = True` and `mgp_model = mgp_model` below.

[4]:
```
from flare.ase.calculator import FLARE_Calculator

flare_calculator = FLARE_Calculator(gp_model,
                                    par = True,
                                    mgp_model = None,
                                    use_mapping = False)

super_cell.set_calculator(flare_calculator)
```

1.2.4.3 Step 3: Set up DFT calculator

For DFT calculator, you can use any calculator provided by ASE, e.g. Quantum Espresso (QE), VASP, etc.

For a quick illustration of our interface, we use the Lennard-Jones (LJ) potential as an example.

[5]:
```
from ase.calculators.lj import LennardJones
lj_calc = LennardJones()
```

Optional: alternatively, set up Quantum Espresso calculator

We also give the code below for setting up the ASE quantum espresso calculator, following the instruction on ASE website.

First, we need to set up our environment variable `ASE_ESPRESSO_COMMAND` to our QE executable, so that ASE can find this calculator. Then set up our input parameters of QE and create an ASE calculator

[6]:
```
import os
from ase.calculators.espresso import Espresso

# ---------------- set up executable ----------------
label = 'C'
input_file = label+'.pwi'
output_file = label+'.pwo'
no_cpus = 32
npool = 32
pw_loc = 'path/to/pw.x'

# serial
os.environ['ASE_ESPRESSO_COMMAND'] = f'{pw_loc} < {input_file} > {output_file}'

## parallel qe using mpirun
# os.environ['ASE_ESPRESSO_COMMAND'] = f'mpirun -np {no_cpus} {pw_loc} -npool {npool}
↪< {input_file} > {output_file}'
```

(continues on next page)

(continued from previous page)

```
## parallel qe using srun (for slurm system)
# os.environ['ASE_ESPRESSO_COMMAND'] = 'srun -n {no_cpus} --mpi=pmi2 {pw_loc} -npool
↪{npool} < {input_file} > {output_file}'


# -------------- set up input parameters --------------
input_data = {'control':   {'prefix': label,
                            'pseudo_dir': './',
                            'outdir': './out',
                            'calculation': 'scf'},
              'system':    {'ibrav': 0,
                            'ecutwfc': 60,
                            'ecutrho': 360},
              'electrons': {'conv_thr': 1.0e-9,
                            'electron_maxstep': 100,
                            'mixing_beta': 0.7}}

# ----------------  pseudo-potentials -----------------
ion_pseudo = {'C': 'C.pz-rrkjus.UPF'}

# -------------- create ASE calculator ----------------
dft_calc = Espresso(pseudopotentials=ion_pseudo, label=label,
                    tstress=True, tprnfor=True, nosym=True,
                    input_data=input_data, kpts=(8, 8, 8))
```

1.2.4.4 Step 4: Set up On-The-Fly MD engine

Finally, our OTF is compatible with 5 MD engines that ASE supports: VelocityVerlet, NVTBerendsen, NPTBerendsen, NPT and Langevin. We can choose any of them, and set up the parameters based on ASE requirements. After everything is set up, we can run the on-the-fly training.

Note: Currently, only VelocityVerlet is tested on real system, NPT may have issue with pressure and stress.

Set up ASE_OTF training engine: 1. Initialize the velocities of atoms as 500K 2. Set up MD arguments as a dictionary based on ASE MD parameters. For VelocityVerlet, we don't need to set up extra parameters.

```
E.g. for NVTBerendsen, we can set `md_kwargs = {'temperature': 500, 'taut': 0.5e3 *␣
↪units.fs}`
```

[7]:
```
from ase import units
from ase.md.velocitydistribution import (MaxwellBoltzmannDistribution,
                                         Stationary, ZeroRotation)

temperature = 500
MaxwellBoltzmannDistribution(super_cell, temperature * units.kB)
Stationary(super_cell)  # zero linear momentum
ZeroRotation(super_cell)  # zero angular momentum

md_engine = 'VelocityVerlet'
md_kwargs = {}
```

3. Set up parameters for On-The-Fly (OTF) training. The descriptions of the parameters are in ASE OTF module.
4. Set up the ASE_OTF training engine, and run
5. Check `otf.out` after the training is done.

[8]:
```
from flare.ase.otf import ASE_OTF

otf_params = {'init_atoms': [0, 1, 2, 3],
              'output_name': 'otf',
              'std_tolerance_factor': 2,
              'max_atoms_added' : 4,
              'freeze_hyps': 10}

test_otf = ASE_OTF(super_cell,
                   timestep = 1 * units.fs,
                   number_of_steps = 3,
                   dft_calc = lj_calc,
                   md_engine = md_engine,
                   md_kwargs = md_kwargs,
                   **otf_params)

test_otf.run()
```

```
INFO:otflog:2020-06-10 14:53:00.574573
INFO:otflog:
GaussianProcess Object
Number of cpu cores: 1
Kernel: ['twobody', 'threebody']
Training points: 0
Cutoffs: {'twobody': 5.0, 'threebody': 3.5}
Model Likelihood: None
Number of hyperparameters: 5
Hyperparameters_array: [0.92961609 0.31637555 0.18391881 0.20456028 0.05      ]
Hyperparameters:
sig2: 0.9296160928171479
ls2: 0.3163755545817859
sig3: 0.18391881167709445
ls3: 0.2045602785530397
noise: 0.05
Hyps_mask train_noise: True
Hyps_mask nspecie: 1
Hyps_mask twobody_start: 0
Hyps_mask ntwobody: 1
Hyps_mask threebody_start: 2
Hyps_mask nthreebody: 1
Hyps_mask kernels: ['twobody', 'threebody']
Hyps_mask cutoffs: {'twobody': 5.0, 'threebody': 3.5}

uncertainty tolerance: 2 times noise hyperparameter
timestep (ps): 0.09822694788464063
number of frames: 3
number of atoms: 8
system species: {'C'}
periodic cell:
[[3.52678 0.      0.     ]
 [0.      3.52678 0.     ]
 [0.      0.      3.52678]]

previous positions (A):
C        0.0000    0.0000    0.0000
C        0.8817    0.8817    0.8817
C        0.0000    1.7634    1.7634
C        0.8817    2.6451    2.6451
```

(continues on next page)

(continued from previous page)

```
C         1.7634    0.0000    1.7634
C         2.6451    0.8817    2.6451
C         1.7634    1.7634    0.0000
C         2.6451    2.6451    0.8817
-------------------------------------------------------------------------------

INFO:otflog:
Calling DFT...

INFO:otflog:DFT run complete.
INFO:otflog:number of DFT calls: 1
INFO:otflog:wall time from start: 0.03 s
INFO:otflog:Adding atom [0, 1, 2, 3] to the training set.
INFO:otflog:Uncertainty: [0. 0. 0.]
INFO:otflog:
GP hyperparameters:
INFO:otflog:Hyp0 : sig2 = 0.9296
INFO:otflog:Hyp1 : ls2 = 0.3164
INFO:otflog:Hyp2 : sig3 = 0.1839
INFO:otflog:Hyp3 : ls3 = 0.2046
INFO:otflog:Hyp4 : noise = 0.0010
INFO:otflog:likelihood: 71.8658
INFO:otflog:likelihood gradient: [-1.79487637e-06 -6.53005436e-06 -8.57610391e-06 -1.
↪76345959e-05
 -1.19999904e+04]
INFO:otflog:wall time from start: 7.23 s
INFO:otflog:
*-Frame: 0
Simulation Time: 0.0 ps
El            Position (A)                    DFT Force (ev/A)                    Std.␣
↪Dev (ev/A)                  Velocities (A/ps)
C         0.0000    0.0000    0.0000         0.0000    0.0000    0.0000         0.0000  ␣
↪ 0.0000    0.0000         0.0417    0.0382    0.0035
C         0.8817    0.8817    0.8817         0.0000    0.0000    0.0000         0.0000  ␣
↪ 0.0000    0.0000        -0.0135    0.0222    0.0320
C         0.0000    1.7634    1.7634         0.0000    0.0000   -0.0000         0.0000  ␣
↪ 0.0000    0.0000         0.0204   -0.0705    0.0194
C         0.8817    2.6451    2.6451        -0.0000   -0.0000   -0.0000         0.0000  ␣
↪ 0.0000    0.0000        -0.0013    0.0346    0.0278
C         1.7634    0.0000    1.7634         0.0000    0.0000   -0.0000         0.0000  ␣
↪ 0.0000    0.0000        -0.0676   -0.0129    0.0242
C         2.6451    0.8817    2.6451        -0.0000   -0.0000   -0.0000         0.0000  ␣
↪ 0.0000    0.0000        -0.0027   -0.0121   -0.0340
C         1.7634    1.7634    0.0000         0.0000   -0.0000    0.0000         0.0000  ␣
↪ 0.0000    0.0000         0.0658   -0.0278   -0.0497
C         2.6451    2.6451    0.8817        -0.0000   -0.0000    0.0000         0.0000  ␣
↪ 0.0000    0.0000        -0.0427    0.0283   -0.0231

temperature: 199.00 K
kinetic energy: 0.180060 eV

INFO:otflog:wall time from start: 7.23 s
INFO:otflog:-------------------------------------------------------------------------
↪------
-Frame: 1
Simulation Time: 0.0982 ps
El            Position (A)                    GP Force (ev/A)                     Std.␣
↪Dev (ev/A)                  Velocities (A/ps)
```

(continues on next page)

(continued from previous page)

```
C         0.0082    0.0075    0.0007       -0.0000   -0.0000   -0.0000       16.1332
↪16.9877    6.7169        0.0417    0.0382    0.0035
C         0.8790    0.8861    0.8880        0.0000   -0.0000   -0.0000       15.3645
↪ 9.5079   17.2434       -0.0135    0.0222    0.0320
C         0.0040    1.7495    1.7672       -0.0000    0.0000   -0.0000       10.8208
↪37.7660    9.7448        0.0204   -0.0705    0.0194
C         0.8814    2.6519    2.6505       -0.0000   -0.0000    0.0000        9.5129
↪20.9930    4.7440       -0.0013    0.0346    0.0278
C         1.7501   -0.0025    1.7682        0.0000    0.0000   -0.0000       23.4305
↪10.7241   12.5271       -0.0676   -0.0129    0.0242
C         2.6446    0.8793    2.6384        0.0000   -0.0000    0.0000        3.1513
↪13.4187    7.3315       -0.0027   -0.0121   -0.0340
C         1.7763    1.7579   -0.0098       -0.0000    0.0000    0.0000       20.5584
↪ 9.0554   17.2228        0.0658   -0.0278   -0.0497
C         2.6367    2.6506    0.8772        0.0000   -0.0000    0.0000       17.8540
↪ 7.7304   12.6641       -0.0427    0.0283   -0.0231

temperature: 199.00 K
kinetic energy: 0.180060 eV

INFO:otflog:wall time from start: 9.65 s
INFO:otflog:
Calling DFT...

INFO:otflog:DFT run complete.
INFO:otflog:number of DFT calls: 2
INFO:otflog:wall time from start: 9.67 s
INFO:otflog:
*-Frame: 1
Simulation Time: 0.0982 ps
El            Position (A)                    DFT Force (ev/A)                  Std.
↪Dev (ev/A)                  Velocities (A/ps)
C         0.0164    0.0150    0.0013        0.0372   -0.0001   -0.0205       16.1332
↪16.9877    6.7169        0.0835    0.0764    0.0069
C         0.8763    0.8904    0.8943       -0.0669   -0.0327    0.0578       15.3645
↪ 9.5079   17.2434       -0.0274    0.0442    0.0641
C         0.0080    1.7356    1.7710        0.0322   -0.1194    0.0093       10.8208
↪37.7660    9.7448        0.0409   -0.1414    0.0388
C         0.8812    2.6587    2.6560        0.0353    0.0737   -0.0165        9.5129
↪20.9930    4.7440       -0.0025    0.0695    0.0555
C         1.7368   -0.0051    1.7729       -0.0330    0.0001    0.0250       23.4305
↪10.7241   12.5271       -0.1353   -0.0259    0.0486
C         2.6440    0.8770    2.6317       -0.0014    0.0643    0.0114        3.1513
↪13.4187    7.3315       -0.0053   -0.0238   -0.0680
C         1.7893    1.7525   -0.0195        0.0479    0.0129   -0.0207       20.5584
↪ 9.0554   17.2228        0.1317   -0.0555   -0.0994
C         2.6283    2.6562    0.8726       -0.0513    0.0013   -0.0459       17.8540
↪ 7.7304   12.6641       -0.0856    0.0565   -0.0465

temperature: 798.57 K
kinetic energy: 0.722563 eV

INFO:otflog:wall time from start: 9.68 s
INFO:otflog:mean absolute error: 0.0340 eV/A
INFO:otflog:mean absolute dft component: 0.0340 eV/A
INFO:otflog:Adding atom [6, 3, 4, 2] to the training set.
INFO:otflog:Uncertainty: [20.55839508  9.05540846 17.22284583]
```

(continues on next page)

(continued from previous page)

```
INFO:otflog:
GP hyperparameters:
INFO:otflog:Hyp0 : sig2 = 0.7038
INFO:otflog:Hyp1 : ls2 = 2.0405
INFO:otflog:Hyp2 : sig3 = 0.0000
INFO:otflog:Hyp3 : ls3 = 9.6547
INFO:otflog:Hyp4 : noise = 0.0010
INFO:otflog:likelihood: 122.4930
INFO:otflog:likelihood gradient: [-1.34065483e+00 -1.79554908e-01 -4.94110742e-02  1.
↪54534584e-10
 -1.82026091e+04]
INFO:otflog:wall time from start: 30.46 s
INFO:otflog:-----------------------------------------------------------------------
↪------
-Frame: 2
Simulation Time: 0.196 ps
El            Position (A)                    GP Force (ev/A)                    Std.␣
↪Dev (ev/A)                  Velocities (A/ps)
C        0.0247    0.0225    0.0020        0.0748   -0.0003   -0.0400        0.0008  ␣
↪ 0.0015    0.0008        0.0000    0.0000    0.0000
C        0.8735    0.8947    0.9007       -0.1357   -0.0645    0.1173        0.0014  ␣
↪ 0.0015    0.0010        0.0000    0.0000    0.0000
C        0.0121    1.7215    1.7748        0.0632   -0.2385    0.0151        0.0010  ␣
↪ 0.0015    0.0016        0.0000    0.0000    0.0000
C        0.8810    2.6657    2.6614        0.0692    0.1497   -0.0328        0.0011  ␣
↪ 0.0013    0.0010        0.0000    0.0000    0.0000
C        1.7235   -0.0076    1.7778       -0.0661   -0.0006    0.0515        0.0015  ␣
↪ 0.0013    0.0008        0.0000    0.0000    0.0000
C        2.6435    0.8748    2.6251       -0.0019    0.1297    0.0235        0.0009  ␣
↪ 0.0017    0.0010        0.0000    0.0000    0.0000
C        1.8023    1.7471   -0.0293        0.0980    0.0221   -0.0451        0.0015  ␣
↪ 0.0018    0.0012        0.0000    0.0000    0.0000
C        2.6197    2.6617    0.8679       -0.1015    0.0024   -0.0895        0.0013  ␣
↪ 0.0012    0.0012        0.0000    0.0000    0.0000

temperature: 0.00 K
kinetic energy: 0.000000 eV

INFO:otflog:wall time from start: 33.38 s
INFO:otflog:--------------------
INFO:otflog:Run complete.
```

1.2.5 After Training

After the on-the-fly training is complete, we can play with the force field we obtained. We are going to do the following things:

1. Parse the on-the-fly training trajectory to collect training data
2. Reconstruct the GP model from the training trajectory
3. Build up Mapped GP (MGP) for accelerated force field, and save coefficient file for LAMMPS
4. Use LAMMPS to run fast simulation using MGP pair style

1.2.5.1 Parse OTF log file

After the on-the-fly training is complete, we have a log file and can use the `otf_parser` module to parse the trajectory.

[3]:
```
import numpy as np
from flare import otf_parser

logdir = '../../../tests/test_files'
file_name = f'{logdir}/AgI_snippet.out'
hyp_no = 2 # use the hyperparameters from the 2nd training step
otf_object = otf_parser.OtfAnalysis(file_name)
```

1.2.5.2 Construct GP model from log file

We can reconstruct GP model from the parsed log file (the on-the-fly training trajectory). Here we build up the GP model with 2+3 body kernel from the on-the-fly log file.

[4]:
```
gp_model = otf_object.make_gp(hyp_no=hyp_no)
gp_model.parallel = True
gp_model.hyp_labels = ['sig2', 'ls2', 'sig3', 'ls3', 'noise']

# write model to a binary file
gp_model.write_model('AgI.gp', format='pickle')
```

```
Final name of the gp instance is default_gp_2
```

The last step `write_model` is to write this GP model into a binary file, so next time we can directly load the model from the pickle file as

[5]:
```
from flare.gp import GaussianProcess

gp_model = GaussianProcess.from_file('AgI.gp.pickle')
```

```
Final name of the gp instance is default_gp_2_2
```

1.2.5.3 Map the GP force field & Dump LAMMPS coefficient file

To use the trained force field with accelerated version MGP, or in LAMMPS, we need to build MGP from GP model. Since 2-body and 3-body are both included, we need to set up the number of grid points for 2-body and 3-body in `grid_params`. We build up energy mapping, thus set `map_force=False`. See MGP tutorial for more explanation of the MGP settings.

[6]:
```
from flare.mgp import MappedGaussianProcess

grid_params = {'twobody':   {'grid_num': [64]},
               'threebody': {'grid_num': [20, 20, 20]}}

data = gp_model.training_statistics
lammps_location = 'AgI_Molten_15.txt'

mgp_model = MappedGaussianProcess(grid_params, data['species'],
    map_force=False, lmp_file_name='AgI_Molten_15.txt', n_cpus=1)
mgp_model.build_map(gp_model)
```

The coefficient file for LAMMPS mgp pair_style is automatically saved once the mapping is done. Saved as `lmp_file_name`.

1.2.5.4 Run LAMMPS with MGP pair style

With the above coefficient file, we can run LAMMPS simulation with the mgp pair style. First download our mgp pair style files, compile your lammps executable with mgp pair style following our instruction.

1. One way to use it is running `lmp_executable < in.lammps > log.lammps` with the executable provided in our repository. When creating the input file, please note to set

```
newton off
pair_style mgp
pair_coeff * * <lmp_file_name> <chemical_symbols> yes/no yes/no
```

An example is using coefficient file `AgI_Molten_15.txt` for AgI system, with two-body (the 1st `yes`) together with three-body (the 2nd `yes`).

```
pair_coeff * * AgI_Molten_15.txt Ag I yes yes
```

Note: if you build force mapping (`map_force=True`) instead of energy mapping, please use

```
pair_style mgpf
```

2. The third way is to use the ASE LAMMPS interface

```
[ ]: from flare.ase.calculator import FLARE_Calculator

# get chemical symbols, masses etc.
species = gp_model.training_statistics['species']
specie_symbol_list = " ".join(species)
masses=[f"{i} {_Z_to_mass[_element_to_Z[species[i]]]}" for i in range(len(species))]

# set up input params
parameters = {'command': os.environ.get('lmp'), # set up executable for ASE
              'newton': 'off',
              'pair_style': 'mgp',
              'pair_coeff': [f'* * {lammps_location} {specie_symbol_list} yes yes'],
              'mass': masses}
files = [lammps_location]

# create ASE calc
lmp_calc = LAMMPS(label=f'tmp_AgI', keep_tmp_files=True, tmp_dir='./tmp/',
        parameters=parameters, files=files, specorder=species)
```

3. The second way to run LAMMPS is using our LAMMPS interface, please set the environment variable `$lmp` to the executable.

```
[ ]: from flare import struc
from flare.lammps import lammps_calculator

# lmp coef file is automatically written now every time MGP is constructed

# create test structure
species = otf_object.gp_species_list[-1]
positions = otf_object.position_list[-1]
forces = otf_object.force_list[-1]
```

(continues on next page)

(continued from previous page)

```
otf_cell = otf_object.header['cell']
structure = struc.Structure(otf_cell, species, positions)

atom_types = [1, 2]
atom_masses = [108, 127]
atom_species = [1, 2] * 27

# create data file
data_file_name = 'tmp.data'
data_text = lammps_calculator.lammps_dat(structure, atom_types,
                                         atom_masses, atom_species)
lammps_calculator.write_text(data_file_name, data_text)

# create lammps input
style_string = 'mgp'
coeff_string = '* * {} Ag I yes yes'.format(lammps_location)
lammps_executable = '$lmp'
dump_file_name = 'tmp.dump'
input_file_name = 'tmp.in'
output_file_name = 'tmp.out'
input_text = \
    lammps_calculator.generic_lammps_input(data_file_name, style_string,
                                           coeff_string, dump_file_name)
lammps_calculator.write_text(input_file_name, input_text)

lammps_calculator.run_lammps(lammps_executable, input_file_name,
                             output_file_name)

lammps_forces = lammps_calculator.lammps_parser(dump_file_name)
```

1.2.6 Compile LAMMPS with MGP Pair Style

Anders Johansson

1.2.6.1 MPI

For when you can't get Kokkos+OpenMP to work.

Compiling

```
cp lammps_plugin/pair_mgp.* /path/to/lammps/src
cd /path/to/lammps/src
make -j$(nproc) mpi
```

You can replace `mpi` with your favourite Makefile, e.g. `intel_cpu_intelmpi`, or use the CMake build system.

Running

```
mpirun /path/to/lammps/src/lmp_mpi -in in.lammps
```

as usual, but your LAMMPS script `in.lammps` needs to specify `newton off`.

1.2.6.2 MPI+OpenMP through Kokkos

For OpenMP parallelisation on your laptop or on one node, or for hybrid parallelisation on multiple nodes.

Compiling

```
cp lammps_plugin/pair_mgp*.* /path/to/lammps/src
cd /path/to/lammps/src
make yes-kokkos
make -j$(nproc) kokkos_omp
```

You can change the compiler flags etc. in `/path/to/lammps/src/MAKE/OPTIONS/Makefile.kokkos_omp`. As of writing, the pair style is not detected by CMake.

Running

With `newton on` in your LAMMPS script:

```
mpirun /path/to/lammps/src/lmp_kokkos_omp -k on t 4 -sf kk -package kokkos newton on␣
↪neigh half -in in.lammps
```

With `newton off` in your LAMMPS script:

```
mpirun /path/to/lammps/src/lmp_kokkos_omp -k on t 4 -sf kk -package kokkos newton off␣
↪neigh full -in in.lammps
```

Replace 4 with the desired number of threads *per MPI task*. Skip `mpirun` when running on one machine.

If you are running on multiple nodes on a cluster, you would typically launch one MPI task per node, and then set the number of threads equal to the number of cores (or hyperthreads) per node. A sample SLURM job script for 4 nodes, each with 48 cores, may look something like this:

```
#SBATCH --nodes=4
#SBATCH --ntasks-per-node=1
#SBATCH --cpus-per-task=48
mpirun -np $SLURM_NTASKS /path/to/lammps/src/lmp_kokkos_omp -k on t $SLURM_CPUS_PER_
↪TASK -sf kk -package kokkos newton off neigh full -in in.lammps
```

When running on Knight's Landing or other heavily hyperthreaded systems, you may want to try using more than one thread per CPU.

1.2.6.3 MPI+CUDA through Kokkos

For running on the GPU on your laptop, or for multiple GPUs on one or more nodes.

Compiling

```
cp lammps_plugin/pair_mgp*.* /path/to/lammps/src
cd /path/to/lammps/src
make yes-kokkos
make -j$(nproc) KOKKOS_ARCH=Volta70 kokkos_cuda_mpi
```

The KOKKOS_ARCH must be changed according to your GPU model. Volta70 is for V100, Pascal60 is for P100, etc.

You can change the compiler flags etc. in /path/to/lammps/src/MAKE/OPTIONS/Makefile.kokkos_cuda_mpi. As of writing, the pair style is not detected by CMake.

Running

With newton on in your LAMMPS script:

```
mpirun /path/to/lammps/src/lmp_kokkos_cuda_mpi -k on g 4 -sf kk -package kokkos␣
↪newton on neigh half -in in.lammps
```

With newton off in your LAMMPS script:

```
mpirun /path/to/lammps/src/lmp_kokkos_cuda_mpi -k on g 4 -sf kk -package kokkos␣
↪newton off neigh full -in in.lammps
```

Replace 4 with the desired number of GPUs *per node*, skip mpirun if you are using 1 GPU. The number of MPI tasks should be set equal to the total number of GPUs.

If you are running on multiple nodes on a cluster, you would typically launch one MPI task per GPU. A sample SLURM job script for 4 nodes, each with 2 GPUs, may look something like this:

```
#SBATCH --nodes=4
#SBATCH --ntasks-per-node=2
#SBATCH --cpus-per-task=1
#SBATCH --gpus-per-node=2
mpirun -np $SLURM_NTASKS /path/to/lammps/src/lmp_kokkos_cuda_mpi -k on g $SLURM_GPUS_
↪PER_NODE -sf kk -package kokkos newton off neigh full -in in.lammps
```

1.2.6.4 Notes on Newton (only relevant with Kokkos)

There are defaults which will kick in if you don't specify anything in the input script and/or skip the -package kokkos newton ... neigh ... flag. You can try these at your own risk, but it is safest to specify everything. See also the documentation.

newton on will probably be faster if you have a 2-body potential, otherwise the alternatives should give roughly equal performance.

1.3 Code Documentation

1.3.1 Gaussian Process Force Fields

1.3.1.1 Structures

The *Structure* object is a collection of atoms in a periodic box. The mandatory inputs are the cell vectors of the box and the chemical species and Cartesian coordinates of the atoms. The atoms are automatically folded back into the primary cell, so the input coordinates don't need to lie inside the box.

class Structure(*cell: np.array, species: List[str] or List[int], positions: np.array, mass_dict: dict = {}, prev_positions: ndarray = [], species_labels: List[str] = []*)

Contains information about a periodic structure of atoms, including the periodic cell boundaries, atomic species, and coordinates. Note that input positions are assumed to be Cartesian.

Parameters

- **cell** (*np.ndarray*) – 3x3 array whose rows are the Bravais lattice vectors of the cell.
- **species** (*List[int] or List[str]*) – List of atomic species, which can be given as either atomic numbers (integers) or chemical symbols (string of one or two characters, e.g. 'He' for Helium).
- **positions** (*np.array*) – Nx3 array of atomic coordinates in Angstrom.
- **mass_dict** (*dict, optional*) – Dictionary of atomic masses used in MD simulations, with species as keywords (either as integers or strings) and masses in amu as values. The format of the species keyword should match the format of the species input. For example, if the species are given as strings, mass_dict might take the form {'H': 1.0, 'He': 2.0}.
- **prev_positions** (*np.ndarray, optional*) – Nx3 array of previous atomic coordinates used in MD simulations. If not specified, prev_positions is set equal to positions.
- **species_labels** (*List[str], optional*) – List of chemical symbols used in the output file of on-the-fly runs. If not specified, species_labels is set equal to species.

coded_species

Python methods

The following are methods implemented in pure Python, i.e. independently of the underlying C++ backend, and are intended to improve the quality of life of the user.

flare.struc.**as_dict**(*self*) → dict
Returns structure as a dictionary; useful for serialization purposes.

Returns Dictionary version of current structure

Return type dict

flare.struc.**as_str**(*self*) → str
Returns string dictionary serialization cast as string.

Returns output of as_dict method cast as string

Return type str

flare.struc.**get_unique_species**(*species: List[Any]) -> (typing.List, typing.List[int]*)
Returns a list of the unique species passed in, and a list of integers indexing them.

Parameters **species** – Species to index uniquely

Returns List of the unique species, and integer indexes

flare.struc.**indices_of_specie**(*self, specie: Union[int, str]*) → List[int]
Return the indices of a given species within atoms of the structure.

Parameters **specie** – Element to target, can be string or integer

Returns The indices in the structure at which this element occurs

Return type List[str]

flare.struc.**to_pmg_structure**(*self*)
Returns FLARE structure as a pymatgen structure.

Returns Pymatgen structure corresponding to current FLARE structure

flare.struc.**to_xyz**(*self*, *extended_xyz: bool = True*, *print_stds: bool = False*, *print_forces: bool = False*, *print_max_stds: bool = False*, *write_file: str = ''*) → str

Convenience function which turns a structure into an extended .xyz file; useful for further input into visualization programs like VESTA or Ovito. Can be saved to an output file via write_file.

Parameters

- **print_stds** – Print the stds associated with the structure.
- **print_forces** –
- **extended_xyz** –
- **print_max_stds** –
- **write_file** –

Returns

1.3.1.2 Atomic Environments

The *AtomicEnvironment* object stores information about the local environment of an atom. *AtomicEnvironment* objects are inputs to the 2-, 3-, and 2+3-body kernels.

class flare.env.**AtomicEnvironment**(*structure: <sphinx.ext.autodoc.importer._MockObject object at 0x7fda8018f860>*, *atom: int*, *cutoffs*, *cutoffs_mask=None*)

Contains information about the local environment of an atom, including arrays of pair and triplet distances and the chemical species of atoms in the environment.

Parameters

- **structure** (*struc.Structure*) – Structure of atoms.
- **atom** (*int*) – Index of the atom in the structure.
- **cutoffs** – 2- and 3-body cutoff radii. 2-body if one cutoff is

given, 2+3-body if two are passed. :type cutoffs: np.ndarray :param cutoffs_mask: a dictionary to store multiple cutoffs if neede

it should be exactly the same as the hyps mask

The cutoffs_mask allows the user to define multiple cutoffs for different bonds, triples, and many body interaction. This dictionary should be consistent with the hyps_mask used in the GuassianProcess object.

- **specie_mask: 118-long integer array descirbing which elements belong to** like groups for determining which bond hyperparameters to use. For instance, [0,0,1,1,0 . . .] assigns H to group 0, He and Li to group 1, and Be to group 0 (the 0th register is ignored).
- **nspecie: Integer, number of different species groups (equal to number of** unique values in specie_mask).
- **ntwobody: Integer, number of different hyperparameter/cutoff sets to** associate with different 2-body pairings of atoms in groups defined in specie_mask.
- **twobody_mask: Array of length nspecie^2, which describes the cutoff to** associate with different pairings of species types. For example, if there are atoms of type 0 and 1, then twobody_mask defines which cutoff to use for parings [0-0, 0-1, 1-0, 1-1]: if we wanted cutoff0 for 0-0 parings and set 1 for 0-1 and 1-1 pairings, then we would make twobody_mask [0, 1, 1, 1].

- **twobody_cutoff_list: Array of length ntwobody, which stores the cutoff** used for different types of bonds defined in twobody_mask
- **ncut3b: Integer, number of different cutoffs sets to associate** with different 3-body pariings of atoms in groups defined in specie_mask.
- **cut3b_mask: Array of length nspecie^2, which describes the cutoff to** associate with different bond types in triplets. For example, in a triplet (C, O, H) , there are three cutoffs. Cutoffs for CH bond, CO bond and OH bond. If C and O are associate with atom group 1 in specie_mask and H are associate with group 0 in specie_mask, the cut3b_mask[1*nspecie+0] determines the C/O-H bond cutoff, and cut3b_mask[1*nspecie+1] determines the C-O bond cutoff. If we want the former one to use the 1st cutoff in threebody_cutoff_list and the later to use the 2nd cutoff in threebody_cutoff_list, the cut3b_mask should be [0, 0, 0, 1].
- **threebody_cutoff_list: Array of length ncut3b, which stores the cutoff** used for different types of bonds in triplets.
- **nmanybody: Integer, number of different cutoffs set to associate with** different coordination numbers.
- manybody_mask: Similar to twobody_mask and cut3b_mask.
- **manybody_cutoff_list: Array of length nmanybody, stores the cutoff used** for different many body terms

Examples can be found at the end of in tests/test_env.py

as_dict (*include_structure: bool = False*)
Returns Atomic Environment object as a dictionary for serialization purposes. Optional to not include the structure to avoid redundant information. :return:

static from_dict (*dictionary*)
Loads in atomic environment object from a dictionary which was serialized by the to_dict method.

Parameters **dictionary** – Dictionary describing atomic environment.

1.3.1.3 Kernels

Single-element Kernels

Single element 2-, 3-, and 2+3-body kernels. The kernel functions to choose:

- Two body:
  - two_body: force kernel
  - two_body_en: energy kernel
  - two_body_grad: gradient of kernel function
  - two_body_force_en: energy force kernel
- Three body:
  - three_body,
  - three_body_grad,
  - three_body_en,
  - three_body_force_en,
- Two plus three body:

  - two_plus_three_body,
  - two_plus_three_body_grad,
  - two_plus_three_en,
  - two_plus_three_force_en
- Two plus three plus many body:
  - two_plus_three_plus_many_body,
  - two_plus_three_plus_many_body_grad,
  - two_plus_three_plus_many_body_en,
  - two_plus_three_plus_many_body_force_en

Example:

```
>>> gp_model = GaussianProcess(kernel_name='2b',
                               <other arguments>)
```

flare.kernels.sc.**many_body**(*env1*, *env2*, *d1*, *d2*, *hyps*, *cutoffs*, *cutoff_func=<function quadratic_cutoff>*)

many-body single-element kernel between two forces.

Parameters

- **env1** (*AtomicEnvironment*) – First local environment.
- **env2** (*AtomicEnvironment*) – Second local environment.
- **d1** (*int*) – Force component of the first environment.
- **d2** (*int*) – Force component of the second environment.
- **hyps** (*np.ndarray*) – Hyperparameters of the kernel function (sig, ls).
- **cutoffs** (*np.ndarray*) – Two-element array containing the 2-, 3-, and many-body cutoffs.
- **cutoff_func** (*Callable*) – Cutoff function of the kernel.

Returns Value of the many-body force/force kernel.

Return type float

flare.kernels.sc.**many_body_en**(*env1*, *env2*, *hyps*, *cutoffs*, *cutoff_func=<function quadratic_cutoff>*)

many-body single-element kernel between two local energies.

Parameters

- **env1** (*AtomicEnvironment*) – First local environment.
- **env2** (*AtomicEnvironment*) – Second local environment.
- **hyps** (*np.ndarray*) – Hyperparameters of the kernel function (sig, ls).
- **cutoffs** (*np.ndarray*) – Two-element array containing the 2-, 3-, and many-body cutoffs.
- **cutoff_func** (*Callable*) – Cutoff function of the kernel.

Returns Value of the many-body energy/energy kernel.

Return type float

flare.kernels.sc.**many_body_en_jit**(*q_array_1*, *q_array_2*, *sig*, *ls*)

many-body single-element energy kernel between accelerated with Numba.

Parameters

- **q_array_1** (*np.ndarray*) – coordination number of the 1st local environment.
- **q_array_2** (*np.ndarray*) – coordination number of the 2nd local environment.
- **sig** (*float*) – many-body signal variance hyperparameter.
- **ls** (*float*) – many-body length scale hyperparameter.

Returns Value of the many-body kernel.

Return type float

flare.kernels.sc.**many_body_force_en**(*env1*, *env2*, *d1*, *hyps*, *cutoffs*, *cutoff_func=<function quadratic_cutoff>*)

many-body single-element kernel between two local energies.

Parameters

- **env1** (*AtomicEnvironment*) – First local environment.
- **env2** (*AtomicEnvironment*) – Second local environment.
- **hyps** (*np.ndarray*) – Hyperparameters of the kernel function (sig, ls).
- **cutoffs** (*np.ndarray*) – Two-element array containing the 2-, 3-, and many-body cutoffs.
- **cutoff_func** (*Callable*) – Cutoff function of the kernel.

Returns Value of the many-body force/energy kernel.

Return type float

flare.kernels.sc.**many_body_force_en_jit**(*q_array_1*, *q_array_2*, *q_neigh_array_1*, *q_neigh_grads*, *d1*, *sig*, *ls*)

many-body single-element kernel between force and energy components accelerated with Numba.

Parameters

- **bond_array_1** (*np.ndarray*) – many-body bond array of the first local environment.
- **bond_array_2** (*np.ndarray*) – many-body bond array of the second local environment.
- **neighbouring_dists_array_1** (*np.ndarray*) – matrix padded with zero values of distances of neighbours for the atoms in the first local environment.
- **num_neighbours_1** (*np.nsdarray*) – number of neighbours of each atom in the first local environment
- **d1** (*int*) – Force component of the first environment.
- **sig** (*float*) – many-body signal variance hyperparameter.
- **ls** (*float*) – many-body length scale hyperparameter.
- **r_cut** (*float*) – many-body cutoff radius.
- **cutoff_func** (*Callable*) – Cutoff function.

Returns Value of the many-body kernel.

Return type float

flare.kernels.sc.**many_body_grad**(*env1*, *env2*, *d1*, *d2*, *hyps*, *cutoffs*, *cutoff_func=<function quadratic_cutoff>*)

many-body single-element kernel between two force components and its gradient with respect to the hyperparameters.

Parameters

- **env1** (*AtomicEnvironment*) – First local environment.
- **env2** (*AtomicEnvironment*) – Second local environment.
- **d1** (*int*) – Force component of the first environment.
- **d2** (*int*) – Force component of the second environment.
- **hyps** (*np.ndarray*) – Hyperparameters of the kernel function (sig, ls).
- **cutoffs** (*np.ndarray*) – Two-element array containing the 2-, 3-, and many-body cutoffs.
- **cutoff_func** (*Callable*) – Cutoff function of the kernel.

Returns

Value of the many-body kernel and its gradient with respect to the hyperparameters.

Return type (float, np.ndarray)

flare.kernels.sc.**many_body_grad_jit**(*q_array_1*, *q_array_2*, *q_neigh_array_1*, *q_neigh_array_2*, *q_neigh_grads_1*, *q_neigh_grads_2*, *d1*, *d2*, *sig*, *ls*)

gradient of many-body single-element kernel between two force components w.r.t. the hyperparameters, accelerated with Numba.

Parameters

- **bond_array_1** (*np.ndarray*) – many-body bond array of the first local environment.
- **bond_array_2** (*np.ndarray*) – many-body bond array of the second local environment.
- **neighbouring_dists_array_1** (*np.ndarray*) – matrix padded with zero values of distances of neighbours for the atoms in the first local environment.
- **neighbouring_dists_array_2** (*np.ndarray*) – matrix padded with zero values of distances of neighbours for the atoms in the second local environment.
- **num_neighbours_1** (*np.nsdarray*) – number of neighbours of each atom in the first local environment
- **num_neighbours_2** (*np.ndarray*) – number of neighbours of each atom in the second local environment
- **d1** (*int*) – Force component of the first environment.
- **d2** (*int*) – Force component of the second environment.
- **sig** (*float*) – many-body signal variance hyperparameter.
- **ls** (*float*) – many-body length scale hyperparameter.
- **r_cut** (*float*) – many-body cutoff radius.
- **cutoff_func** (*Callable*) – Cutoff function.

Returns Value of the many-body kernel and its gradient w.r.t. sig and ls

Return type array

flare.kernels.sc.**many_body_jit**(*q_array_1*, *q_array_2*, *q_neigh_array_1*, *q_neigh_array_2*, *q_neigh_grads_1*, *q_neigh_grads_2*, *d1*, *d2*, *sig*, *ls*)

many-body single-element kernel between two force components accelerated with Numba.

Parameters

- **bond_array_1** (*np.ndarray*) – many-body bond array of the first local environment.

- **bond_array_2** (*np.ndarray*) – many-body bond array of the second local environment.
- **neighbouring_dists_array_1** (*np.ndarray*) – matrix padded with zero values of distances of neighbours for the atoms in the first local environment.
- **neighbouring_dists_array_2** (*np.ndarray*) – matrix padded with zero values of distances of neighbours for the atoms in the second local environment.
- **num_neighbours_1** (*np.nsdarray*) – number of neighbours of each atom in the first local environment
- **num_neighbours_2** (*np.ndarray*) – number of neighbours of each atom in the second local environment
- **d1** (*int*) – Force component of the first environment.
- **d2** (*int*) – Force component of the second environment.
- **sig** (*float*) – many-body signal variance hyperparameter.
- **ls** (*float*) – many-body length scale hyperparameter.
- **r_cut** (*float*) – many-body cutoff radius.
- **cutoff_func** (*Callable*) – Cutoff function.

Returns Value of the many-body kernel.

Return type float

flare.kernels.sc.**three_body**(*env1*, *env2*, *d1*, *d2*, *hyps*, *cutoffs*, *cutoff_func=<function quadratic_cutoff>*)

3-body single-element kernel between two force components.

Parameters

- **env1** (*AtomicEnvironment*) – First local environment.
- **env2** (*AtomicEnvironment*) – Second local environment.
- **d1** (*int*) – Force component of the first environment.
- **d2** (*int*) – Force component of the second environment.
- **hyps** (*np.ndarray*) – Hyperparameters of the kernel function (sig, ls).
- **cutoffs** (*np.ndarray*) – Two-element array containing the 2- and 3-body cutoffs.
- **cutoff_func** (*Callable*) – Cutoff function of the kernel.

Returns Value of the 3-body kernel.

Return type float

flare.kernels.sc.**three_body_en**(*env1*, *env2*, *hyps*, *cutoffs*, *cutoff_func=<function quadratic_cutoff>*)

3-body single-element kernel between two local energies.

Parameters

- **env1** (*AtomicEnvironment*) – First local environment.
- **env2** (*AtomicEnvironment*) – Second local environment.
- **hyps** (*np.ndarray*) – Hyperparameters of the kernel function (sig, ls).
- **cutoffs** (*np.ndarray*) – Two-element array containing the 2- and 3-body cutoffs.
- **cutoff_func** (*Callable*) – Cutoff function of the kernel.

Returns Value of the 3-body force/energy kernel.

Return type float

flare.kernels.sc.**three_body_en_jit**(*bond_array_1*, *bond_array_2*, *cross_bond_inds_1*, *cross_bond_inds_2*, *cross_bond_dists_1*, *cross_bond_dists_2*, *triplets_1*, *triplets_2*, *sig*, *ls*, *r_cut*, *cutoff_func*)

3-body single-element kernel between two local energies accelerated with Numba.

Parameters

- **bond_array_1** (*np.ndarray*) – 3-body bond array of the first local environment.
- **bond_array_2** (*np.ndarray*) – 3-body bond array of the second local environment.
- **cross_bond_inds_1** (*np.ndarray*) – Two dimensional array whose row m contains the indices of atoms n > m in the first local environment that are within a distance r_cut of both atom n and the central atom.
- **cross_bond_inds_2** (*np.ndarray*) – Two dimensional array whose row m contains the indices of atoms n > m in the second local environment that are within a distance r_cut of both atom n and the central atom.
- **cross_bond_dists_1** (*np.ndarray*) – Two dimensional array whose row m contains the distances from atom m of atoms n > m in the first local environment that are within a distance r_cut of both atom n and the central atom.
- **cross_bond_dists_2** (*np.ndarray*) – Two dimensional array whose row m contains the distances from atom m of atoms n > m in the second local environment that are within a distance r_cut of both atom n and the central atom.
- **triplets_1** (*np.ndarray*) – One dimensional array of integers whose entry m is the number of atoms in the first local environment that are within a distance r_cut of atom m.
- **triplets_2** (*np.ndarray*) – One dimensional array of integers whose entry m is the number of atoms in the second local environment that are within a distance r_cut of atom m.
- **sig** (*float*) – 3-body signal variance hyperparameter.
- **ls** (*float*) – 3-body length scale hyperparameter.
- **r_cut** (*float*) – 3-body cutoff radius.
- **cutoff_func** (*Callable*) – Cutoff function.

Returns Value of the 3-body local energy kernel.

Return type float

flare.kernels.sc.**three_body_force_en**(*env1*, *env2*, *d1*, *hyps*, *cutoffs*, *cutoff_func=<function quadratic_cutoff>*)

3-body single-element kernel between a force component and a local energy.

Parameters

- **env1** (*AtomicEnvironment*) – Local environment associated with the force component.
- **env2** (*AtomicEnvironment*) – Local environment associated with the local energy.
- **d1** (*int*) – Force component of the first environment.
- **hyps** (*np.ndarray*) – Hyperparameters of the kernel function (sig, ls).
- **cutoffs** (*np.ndarray*) – Two-element array containing the 2- and 3-body cutoffs.
- **cutoff_func** (*Callable*) – Cutoff function of the kernel.

Returns Value of the 3-body force/energy kernel.

Return type float

flare.kernels.sc.**three_body_force_en_jit**(*bond_array_1*, *bond_array_2*, *cross_bond_inds_1*, *cross_bond_inds_2*, *cross_bond_dists_1*, *cross_bond_dists_2*, *triplets_1*, *triplets_2*, *d1*, *sig*, *ls*, *r_cut*, *cutoff_func*)

3-body single-element kernel between a force component and a local energy accelerated with Numba.

Parameters

- **bond_array_1** (*np.ndarray*) – 3-body bond array of the first local environment.
- **bond_array_2** (*np.ndarray*) – 3-body bond array of the second local environment.
- **cross_bond_inds_1** (*np.ndarray*) – Two dimensional array whose row m contains the indices of atoms n > m in the first local environment that are within a distance r_cut of both atom n and the central atom.
- **cross_bond_inds_2** (*np.ndarray*) – Two dimensional array whose row m contains the indices of atoms n > m in the second local environment that are within a distance r_cut of both atom n and the central atom.
- **cross_bond_dists_1** (*np.ndarray*) – Two dimensional array whose row m contains the distances from atom m of atoms n > m in the first local environment that are within a distance r_cut of both atom n and the central atom.
- **cross_bond_dists_2** (*np.ndarray*) – Two dimensional array whose row m contains the distances from atom m of atoms n > m in the second local environment that are within a distance r_cut of both atom n and the central atom.
- **triplets_1** (*np.ndarray*) – One dimensional array of integers whose entry m is the number of atoms in the first local environment that are within a distance r_cut of atom m.
- **triplets_2** (*np.ndarray*) – One dimensional array of integers whose entry m is the number of atoms in the second local environment that are within a distance r_cut of atom m.
- **d1** (*int*) – Force component of the first environment (1=x, 2=y, 3=z).
- **sig** (*float*) – 3-body signal variance hyperparameter.
- **ls** (*float*) – 3-body length scale hyperparameter.
- **r_cut** (*float*) – 3-body cutoff radius.
- **cutoff_func** (*Callable*) – Cutoff function.

Returns Value of the 3-body force/energy kernel.

Return type float

flare.kernels.sc.**three_body_grad**(*env1*, *env2*, *d1*, *d2*, *hyps*, *cutoffs*, *cutoff_func=<function quadratic_cutoff>*)

3-body single-element kernel between two force components and its gradient with respect to the hyperparameters.

Parameters

- **env1** (*AtomicEnvironment*) – First local environment.
- **env2** (*AtomicEnvironment*) – Second local environment.
- **d1** (*int*) – Force component of the first environment.

- **d2** (*int*) – Force component of the second environment.
- **hyps** (*np.ndarray*) – Hyperparameters of the kernel function (sig, ls).
- **cutoffs** (*np.ndarray*) – Two-element array containing the 2- and 3-body cutoffs.
- **cutoff_func** (*Callable*) – Cutoff function of the kernel.

Returns

Value of the 3-body kernel and its gradient with respect to the hyperparameters.

Return type (float, np.ndarray)

flare.kernels.sc.**three_body_grad_jit**(*bond_array_1*, *bond_array_2*, *cross_bond_inds_1*, *cross_bond_inds_2*, *cross_bond_dists_1*, *cross_bond_dists_2*, *triplets_1*, *triplets_2*, *d1*, *d2*, *sig*, *ls*, *r_cut*, *cutoff_func*)

3-body single-element kernel between two force components and its gradient with respect to the hyperparameters.

Parameters

- **bond_array_1** (*np.ndarray*) – 3-body bond array of the first local environment.
- **bond_array_2** (*np.ndarray*) – 3-body bond array of the second local environment.
- **cross_bond_inds_1** (*np.ndarray*) – Two dimensional array whose row m contains the indices of atoms n > m in the first local environment that are within a distance r_cut of both atom n and the central atom.
- **cross_bond_inds_2** (*np.ndarray*) – Two dimensional array whose row m contains the indices of atoms n > m in the second local environment that are within a distance r_cut of both atom n and the central atom.
- **cross_bond_dists_1** (*np.ndarray*) – Two dimensional array whose row m contains the distances from atom m of atoms n > m in the first local environment that are within a distance r_cut of both atom n and the central atom.
- **cross_bond_dists_2** (*np.ndarray*) – Two dimensional array whose row m contains the distances from atom m of atoms n > m in the second local environment that are within a distance r_cut of both atom n and the central atom.
- **triplets_1** (*np.ndarray*) – One dimensional array of integers whose entry m is the number of atoms in the first local environment that are within a distance r_cut of atom m.
- **triplets_2** (*np.ndarray*) – One dimensional array of integers whose entry m is the number of atoms in the second local environment that are within a distance r_cut of atom m.
- **d1** (*int*) – Force component of the first environment.
- **d2** (*int*) – Force component of the second environment.
- **sig** (*float*) – 3-body signal variance hyperparameter.
- **ls** (*float*) – 3-body length scale hyperparameter.
- **r_cut** (*float*) – 3-body cutoff radius.
- **cutoff_func** (*Callable*) – Cutoff function.

Returns Value of the 3-body kernel and its gradient with respect to the hyperparameters.

Return type (float, float)

flare.kernels.sc.**three_body_jit**(*bond_array_1*, *bond_array_2*, *cross_bond_inds_1*, *cross_bond_inds_2*, *cross_bond_dists_1*, *cross_bond_dists_2*, *triplets_1*, *triplets_2*, *d1*, *d2*, *sig*, *ls*, *r_cut*, *cutoff_func*)

3-body single-element kernel between two force components accelerated with Numba.

Parameters

- **bond_array_1** (*np.ndarray*) – 3-body bond array of the first local environment.
- **bond_array_2** (*np.ndarray*) – 3-body bond array of the second local environment.
- **cross_bond_inds_1** (*np.ndarray*) – Two dimensional array whose row m contains the indices of atoms n > m in the first local environment that are within a distance r_cut of both atom n and the central atom.
- **cross_bond_inds_2** (*np.ndarray*) – Two dimensional array whose row m contains the indices of atoms n > m in the second local environment that are within a distance r_cut of both atom n and the central atom.
- **cross_bond_dists_1** (*np.ndarray*) – Two dimensional array whose row m contains the distances from atom m of atoms n > m in the first local environment that are within a distance r_cut of both atom n and the central atom.
- **cross_bond_dists_2** (*np.ndarray*) – Two dimensional array whose row m contains the distances from atom m of atoms n > m in the second local environment that are within a distance r_cut of both atom n and the central atom.
- **triplets_1** (*np.ndarray*) – One dimensional array of integers whose entry m is the number of atoms in the first local environment that are within a distance r_cut of atom m.
- **triplets_2** (*np.ndarray*) – One dimensional array of integers whose entry m is the number of atoms in the second local environment that are within a distance r_cut of atom m.
- **d1** (*int*) – Force component of the first environment.
- **d2** (*int*) – Force component of the second environment.
- **sig** (*float*) – 3-body signal variance hyperparameter.
- **ls** (*float*) – 3-body length scale hyperparameter.
- **r_cut** (*float*) – 3-body cutoff radius.
- **cutoff_func** (*Callable*) – Cutoff function.

Returns Value of the 3-body kernel.

Return type float

flare.kernels.sc.**two_body**(*env1*, *env2*, *d1*, *d2*, *hyps*, *cutoffs*, *cutoff_func=<function quadratic_cutoff>*)

2-body single-element kernel between two force components.

Parameters

- **env1** (*AtomicEnvironment*) – First local environment.
- **env2** (*AtomicEnvironment*) – Second local environment.
- **d1** (*int*) – Force component of the first environment.
- **d2** (*int*) – Force component of the second environment.
- **hyps** (*np.ndarray*) – Hyperparameters of the kernel function (sig, ls).
- **cutoffs** (*np.ndarray*) – One-element array containing the 2-body cutoff.

- **cutoff_func** (*Callable*) – Cutoff function of the kernel.

Returns Value of the 2-body kernel.

Return type float

flare.kernels.sc.**two_body_en**(*env1*, *env2*, *hyps*, *cutoffs*, *cutoff_func=<function quadratic_cutoff>*)

2-body single-element kernel between two local energies.

Parameters

- **env1** (*AtomicEnvironment*) – First local environment.
- **env2** (*AtomicEnvironment*) – Second local environment.
- **hyps** (*np.ndarray*) – Hyperparameters of the kernel function (sig, ls).
- **cutoffs** (*np.ndarray*) – One-element array containing the 2-body cutoff.
- **cutoff_func** (*Callable*) – Cutoff function of the kernel.

Returns Value of the 2-body force/energy kernel.

Return type float

flare.kernels.sc.**two_body_en_jit**(*bond_array_1*, *bond_array_2*, *sig*, *ls*, *r_cut*, *cutoff_func*)

2-body single-element kernel between two local energies accelerated with Numba.

Parameters

- **bond_array_1** (*np.ndarray*) – 2-body bond array of the first local environment.
- **bond_array_2** (*np.ndarray*) – 2-body bond array of the second local environment.
- **sig** (*float*) – 2-body signal variance hyperparameter.
- **ls** (*float*) – 2-body length scale hyperparameter.
- **r_cut** (*float*) – 2-body cutoff radius.
- **cutoff_func** (*Callable*) – Cutoff function.

Returns Value of the 2-body local energy kernel.

Return type float

flare.kernels.sc.**two_body_force_en**(*env1*, *env2*, *d1*, *hyps*, *cutoffs*, *cutoff_func=<function quadratic_cutoff>*)

2-body single-element kernel between a force component and a local energy.

Parameters

- **env1** (*AtomicEnvironment*) – Local environment associated with the force component.
- **env2** (*AtomicEnvironment*) – Local environment associated with the local energy.
- **d1** (*int*) – Force component of the first environment.
- **hyps** (*np.ndarray*) – Hyperparameters of the kernel function (sig, ls).
- **cutoffs** (*np.ndarray*) – One-element array containing the 2-body cutoff.
- **cutoff_func** (*Callable*) – Cutoff function of the kernel.

Returns Value of the 2-body force/energy kernel.

Return type float

flare.kernels.sc.**two_body_force_en_jit**(*bond_array_1*, *bond_array_2*, *d1*, *sig*, *ls*, *r_cut*, *cutoff_func*)

2-body single-element kernel between a force component and a local energy accelerated with Numba.

Parameters

- **bond_array_1** (*np.ndarray*) – 2-body bond array of the first local environment.
- **bond_array_2** (*np.ndarray*) – 2-body bond array of the second local environment.
- **d1** (*int*) – Force component of the first environment (1=x, 2=y, 3=z).
- **sig** (*float*) – 2-body signal variance hyperparameter.
- **ls** (*float*) – 2-body length scale hyperparameter.
- **r_cut** (*float*) – 2-body cutoff radius.
- **cutoff_func** (*Callable*) – Cutoff function.

Returns Value of the 2-body force/energy kernel.

Return type float

flare.kernels.sc.**two_body_grad**(*env1*, *env2*, *d1*, *d2*, *hyps*, *cutoffs*, *cutoff_func=<function quadratic_cutoff>*)

2-body single-element kernel between two force components and its gradient with respect to the hyperparameters.

Parameters

- **env1** (*AtomicEnvironment*) – First local environment.
- **env2** (*AtomicEnvironment*) – Second local environment.
- **d1** (*int*) – Force component of the first environment.
- **d2** (*int*) – Force component of the second environment.
- **hyps** (*np.ndarray*) – Hyperparameters of the kernel function (sig, ls).
- **cutoffs** (*np.ndarray*) – One-element array containing the 2-body cutoff.
- **cutoff_func** (*Callable*) – Cutoff function of the kernel.

Returns

Value of the 2-body kernel and its gradient with respect to the hyperparameters.

Return type (float, np.ndarray)

flare.kernels.sc.**two_body_grad_jit**(*bond_array_1*, *bond_array_2*, *d1*, *d2*, *sig*, *ls*, *r_cut*, *cutoff_func*)

2-body single-element kernel between two force components and its gradient with respect to the hyperparameters.

Parameters

- **bond_array_1** (*np.ndarray*) – 2-body bond array of the first local environment.
- **bond_array_2** (*np.ndarray*) – 2-body bond array of the second local environment.
- **d1** (*int*) – Force component of the first environment (1=x, 2=y, 3=z).
- **d2** (*int*) – Force component of the second environment (1=x, 2=y, 3=z).
- **sig** (*float*) – 2-body signal variance hyperparameter.
- **ls** (*float*) – 2-body length scale hyperparameter.

- **r_cut** (*float*) – 2-body cutoff radius.
- **cutoff_func** (*Callable*) – Cutoff function.

Returns Value of the 2-body kernel and its gradient with respect to the hyperparameters.

Return type (float, float)

flare.kernels.sc.**two_body_jit**(*bond_array_1*, *bond_array_2*, *d1*, *d2*, *sig*, *ls*, *r_cut*, *cutoff_func*)

2-body single-element kernel between two force components accelerated with Numba.

Parameters

- **bond_array_1** (*np.ndarray*) – 2-body bond array of the first local environment.
- **bond_array_2** (*np.ndarray*) – 2-body bond array of the second local environment.
- **d1** (*int*) – Force component of the first environment (1=x, 2=y, 3=z).
- **d2** (*int*) – Force component of the second environment (1=x, 2=y, 3=z).
- **sig** (*float*) – 2-body signal variance hyperparameter.
- **ls** (*float*) – 2-body length scale hyperparameter.
- **r_cut** (*float*) – 2-body cutoff radius.
- **cutoff_func** (*Callable*) – Cutoff function.

Returns Value of the 2-body kernel.

Return type float

flare.kernels.sc.**two_plus_three_body**(*env1: flare.env.AtomicEnvironment*, *env2: flare.env.AtomicEnvironment*, *d1: int*, *d2: int*, *hyps*, *cutoffs*, *cutoff_func=<function quadratic_cutoff>*)

2+3-body single-element kernel between two force components.

Parameters

- **env1** (*AtomicEnvironment*) – First local environment.
- **env2** (*AtomicEnvironment*) – Second local environment.
- **d1** (*int*) – Force component of the first environment.
- **d2** (*int*) – Force component of the second environment.
- **hyps** (*np.ndarray*) – Hyperparameters of the kernel function (sig1, ls1, sig2, ls2, sig_n).
- **cutoffs** (*np.ndarray*) – Two-element array containing the 2- and 3-body cutoffs.
- **cutoff_func** (*Callable*) – Cutoff function of the kernel.

Returns Value of the 2+3-body kernel.

Return type float

flare.kernels.sc.**two_plus_three_body_grad**(*env1*, *env2*, *d1*, *d2*, *hyps*, *cutoffs*, *cutoff_func=<function quadratic_cutoff>*)

2+3-body single-element kernel between two force components and its gradient with respect to the hyperparameters.

Parameters

- **env1** (*AtomicEnvironment*) – First local environment.
- **env2** (*AtomicEnvironment*) – Second local environment.
- **d1** (*int*) – Force component of the first environment.

- **d2** (*int*) – Force component of the second environment.
- **hyps** (*np.ndarray*) – Hyperparameters of the kernel function (sig1, ls1, sig2, ls2, sig_n).
- **cutoffs** (*np.ndarray*) – Two-element array containing the 2- and 3-body cutoffs.
- **cutoff_func** (*Callable*) – Cutoff function of the kernel.

Returns

Value of the 2+3-body kernel and its gradient with respect to the hyperparameters.

Return type (float, np.ndarray)

flare.kernels.sc.**two_plus_three_en**(*env1*, *env2*, *hyps*, *cutoffs*, *cutoff_func=<function quadratic_cutoff>*)

2+3-body single-element kernel between two local energies.

Parameters

- **env1** (*AtomicEnvironment*) – First local environment.
- **env2** (*AtomicEnvironment*) – Second local environment.
- **hyps** (*np.ndarray*) – Hyperparameters of the kernel function (sig1, ls1, sig2, ls2).
- **cutoffs** (*np.ndarray*) – Two-element array containing the 2- and 3-body cutoffs.
- **cutoff_func** (*Callable*) – Cutoff function of the kernel.

Returns Value of the 2+3-body force/energy kernel.

Return type float

flare.kernels.sc.**two_plus_three_force_en**(*env1*, *env2*, *d1*, *hyps*, *cutoffs*, *cutoff_func=<function quadratic_cutoff>*)

2+3-body single-element kernel between a force component and a local energy.

Parameters

- **env1** (*AtomicEnvironment*) – Local environment associated with the force component.
- **env2** (*AtomicEnvironment*) – Local environment associated with the local energy.
- **d1** (*int*) – Force component of the first environment.
- **hyps** (*np.ndarray*) – Hyperparameters of the kernel function (sig1, ls1, sig2, ls2).
- **cutoffs** (*np.ndarray*) – Two-element array containing the 2- and 3-body cutoffs.
- **cutoff_func** (*Callable*) – Cutoff function of the kernel.

Returns Value of the 2+3-body force/energy kernel.

Return type float

flare.kernels.sc.**two_plus_three_plus_many_body**(*env1: flare.env.AtomicEnvironment*, *env2: flare.env.AtomicEnvironment*, *d1: int*, *d2: int*, *hyps*, *cutoffs*, *cutoff_func=<function quadratic_cutoff>*)

2+3-body single-element kernel between two force components.

Parameters

- **env1** (*AtomicEnvironment*) – First local environment.
- **env2** (*AtomicEnvironment*) – Second local environment.
- **d1** (*int*) – Force component of the first environment.

- **d2** (*int*) – Force component of the second environment.
- **hyps** (*np.ndarray*) – Hyperparameters of the kernel function (sig2, ls2, sig3, ls3, sigm, lsm, sig_n).
- **cutoffs** (*np.ndarray*) – Two-element array containing the 2- and 3-body cutoffs.
- **cutoff_func** (*Callable*) – Cutoff function of the kernel.

Returns Value of the 2+3+many-body kernel.

Return type float

flare.kernels.sc.**two_plus_three_plus_many_body_en**(*env1: flare.env.AtomicEnvironment*, *env2: flare.env.AtomicEnvironment*, *hyps*, *cutoffs*, *cutoff_func=<function quadratic_cutoff>*)

2+3+many-body single-element energy kernel.

Parameters

- **env1** (*AtomicEnvironment*) – First local environment.
- **env2** (*AtomicEnvironment*) – Second local environment.
- **hyps** (*np.ndarray*) – Hyperparameters of the kernel function (sig2, ls2, sig3, ls3, sigm, lsm, sig_n).
- **cutoffs** (*np.ndarray*) – Two-element array containing the 2- and 3-body cutoffs.
- **cutoff_func** (*Callable*) – Cutoff function of the kernel.

Returns Value of the 2+3+many-body kernel.

Return type float

flare.kernels.sc.**two_plus_three_plus_many_body_force_en**(*env1: flare.env.AtomicEnvironment*, *env2: flare.env.AtomicEnvironment*, *d1: int*, *hyps*, *cutoffs*, *cutoff_func=<function quadratic_cutoff>*)

2+3+many-body single-element kernel between two force and energy components.

Parameters

- **env1** (*AtomicEnvironment*) – First local environment.
- **env2** (*AtomicEnvironment*) – Second local environment.
- **d1** (*int*) – Force component of the first environment.
- **hyps** (*np.ndarray*) – Hyperparameters of the kernel function (sig2, ls2, sig3, ls3, sigm, lsm, sig_n).
- **cutoffs** (*np.ndarray*) – Two-element array containing the 2- and 3-body cutoffs.
- **cutoff_func** (*Callable*) – Cutoff function of the kernel.

Returns Value of the 2+3+many-body kernel.

Return type float

flare.kernels.sc.**two_plus_three_plus_many_body_grad**(*env1: flare.env.AtomicEnvironment*, *env2: flare.env.AtomicEnvironment*, *d1: int*, *d2: int*, *hyps*, *cutoffs*, *cutoff_func=<function quadratic_cutoff>*)

2+3+many-body single-element kernel between two force components.

Parameters

- **env1** (*AtomicEnvironment*) – First local environment.
- **env2** (*AtomicEnvironment*) – Second local environment.
- **d1** (*int*) – Force component of the first environment.
- **d2** (*int*) – Force component of the second environment.
- **hyps** (*np.ndarray*) – Hyperparameters of the kernel function (sig2, ls2, sig3, ls3, sigm, lsm, sig_n).
- **cutoffs** (*np.ndarray*) – Two-element array containing the 2- and 3-body cutoffs.
- **cutoff_func** (*Callable*) – Cutoff function of the kernel.

Returns Value of the 2+3+many-body kernel.

Return type float

Multi-element Kernels (simple)

Multi-element 2-, 3-, and 2+3-body kernels that restrict all signal variance hyperparameters to a single value.

flare.kernels.mc_simple.**many_body_mc**(*env1: flare.env.AtomicEnvironment*, *env2: flare.env.AtomicEnvironment*, *d1: int*, *d2: int*, *hyps: ndarray*, *cutoffs: ndarray*, *cutoff_func: Callable = <function quadratic_cutoff>*) → float

many-body multi-element kernel between two force components.

Parameters

- **env1** (*AtomicEnvironment*) – First local environment.
- **env2** (*AtomicEnvironment*) – Second local environment.
- **d1** (*int*) – Force component of the first environment.
- **d2** (*int*) – Force component of the second environment.
- **hyps** (*np.ndarray*) – Hyperparameters of the kernel function (sig, ls).
- **cutoffs** (*np.ndarray*) – Two-element array containing the 2- and 3-body cutoffs.
- **cutoff_func** (*Callable*) – Cutoff function of the kernel.

Returns Value of the 3-body kernel.

Return type float

flare.kernels.mc_simple.**many_body_mc_en**(*env1: flare.env.AtomicEnvironment*, *env2: flare.env.AtomicEnvironment*, *hyps: ndarray*, *cutoffs: ndarray*, *cutoff_func: Callable = <function quadratic_cutoff>*) → float

many-body multi-element kernel between two local energies.

Parameters

- **env1** (*AtomicEnvironment*) – First local environment.
- **env2** (*AtomicEnvironment*) – Second local environment.
- **hyps** (*np.ndarray*) – Hyperparameters of the kernel function (sig, ls).
- **cutoffs** (*np.ndarray*) – One-element array containing the 2-body cutoff.
- **cutoff_func** (*Callable*) – Cutoff function of the kernel.

Returns Value of the 2-body force/energy kernel.

Return type float

flare.kernels.mc_simple.**many_body_mc_en_jit**(*q_array_1*, *q_array_2*, *c1*, *c2*, *species1*, *species2*, *sig*, *ls*)

many-body many-element kernel between energy components accelerated with Numba.

Parameters

- **bond_array_1** (*np.ndarray*) – many-body bond array of the first local environment.
- **bond_array_2** (*np.ndarray*) – many-body bond array of the second local environment.
- **c1** (*int*) – atomic species of the central atom in env 1
- **c2** (*int*) – atomic species of the central atom in env 2
- **etypes1** (*np.ndarray*) – atomic species of atoms in env 1
- **etypes2** (*np.ndarray*) – atomic species of atoms in env 2
- **species1** (*np.ndarray*) – all the atomic species present in trajectory 1
- **species2** (*np.ndarray*) – all the atomic species present in trajectory 2
- **sig** (*float*) – many-body signal variance hyperparameter.
- **ls** (*float*) – many-body length scale hyperparameter.
- **r_cut** (*float*) – many-body cutoff radius.
- **cutoff_func** (*Callable*) – Cutoff function.

Returns Value of the many-body kernel.

Return type float

flare.kernels.mc_simple.**many_body_mc_force_en**(*env1*, *env2*, *d1*, *hyps*, *cutoffs*, *cutoff_func=<function quadratic_cutoff>*)

many-body single-element kernel between two local energies.

Parameters

- **env1** (*AtomicEnvironment*) – First local environment.
- **env2** (*AtomicEnvironment*) – Second local environment.
- **hyps** (*np.ndarray*) – Hyperparameters of the kernel function (sig, ls).
- **cutoffs** (*np.ndarray*) – Two-element array containing the 2-, 3-, and many-body cutoffs.

- **cutoff_func** (*Callable*) – Cutoff function of the kernel.

Returns Value of the many-body force/energy kernel.

Return type float

flare.kernels.mc_simple.**many_body_mc_force_en_jit**(*q_array_1*, *q_array_2*, *q_neigh_array_1*, *q_neigh_grads_1*, *c1*, *c2*, *etypes1*, *species1*, *species2*, *d1*, *sig*, *ls*)

many-body many-element kernel between force and energy components accelerated with Numba.

Parameters

- **c1** (*int*) – atomic species of the central atom in env 1
- **c2** (*int*) – atomic species of the central atom in env 2
- **etypes1** (*np.ndarray*) – atomic species of atoms in env 1
- **species1** (*np.ndarray*) – all the atomic species present in trajectory 1
- **species2** (*np.ndarray*) – all the atomic species present in trajectory 2
- **d1** (*int*) – Force component of the first environment.
- **sig** (*float*) – many-body signal variance hyperparameter.
- **ls** (*float*) – many-body length scale hyperparameter.

Returns Value of the many-body kernel.

Return type float

flare.kernels.mc_simple.**many_body_mc_grad**(*env1: flare.env.AtomicEnvironment*, *env2: flare.env.AtomicEnvironment*, *d1: int*, *d2: int*, *hyps: ndarray*, *cutoffs: ndarray*, *cutoff_func: Callable = <function quadratic_cutoff>*) → float

gradient manybody-body multi-element kernel between two force components.

flare.kernels.mc_simple.**many_body_mc_grad_jit**(*q_array_1*, *q_array_2*, *q_neigh_array_1*, *q_neigh_array_2*, *q_neigh_grads_1*, *q_neigh_grads_2*, *c1*, *c2*, *etypes1*, *etypes2*, *species1*, *species2*, *d1*, *d2*, *sig*, *ls*)

gradient of many-body multi-element kernel between two force components w.r.t. the hyperparameters, accelerated with Numba.

Parameters

- **bond_array_1** (*np.ndarray*) – many-body bond array of the first local environment.
- **bond_array_2** (*np.ndarray*) – many-body bond array of the second local environment.
- **neigh_dists_1** (*np.ndarray*) – matrix padded with zero values of distances of neighbours for the atoms in the first local environment.
- **neigh_dists_2** (*np.ndarray*) – matrix padded with zero values of distances of neighbours for the atoms in the second local environment.
- **num_neigh_1** (*np.ndarray*) – number of neighbours of each atom in the first local environment
- **num_neigh_2** (*np.ndarray*) – number of neighbours of each atom in the second local environment

- **c1** (*int*) – atomic species of the central atom in env 1
- **c2** (*int*) – atomic species of the central atom in env 2
- **etypes1** (*np.ndarray*) – atomic species of atoms in env 1
- **etypes2** (*np.ndarray*) – atomic species of atoms in env 2
- **etypes_neigh_1** (*np.ndarray*) – atomic species of atoms in the neighbourhoods of atoms in env 1
- **etypes_neigh_2** (*np.ndarray*) – atomic species of atoms in the neighbourhoods of atoms in env 2
- **species1** (*np.ndarray*) – all the atomic species present in trajectory 1
- **species2** (*np.ndarray*) – all the atomic species present in trajectory 2
- **d1** (*int*) – Force component of the first environment.
- **d2** (*int*) – Force component of the second environment.
- **sig** (*float*) – many-body signal variance hyperparameter.
- **ls** (*float*) – many-body length scale hyperparameter.
- **r_cut** (*float*) – many-body cutoff radius.
- **cutoff_func** (*Callable*) – Cutoff function.

Returns Value of the many-body kernel and its gradient w.r.t. sig and ls

Return type array

flare.kernels.mc_simple.**many_body_mc_jit**(*q_array_1*, *q_array_2*, *q_neigh_array_1*, *q_neigh_array_2*, *q_neigh_grads_1*, *q_neigh_grads_2*, *c1*, *c2*, *etypes1*, *etypes2*, *species1*, *species2*, *d1*, *d2*, *sig*, *ls*)

many-body multi-element kernel between two force components accelerated with Numba.

Parameters

- **bond_array_1** (*np.ndarray*) – many-body bond array of the first local environment.
- **bond_array_2** (*np.ndarray*) – many-body bond array of the second local environment.
- **neigh_dists_1** (*np.ndarray*) – matrix padded with zero values of distances of neighbours for the atoms in the first local environment.
- **neigh_dists_2** (*np.ndarray*) – matrix padded with zero values of distances of neighbours for the atoms in the second local environment.
- **num_neigh_1** (*np.ndarray*) – number of neighbours of each atom in the first local environment
- **num_neigh_2** (*np.ndarray*) – number of neighbours of each atom in the second local environment
- **c1** (*int*) – atomic species of the central atom in env 1
- **c2** (*int*) – atomic species of the central atom in env 2
- **etypes1** (*np.ndarray*) – atomic species of atoms in env 1
- **etypes2** (*np.ndarray*) – atomic species of atoms in env 2
- **etypes_neigh_1** (*np.ndarray*) – atomic species of atoms in the neighbourhoods of atoms in env 1

- **etypes_neigh_2** (*np.ndarray*) – atomic species of atoms in the neighbourhoods of atoms in env 2
- **species1** (*np.ndarray*) – all the atomic species present in trajectory 1
- **species2** (*np.ndarray*) – all the atomic species present in trajectory 2
- **d1** (*int*) – Force component of the first environment.
- **d2** (*int*) – Force component of the second environment.
- **sig** (*float*) – many-body signal variance hyperparameter.
- **ls** (*float*) – many-body length scale hyperparameter.
- **r_cut** (*float*) – many-body cutoff radius.
- **cutoff_func** (*Callable*) – Cutoff function.

Returns Value of the many-body kernel.

Return type float

flare.kernels.mc_simple.**three_body_mc**(*env1: flare.env.AtomicEnvironment*, *env2: flare.env.AtomicEnvironment*, *d1: int*, *d2: int*, *hyps: ndarray*, *cutoffs: ndarray*, *cutoff_func: Callable = <function quadratic_cutoff>*) → float

3-body multi-element kernel between two force components.

Parameters

- **env1** (*AtomicEnvironment*) – First local environment.
- **env2** (*AtomicEnvironment*) – Second local environment.
- **d1** (*int*) – Force component of the first environment.
- **d2** (*int*) – Force component of the second environment.
- **hyps** (*np.ndarray*) – Hyperparameters of the kernel function (sig, ls).
- **cutoffs** (*np.ndarray*) – Two-element array containing the 2- and 3-body cutoffs.
- **cutoff_func** (*Callable*) – Cutoff function of the kernel.

Returns Value of the 3-body kernel.

Return type float

flare.kernels.mc_simple.**three_body_mc_en**(*env1: flare.env.AtomicEnvironment*, *env2: flare.env.AtomicEnvironment*, *hyps: ndarray*, *cutoffs: ndarray*, *cutoff_func: Callable = <function quadratic_cutoff>*) → float

3-body multi-element kernel between two local energies.

Parameters

- **env1** (*AtomicEnvironment*) – First local environment.
- **env2** (*AtomicEnvironment*) – Second local environment.
- **hyps** (*np.ndarray*) – Hyperparameters of the kernel function (sig, ls).
- **cutoffs** (*np.ndarray*) – Two-element array containing the 2- and 3-body cutoffs.
- **cutoff_func** (*Callable*) – Cutoff function of the kernel.

Returns Value of the 3-body force/energy kernel.

Return type float

flare.kernels.mc_simple.**three_body_mc_en_jit**(*bond_array_1*, *c1*, *etypes1*, *bond_array_2*, *c2*, *etypes2*, *cross_bond_inds_1*, *cross_bond_inds_2*, *cross_bond_dists_1*, *cross_bond_dists_2*, *triplets_1*, *triplets_2*, *sig*, *ls*, *r_cut*, *cutoff_func*)

3-body multi-element kernel between two local energies accelerated with Numba.

Parameters

- **bond_array_1** (*np.ndarray*) – 3-body bond array of the first local environment.
- **c1** (*int*) – Species of the central atom of the first local environment.
- **etypes1** (*np.ndarray*) – Species of atoms in the first local environment.
- **bond_array_2** (*np.ndarray*) – 3-body bond array of the second local environment.
- **c2** (*int*) – Species of the central atom of the second local environment.
- **etypes2** (*np.ndarray*) – Species of atoms in the second local environment.
- **cross_bond_inds_1** (*np.ndarray*) – Two dimensional array whose row m contains the indices of atoms n > m in the first local environment that are within a distance r_cut of both atom n and the central atom.
- **cross_bond_inds_2** (*np.ndarray*) – Two dimensional array whose row m contains the indices of atoms n > m in the second local environment that are within a distance r_cut of both atom n and the central atom.
- **cross_bond_dists_1** (*np.ndarray*) – Two dimensional array whose row m contains the distances from atom m of atoms n > m in the first local environment that are within a distance r_cut of both atom n and the central atom.
- **cross_bond_dists_2** (*np.ndarray*) – Two dimensional array whose row m contains the distances from atom m of atoms n > m in the second local environment that are within a distance r_cut of both atom n and the central atom.
- **triplets_1** (*np.ndarray*) – One dimensional array of integers whose entry m is the number of atoms in the first local environment that are within a distance r_cut of atom m.
- **triplets_2** (*np.ndarray*) – One dimensional array of integers whose entry m is the number of atoms in the second local environment that are within a distance r_cut of atom m.
- **sig** (*float*) – 3-body signal variance hyperparameter.
- **ls** (*float*) – 3-body length scale hyperparameter.
- **r_cut** (*float*) – 3-body cutoff radius.
- **cutoff_func** (*Callable*) – Cutoff function.

Returns Value of the 3-body local energy kernel.

Return type float

flare.kernels.mc_simple.**three_body_mc_force_en**(*env1: flare.env.AtomicEnvironment*, *env2: flare.env.AtomicEnvironment*, *d1: int*, *hyps: ndarray*, *cutoffs: ndarray*, *cutoff_func: Callable = <function quadratic_cutoff>*) → float

3-body multi-element kernel between a force component and a local energy.

Parameters

- **env1** (*AtomicEnvironment*) – Local environment associated with the force component.
- **env2** (*AtomicEnvironment*) – Local environment associated with the local energy.
- **d1** (*int*) – Force component of the first environment.
- **hyps** (*np.ndarray*) – Hyperparameters of the kernel function (sig, ls).
- **cutoffs** (*np.ndarray*) – Two-element array containing the 2- and 3-body cutoffs.
- **cutoff_func** (*Callable*) – Cutoff function of the kernel.

Returns Value of the 3-body force/energy kernel.

Return type float

flare.kernels.mc_simple.**three_body_mc_force_en_jit**(*bond_array_1*, *c1*, *etypes1*, *bond_array_2*, *c2*, *etypes2*, *cross_bond_inds_1*, *cross_bond_inds_2*, *cross_bond_dists_1*, *cross_bond_dists_2*, *triplets_1*, *triplets_2*, *d1*, *sig*, *ls*, *r_cut*, *cutoff_func*)

3-body multi-element kernel between a force component and a local energy accelerated with Numba.

Parameters

- **bond_array_1** (*np.ndarray*) – 3-body bond array of the first local environment.
- **c1** (*int*) – Species of the central atom of the first local environment.
- **etypes1** (*np.ndarray*) – Species of atoms in the first local environment.
- **bond_array_2** (*np.ndarray*) – 3-body bond array of the second local environment.
- **c2** (*int*) – Species of the central atom of the second local environment.
- **etypes2** (*np.ndarray*) – Species of atoms in the second local environment.
- **cross_bond_inds_1** (*np.ndarray*) – Two dimensional array whose row m contains the indices of atoms n > m in the first local environment that are within a distance r_cut of both atom n and the central atom.
- **cross_bond_inds_2** (*np.ndarray*) – Two dimensional array whose row m contains the indices of atoms n > m in the second local environment that are within a distance r_cut of both atom n and the central atom.
- **cross_bond_dists_1** (*np.ndarray*) – Two dimensional array whose row m contains the distances from atom m of atoms n > m in the first local environment that are within a distance r_cut of both atom n and the central atom.
- **cross_bond_dists_2** (*np.ndarray*) – Two dimensional array whose row m contains the distances from atom m of atoms n > m in the second local environment that are within a distance r_cut of both atom n and the central atom.
- **triplets_1** (*np.ndarray*) – One dimensional array of integers whose entry m is the number of atoms in the first local environment that are within a distance r_cut of atom m.
- **triplets_2** (*np.ndarray*) – One dimensional array of integers whose entry m is the number of atoms in the second local environment that are within a distance r_cut of atom m.
- **d1** (*int*) – Force component of the first environment (1=x, 2=y, 3=z).
- **sig** (*float*) – 3-body signal variance hyperparameter.

- **ls** (*float*) – 3-body length scale hyperparameter.
- **r_cut** (*float*) – 3-body cutoff radius.
- **cutoff_func** (*Callable*) – Cutoff function.

Returns Value of the 3-body force/energy kernel.

Return type float

flare.kernels.mc_simple.**three_body_mc_grad**(*env1: flare.env.AtomicEnvironment*, *env2: flare.env.AtomicEnvironment*, *d1: int*, *d2: int*, *hyps: ndarray*, *cutoffs: ndarray*, *cutoff_func: Callable = <function quadratic_cutoff>) -> ('float', 'ndarray')*

3-body multi-element kernel between two force components and its gradient with respect to the hyperparameters.

Parameters

- **env1** (*AtomicEnvironment*) – First local environment.
- **env2** (*AtomicEnvironment*) – Second local environment.
- **d1** (*int*) – Force component of the first environment.
- **d2** (*int*) – Force component of the second environment.
- **hyps** (*np.ndarray*) – Hyperparameters of the kernel function (sig, ls).
- **cutoffs** (*np.ndarray*) – Two-element array containing the 2- and 3-body cutoffs.
- **cutoff_func** (*Callable*) – Cutoff function of the kernel.

Returns Value of the 3-body kernel and its gradient with respect to the hyperparameters.

Return type (float, np.ndarray)

flare.kernels.mc_simple.**three_body_mc_grad_jit**(*bond_array_1*, *c1*, *etypes1*, *bond_array_2*, *c2*, *etypes2*, *cross_bond_inds_1*, *cross_bond_inds_2*, *cross_bond_dists_1*, *cross_bond_dists_2*, *triplets_1*, *triplets_2*, *d1*, *d2*, *sig*, *ls*, *r_cut*, *cutoff_func*)

3-body multi-element kernel between two force components and its gradient with respect to the hyperparameters.

Parameters

- **bond_array_1** (*np.ndarray*) – 3-body bond array of the first local environment.
- **c1** (*int*) – Species of the central atom of the first local environment.
- **etypes1** (*np.ndarray*) – Species of atoms in the first local environment.
- **bond_array_2** (*np.ndarray*) – 3-body bond array of the second local environment.
- **c2** (*int*) – Species of the central atom of the second local environment.
- **etypes2** (*np.ndarray*) – Species of atoms in the second local environment.
- **cross_bond_inds_1** (*np.ndarray*) – Two dimensional array whose row m contains the indices of atoms n > m in the first local environment that are within a distance r_cut of both atom n and the central atom.

- **cross_bond_inds_2** (*np.ndarray*) – Two dimensional array whose row m contains the indices of atoms n > m in the second local environment that are within a distance r_cut of both atom n and the central atom.
- **cross_bond_dists_1** (*np.ndarray*) – Two dimensional array whose row m contains the distances from atom m of atoms n > m in the first local environment that are within a distance r_cut of both atom n and the central atom.
- **cross_bond_dists_2** (*np.ndarray*) – Two dimensional array whose row m contains the distances from atom m of atoms n > m in the second local environment that are within a distance r_cut of both atom n and the central atom.
- **triplets_1** (*np.ndarray*) – One dimensional array of integers whose entry m is the number of atoms in the first local environment that are within a distance r_cut of atom m.
- **triplets_2** (*np.ndarray*) – One dimensional array of integers whose entry m is the number of atoms in the second local environment that are within a distance r_cut of atom m.
- **d1** (*int*) – Force component of the first environment.
- **d2** (*int*) – Force component of the second environment.
- **sig** (*float*) – 3-body signal variance hyperparameter.
- **ls** (*float*) – 3-body length scale hyperparameter.
- **r_cut** (*float*) – 3-body cutoff radius.
- **cutoff_func** (*Callable*) – Cutoff function.

Returns Value of the 3-body kernel and its gradient with respect to the hyperparameters.

Return type (float, float)

flare.kernels.mc_simple.**three_body_mc_jit**(*bond_array_1*, *c1*, *etypes1*, *bond_array_2*, *c2*, *etypes2*, *cross_bond_inds_1*, *cross_bond_inds_2*, *cross_bond_dists_1*, *cross_bond_dists_2*, *triplets_1*, *triplets_2*, *d1*, *d2*, *sig*, *ls*, *r_cut*, *cutoff_func*)

3-body multi-element kernel between two force components accelerated with Numba.

Parameters

- **bond_array_1** (*np.ndarray*) – 3-body bond array of the first local environment.
- **c1** (*int*) – Species of the central atom of the first local environment.
- **etypes1** (*np.ndarray*) – Species of atoms in the first local environment.
- **bond_array_2** (*np.ndarray*) – 3-body bond array of the second local environment.
- **c2** (*int*) – Species of the central atom of the second local environment.
- **etypes2** (*np.ndarray*) – Species of atoms in the second local environment.
- **cross_bond_inds_1** (*np.ndarray*) – Two dimensional array whose row m contains the indices of atoms n > m in the first local environment that are within a distance r_cut of both atom n and the central atom.
- **cross_bond_inds_2** (*np.ndarray*) – Two dimensional array whose row m contains the indices of atoms n > m in the second local environment that are within a distance r_cut of both atom n and the central atom.
- **cross_bond_dists_1** (*np.ndarray*) – Two dimensional array whose row m contains the distances from atom m of atoms n > m in the first local environment that are within a distance r_cut of both atom n and the central atom.

- **cross_bond_dists_2** (*np.ndarray*) – Two dimensional array whose row m contains the distances from atom m of atoms n > m in the second local environment that are within a distance r_cut of both atom n and the central atom.
- **triplets_1** (*np.ndarray*) – One dimensional array of integers whose entry m is the number of atoms in the first local environment that are within a distance r_cut of atom m.
- **triplets_2** (*np.ndarray*) – One dimensional array of integers whose entry m is the number of atoms in the second local environment that are within a distance r_cut of atom m.
- **d1** (*int*) – Force component of the first environment.
- **d2** (*int*) – Force component of the second environment.
- **sig** (*float*) – 3-body signal variance hyperparameter.
- **ls** (*float*) – 3-body length scale hyperparameter.
- **r_cut** (*float*) – 3-body cutoff radius.
- **cutoff_func** (*Callable*) – Cutoff function.

Returns Value of the 3-body kernel.

Return type float

flare.kernels.mc_simple.**three_body_se_jit**(*bond_array_1*, *c1*, *etypes1*, *bond_array_2*, *c2*, *etypes2*, *cross_bond_inds_1*, *cross_bond_inds_2*, *cross_bond_dists_1*, *cross_bond_dists_2*, *triplets_1*, *triplets_2*, *sig*, *ls*, *r_cut*, *cutoff_func*)

3-body multi-element kernel between a force component and a local energy accelerated with Numba.

Parameters

- **bond_array_1** (*np.ndarray*) – 3-body bond array of the first local environment.
- **c1** (*int*) – Species of the central atom of the first local environment.
- **etypes1** (*np.ndarray*) – Species of atoms in the first local environment.
- **bond_array_2** (*np.ndarray*) – 3-body bond array of the second local environment.
- **c2** (*int*) – Species of the central atom of the second local environment.
- **etypes2** (*np.ndarray*) – Species of atoms in the second local environment.
- **cross_bond_inds_1** (*np.ndarray*) – Two dimensional array whose row m contains the indices of atoms n > m in the first local environment that are within a distance r_cut of both atom n and the central atom.
- **cross_bond_inds_2** (*np.ndarray*) – Two dimensional array whose row m contains the indices of atoms n > m in the second local environment that are within a distance r_cut of both atom n and the central atom.
- **cross_bond_dists_1** (*np.ndarray*) – Two dimensional array whose row m contains the distances from atom m of atoms n > m in the first local environment that are within a distance r_cut of both atom n and the central atom.
- **cross_bond_dists_2** (*np.ndarray*) – Two dimensional array whose row m contains the distances from atom m of atoms n > m in the second local environment that are within a distance r_cut of both atom n and the central atom.
- **triplets_1** (*np.ndarray*) – One dimensional array of integers whose entry m is the number of atoms in the first local environment that are within a distance r_cut of atom m.

- **triplets_2** (*np.ndarray*) – One dimensional array of integers whose entry m is the number of atoms in the second local environment that are within a distance r_cut of atom m.
- **sig** (*float*) – 3-body signal variance hyperparameter.
- **ls** (*float*) – 3-body length scale hyperparameter.
- **r_cut** (*float*) – 3-body cutoff radius.
- **cutoff_func** (*Callable*) – Cutoff function.

Returns Value of the 3-body force/energy kernel.

Return type float

flare.kernels.mc_simple.**three_body_sf_jit**(*bond_array_1*, *c1*, *etypes1*, *bond_array_2*, *c2*, *etypes2*, *cross_bond_inds_1*, *cross_bond_inds_2*, *cross_bond_dists_1*, *cross_bond_dists_2*, *triplets_1*, *triplets_2*, *sig*, *ls*, *r_cut*, *cutoff_func*)

3-body multi-element kernel between two force components accelerated with Numba.

Parameters

- **bond_array_1** (*np.ndarray*) – 3-body bond array of the first local environment.
- **c1** (*int*) – Species of the central atom of the first local environment.
- **etypes1** (*np.ndarray*) – Species of atoms in the first local environment.
- **bond_array_2** (*np.ndarray*) – 3-body bond array of the second local environment.
- **c2** (*int*) – Species of the central atom of the second local environment.
- **etypes2** (*np.ndarray*) – Species of atoms in the second local environment.
- **cross_bond_inds_1** (*np.ndarray*) – Two dimensional array whose row m contains the indices of atoms n > m in the first local environment that are within a distance r_cut of both atom n and the central atom.
- **cross_bond_inds_2** (*np.ndarray*) – Two dimensional array whose row m contains the indices of atoms n > m in the second local environment that are within a distance r_cut of both atom n and the central atom.
- **cross_bond_dists_1** (*np.ndarray*) – Two dimensional array whose row m contains the distances from atom m of atoms n > m in the first local environment that are within a distance r_cut of both atom n and the central atom.
- **cross_bond_dists_2** (*np.ndarray*) – Two dimensional array whose row m contains the distances from atom m of atoms n > m in the second local environment that are within a distance r_cut of both atom n and the central atom.
- **triplets_1** (*np.ndarray*) – One dimensional array of integers whose entry m is the number of atoms in the first local environment that are within a distance r_cut of atom m.
- **triplets_2** (*np.ndarray*) – One dimensional array of integers whose entry m is the number of atoms in the second local environment that are within a distance r_cut of atom m.
- **sig** (*float*) – 3-body signal variance hyperparameter.
- **ls** (*float*) – 3-body length scale hyperparameter.
- **r_cut** (*float*) – 3-body cutoff radius.
- **cutoff_func** (*Callable*) – Cutoff function.

Returns Value of the 3-body kernel.

Return type float

flare.kernels.mc_simple.**three_body_ss_jit**(*bond_array_1*, *c1*, *etypes1*, *bond_array_2*, *c2*, *etypes2*, *cross_bond_inds_1*, *cross_bond_inds_2*, *cross_bond_dists_1*, *cross_bond_dists_2*, *triplets_1*, *triplets_2*, *sig*, *ls*, *r_cut*, *cutoff_func*)

3-body multi-element kernel between two force components accelerated with Numba.

Parameters

- **bond_array_1** (*np.ndarray*) – 3-body bond array of the first local environment.
- **c1** (*int*) – Species of the central atom of the first local environment.
- **etypes1** (*np.ndarray*) – Species of atoms in the first local environment.
- **bond_array_2** (*np.ndarray*) – 3-body bond array of the second local environment.
- **c2** (*int*) – Species of the central atom of the second local environment.
- **etypes2** (*np.ndarray*) – Species of atoms in the second local environment.
- **cross_bond_inds_1** (*np.ndarray*) – Two dimensional array whose row m contains the indices of atoms n > m in the first local environment that are within a distance r_cut of both atom n and the central atom.
- **cross_bond_inds_2** (*np.ndarray*) – Two dimensional array whose row m contains the indices of atoms n > m in the second local environment that are within a distance r_cut of both atom n and the central atom.
- **cross_bond_dists_1** (*np.ndarray*) – Two dimensional array whose row m contains the distances from atom m of atoms n > m in the first local environment that are within a distance r_cut of both atom n and the central atom.
- **cross_bond_dists_2** (*np.ndarray*) – Two dimensional array whose row m contains the distances from atom m of atoms n > m in the second local environment that are within a distance r_cut of both atom n and the central atom.
- **triplets_1** (*np.ndarray*) – One dimensional array of integers whose entry m is the number of atoms in the first local environment that are within a distance r_cut of atom m.
- **triplets_2** (*np.ndarray*) – One dimensional array of integers whose entry m is the number of atoms in the second local environment that are within a distance r_cut of atom m.
- **sig** (*float*) – 3-body signal variance hyperparameter.
- **ls** (*float*) – 3-body length scale hyperparameter.
- **r_cut** (*float*) – 3-body cutoff radius.
- **cutoff_func** (*Callable*) – Cutoff function.

Returns Value of the 3-body kernel.

Return type float

flare.kernels.mc_simple.**two_body_mc**(*env1: flare.env.AtomicEnvironment*, *env2: flare.env.AtomicEnvironment*, *d1: float*, *d2: float*, *hyps: ndarray*, *cutoffs: ndarray*, *cutoff_func: Callable = <function quadratic_cutoff>*) → float

2-body multi-element kernel between two force components.

Parameters

- **env1** (*AtomicEnvironment*) – First local environment.

- **env2** (*AtomicEnvironment*) – Second local environment.
- **d1** (*int*) – Force component of the first environment.
- **d2** (*int*) – Force component of the second environment.
- **hyps** (*np.ndarray*) – Hyperparameters of the kernel function (sig, ls).
- **cutoffs** (*np.ndarray*) – One-element array containing the 2-body cutoff.
- **cutoff_func** (*Callable*) – Cutoff function of the kernel.

Returns Value of the 2-body kernel.

Return type float

flare.kernels.mc_simple.**two_body_mc_en**(*env1: flare.env.AtomicEnvironment*, *env2: flare.env.AtomicEnvironment*, *hyps: ndarray*, *cutoffs: ndarray*, *cutoff_func: Callable = <function quadratic_cutoff>*) → float

2-body multi-element kernel between two local energies.

Parameters

- **env1** (*AtomicEnvironment*) – First local environment.
- **env2** (*AtomicEnvironment*) – Second local environment.
- **hyps** (*np.ndarray*) – Hyperparameters of the kernel function (sig, ls).
- **cutoffs** (*np.ndarray*) – One-element array containing the 2-body cutoff.
- **cutoff_func** (*Callable*) – Cutoff function of the kernel.

Returns Value of the 2-body force/energy kernel.

Return type float

flare.kernels.mc_simple.**two_body_mc_en_jit**(*bond_array_1*, *c1*, *etypes1*, *bond_array_2*, *c2*, *etypes2*, *sig*, *ls*, *r_cut*, *cutoff_func*)

2-body multi-element kernel between two local energies accelerated with Numba.

Parameters

- **bond_array_1** (*np.ndarray*) – 2-body bond array of the first local environment.
- **c1** (*int*) – Species of the central atom of the first local environment.
- **etypes1** (*np.ndarray*) – Species of atoms in the first local environment.
- **bond_array_2** (*np.ndarray*) – 2-body bond array of the second local environment.
- **c2** (*int*) – Species of the central atom of the second local environment.
- **etypes2** (*np.ndarray*) – Species of atoms in the second local environment.
- **sig** (*float*) – 2-body signal variance hyperparameter.
- **ls** (*float*) – 2-body length scale hyperparameter.
- **r_cut** (*float*) – 2-body cutoff radius.
- **cutoff_func** (*Callable*) – Cutoff function.

Returns Value of the 2-body local energy kernel.

Return type float

flare.kernels.mc_simple.**two_body_mc_force_en**(*env1: flare.env.AtomicEnvironment*, *env2: flare.env.AtomicEnvironment*, *d1: int*, *hyps: ndarray*, *cutoffs: ndarray*, *cutoff_func: Callable = <function quadratic_cutoff>*) → float

2-body multi-element kernel between a force component and a local energy.

Parameters

- **env1** (*AtomicEnvironment*) – Local environment associated with the force component.
- **env2** (*AtomicEnvironment*) – Local environment associated with the local energy.
- **d1** (*int*) – Force component of the first environment.
- **hyps** (*np.ndarray*) – Hyperparameters of the kernel function (sig, ls).
- **cutoffs** (*np.ndarray*) – One-element array containing the 2-body cutoff.
- **cutoff_func** (*Callable*) – Cutoff function of the kernel.

Returns Value of the 2-body force/energy kernel.

Return type float

flare.kernels.mc_simple.**two_body_mc_force_en_jit**(*bond_array_1*, *c1*, *etypes1*, *bond_array_2*, *c2*, *etypes2*, *d1*, *sig*, *ls*, *r_cut*, *cutoff_func*)

2-body multi-element kernel between a force component and a local energy accelerated with Numba.

Parameters

- **bond_array_1** (*np.ndarray*) – 2-body bond array of the first local environment.
- **c1** (*int*) – Species of the central atom of the first local environment.
- **etypes1** (*np.ndarray*) – Species of atoms in the first local environment.
- **bond_array_2** (*np.ndarray*) – 2-body bond array of the second local environment.
- **c2** (*int*) – Species of the central atom of the second local environment.
- **etypes2** (*np.ndarray*) – Species of atoms in the second local environment.
- **d1** (*int*) – Force component of the first environment (1=x, 2=y, 3=z).
- **sig** (*float*) – 2-body signal variance hyperparameter.
- **ls** (*float*) – 2-body length scale hyperparameter.
- **r_cut** (*float*) – 2-body cutoff radius.
- **cutoff_func** (*Callable*) – Cutoff function.

Returns Value of the 2-body force/energy kernel.

Return type float

flare.kernels.mc_simple.**two_body_mc_grad**(*env1: flare.env.AtomicEnvironment*, *env2: flare.env.AtomicEnvironment*, *d1: int*, *d2: int*, *hyps: ndarray*, *cutoffs: ndarray*, *cutoff_func: Callable = <function quadratic_cutoff>) -> (<class 'float'>, 'ndarray'*)

2-body multi-element kernel between two force components and its gradient with respect to the hyperparameters.

Parameters

- **env1** (*AtomicEnvironment*) – First local environment.
- **env2** (*AtomicEnvironment*) – Second local environment.
- **d1** (*int*) – Force component of the first environment.
- **d2** (*int*) – Force component of the second environment.
- **hyps** (*np.ndarray*) – Hyperparameters of the kernel function (sig, ls).
- **cutoffs** (*np.ndarray*) – One-element array containing the 2-body cutoff.
- **cutoff_func** (*Callable*) – Cutoff function of the kernel.

Returns Value of the 2-body kernel and its gradient with respect to the hyperparameters.

Return type (float, np.ndarray)

`flare.kernels.mc_simple.`**`two_body_mc_grad_jit`**(*bond_array_1*, *c1*, *etypes1*, *bond_array_2*, *c2*, *etypes2*, *d1*, *d2*, *sig*, *ls*, *r_cut*, *cutoff_func*)

2-body multi-element kernel between two force components and its gradient with respect to the hyperparameters.

Parameters

- **bond_array_1** (*np.ndarray*) – 2-body bond array of the first local environment.
- **c1** (*int*) – Species of the central atom of the first local environment.
- **etypes1** (*np.ndarray*) – Species of atoms in the first local environment.
- **bond_array_2** (*np.ndarray*) – 2-body bond array of the second local environment.
- **c2** (*int*) – Species of the central atom of the second local environment.
- **etypes2** (*np.ndarray*) – Species of atoms in the second local environment.
- **d1** (*int*) – Force component of the first environment (1=x, 2=y, 3=z).
- **d2** (*int*) – Force component of the second environment (1=x, 2=y, 3=z).
- **sig** (*float*) – 2-body signal variance hyperparameter.
- **ls** (*float*) – 2-body length scale hyperparameter.
- **r_cut** (*float*) – 2-body cutoff radius.
- **cutoff_func** (*Callable*) – Cutoff function.

Returns Value of the 2-body kernel and its gradient with respect to the hyperparameters.

Return type (float, float)

`flare.kernels.mc_simple.`**`two_body_mc_jit`**(*bond_array_1*, *c1*, *etypes1*, *bond_array_2*, *c2*, *etypes2*, *d1*, *d2*, *sig*, *ls*, *r_cut*, *cutoff_func*)

2-body multi-element kernel between two force components accelerated with Numba.

Parameters

- **bond_array_1** (*np.ndarray*) – 2-body bond array of the first local environment.
- **c1** (*int*) – Species of the central atom of the first local environment.
- **etypes1** (*np.ndarray*) – Species of atoms in the first local environment.
- **bond_array_2** (*np.ndarray*) – 2-body bond array of the second local environment.
- **c2** (*int*) – Species of the central atom of the second local environment.

- **etypes2** (*np.ndarray*) – Species of atoms in the second local environment.
- **d1** (*int*) – Force component of the first environment (1=x, 2=y, 3=z).
- **d2** (*int*) – Force component of the second environment (1=x, 2=y, 3=z).
- **sig** (*float*) – 2-body signal variance hyperparameter.
- **ls** (*float*) – 2-body length scale hyperparameter.
- **r_cut** (*float*) – 2-body cutoff radius.
- **cutoff_func** (*Callable*) – Cutoff function.

Returns Value of the 2-body kernel.

Return type float

flare.kernels.mc_simple.**two_body_mc_stress_en_jit**(*bond_array_1*, *c1*, *etypes1*, *bond_array_2*, *c2*, *etypes2*, *d1*, *d2*, *sig*, *ls*, *r_cut*, *cutoff_func*)

2-body multi-element kernel between a partial stress component and a local energy accelerated with Numba.

Parameters

- **bond_array_1** (*np.ndarray*) – 2-body bond array of the first local environment.
- **c1** (*int*) – Species of the central atom of the first local environment.
- **etypes1** (*np.ndarray*) – Species of atoms in the first local environment.
- **bond_array_2** (*np.ndarray*) – 2-body bond array of the second local environment.
- **c2** (*int*) – Species of the central atom of the second local environment.
- **etypes2** (*np.ndarray*) – Species of atoms in the second local environment.
- **d1** (*int*) – First stress component of the first environment (1=x, 2=y, 3=z).
- **d2** (*int*) – Second stress component of the first environment (1=x, 2=y, 3=z).
- **sig** (*float*) – 2-body signal variance hyperparameter.
- **ls** (*float*) – 2-body length scale hyperparameter.
- **r_cut** (*float*) – 2-body cutoff radius.
- **cutoff_func** (*Callable*) – Cutoff function.

Returns Value of the 2-body partial-stress/energy kernel.

Return type float

flare.kernels.mc_simple.**two_body_mc_stress_force_jit**(*bond_array_1*, *c1*, *etypes1*, *bond_array_2*, *c2*, *etypes2*, *d1*, *d2*, *d3*, *sig*, *ls*, *r_cut*, *cutoff_func*)

2-body multi-element kernel between two force components accelerated with Numba.

Parameters

- **bond_array_1** (*np.ndarray*) – 2-body bond array of the first local environment.
- **c1** (*int*) – Species of the central atom of the first local environment.
- **etypes1** (*np.ndarray*) – Species of atoms in the first local environment.
- **bond_array_2** (*np.ndarray*) – 2-body bond array of the second local environment.
- **c2** (*int*) – Species of the central atom of the second local environment.

- **etypes2** (*np.ndarray*) – Species of atoms in the second local environment.
- **d1** (*int*) – First stress component of the first environment (1=x, 2=y, 3=z).
- **d2** (*int*) – Second stress component of the first environment (1=x, 2=y, 3=z).
- **d3** (*int*) – Force component of the second environment (1=x, 2=y, 3=z).
- **sig** (*float*) – 2-body signal variance hyperparameter.
- **ls** (*float*) – 2-body length scale hyperparameter.
- **r_cut** (*float*) – 2-body cutoff radius.
- **cutoff_func** (*Callable*) – Cutoff function.

Returns Value of the 2-body kernel.

Return type float

flare.kernels.mc_simple.**two_body_mc_stress_stress_jit**(*bond_array_1*, *c1*, *etypes1*, *bond_array_2*, *c2*, *etypes2*, *d1*, *d2*, *d3*, *d4*, *sig*, *ls*, *r_cut*, *cutoff_func*)

2-body multi-element kernel between two partial stress components accelerated with Numba.

Parameters

- **bond_array_1** (*np.ndarray*) – 2-body bond array of the first local environment.
- **c1** (*int*) – Species of the central atom of the first local environment.
- **etypes1** (*np.ndarray*) – Species of atoms in the first local environment.
- **bond_array_2** (*np.ndarray*) – 2-body bond array of the second local environment.
- **c2** (*int*) – Species of the central atom of the second local environment.
- **etypes2** (*np.ndarray*) – Species of atoms in the second local environment.
- **d1** (*int*) – First stress component of the first environment (1=x, 2=y, 3=z).
- **d2** (*int*) – Second stress component of the first environment (1=x, 2=y, 3=z).
- **d3** (*int*) – First stress component of the second environment (1=x, 2=y, 3=z).
- **d4** (*int*) – Second stress component of the second environment (1=x, 2=y, 3=z).
- **sig** (*float*) – 2-body signal variance hyperparameter.
- **ls** (*float*) – 2-body length scale hyperparameter.
- **r_cut** (*float*) – 2-body cutoff radius.
- **cutoff_func** (*Callable*) – Cutoff function.

Returns Value of the 2-body kernel.

Return type float

flare.kernels.mc_simple.**two_plus_three_body_mc**(*env1: flare.env.AtomicEnvironment*, *env2: flare.env.AtomicEnvironment*, *d1: int*, *d2: int*, *hyps: ndarray*, *cutoffs: ndarray*, *cutoff_func: Callable = <function quadratic_cutoff>*) → float

2+3-body multi-element kernel between two force components.

Parameters

- **env1** (*AtomicEnvironment*) – First local environment.
- **env2** (*AtomicEnvironment*) – Second local environment.
- **d1** (*int*) – Force component of the first environment.
- **d2** (*int*) – Force component of the second environment.
- **hyps** (*np.ndarray*) – Hyperparameters of the kernel function (sig1, ls1, sig2, ls2, sig_n).
- **cutoffs** (*np.ndarray*) – Two-element array containing the 2- and 3-body cutoffs.
- **cutoff_func** (*Callable*) – Cutoff function of the kernel.

Returns Value of the 2+3-body kernel.

Return type float

flare.kernels.mc_simple.**two_plus_three_body_mc_grad**(*env1: flare.env.AtomicEnvironment*, *env2: flare.env.AtomicEnvironment*, *d1: int*, *d2: int*, *hyps: ndarray*, *cutoffs: ndarray*, *cutoff_func: Callable = <function quadratic_cutoff>*) *-> ('float', 'ndarray')*

2+3-body multi-element kernel between two force components and its gradient with respect to the hyperparameters.

Parameters

- **env1** (*AtomicEnvironment*) – First local environment.
- **env2** (*AtomicEnvironment*) – Second local environment.
- **d1** (*int*) – Force component of the first environment.
- **d2** (*int*) – Force component of the second environment.
- **hyps** (*np.ndarray*) – Hyperparameters of the kernel function (sig1, ls1, sig2, ls2, sig_n).
- **cutoffs** (*np.ndarray*) – Two-element array containing the 2- and 3-body cutoffs.
- **cutoff_func** (*Callable*) – Cutoff function of the kernel.

Returns Value of the 2+3-body kernel and its gradient with respect to the hyperparameters.

Return type (float, np.ndarray)

flare.kernels.mc_simple.**two_plus_three_mc_en**(*env1: flare.env.AtomicEnvironment*, *env2: flare.env.AtomicEnvironment*, *hyps: ndarray*, *cutoffs: ndarray*, *cutoff_func: Callable = <function quadratic_cutoff>*) → float

2+3-body multi-element kernel between two local energies.

Parameters

- **env1** (*AtomicEnvironment*) – First local environment.
- **env2** (*AtomicEnvironment*) – Second local environment.
- **hyps** (*np.ndarray*) – Hyperparameters of the kernel function (sig1, ls1, sig2, ls2).
- **cutoffs** (*np.ndarray*) – Two-element array containing the 2- and 3-body cutoffs.

- **cutoff_func** (*Callable*) – Cutoff function of the kernel.

Returns Value of the 2+3-body energy/energy kernel.

Return type float

flare.kernels.mc_simple.**two_plus_three_mc_force_en**(*env1: flare.env.AtomicEnvironment*, *env2: flare.env.AtomicEnvironment*, *d1: int*, *hyps: ndarray*, *cutoffs: ndarray*, *cutoff_func: Callable = <function quadratic_cutoff>*) → float

2+3-body multi-element kernel between a force component and a local energy.

Parameters

- **env1** (*AtomicEnvironment*) – Local environment associated with the force component.
- **env2** (*AtomicEnvironment*) – Local environment associated with the local energy.
- **d1** (*int*) – Force component of the first environment.
- **hyps** (*np.ndarray*) – Hyperparameters of the kernel function (sig1, ls1, sig2, ls2).
- **cutoffs** (*np.ndarray*) – Two-element array containing the 2- and 3-body cutoffs.
- **cutoff_func** (*Callable*) – Cutoff function of the kernel.

Returns Value of the 2+3-body force/energy kernel.

Return type float

flare.kernels.mc_simple.**two_plus_three_plus_many_body_mc**(*env1: flare.env.AtomicEnvironment*, *env2: flare.env.AtomicEnvironment*, *d1: int*, *d2: int*, *hyps*, *cutoffs*, *cutoff_func=<function quadratic_cutoff>*)

2+3-body single-element kernel between two force components.

Parameters

- **env1** (*AtomicEnvironment*) – First local environment.
- **env2** (*AtomicEnvironment*) – Second local environment.
- **d1** (*int*) – Force component of the first environment.
- **d2** (*int*) – Force component of the second environment.
- **hyps** (*np.ndarray*) – Hyperparameters of the kernel function (sig1, ls1, sig2, ls2, sig3, ls3, sig_n).
- **cutoffs** (*np.ndarray*) – Two-element array containing the 2- and 3-body cutoffs.
- **cutoff_func** (*Callable*) – Cutoff function of the kernel.

Returns Value of the 2+3+many-body kernel.

Return type float

flare.kernels.mc_simple.**two_plus_three_plus_many_body_mc_en**(*env1: flare.env.AtomicEnvironment*, *env2: flare.env.AtomicEnvironment*, *hyps*, *cutoffs*, *cutoff_func=<function quadratic_cutoff>*)

2+3+many-body single-element energy kernel.

Parameters

- **env1** (*AtomicEnvironment*) – First local environment.
- **env2** (*AtomicEnvironment*) – Second local environment.
- **hyps** (*np.ndarray*) – Hyperparameters of the kernel function (sig1, ls1, sig2, ls2, sig3, ls3, sig_n).
- **cutoffs** (*np.ndarray*) – Two-element array containing the 2- and 3-body cutoffs.
- **cutoff_func** (*Callable*) – Cutoff function of the kernel.

Returns Value of the 2+3+many-body kernel.

Return type float

flare.kernels.mc_simple.**two_plus_three_plus_many_body_mc_force_en**(*env1: flare.env.AtomicEnvironment*, *env2: flare.env.AtomicEnvironment*, *d1: int*, *hyps*, *cutoffs*, *cutoff_func=<function quadratic_cutoff>*)

2+3+many-body single-element kernel between two force and energy components.

Parameters

- **env1** (*AtomicEnvironment*) – First local environment.
- **env2** (*AtomicEnvironment*) – Second local environment.
- **d1** (*int*) – Force component of the first environment.
- **hyps** (*np.ndarray*) – Hyperparameters of the kernel function (sig1, ls1, sig2, ls2, sig3, ls3, sig_n).
- **cutoffs** (*np.ndarray*) – Two-element array containing the 2- and 3-body cutoffs.
- **cutoff_func** (*Callable*) – Cutoff function of the kernel.

Returns Value of the 2+3+many-body kernel.

Return type float

flare.kernels.mc_simple.**two_plus_three_plus_many_body_mc_grad**(*env1: flare.env.AtomicEnvironment*, *env2: flare.env.AtomicEnvironment*, *d1: int*, *d2: int*, *hyps*, *cutoffs*, *cutoff_func=<function quadratic_cutoff>*)

2+3+many-body single-element kernel between two force components.

Parameters

- **env1** (*AtomicEnvironment*) – First local environment.
- **env2** (*AtomicEnvironment*) – Second local environment.
- **d1** (*int*) – Force component of the first environment.
- **d2** (*int*) – Force component of the second environment.
- **hyps** (*np.ndarray*) – Hyperparameters of the kernel function (sig1, ls1, sig2, ls2, sig3, ls3, sig_n).
- **cutoffs** (*np.ndarray*) – Two-element array containing the 2- and 3-body cutoffs.
- **cutoff_func** (*Callable*) – Cutoff function of the kernel.

Returns Value of the 2+3+many-body kernel.

Return type float

Multi-element Kernels (Separate Parameters)

Multicomponent kernels (simple) restrict all signal variance and length scale of hyperparameters to a single value. The kernels in this module allow you to have different sets of hyperparameters and cutoffs for different interactions, and have flexible groupings of elements. It also allows you to do partial hyper-parameter training, keeping some components fixed.

To use this set of kernels, we need a hyps_mask dictionary for GaussianProcess, MappedGaussianProcess, and AtomicEnvironment (if you also set up different cutoffs). A simple example is shown below.

Examples

```
>>> from flare.util.parameter_helper import ParameterHelper
>>> from flare.gp import GaussianProcess
```

```
>>> pm = ParameterHelper(species=['O', 'C', 'H'],
...                      kernels={'twobody':[['*', '*'], ['O','O']],
...                               'threebody':[['*', '*', '*'], ['O','O', 'O']]},
...                      parameters={'twobody0':[1, 0.5, 1], 'twobody1':[2, 0.2, 2],
...                            'triplet0':[1, 0.5], 'triplet1':[2, 0.2],
...                            'cutoff_twobody':2, 'cutoff_threebody':1, 'noise': 0.
↪05},
...                      constraints={'twobody0':[False, True]})
>>> hyps_mask = pm1.as_dict()
>>> hyps = hyps_mask.pop('hyps')
>>> cutoffs = hyps_mask.pop('cutoffs')
>>> hyp_labels = hyps_mask.pop('hyp_labels')
```

(continues on next page)

(continued from previous page)

```
>>> kernels = hyps_mask['kernels']
>>> gp_model = GaussianProcess(kernels=kernels,
...                            hyps=hyps, cutoffs=cutoffs,
...                            hyp_labels=hyp_labels,
...                            parallel=True, per_atom_par=False,
...                            n_cpus=n_cpus,
...                            multihyps=True, hyps_mask=hm)
```

In the example above, Parameters class generates the arrays needed for these kernels and store all the grouping and mapping information in the hyps_mask dictionary. It stores following keys and values:

- **spec_mask: 118-long integer array descirbing which elements belong to** like groups for determining which bond hyperparameters to use. For instance, [0,0,1,1,0 . . .] assigns H to group 0, He and Li to group 1, and Be to group 0 (the 0th register is ignored).
- **nspec: Integer, number of different species groups (equal to number of** unique values in spec_mask).
- **nbond: Integer, number of different hyperparameter sets to associate with** different 2-body pairings of atoms in groups defined in spec_mask.
- **bond_mask: Array of length nspec^2, which describes the hyperparameter sets to** associate with different pairings of species types. For example, if there are atoms of type 0 and 1, then bond_mask defines which hyperparameters to use for parings [0-0, 0-1, 1-0, 1-1]: if we wanted hyperparameter set 0 for 0-0 parings and set 1 for 0-1 and 1-1 pairings, then we would make bond_mask [0, 1, 1, 1].
- **ntriplet: Integer, number of different hyperparameter sets to associate** with different 3-body pariings of atoms in groups defined in spec_mask.
- **triplet_mask: Similar to bond mask: Triplet pairings of type 0 and 1 atoms** would go {0-0-0, 0-0-1, 0-1-0, 0-1-1, 1-0-0, 1-0-1, 1-1-0, 1-1-1}, and if we wanted hyp. set 0 for triplets with only atoms of type 0 and hyp. set 1 for all the rest, then the triplet_mask array would read [0,1,1,1,1,1,1,1]. The user should make sure that the mask has a permutational symmetry.
- **cutoff_2b: Array of length nbond, which stores the cutoff used for different** types of bonds defined in bond_mask
- **ncut3b: Integer, number of different cutoffs sets to associate** with different 3-body pariings of atoms in groups defined in spec_mask.
- **cut3b_mask: Array of length nspec^2, which describes the cutoff to** associate with different bond types in triplets. For example, in a triplet (C, O, H) , there are three cutoffs. Cutoffs for CH bond, CO bond and OH bond. If C and O are associate with atom group 1 in spec_mask and H are associate with group 0 in spec_mask, the cut3b_mask[1*nspec+0] determines the C/O-H bond cutoff, and cut3b_mask[1*nspec+1] determines the C-O bond cutoff. If we want the former one to use the 1st cutoff in cutoff_3b and the later to use the 2nd cutoff in cutoff_3b, the cut3b_mask should be [0, 0, 0, 1]
- **cutoff_3b: Array of length ncut3b, which stores the cutoff used for different** types of bonds in triplets.
- **nmb** [Integer, number of different cutoffs set to associate with different coordination] numbers
- mb_mask: similar to bond_mask and cut3b_mask.
- cutoff_mb: Array of length nmb, stores the cutoff used for different many body terms

For selective optimization. one can define 'map', 'train_noise' and 'original' to identify which element to be optimized. All three have to be defined. train_noise = Bool (True/False), whether the noise parameter can be optimized original: np.array. Full set of initial values for hyperparmeters map: np.array, array to map the hyper parameter back to the full set. map[i]=j means the i-th element in hyps should be the j-th element in hyps_mask['original']

For example, the full set of hyper parmeters may include [ls21, ls22, sig21, sig22, ls3 sg3, noise] but suppose you wanted only the set 21 optimized. The full set of hyperparameters is defined in 'original'; include all those you

want to leave static, and set initial guesses for those you want to vary. Have the 'map' list contain the indices of the hyperparameters in 'original' that correspond to the hyperparameters you want to vary. Have a hyps list which contain those which you want to vary. Below, ls21, ls22 etc... represent floating-point variables which correspond to the initial guesses / static values. You would then pass in:

hyps = [ls21, sig21] hyps_mask = { ..., 'train_noise': False, 'map':[0, 2],

'original': [ls21, ls22, sig21, sig22, ls3, sg3, noise]}

the hyps argument should only contain the values that need to be optimized. If you want noise to be trained as well include noise as the final hyperparameter value in hyps.

flare.kernels.mc_sephyps.**many_body_mc**(*env1*, *env2*, *d1*, *d2*, *cutoff_2b*, *cutoff_3b*, *cutoff_mb*, *nspec*, *spec_mask*, *nbond*, *bond_mask*, *ntriplet*, *triplet_mask*, *ncut3b*, *cut3b_mask*, *nmb*, *mb_mask*, *sig2*, *ls2*, *sig3*, *ls3*, *sigm*, *lsm*, *cutoff_func=<function quadratic_cutoff>*)

many-body multi-element kernel between two force components.

Parameters

- **env1** (*AtomicEnvironment*) – First local environment.
- **env2** (*AtomicEnvironment*) – Second local environment.
- **d1** (*int*) – Force component of the first environment.
- **d2** (*int*) – Force component of the second environment.
- **cutoff_2b** – dummy
- **cutoff_3b** – dummy
- **cutoff_mb** (*float, np.ndarray*) – cutoff(s) for coordination-based manybody interaction
- **nspec** (*int*) – number of different species groups
- **spec_mask** (*np.ndarray*) – 118-long integer array that determines specie group
- **nbond** – dummy
- **bond_mask** – dummy
- **ntriplet** – dummy
- **triplet_mask** – dummy
- **ncut3b** – dummy
- **cut3b_mask** – dummy
- **nmb** (*int*) – number of different hyperparameter sets to associate with manybody pairings
- **mb_mask** (*np.ndarray*) – nspec^2 long integer array
- **sig2** – dummy
- **ls2** – dummy
- **sig3** – dummy
- **ls3** – dummy
- **sigm** (*np.ndarray*) – signal variances associates with manybody term
- **lsm** (*np.ndarray*) – length scales associates with manybody term
- **cutoff_func** (*Callable*) – Cutoff function of the kernel.

Returns Value of the 2+3+many-body kernel.

Return type float

flare.kernels.mc_sephyps.**many_body_mc_en**(*env1*, *env2*, *cutoff_2b*, *cutoff_3b*, *cutoff_mb*, *nspec*, *spec_mask*, *nbond*, *bond_mask*, *ntriplet*, *triplet_mask*, *ncut3b*, *cut3b_mask*, *nmb*, *mb_mask*, *sig2*, *ls2*, *sig3*, *ls3*, *sigm*, *lsm*, *cutoff_func=<function quadratic_cutoff>*)

many-body multi-element kernel between two local energies.

Parameters

- **env1** (*AtomicEnvironment*) – First local environment.
- **env2** (*AtomicEnvironment*) – Second local environment.
- **hyps** (*np.ndarray*) – Hyperparameters of the kernel function (sig, ls).
- **cutoffs** (*np.ndarray*) – One-element array containing the 2-body cutoff.
- **cutoff_func** (*Callable*) – Cutoff function of the kernel.

Returns Value of the 2-body force/energy kernel.

Return type float

flare.kernels.mc_sephyps.**many_body_mc_force_en**(*env1*, *env2*, *d1*, *cutoff_2b*, *cutoff_3b*, *cutoff_mb*, *nspec*, *spec_mask*, *nbond*, *bond_mask*, *ntriplet*, *triplet_mask*, *ncut3b*, *cut3b_mask*, *nmb*, *mb_mask*, *sig2*, *ls2*, *sig3*, *ls3*, *sigm*, *lsm*, *cutoff_func=<function quadratic_cutoff>*)

many-body single-element kernel between two local energies.

Parameters

- **env1** (*AtomicEnvironment*) – First local environment.
- **env2** (*AtomicEnvironment*) – Second local environment.
- **hyps** (*np.ndarray*) – Hyperparameters of the kernel function (sig, ls).
- **cutoffs** (*np.ndarray*) – Two-element array containing the 2-, 3-, and many-body cutoffs.
- **cutoff_func** (*Callable*) – Cutoff function of the kernel.

Returns Value of the many-body force/energy kernel.

Return type float

flare.kernels.mc_sephyps.**many_body_mc_grad**(*env1*, *env2*, *d1*, *d2*, *cutoff_2b*, *cutoff_3b*, *cutoff_mb*, *nspec*, *spec_mask*, *nbond*, *bond_mask*, *ntriplet*, *triplet_mask*, *ncut3b*, *cut3b_mask*, *nmb*, *mb_mask*, *sig2*, *ls2*, *sig3*, *ls3*, *sigm*, *lsm*, *cutoff_func=<function quadratic_cutoff>*)

manybody multi-element kernel between two force components and its gradient with respect to the hyperparameters.

Parameters

- **env1** (*AtomicEnvironment*) – First local environment.
- **env2** (*AtomicEnvironment*) – Second local environment.
- **d1** (*int*) – Force component of the first environment.

- **d2** (*int*) – Force component of the second environment.
- **cutoff_2b** – dummy
- **cutoff_3b** – dummy
- **cutoff_mb** (*float, np.ndarray*) – cutoff(s) for coordination-based manybody interaction
- **nspec** (*int*) – number of different species groups
- **spec_mask** (*np.ndarray*) – 118-long integer array that determines specie group
- **nbond** – dummy
- **bond_mask** – dummy
- **ntriplet** – dummy
- **triplet_mask** – dummy
- **ncut3b** – dummy
- **cut3b_mask** – dummy
- **nmb** (*int*) – number of different hyperparameter sets to associate with manybody pairings
- **mb_mask** (*np.ndarray*) – nspec^2 long integer array
- **sig2** – dummy
- **ls2** – dummy
- **sig3** – dummy
- **ls3** – dummy
- **sigm** (*np.ndarray*) – signal variances associates with manybody term
- **lsm** (*np.ndarray*) – length scales associates with manybody term
- **cutoff_func** (*Callable*) – Cutoff function of the kernel.

Returns Value of the 2+3+manybody kernel and its gradient with respect to the hyperparameters.

Return type (float, np.ndarray)

flare.kernels.mc_sephyps.**three_body_mc**(*env1*, *env2*, *d1*, *d2*, *cutoff_2b*, *cutoff_3b*, *cutoff_mb*, *nspec*, *spec_mask*, *nbond*, *bond_mask*, *ntriplet*, *triplet_mask*, *ncut3b*, *cut3b_mask*, *nmb*, *mb_mask*, *sig2*, *ls2*, *sig3*, *ls3*, *sigm*, *lsm*, *cutoff_func=<function quadratic_cutoff>*)

3-body multi-element kernel between two force components.

Parameters

- **env1** (*AtomicEnvironment*) – First local environment.
- **env2** (*AtomicEnvironment*) – Second local environment.
- **d1** (*int*) – Force component of the first environment.
- **d2** (*int*) – Force component of the second environment.
- **cutoff_2b** – dummy
- **cutoff_3b** (*float, np.ndarray*) – cutoff(s) for three-body interaction
- **nspec** (*int*) – number of different species groups
- **spec_mask** (*np.ndarray*) – 118-long integer array that determines specie group

- **nbond** – dummy
- **bond_mask** – dummy
- **ntriplet** (*int*) – number of different hyperparameter sets to associate with 3-body pairings
- **triplet_mask** (*np.ndarray*) – nspec^3 long integer array
- **ncut3b** (*int*) – number of different 3-body cutoff sets to associate with 3-body pairings
- **cut3b_mask** (*np.ndarray*) – nspec^2 long integer array
- **sig2** – dummy
- **ls2** – dummy
- **sig3** (*np.ndarray*) – signal variances associates with three-body term
- **ls3** (*np.ndarray*) – length scales associates with three-body term
- **cutoff_func** (*Callable*) – Cutoff function of the kernel.

Returns Value of the 2+3-body force/force kernel.

Return type float

flare.kernels.mc_sephyps.**three_body_mc_en**(*env1*, *env2*, *cutoff_2b*, *cutoff_3b*, *cutoff_mb*, *nspec*, *spec_mask*, *nbond*, *bond_mask*, *ntriplet*, *triplet_mask*, *ncut3b*, *cut3b_mask*, *nmb*, *mb_mask*, *sig2*, *ls2*, *sig3*, *ls3*, *sigm*, *lsm*, *cutoff_func=<function quadratic_cutoff>*)

3-body multi-element kernel between two local energies

Parameters

- **env1** (*AtomicEnvironment*) – First local environment.
- **env2** (*AtomicEnvironment*) – Second local environment.
- **d1** (*int*) – Force component of the first environment.
- **d2** (*int*) – Force component of the second environment.
- **cutoff_2b** – dummy
- **cutoff_3b** (*float, np.ndarray*) – cutoff(s) for three-body interaction
- **nspec** (*int*) – number of different species groups
- **spec_mask** (*np.ndarray*) – 118-long integer array that determines specie group
- **nbond** – dummy
- **bond_mask** – dummy
- **ntriplet** (*int*) – number of different hyperparameter sets to associate with 3-body pairings
- **triplet_mask** (*np.ndarray*) – nspec^3 long integer array
- **ncut3b** (*int*) – number of different 3-body cutoff sets to associate with 3-body pairings
- **cut3b_mask** (*np.ndarray*) – nspec^2 long integer array
- **sig2** – dummy
- **ls2** – dummy
- **sig3** (*np.ndarray*) – signal variances associates with three-body term
- **ls3** (*np.ndarray*) – length scales associates with three-body term

- **cutoff_func** (*Callable*) – Cutoff function of the kernel.

Returns Value of the 2+3-body energy/energy kernel.

Return type float

flare.kernels.mc_sephyps.**three_body_mc_force_en**(*env1*, *env2*, *d1*, *cutoff_2b*, *cutoff_3b*, *cutoff_mb*, *nspec*, *spec_mask*, *nbond*, *bond_mask*, *ntriplet*, *triplet_mask*, *ncut3b*, *cut3b_mask*, *nmb*, *mb_mask*, *sig2*, *ls2*, *sig3*, *ls3*, *sigm*, *lsm*, *cutoff_func=<function quadratic_cutoff>*)

3-body multi-element kernel between a force component and local energies

Parameters

- **env1** (*AtomicEnvironment*) – First local environment.
- **env2** (*AtomicEnvironment*) – Second local environment.
- **d1** (*int*) – Force component of the first environment.
- **d2** (*int*) – Force component of the second environment.
- **cutoff_2b** – dummy
- **cutoff_3b** (*float, np.ndarray*) – cutoff(s) for three-body interaction
- **nspec** (*int*) – number of different species groups
- **spec_mask** (*np.ndarray*) – 118-long integer array that determines specie group
- **nbond** – dummy
- **bond_mask** – dummy
- **ntriplet** (*int*) – number of different hyperparameter sets to associate with 3-body pairings
- **triplet_mask** (*np.ndarray*) – nspec^3 long integer array
- **ncut3b** (*int*) – number of different 3-body cutoff sets to associate with 3-body pairings
- **cut3b_mask** (*np.ndarray*) – nspec^2 long integer array
- **sig2** – dummy
- **ls2** – dummy
- **sig3** (*np.ndarray*) – signal variances associates with three-body term
- **ls3** (*np.ndarray*) – length scales associates with three-body term
- **cutoff_func** (*Callable*) – Cutoff function of the kernel.

Returns Value of the 2+3-body force/energy kernel.

Return type float

flare.kernels.mc_sephyps.**three_body_mc_force_en_jit**(*bond_array_1*, *c1*, *etypes1*, *bond_array_2*, *c2*, *etypes2*, *cross_bond_inds_1*, *cross_bond_inds_2*, *cross_bond_dists_1*, *cross_bond_dists_2*, *triplets_1*, *triplets_2*, *d1*, *sig*, *ls*, *r_cut*, *cutoff_func*, *nspec*, *spec_mask*, *triplet_mask*, *cut3b_mask*)

Kernel for 3-body force/energy comparisons.

flare.kernels.mc_sephyps.**three_body_mc_grad**(*env1*, *env2*, *d1*, *d2*, *cutoff_2b*, *cutoff_3b*, *cutoff_mb*, *nspec*, *spec_mask*, *nbond*, *bond_mask*, *ntriplet*, *triplet_mask*, *ncut3b*, *cut3b_mask*, *nmb*, *mb_mask*, *sig2*, *ls2*, *sig3*, *ls3*, *sigm*, *lsm*, *cutoff_func=<function quadratic_cutoff>*)

3-body multi-element kernel between two force components and its gradient with respect to the hyperparameters.

Parameters

- **env1** (*AtomicEnvironment*) – First local environment.
- **env2** (*AtomicEnvironment*) – Second local environment.
- **d1** (*int*) – Force component of the first environment.
- **d2** (*int*) – Force component of the second environment.
- **cutoff_2b** – dummy
- **cutoff_3b** (*float, np.ndarray*) – cutoff(s) for three-body interaction
- **nspec** (*int*) – number of different species groups
- **spec_mask** (*np.ndarray*) – 118-long integer array that determines specie group
- **nbond** – dummy
- **bond_mask** – dummy
- **ntriplet** (*int*) – number of different hyperparameter sets to associate with 3-body pairings
- **triplet_mask** (*np.ndarray*) – nspec^3 long integer array
- **ncut3b** (*int*) – number of different 3-body cutoff sets to associate with 3-body pairings
- **cut3b_mask** (*np.ndarray*) – nspec^2 long integer array
- **sig2** – dummy
- **ls2** – dummy
- **sig3** (*np.ndarray*) – signal variances associates with three-body term
- **ls3** (*np.ndarray*) – length scales associates with three-body term
- **cutoff_func** (*Callable*) – Cutoff function of the kernel.

Returns Value of the 2+3+manybody kernel and its gradient with respect to the hyperparameters.

Return type (float, np.ndarray)

flare.kernels.mc_sephyps.**three_body_mc_grad_jit**(*bond_array_1*, *c1*, *etypes1*, *bond_array_2*, *c2*, *etypes2*, *cross_bond_inds_1*, *cross_bond_inds_2*, *cross_bond_dists_1*, *cross_bond_dists_2*, *triplets_1*, *triplets_2*, *d1*, *d2*, *sig*, *ls*, *r_cut*, *cutoff_func*, *nspec*, *spec_mask*, *ntriplet*, *triplet_mask*, *cut3b_mask*)

Kernel gradient for 3-body force comparisons.

flare.kernels.mc_sephyps.**two_body_mc**(*env1*, *env2*, *d1*, *d2*, *cutoff_2b*, *cutoff_3b*, *cutoff_mb*, *nspec*, *spec_mask*, *nbond*, *bond_mask*, *ntriplet*, *triplet_mask*, *ncut3b*, *cut3b_mask*, *nmb*, *mb_mask*, *sig2*, *ls2*, *sig3*, *ls3*, *sigm*, *lsm*, *cutoff_func=<function quadratic_cutoff>*)

2-body multi-element kernel between two force components.

Parameters

- **env1** (*AtomicEnvironment*) – First local environment.
- **env2** (*AtomicEnvironment*) – Second local environment.
- **d1** (*int*) – Force component of the first environment.
- **d2** (*int*) – Force component of the second environment.
- **cutoff_2b** (*float, np.ndarray*) – cutoff(s) for two-body interaction
- **cutoff_3b** – dummy
- **nspec** (*int*) – number of different species groups
- **spec_mask** (*np.ndarray*) – 118-long integer array that determines specie group
- **nbond** (*int*) – number of different hyperparameter sets to associate with two-body pairings
- **bond_mask** (*np.ndarray*) – nspec^2 long integer array
- **ntriplet** – dummy
- **triplet_mask** – dummy
- **ncut3b** – dummy
- **cut3b_mask** – dummy
- **sig2** – dummy
- **ls2** – dummy
- **sig3** – dummy
- **ls3** – dummy
- **cutoff_func** (*Callable*) – Cutoff function of the kernel.

Returns Value of the 2-body force/force kernel.

Return type float

flare.kernels.mc_sephyps.**two_body_mc_en**(*env1*, *env2*, *cutoff_2b*, *cutoff_3b*, *cutoff_mb*, *nspec*, *spec_mask*, *nbond*, *bond_mask*, *ntriplet*, *triplet_mask*, *ncut3b*, *cut3b_mask*, *nmb*, *mb_mask*, *sig2*, *ls2*, *sig3*, *ls3*, *sigm*, *lsm*, *cutoff_func=<function quadratic_cutoff>*)

2-body multi-element kernel between two local energies

Parameters

- **env1** (*AtomicEnvironment*) – First local environment.
- **env2** (*AtomicEnvironment*) – Second local environment.
- **d1** (*int*) – Force component of the first environment.
- **d2** (*int*) – Force component of the second environment.
- **cutoff_2b** (*float, np.ndarray*) – cutoff(s) for two-body interaction
- **cutoff_3b** – dummy
- **nspec** (*int*) – number of different species groups
- **spec_mask** (*np.ndarray*) – 118-long integer array that determines specie group
- **nbond** (*int*) – number of different hyperparameter sets to associate with two-body pairings
- **bond_mask** (*np.ndarray*) – nspec^2 long integer array
- **ntriplet** – dummy
- **triplet_mask** – dummy
- **ncut3b** – dummy
- **cut3b_mask** – dummy
- **sig2** – dummy
- **ls2** – dummy
- **sig3** – dummy
- **ls3** – dummy
- **cutoff_func** (*Callable*) – Cutoff function of the kernel.

Returns Value of the 2-body energy/energy kernel.

Return type float

flare.kernels.mc_sephyps.**two_body_mc_en_jit**(*bond_array_1*, *c1*, *etypes1*, *bond_array_2*, *c2*, *etypes2*, *sig*, *ls*, *r_cut*, *cutoff_func*, *nspec*, *spec_mask*, *bond_mask*)

Multicomponent two-body energy/energy kernel accelerated with Numba's njit decorator.

flare.kernels.mc_sephyps.**two_body_mc_force_en**(*env1*, *env2*, *d1*, *cutoff_2b*, *cutoff_3b*, *cutoff_mb*, *nspec*, *spec_mask*, *nbond*, *bond_mask*, *ntriplet*, *triplet_mask*, *ncut3b*, *cut3b_mask*, *nmb*, *mb_mask*, *sig2*, *ls2*, *sig3*, *ls3*, *sigm*, *lsm*, *cutoff_func=<function quadratic_cutoff>*)

2-body multi-element kernel between a force components and local energy

Parameters

- **env1** (*AtomicEnvironment*) – First local environment.

- **env2** (*AtomicEnvironment*) – Second local environment.
- **d1** (*int*) – Force component of the first environment.
- **d2** (*int*) – Force component of the second environment.
- **cutoff_2b** (*float, np.ndarray*) – cutoff(s) for two-body interaction
- **cutoff_3b** – dummy
- **nspec** (*int*) – number of different species groups
- **spec_mask** (*np.ndarray*) – 118-long integer array that determines specie group
- **nbond** (*int*) – number of different hyperparameter sets to associate with two-body pairings
- **bond_mask** (*np.ndarray*) – nspec^2 long integer array
- **ntriplet** – dummy
- **triplet_mask** – dummy
- **ncut3b** – dummy
- **cut3b_mask** – dummy
- **sig2** – dummy
- **ls2** – dummy
- **sig3** – dummy
- **ls3** – dummy
- **cutoff_func** (*Callable*) – Cutoff function of the kernel.

Returns Value of the 2-body force/energy kernel.

Return type float

flare.kernels.mc_sephyps.**two_body_mc_force_en_jit**(*bond_array_1*, *c1*, *etypes1*, *bond_array_2*, *c2*, *etypes2*, *d1*, *sig*, *ls*, *r_cut*, *cutoff_func*, *nspec*, *spec_mask*, *bond_mask*)

Multicomponent two-body force/energy kernel accelerated with Numba's njit decorator.

flare.kernels.mc_sephyps.**two_body_mc_grad**(*env1*, *env2*, *d1*, *d2*, *cutoff_2b*, *cutoff_3b*, *cutoff_mb*, *nspec*, *spec_mask*, *nbond*, *bond_mask*, *ntriplet*, *triplet_mask*, *ncut3b*, *cut3b_mask*, *nmb*, *mb_mask*, *sig2*, *ls2*, *sig3*, *ls3*, *sigm*, *lsm*, *cutoff_func=<function quadratic_cutoff>*)

2-body multi-element kernel between two force components and its gradient with respect to the hyperparameters.

Parameters

- **env1** (*AtomicEnvironment*) – First local environment.
- **env2** (*AtomicEnvironment*) – Second local environment.
- **d1** (*int*) – Force component of the first environment.
- **d2** (*int*) – Force component of the second environment.
- **cutoff_2b** (*float, np.ndarray*) – cutoff(s) for two-body interaction
- **cutoff_3b** – dummy
- **nspec** (*int*) – number of different species groups

- **spec_mask** (*np.ndarray*) – 118-long integer array that determines specie group
- **nbond** (*int*) – number of different hyperparameter sets to associate with two-body pairings
- **bond_mask** (*np.ndarray*) – nspec^2 long integer array
- **ntriplet** – dummy
- **triplet_mask** – dummy
- **ncut3b** – dummy
- **cut3b_mask** – dummy
- **sig2** (*np.ndarray*) – signal variances associates with two-body term
- **ls2** (*np.ndarray*) – length scales associates with two-body term
- **sig3** – dummy
- **ls3** – dummy
- **cutoff_func** (*Callable*) – Cutoff function of the kernel.

Returns Value of the 2-body kernel and its gradient with respect to the hyperparameters.

Return type (float, np.ndarray)

flare.kernels.mc_sephyps.**two_body_mc_grad_jit**(*bond_array_1*, *c1*, *etypes1*, *bond_array_2*, *c2*, *etypes2*, *d1*, *d2*, *sig*, *ls*, *r_cut*, *cutoff_func*, *nspec*, *spec_mask*, *nbond*, *bond_mask*)

Multicomponent two-body force/force kernel gradient accelerated with Numba's njit decorator.

flare.kernels.mc_sephyps.**two_body_mc_jit**(*bond_array_1*, *c1*, *etypes1*, *bond_array_2*, *c2*, *etypes2*, *d1*, *d2*, *sig*, *ls*, *r_cut*, *cutoff_func*, *nspec*, *spec_mask*, *bond_mask*)

Multicomponent two-body force/force kernel accelerated with Numba's njit decorator. Loops over bonds in two environments and adds to the kernel if bonds are of the same type.

flare.kernels.mc_sephyps.**two_plus_three_body_mc**(*env1*, *env2*, *d1*, *d2*, *cutoff_2b*, *cutoff_3b*, *cutoff_mb*, *nspec*, *spec_mask*, *nbond*, *bond_mask*, *ntriplet*, *triplet_mask*, *ncut3b*, *cut3b_mask*, *nmb*, *mb_mask*, *sig2*, *ls2*, *sig3*, *ls3*, *sigm*, *lsm*, *cutoff_func=<function quadratic_cutoff>*)

2+3-body multi-element kernel between two force components.

Parameters

- **env1** (*AtomicEnvironment*) – First local environment.
- **env2** (*AtomicEnvironment*) – Second local environment.
- **d1** (*int*) – Force component of the first environment.
- **d2** (*int*) – Force component of the second environment.
- **cutoff_2b** (*float, np.ndarray*) – cutoff(s) for two-body interaction
- **cutoff_3b** (*float, np.ndarray*) – cutoff(s) for three-body interaction
- **cutoff_mb** (*float, np.ndarray*) – cutoff(s) for coordination-based manybody interaction
- **nspec** (*int*) – number of different species groups

- **spec_mask** (*np.ndarray*) – 118-long integer array that determines specie group
- **nbond** (*int*) – number of different hyperparameter sets to associate with two-body pairings
- **bond_mask** (*np.ndarray*) – nspec^2 long integer array
- **ntriplet** (*int*) – number of different hyperparameter sets to associate with 3-body pairings
- **triplet_mask** (*np.ndarray*) – nspec^3 long integer array
- **ncut3b** (*int*) – number of different 3-body cutoff sets to associate with 3-body pairings
- **cut3b_mask** (*np.ndarray*) – nspec^2 long integer array
- **sig2** (*np.ndarray*) – signal variances associates with two-body term
- **ls2** (*np.ndarray*) – length scales associates with two-body term
- **sig3** (*np.ndarray*) – signal variances associates with three-body term
- **ls3** (*np.ndarray*) – length scales associates with three-body term
- **cutoff_func** (*Callable*) – Cutoff function of the kernel.

Returns Value of the 2+3-body force/force kernel.

Return type float

flare.kernels.mc_sephyps.**two_plus_three_body_mc_grad**(*env1*, *env2*, *d1*, *d2*, *cutoff_2b*, *cutoff_3b*, *cutoff_mb*, *nspec*, *spec_mask*, *nbond*, *bond_mask*, *ntriplet*, *triplet_mask*, *ncut3b*, *cut3b_mask*, *nmb*, *mb_mask*, *sig2*, *ls2*, *sig3*, *ls3*, *sigm*, *lsm*, *cutoff_func=<function quadratic_cutoff>*)

2+3-body multi-element kernel between two force components and its gradient with respect to the hyperparameters.

Parameters

- **env1** (*AtomicEnvironment*) – First local environment.
- **env2** (*AtomicEnvironment*) – Second local environment.
- **d1** (*int*) – Force component of the first environment.
- **d2** (*int*) – Force component of the second environment.
- **cutoff_2b** (*float, np.ndarray*) – cutoff(s) for two-body interaction
- **cutoff_3b** (*float, np.ndarray*) – cutoff(s) for three-body interaction
- **nspec** (*int*) – number of different species groups
- **spec_mask** (*np.ndarray*) – 118-long integer array that determines specie group
- **nbond** (*int*) – number of different hyperparameter sets to associate with two-body pairings
- **bond_mask** (*np.ndarray*) – nspec^2 long integer array
- **ntriplet** (*int*) – number of different hyperparameter sets to associate with 3-body pairings
- **triplet_mask** (*np.ndarray*) – nspec^3 long integer array
- **ncut3b** (*int*) – number of different 3-body cutoff sets to associate with 3-body pairings
- **cut3b_mask** (*np.ndarray*) – nspec^2 long integer array

- **sig2** (*np.ndarray*) – signal variances associates with two-body term
- **ls2** (*np.ndarray*) – length scales associates with two-body term
- **sig3** (*np.ndarray*) – signal variances associates with three-body term
- **ls3** (*np.ndarray*) – length scales associates with three-body term
- **cutoff_func** (*Callable*) – Cutoff function of the kernel.

Returns Value of the 2+3-body kernel and its gradient with respect to the hyperparameters.

Return type (float, np.ndarray)

flare.kernels.mc_sephyps.**two_plus_three_mc_en**(*env1*, *env2*, *cutoff_2b*, *cutoff_3b*, *cutoff_mb*, *nspec*, *spec_mask*, *nbond*, *bond_mask*, *ntriplet*, *triplet_mask*, *ncut3b*, *cut3b_mask*, *nmb*, *mb_mask*, *sig2*, *ls2*, *sig3*, *ls3*, *sigm*, *lsm*, *cutoff_func=<function quadratic_cutoff>*)

2+3-body multi-element kernel between two local energies

Parameters

- **env1** (*AtomicEnvironment*) – First local environment.
- **env2** (*AtomicEnvironment*) – Second local environment.
- **d1** (*int*) – Force component of the first environment.
- **d2** (*int*) – Force component of the second environment.
- **cutoff_2b** (*float, np.ndarray*) – cutoff(s) for two-body interaction
- **cutoff_3b** (*float, np.ndarray*) – cutoff(s) for three-body interaction
- **cutoff_mb** (*float, np.ndarray*) – cutoff(s) for coordination-based manybody interaction
- **nspec** (*int*) – number of different species groups
- **spec_mask** (*np.ndarray*) – 118-long integer array that determines specie group
- **nbond** (*int*) – number of different hyperparameter sets to associate with two-body pairings
- **bond_mask** (*np.ndarray*) – nspec^2 long integer array
- **ntriplet** (*int*) – number of different hyperparameter sets to associate with 3-body pairings
- **triplet_mask** (*np.ndarray*) – nspec^3 long integer array
- **ncut3b** (*int*) – number of different 3-body cutoff sets to associate with 3-body pairings
- **cut3b_mask** (*np.ndarray*) – nspec^2 long integer array
- **sig2** (*np.ndarray*) – signal variances associates with two-body term
- **ls2** (*np.ndarray*) – length scales associates with two-body term
- **sig3** (*np.ndarray*) – signal variances associates with three-body term
- **ls3** (*np.ndarray*) – length scales associates with three-body term
- **cutoff_func** (*Callable*) – Cutoff function of the kernel.

Returns Value of the 2+3-body energy/energy kernel.

Return type float

flare.kernels.mc_sephyps.**two_plus_three_mc_force_en**(*env1*, *env2*, *d1*, *cutoff_2b*, *cutoff_3b*, *cutoff_mb*, *nspec*, *spec_mask*, *nbond*, *bond_mask*, *ntriplet*, *triplet_mask*, *ncut3b*, *cut3b_mask*, *nmb*, *mb_mask*, *sig2*, *ls2*, *sig3*, *ls3*, *sigm*, *lsm*, *cutoff_func=<function quadratic_cutoff>*)

2+3-body multi-element kernel between force and local energy

Parameters

- **env1** (*AtomicEnvironment*) – First local environment.
- **env2** (*AtomicEnvironment*) – Second local environment.
- **d1** (*int*) – Force component of the first environment.
- **d2** (*int*) – Force component of the second environment.
- **cutoff_2b** (*float, np.ndarray*) – cutoff(s) for two-body interaction
- **cutoff_3b** (*float, np.ndarray*) – cutoff(s) for three-body interaction
- **cutoff_mb** (*float, np.ndarray*) – cutoff(s) for coordination-based manybody interaction
- **nspec** (*int*) – number of different species groups
- **spec_mask** (*np.ndarray*) – 118-long integer array that determines specie group
- **nbond** (*int*) – number of different hyperparameter sets to associate with two-body pairings
- **bond_mask** (*np.ndarray*) – nspec^2 long integer array
- **ntriplet** (*int*) – number of different hyperparameter sets to associate with 3-body pairings
- **triplet_mask** (*np.ndarray*) – nspec^3 long integer array
- **ncut3b** (*int*) – number of different 3-body cutoff sets to associate with 3-body pairings
- **cut3b_mask** (*np.ndarray*) – nspec^2 long integer array
- **sig2** (*np.ndarray*) – signal variances associates with two-body term
- **ls2** (*np.ndarray*) – length scales associates with two-body term
- **sig3** (*np.ndarray*) – signal variances associates with three-body term
- **ls3** (*np.ndarray*) – length scales associates with three-body term
- **cutoff_func** (*Callable*) – Cutoff function of the kernel.

Returns Value of the 2+3-body force/energy kernel.

Return type float

flare.kernels.mc_sephyps.**two_three_many_body_mc**(*env1*, *env2*, *d1*, *d2*, *cutoff_2b*, *cutoff_3b*, *cutoff_mb*, *nspec*, *spec_mask*, *nbond*, *bond_mask*, *ntriplet*, *triplet_mask*, *ncut3b*, *cut3b_mask*, *nmb*, *mb_mask*, *sig2*, *ls2*, *sig3*, *ls3*, *sigm*, *lsm*, *cutoff_func=<function quadratic_cutoff>*)

2+3+manybody multi-element kernel between two force components.

Parameters

- **env1** (*AtomicEnvironment*) – First local environment.
- **env2** (*AtomicEnvironment*) – Second local environment.
- **d1** (*int*) – Force component of the first environment.
- **d2** (*int*) – Force component of the second environment.
- **cutoff_2b** (*float, np.ndarray*) – cutoff(s) for two-body interaction
- **cutoff_3b** (*float, np.ndarray*) – cutoff(s) for three-body interaction
- **cutoff_mb** (*float, np.ndarray*) – cutoff(s) for coordination-based manybody interaction
- **nspec** (*int*) – number of different species groups
- **spec_mask** (*np.ndarray*) – 118-long integer array that determines specie group
- **nbond** (*int*) – number of different hyperparameter sets to associate with two-body pairings
- **bond_mask** (*np.ndarray*) – nspec^2 long integer array
- **ntriplet** (*int*) – number of different hyperparameter sets to associate with 3-body pairings
- **triplet_mask** (*np.ndarray*) – nspec^3 long integer array
- **ncut3b** (*int*) – number of different 3-body cutoff sets to associate with 3-body pairings
- **cut3b_mask** (*np.ndarray*) – nspec^2 long integer array
- **nmb** (*int*) – number of different hyperparameter sets to associate with manybody pairings
- **mb_mask** (*np.ndarray*) – nspec^2 long integer array
- **sig2** (*np.ndarray*) – signal variances associates with two-body term
- **ls2** (*np.ndarray*) – length scales associates with two-body term
- **sig3** (*np.ndarray*) – signal variances associates with three-body term
- **ls3** (*np.ndarray*) – length scales associates with three-body term
- **sigm** (*np.ndarray*) – signal variances associates with manybody term
- **lsm** (*np.ndarray*) – length scales associates with manybody term
- **cutoff_func** (*Callable*) – Cutoff function of the kernel.

Returns Value of the 2+3+many-body kernel.

Return type float

flare.kernels.mc_sephyps.**two_three_many_body_mc_grad**(*env1*, *env2*, *d1*, *d2*, *cutoff_2b*, *cutoff_3b*, *cutoff_mb*, *nspec*, *spec_mask*, *nbond*, *bond_mask*, *ntriplet*, *triplet_mask*, *ncut3b*, *cut3b_mask*, *nmb*, *mb_mask*, *sig2*, *ls2*, *sig3*, *ls3*, *sigm*, *lsm*, *cutoff_func=<function quadratic_cutoff>*)

2+3+manybody multi-element kernel between two force components and its gradient with respect to the hyperparameters.

Parameters

- **env1** (*AtomicEnvironment*) – First local environment.
- **env2** (*AtomicEnvironment*) – Second local environment.

- **d1** (*int*) – Force component of the first environment.
- **d2** (*int*) – Force component of the second environment.
- **cutoff_2b** (*float, np.ndarray*) – cutoff(s) for two-body interaction
- **cutoff_3b** (*float, np.ndarray*) – cutoff(s) for three-body interaction
- **cutoff_mb** (*float, np.ndarray*) – cutoff(s) for coordination-based manybody interaction
- **nspec** (*int*) – number of different species groups
- **spec_mask** (*np.ndarray*) – 118-long integer array that determines specie group
- **nbond** (*int*) – number of different hyperparameter sets to associate with two-body pairings
- **bond_mask** (*np.ndarray*) – nspec^2 long integer array
- **ntriplet** (*int*) – number of different hyperparameter sets to associate with 3-body pairings
- **triplet_mask** (*np.ndarray*) – nspec^3 long integer array
- **ncut3b** (*int*) – number of different 3-body cutoff sets to associate with 3-body pairings
- **cut3b_mask** (*np.ndarray*) – nspec^2 long integer array
- **nmb** (*int*) – number of different hyperparameter sets to associate with manybody pairings
- **mb_mask** (*np.ndarray*) – nspec^2 long integer array
- **sig2** (*np.ndarray*) – signal variances associates with two-body term
- **ls2** (*np.ndarray*) – length scales associates with two-body term
- **sig3** (*np.ndarray*) – signal variances associates with three-body term
- **ls3** (*np.ndarray*) – length scales associates with three-body term
- **sigm** (*np.ndarray*) – signal variances associates with manybody term
- **lsm** (*np.ndarray*) – length scales associates with manybody term
- **cutoff_func** (*Callable*) – Cutoff function of the kernel.

Returns Value of the 2+3+manybody kernel and its gradient with respect to the hyperparameters.

Return type (float, np.ndarray)

flare.kernels.mc_sephyps.**two_three_many_mc_en**(*env1*, *env2*, *cutoff_2b*, *cutoff_3b*, *cutoff_mb*, *nspec*, *spec_mask*, *nbond*, *bond_mask*, *ntriplet*, *triplet_mask*, *ncut3b*, *cut3b_mask*, *nmb*, *mb_mask*, *sig2*, *ls2*, *sig3*, *ls3*, *sigm*, *lsm*, *cutoff_func=<function quadratic_cutoff>*)

2+3+many-body multi-element kernel between two local energies.

Parameters

- **env1** (*AtomicEnvironment*) – First local environment.
- **env2** (*AtomicEnvironment*) – Second local environment.
- **d1** (*int*) – Force component of the first environment.
- **d2** (*int*) – Force component of the second environment.
- **cutoff_2b** (*float, np.ndarray*) – cutoff(s) for two-body interaction
- **cutoff_3b** (*float, np.ndarray*) – cutoff(s) for three-body interaction

- **cutoff_mb** (*float, np.ndarray*) – cutoff(s) for coordination-based manybody interaction
- **nspec** (*int*) – number of different species groups
- **spec_mask** (*np.ndarray*) – 118-long integer array that determines specie group
- **nbond** (*int*) – number of different hyperparameter sets to associate with two-body pairings
- **bond_mask** (*np.ndarray*) – nspec^2 long integer array
- **ntriplet** (*int*) – number of different hyperparameter sets to associate with 3-body pairings
- **triplet_mask** (*np.ndarray*) – nspec^3 long integer array
- **ncut3b** (*int*) – number of different 3-body cutoff sets to associate with 3-body pairings
- **cut3b_mask** (*np.ndarray*) – nspec^2 long integer array
- **nmb** (*int*) – number of different hyperparameter sets to associate with manybody pairings
- **mb_mask** (*np.ndarray*) – nspec^2 long integer array
- **sig2** (*np.ndarray*) – signal variances associates with two-body term
- **ls2** (*np.ndarray*) – length scales associates with two-body term
- **sig3** (*np.ndarray*) – signal variances associates with three-body term
- **ls3** (*np.ndarray*) – length scales associates with three-body term
- **sigm** (*np.ndarray*) – signal variances associates with manybody term
- **lsm** (*np.ndarray*) – length scales associates with manybody term
- **cutoff_func** (*Callable*) – Cutoff function of the kernel.

Returns Value of the 2+3+many-body energy/energy kernel.

Return type float

flare.kernels.mc_sephyps.**two_three_many_mc_force_en**(*env1*, *env2*, *d1*, *cutoff_2b*, *cutoff_3b*, *cutoff_mb*, *nspec*, *spec_mask*, *nbond*, *bond_mask*, *ntriplet*, *triplet_mask*, *ncut3b*, *cut3b_mask*, *nmb*, *mb_mask*, *sig2*, *ls2*, *sig3*, *ls3*, *sigm*, *lsm*, *cutoff_func=<function quadratic_cutoff>*)

2+3+manybody multi-element kernel between a force component and a local energy.

Parameters

- **env1** (*AtomicEnvironment*) – First local environment.
- **env2** (*AtomicEnvironment*) – Second local environment.
- **d1** (*int*) – Force component of the first environment.
- **d2** (*int*) – Force component of the second environment.
- **cutoff_2b** (*float, np.ndarray*) – cutoff(s) for two-body interaction
- **cutoff_3b** (*float, np.ndarray*) – cutoff(s) for three-body interaction
- **cutoff_mb** (*float, np.ndarray*) – cutoff(s) for coordination-based manybody interaction
- **nspec** (*int*) – number of different species groups
- **spec_mask** (*np.ndarray*) – 118-long integer array that determines specie group

- **nbond** (*int*) – number of different hyperparameter sets to associate with two-body pairings
- **bond_mask** (*np.ndarray*) – nspec^2 long integer array
- **ntriplet** (*int*) – number of different hyperparameter sets to associate with 3-body pairings
- **triplet_mask** (*np.ndarray*) – nspec^3 long integer array
- **ncut3b** (*int*) – number of different 3-body cutoff sets to associate with 3-body pairings
- **cut3b_mask** (*np.ndarray*) – nspec^2 long integer array
- **nmb** (*int*) – number of different hyperparameter sets to associate with manybody pairings
- **mb_mask** (*np.ndarray*) – nspec^2 long integer array
- **sig2** (*np.ndarray*) – signal variances associates with two-body term
- **ls2** (*np.ndarray*) – length scales associates with two-body term
- **sig3** (*np.ndarray*) – signal variances associates with three-body term
- **ls3** (*np.ndarray*) – length scales associates with three-body term
- **sigm** (*np.ndarray*) – signal variances associates with manybody term
- **lsm** (*np.ndarray*) – length scales associates with manybody term
- **cutoff_func** (*Callable*) – Cutoff function of the kernel.

Returns Value of the 2+3+many-body force/energy kernel.

Return type float

Implementation of three-body kernels using different cutoffs.

The kernels are slightly slower.

flare.kernels.mc_3b_sepcut.**three_body_mc_force_en_sepcut_jit**(*bond_array_1*, *c1*, *etypes1*, *bond_array_2*, *c2*, *etypes2*, *cross_bond_inds_1*, *cross_bond_inds_2*, *cross_bond_dists_1*, *cross_bond_dists_2*, *triplets_1*, *triplets_2*, *d1*, *sig*, *ls*, *r_cut*, *cutoff_func*, *nspec*, *spec_mask*, *triplet_mask*, *cut3b_mask*)

Kernel for 3-body force/energy comparisons.

flare.kernels.mc_3b_sepcut.**three_body_mc_grad_sepcut_jit**(*bond_array_1*, *c1*, *etypes1*, *bond_array_2*, *c2*, *etypes2*, *cross_bond_inds_1*, *cross_bond_inds_2*, *cross_bond_dists_1*, *cross_bond_dists_2*, *triplets_1*, *triplets_2*, *d1*, *d2*, *sig*, *ls*, *r_cut*, *cutoff_func*, *nspec*, *spec_mask*, *ntriplet*, *triplet_mask*, *cut3b_mask*)

Kernel gradient for 3-body force comparisons.

Implementation of three-body kernels using different cutoffs.

The kernels are slightly slower.

flare.kernels.mc_mb_sepcut.**many_body_mc_en_sepcut_jit**(*q_array_1*, *q_array_2*, *c1*, *c2*, *species1*, *species2*, *sig*, *ls*, *nspec*, *spec_mask*, *mb_mask*)

many-body many-element kernel between energy components accelerated with Numba.

Parameters

- **bond_array_1** (*np.ndarray*) – many-body bond array of the first local environment.
- **bond_array_2** (*np.ndarray*) – many-body bond array of the second local environment.
- **c1** (*int*) – atomic species of the central atom in env 1
- **c2** (*int*) – atomic species of the central atom in env 2
- **etypes1** (*np.ndarray*) – atomic species of atoms in env 1
- **etypes2** (*np.ndarray*) – atomic species of atoms in env 2
- **species1** (*np.ndarray*) – all the atomic species present in trajectory 1
- **species2** (*np.ndarray*) – all the atomic species present in trajectory 2
- **sig** (*float*) – many-body signal variance hyperparameter.
- **ls** (*float*) – many-body length scale hyperparameter.
- **r_cut** (*float*) – many-body cutoff radius.
- **cutoff_func** (*Callable*) – Cutoff function.

Returns Value of the many-body kernel.

Return type float

flare.kernels.mc_mb_sepcut.**many_body_mc_force_en_sepcut_jit**(*q_array_1*, *q_array_2*, *q_neigh_array_1*, *q_neigh_grads_1*, *c1*, *c2*, *etypes1*, *species1*, *species2*, *d1*, *sig*, *ls*, *nspec*, *spec_mask*, *mb_mask*)

many-body many-element kernel between force and energy components accelerated with Numba.

Parameters

- **To be complete**
- **c1** (*int*) – atomic species of the central atom in env 1
- **c2** (*int*) – atomic species of the central atom in env 2
- **etypes1** (*np.ndarray*) – atomic species of atoms in env 1
- **species1** (*np.ndarray*) – all the atomic species present in trajectory 1
- **species2** (*np.ndarray*) – all the atomic species present in trajectory 2
- **d1** (*int*) – Force component of the first environment.
- **sig** (*float*) – many-body signal variance hyperparameter.
- **ls** (*float*) – many-body length scale hyperparameter.

Returns Value of the many-body kernel.

Return type float

`flare.kernels.mc_mb_sepcut.`**`many_body_mc_grad_sepcut_jit`**(*q_array_1*, *q_array_2*, *q_neigh_array_1*, *q_neigh_array_2*, *q_neigh_grads_1*, *q_neigh_grads_2*, *c1*, *c2*, *etypes1*, *etypes2*, *species1*, *species2*, *d1*, *d2*, *sig*, *ls*, *nspec*, *spec_mask*, *nmb*, *mb_mask*)

gradient of many-body multi-element kernel between two force components w.r.t. the hyperparameters, accelerated with Numba.

Parameters

- **c1** (*int*) – atomic species of the central atom in env 1
- **c2** (*int*) – atomic species of the central atom in env 2
- **etypes1** (*np.ndarray*) – atomic species of atoms in env 1
- **etypes2** (*np.ndarray*) – atomic species of atoms in env 2
- **species1** (*np.ndarray*) – all the atomic species present in trajectory 1
- **species2** (*np.ndarray*) – all the atomic species present in trajectory 2
- **d1** (*int*) – Force component of the first environment.
- **d2** (*int*) – Force component of the second environment.
- **sig** (*float*) – many-body signal variance hyperparameter.
- **ls** (*float*) – many-body length scale hyperparameter.

Returns Value of the many-body kernel and its gradient w.r.t. sig and ls

Return type array

flare.kernels.mc_mb_sepcut.**many_body_mc_sepcut_jit**(*q_array_1*, *q_array_2*, *q_neigh_array_1*, *q_neigh_array_2*, *q_neigh_grads_1*, *q_neigh_grads_2*, *c1*, *c2*, *etypes1*, *etypes2*, *species1*, *species2*, *d1*, *d2*, *sig*, *ls*, *nspec*, *spec_mask*, *mb_mask*)

many-body multi-element kernel between two force components accelerated with Numba.

Parameters

- **c1** (*int*) – atomic species of the central atom in env 1
- **c2** (*int*) – atomic species of the central atom in env 2
- **etypes1** (*np.ndarray*) – atomic species of atoms in env 1
- **etypes2** (*np.ndarray*) – atomic species of atoms in env 2
- **species1** (*np.ndarray*) – all the atomic species present in trajectory 1
- **species2** (*np.ndarray*) – all the atomic species present in trajectory 2
- **d1** (*int*) – Force component of the first environment.
- **d2** (*int*) – Force component of the second environment.
- **sig** (*float*) – many-body signal variance hyperparameter.
- **ls** (*float*) – many-body length scale hyperparameter.

Returns Value of the many-body kernel.

Return type float

Cutoff Functions

The cutoffs module gives a few different options for smoothly sending the GP kernel to zero near the boundary of the cutoff sphere.

flare.kernels.cutoffs.**cosine_cutoff**(*r_cut: float*, *ri: float*, *ci: float*, *d: float = 1*)

A cosine cutoff that returns 1 up to r_cut - d, and assigns a cosine envelope to values of r between r_cut - d and r_cut. Based on Eq. 24 of Albert P. Bartók and Gábor Csányi. "Gaussian approximation potentials: A brief tutorial introduction." International Journal of Quantum Chemistry 115.16 (2015): 1051-1057.

Parameters

- **r_cut** (*float*) – Cutoff value (in angstrom).
- **ri** (*float*) – Interatomic distance.
- **ci** (*float*) – Cartesian coordinate divided by the distance.

Returns Cutoff value and its derivative.

Return type (float, float)

flare.kernels.cutoffs.**cubic_cutoff**(*r_cut: float*, *ri: float*, *ci: float*)

A cubic cutoff that goes to zero smoothly at the cutoff boundary.

Parameters

- **r_cut** (*float*) – Cutoff value (in angstrom).

- **ri** (*float*) – Interatomic distance.
- **ci** (*float*) – Cartesian coordinate divided by the distance.

Returns Cutoff value and its derivative.

Return type (float, float)

flare.kernels.cutoffs.**hard_cutoff**(*r_cut: float*, *ri: float*, *ci: float*)
A hard cutoff that assigns a value of 1 to all interatomic distances.

Parameters

- **r_cut** (*float*) – Cutoff value (in angstrom).
- **ri** (*float*) – Interatomic distance.
- **ci** (*float*) – Cartesian coordinate divided by the distance.

Returns Cutoff value and its derivative.

Return type (float, float)

flare.kernels.cutoffs.**quadratic_cutoff**(*r_cut: float*, *ri: float*, *ci: float*)
A quadratic cutoff that goes to zero smoothly at the cutoff boundary.

Parameters

- **r_cut** (*float*) – Cutoff value (in angstrom).
- **ri** (*float*) – Interatomic distance.
- **ci** (*float*) – Cartesian coordinate divided by the distance.

Returns Cutoff value and its derivative.

Return type (float, float)

flare.kernels.cutoffs.**quadratic_cutoff_bound**(*r_cut: float*, *ri: float*, *ci: float*)
A quadratic cutoff that goes to zero smoothly at the cutoff boundary.

Parameters

- **r_cut** (*float*) – Cutoff value (in angstrom).
- **ri** (*float*) – Interatomic distance.
- **ci** (*float*) – Cartesian coordinate divided by the distance.

Returns Cutoff value and its derivative.

Return type (float, float)

Helper Functions

flare.kernels.kernels.**coordination_number**(*rij*, *cij*, *r_cut*, *cutoff_func*)

Pairwise contribution to many-body descriptor based on number of atoms in the environment

Parameters

- **rij** (*float*) – distance between atoms i and j
- **cij** (*float*) – Component of versor of rij along given direction
- **r_cut** (*float*) – cutoff hyperparameter

- **cutoff_func** (*callable*) – cutoff function

Returns the value of the pairwise many-body contribution float: the value of the derivative of the pairwise many-body contribution w.r.t. the central atom displacement

Return type float

flare.kernels.kernels.**force_helper**(*A*, *B*, *C*, *D*, *fi*, *fj*, *fdi*, *fdj*, *ls1*, *ls2*, *ls3*, *sig2*)

Helper function for computing the force/force kernel between two pairs or triplets of atoms of the same type.

See Table IV of the SI of the FLARE paper for definitions of intermediate quantities.

Returns

Force/force kernel between two pairs or triplets of atoms of the same type.

Return type float

flare.kernels.kernels.**k_sq_exp_dev**(*q1*, *q2*, *sig*, *ls*)

First Gradient of generic squared exponential kernel on two many body functions

Parameters

- **q1** (*float*) – the many body descriptor of the first local environment
- **q2** (*float*) – the many body descriptor of the second local environment
- **sig** (*float*) – amplitude hyperparameter
- **ls2** (*float*) – squared lenghtscale hyperparameter

Returns the value of the derivative of the squared exponential kernel

Return type float

flare.kernels.kernels.**k_sq_exp_double_dev**(*q1*, *q2*, *sig*, *ls*)

Second Gradient of generic squared exponential kernel on two many body functions

Parameters

- **q1** (*float*) – the many body descriptor of the first local environment
- **q2** (*float*) – the many body descriptor of the second local environment
- **sig** (*float*) – amplitude hyperparameter
- **ls2** (*float*) – squared lenghtscale hyperparameter

Returns the value of the double derivative of the squared exponential kernel

Return type float

flare.kernels.kernels.**mb_grad_helper_ls**(*q1*, *q2*, *qi*, *qj*, *sig*, *ls*)

Helper function for many body gradient collecting all the derivatives of the force-force many body kernel w.r.t. ls

flare.kernels.kernels.**mb_grad_helper_ls_**(*qdiffsq*, *sig*, *ls*)

Derivative of a many body force-force kernel w.r.t. ls

flare.kernels.kernels.**q_value**(*distances*, *r_cut*, *cutoff_func*, *q_func=<function coordination_number>*)

Compute value of many-body descriptor based on distances of atoms in the local many-body environment.

Parameters

- **distances** (*np.ndarray*) – distances between atoms i and j
- **r_cut** (*float*) – cutoff hyperparameter

- **cutoff_func** (*callable*) – cutoff function
- **q_func** (*callable*) – many-body pairwise descrptor function

Returns the value of the many-body descriptor

Return type float

flare.kernels.kernels.**q_value_mc**(*distances*, *r_cut*, *ref_species*, *species*, *cutoff_func*, *q_func=<function coordination_number>*)

Compute value of many-body many components descriptor based on distances of atoms in the local many-body environment.

Parameters

- **distances** (*np.ndarray*) – distances between atoms i and j
- **r_cut** (*float*) – cutoff hyperparameter
- **ref_species** (*int*) – species to consider to compute the contribution
- **species** (*np.ndarray*) – atomic species of neighbours
- **cutoff_func** (*callable*) – cutoff function
- **q_func** (*callable*) – many-body pairwise descrptor function

Returns the value of the many-body descriptor

Return type float

class flare.gp.**GaussianProcess**(*kernels: List[str] = ['two', 'three'], component: str = 'mc', hyps: ndarray = None, cutoffs={}, hyps_mask: dict = {}, hyp_labels: List[T] = None, opt_algorithm: str = 'L-BFGS-B', maxiter: int = 10, parallel: bool = False, per_atom_par: bool = True, n_cpus: int = 1, n_sample: int = 100, output: flare.output.Output = None, name='default_gp', energy_noise: float = 0.01, **kwargs*)

Gaussian process force field. Implementation is based on Algorithm 2.1 (pg. 19) of "Gaussian Processes for Machine Learning" by Rasmussen and Williams.

Parameters

- **kernels** (*list, optional*) – Determine the type of kernels. Example: ['2', '3'], ['2', '3', 'mb'], ['2']. Defaults to ['2', '3']
- **component** (*str, optional*) – Determine single- ("sc") or multi- component ("mc") kernel to use. Defaults to "mc"
- **hyps** (*np.ndarray, optional*) – Hyperparameters of the GP.
- **cutoffs** (*Dict, optional*) – Cutoffs of the GP kernel.
- **hyp_labels** (*List, optional*) – List of hyperparameter labels. Defaults to None.
- **opt_algorithm** (*str, optional*) – Hyperparameter optimization algorithm. Defaults to 'L-BFGS-B'.
- **maxiter** (*int, optional*) – Maximum number of iterations of the hyperparameter optimization algorithm. Defaults to 10.
- **parallel** (*bool, optional*) – If True, the covariance matrix K of the GP is computed in parallel. Defaults to False.
- **n_cpus** (*int, optional*) – Number of cpus used for parallel calculations. Defaults to 1 (serial)
- **n_sample** (*int, optional*) – Size of submatrix to use when parallelizing predictions.

- **output** (*Output, optional*) – Output object used to dump hyperparameters during optimization. Defaults to None.
- **hyps_mask** (*dict, optional*) – hyps_mask can set up which hyper parameter is used for what interaction. Details see kernels/mc_sephyps.py
- **name** (*str, optional*) – Name for the GP instance.

add_one_env(*env: flare.env.AtomicEnvironment*, *force*, *train: bool = False*, ***kwargs*)
Add a single local environment to the training set of the GP.

Parameters

- **env** (*AtomicEnvironment*) – Local environment to be added to the training set of the GP.
- **force** (*np.ndarray*) – Force on the central atom of the local environment in the form of a 3-component Numpy array containing the x, y, and z components.
- **train** (*bool*) – If True, the GP is trained after the local environment is added.

adjust_cutoffs(*new_cutoffs: Union[list, tuple, np.ndarray], reset_L_alpha=True, train=True, new_hyps_mask=None*)
Loop through atomic environment objects stored in the training data, and re-compute cutoffs for each. Useful if you want to gauge the impact of cutoffs given a certain training set! Unless you know *exactly* what you are doing for some development or test purpose, it is **highly** suggested that you call set_L_alpha and re-optimize your hyperparameters afterwards as is default here.

Parameters **new_cutoffs** –

Returns

as_dict()
Dictionary representation of the GP model.

static backward_arguments(*kwargs*, *new_args={}*)
update the initialize arguments that were renamed

static backward_attributes(*dictionary*)
add new attributes to old instance or update attribute types

check_L_alpha()
Check that the alpha vector is up to date with the training set. If not, update_L_alpha is called.

check_instantiation()
Runs a series of checks to ensure that the user has not supplied contradictory arguments which will result in undefined behavior with multiple hyperparameters. :return:

compute_matrices()
When covariance matrix is known, reconstruct other matrices. Used in re-loading large GPs. :return:

static from_dict(*dictionary*)
Create GP object from dictionary representation.

static from_file(*filename: str*, *format: str = ''*)
One-line convenience method to load a GP from a file stored using write_file

Parameters

- **filename** (*str*) – path to GP model
- **format** (*str*) – json or pickle if format is not in filename

Returns

par
Backwards compability attribute :return:

predict (*x_t: flare.env.AtomicEnvironment*, *d: int*) → [<class 'float'>, <class 'float'>]
Predict a force component of the central atom of a local environment.

Parameters

- **x_t** (*AtomicEnvironment*) – Input local environment.
- **d** (*int*) – Force component to be predicted (1 is x, 2 is y, and 3 is z).

Returns Mean and epistemic variance of the prediction.

Return type (float, float)

predict_efs (*x_t: flare.env.AtomicEnvironment*)
Predict the local energy, forces, and partial stresses of an atomic environment and their predictive variances.

predict_local_energy (*x_t: flare.env.AtomicEnvironment*) → float
Predict the local energy of a local environment.

Parameters **x_t** (*AtomicEnvironment*) – Input local environment.

Returns Local energy predicted by the GP.

Return type float

predict_local_energy_and_var (*x_t: flare.env.AtomicEnvironment*)
Predict the local energy of a local environment and its uncertainty.

Parameters **x_t** (*AtomicEnvironment*) – Input local environment.

Returns Mean and predictive variance predicted by the GP.

Return type (float, float)

remove_force_data (*indexes: Union[int, List[int]], update_matrices: bool = True*) → Tuple[List[<sphinx.ext.autodoc.importer._MockObject object at 0x7fda8018f860>], List[ndarray]]
Remove force components from the model. Convenience function which deletes individual data points.

Matrices should *always* be updated if you intend to use the GP to make predictions afterwards. This might be time consuming for large GPs, so, it is provided as an option, but, only do so with extreme caution. (Undefined behavior may result if you try to make predictions and/or add to the training set afterwards).

Returns training data which was removed akin to a pop method, in order of lowest to highest index passed in.

Parameters

- **indexes** – Indexes of envs in training data to remove.
- **update_matrices** – If false, will not update the GP's matrices afterwards (which can be time consuming for large models). This should essentially always be true except for niche development applications.

Returns

set_L_alpha ()
Invert the covariance matrix, setting L (a lower triangular matrix s.t. L L^T = (K + sig_n^2 I)) and alpha, the inverse covariance matrix multiplied by the vector of training labels. The forces and variances are later obtained using alpha.

train (*logger=None*, *custom_bounds=None*, *grad_tol: float = 0.0001*, *x_tol: float = 1e-05*, *line_steps: int = 20*, *print_progress: bool = False*)
Train Gaussian Process model on training data. Tunes the hyperparameters to maximize the likelihood, then computes L and alpha (related to the covariance matrix of the training set).

Parameters

- **logger** (*logging.logger*) – logger object specifying where to write the progress of the optimization.
- **custom_bounds** (*np.ndarray*) – Custom bounds on the hyperparameters.
- **grad_tol** (*float*) – Tolerance of the hyperparameter gradient that determines when hyperparameter optimization is terminated.
- **x_tol** (*float*) – Tolerance on the x values used to decide when Nelder-Mead hyperparameter optimization is terminated.
- **line_steps** (*int*) – Maximum number of line steps for L-BFGS hyperparameter optimization.

training_statistics
Return a dictionary with statistics about the current training data. Useful for quickly summarizing info about the GP. :return:

update_L_alpha()
Update the GP's L matrix and alpha vector without recalculating the entire covariance matrix K.

update_db(*struc: <sphinx.ext.autodoc.importer._MockObject object at 0x7fda8018f860>, forces: List[T], custom_range: List[int] = (), energy: float = None*)
Given a structure and forces, add local environments from the structure to the training set of the GP. If energy is given, add the entire structure to the training set.

Parameters

- **struc** (*Structure*) – Input structure. Local environments of atoms in this structure will be added to the training set of the GP.
- **forces** (*np.ndarray*) – Forces on atoms in the structure.
- **custom_range** (*List[int]*) – Indices of atoms whose local environments will be added to the training set of the GP.
- **energy** (*float*) – Energy of the structure.

write_model(*name: str, format: str = None, split_matrix_size_cutoff: int = 5000*)
Write model in a variety of formats to a file for later re-use. JSON files are open to visual inspection and are easier to use across different versions of FLARE or GP implementations. However, they are larger and loading them in takes longer (by setting up a new GP from the specifications). Pickled files can be faster to read & write, and they take up less memory.

Parameters

- **name** (*str*) – Output name.
- **format** (*str*) – Output format.
- **split_matrix_size_cutoff** (*int*) – If there are more than this
- **number of training points in the set, save the matrices seperately.**

1.3.1.4 Predict

Helper functions which obtain forces and energies corresponding to atoms in structures. These functions automatically cast atoms into their respective atomic environments.

flare.predict.**predict_on_atom**(*param: Tuple[<sphinx.ext.autodoc.importer._MockObject object at 0x7fda8018f860>, int, flare.gp.GaussianProcess]) -> ('np.ndarray', 'np.ndarray')*

Return the forces/std. dev. uncertainty associated with an individual atom in a structure, without necessarily having cast it to a chemical environment. In order to work with other functions, all arguments are passed in as a tuple.

Parameters **param** (*Tuple(Structure, integer, GaussianProcess)*) – tuple of FLARE Structure, atom index, and Gaussian Process object

Returns 3-element force array and associated uncertainties

Return type (np.ndarray, np.ndarray)

flare.predict.**predict_on_atom_efs**(*param*)

Predict the local energy, forces, and partial stresses and predictive variances of a chemical environment.

flare.predict.**predict_on_atom_en**(*param: Tuple[<sphinx.ext.autodoc.importer._MockObject object at 0x7fda8018f860>, int, flare.gp.GaussianProcess]) -> ('np.ndarray', 'np.ndarray', <class 'float'>)*

Return the forces/std. dev. uncertainty / energy associated with an individual atom in a structure, without necessarily having cast it to a chemical environment. In order to work with other functions, all arguments are passed in as a tuple.

Parameters **param** (*Tuple(Structure, integer, GaussianProcess)*) – tuple of FLARE Structure, atom index, and Gaussian Process object

Returns 3-element force array, associated uncertainties, and local energy

Return type (np.ndarray, np.ndarray, float)

flare.predict.**predict_on_atom_en_std**(*param*)

Predict local energy and predictive std of a chemical environment.

flare.predict.**predict_on_structure**(*structure: <sphinx.ext.autodoc.importer._MockObject object at 0x7fda8018f860>, gp: flare.gp.GaussianProcess, n_cpus: int = None, write_to_structure: bool = True, selective_atoms: List[int] = None, skipped_atom_value=0) -> ('np.ndarray', 'np.ndarray')*

Return the forces/std. dev. uncertainty associated with each individual atom in a structure. Forces are stored directly to the structure and are also returned.

Parameters

- **structure** – FLARE structure to obtain forces for, with N atoms
- **gp** – Gaussian Process model
- **write_to_structure** – Write results to structure's forces, std attributes
- **selective_atoms** – Only predict on these atoms; e.g. [0,1,2] will only predict and return for those atoms
- **skipped_atom_value** – What value to use for atoms that are skipped. Defaults to 0 but other options could be e.g. NaN. Will NOT write this to the structure if write_to_structure is True.

Returns N x 3 numpy array of foces, Nx3 numpy array of uncertainties

Return type (np.ndarray, np.ndarray)

flare.predict.**predict_on_structure_en**(*structure: <sphinx.ext.autodoc.importer._MockObject object at 0x7fda8018f860>, gp: flare.gp.GaussianProcess, n_cpus: int = None, write_to_structure: bool = True, selective_atoms: List[int] = None, skipped_atom_value=0) -> ('np.ndarray', 'np.ndarray', 'np.ndarray')*

Return the forces/std. dev. uncertainty / local energy associated with each individual atom in a structure. Forces are stored directly to the structure and are also returned.

Parameters

- **structure** – FLARE structure to obtain forces for, with N atoms
- **gp** – Gaussian Process model
- **n_cpus** – Dummy parameter passed as an argument to allow for flexibility when the callable may or may not be parallelized

Returns N x 3 array of forces, N x 3 array of uncertainties, N-length array of energies

Return type (np.ndarray, np.ndarray, np.ndarray)

flare.predict.**predict_on_structure_mgp**(*structure, mgp, output=None, output_name=None, n_cpus=None, write_to_structure=True, selective_atoms: List[int] = None, skipped_atom_value=0*)

Assign forces to structure based on an mgp

flare.predict.**predict_on_structure_par**(*structure: <sphinx.ext.autodoc.importer._MockObject object at 0x7fda8018f860>, gp: flare.gp.GaussianProcess, n_cpus: int = None, write_to_structure: bool = True, selective_atoms: List[int] = None, skipped_atom_value=0) -> ('np.ndarray', 'np.ndarray')*

Return the forces/std. dev. uncertainty associated with each individual atom in a structure. Forces are stored directly to the structure and are also returned.

Parameters

- **structure** – FLARE structure to obtain forces for, with N atoms
- **gp** – Gaussian Process model
- **n_cpus** – Number of cores to parallelize over
- **write_to_structure** – Write results to structure's forces, std attributes
- **selective_atoms** – Only predict on these atoms; e.g. [0,1,2] will only predict and return for those atoms
- **skipped_atom_value** – What value to use for atoms that are skipped. Defaults to 0 but other options could be e.g. NaN. Will NOT write this to the structure if write_to_structure is True.

Returns N x 3 array of forces, N x 3 array of uncertainties

Return type (np.ndarray, np.ndarray)

flare.predict.**predict_on_structure_par_en**(*structure: <sphinx.ext.autodoc.importer._MockObject object at 0x7fda8018f860>, gp: flare.gp.GaussianProcess, n_cpus: int = None, write_to_structure: bool = True, selective_atoms: List[int] = None, skipped_atom_value=0) -> ('np.ndarray', 'np.ndarray', 'np.ndarray')*

Return the forces/std. dev. uncertainty / local energy associated with each individual atom in a structure, parallelized over atoms. Forces are stored directly to the structure and are also returned.

Parameters

- **structure** – FLARE structure to obtain forces for, with N atoms
- **gp** – Gaussian Process model
- **n_cpus** – Number of cores to parallelize over

Returns N x 3 array of forces, N x 3 array of uncertainties, N-length array of energies

Return type (np.ndarray, np.ndarray, np.ndarray)

1.3.1.5 Helper functions for GP

flare.gp_algebra.**efs_energy_vector**(*name, efs_energy_kernel, x, hyps, cutoffs=None, hyps_mask=None, n_cpus=1, n_sample=100*)

Returns covariances between the local eneregy, force components, and partial stresses of a test environment and the total energy labels in the training set.

flare.gp_algebra.**efs_force_vector**(*name, efs_force_kernel, x, hyps, cutoffs=None, hyps_mask=None, n_cpus=1, n_sample=100*)

Returns covariances between the local eneregy, force components, and partial stresses of a test environment and the force labels in the training set.

flare.gp_algebra.**energy_energy_vector**(*name, kernel, x, hyps, cutoffs=None, hyps_mask=None, n_cpus=1, n_sample=100*)

Get a vector of covariances between the local energy of a test environment and the total energy labels in the training set.

flare.gp_algebra.**energy_energy_vector_unit**(*name, s, e, x, kernel, hyps, cutoffs=None, hyps_mask=None, d_1=None*)

Gets part of the energy/energy vector.

flare.gp_algebra.**energy_force_vector**(*name, kernel, x, hyps, cutoffs=None, hyps_mask=None, n_cpus=1, n_sample=100*)

Get the vector of covariances between the local energy of a test environment and the force labels in the training set.

flare.gp_algebra.**energy_force_vector_unit**(*name, s, e, x, kernel, hyps, cutoffs=None, hyps_mask=None, d_1=None*)

Gets part of the energy/force vector.

flare.gp_algebra.**force_energy_vector**(*name, kernel, x, d_1, hyps, cutoffs=None, hyps_mask=None, n_cpus=1, n_sample=100*)

Get a vector of covariances between a force component of a test environment and the total energy labels in the training set.

flare.gp_algebra.**force_energy_vector_unit**(*name, s, e, x, kernel, hyps, cutoffs, hyps_mask, d_1*)

Gets part of the force/energy vector.

flare.gp_algebra.**force_force_vector**(*name*, *kernel*, *x*, *d_1*, *hyps*, *cutoffs=None*, *hyps_mask=None*, *n_cpus=1*, *n_sample=100*)

Get a vector of covariances between a force component of a test environment and the force labels in the training set.

flare.gp_algebra.**force_force_vector_unit**(*name*, *s*, *e*, *x*, *kernel*, *hyps*, *cutoffs*, *hyps_mask*, *d_1*)

Gets part of the force/force vector.

flare.gp_algebra.**get_force_block**(*hyps: <sphinx.ext.autodoc.importer._MockObject object at 0x7fda802e1630>*, *name: str*, *kernel*, *cutoffs=None*, *hyps_mask=None*, *n_cpus=1*, *n_sample=100*)

parallel version of get_ky_mat :param hyps: list of hyper-parameters :param name: name of the gp instance. :param kernel: function object of the kernel :param cutoffs: The cutoff values used for the atomic environments :type cutoffs: list of 2 float numbers :param hyps_mask: dictionary used for multi-group hyperparmeters

Returns covariance matrix

flare.gp_algebra.**get_force_block_pack**(*hyps: <sphinx.ext.autodoc.importer._MockObject object at 0x7fda802e16d8>*, *name: str*, *s1: int*, *e1: int*, *s2: int*, *e2: int*, *same: bool*, *kernel*, *cutoffs*, *hyps_mask*)

Compute covariance matrix element between set1 and set2 :param hyps: list of hyper-parameters :param name: name of the gp instance. :param same: whether the row and column are the same :param kernel: function object of the kernel :param cutoffs: The cutoff values used for the atomic environments :type cutoffs: list of 2 float numbers :param hyps_mask: dictionary used for multi-group hyperparmeters

Returns covariance matrix

flare.gp_algebra.**get_ky_and_hyp**(*hyps: <sphinx.ext.autodoc.importer._MockObject object at 0x7fda804ec0f0>*, *name*, *kernel_grad*, *cutoffs=None*, *hyps_mask=None*, *n_cpus=1*, *n_sample=100*)

parallel version of get_ky_and_hyp

Parameters

- **hyps** – list of hyper-parameters
- **name** – name of the gp instance.
- **kernel_grad** – function object of the kernel gradient
- **cutoffs** (*list of 2 float numbers*) – The cutoff values used for the atomic environments
- **hyps_mask** – dictionary used for multi-group hyperparmeters
- **n_cpus** – number of cpus to use.
- **n_sample** – the size of block for matrix to compute

Returns hyp_mat, ky_mat

flare.gp_algebra.**get_ky_and_hyp_pack**(*name*, *s1*, *e1*, *s2*, *e2*, *same: bool*, *hyps: <sphinx.ext.autodoc.importer._MockObject object at 0x7fda804ec390>*, *kernel_grad*, *cutoffs=None*, *hyps_mask=None*)

computes a block of ky matrix and its derivative to hyper-parameter If the cpu set up is None, it uses as much as posible cpus

Parameters

- **hyps** – list of hyper-parameters
- **name** – name of the gp instance.

- **kernel_grad** – function object of the kernel gradient
- **cutoffs** (*list of 2 float numbers*) – The cutoff values used for the atomic environments
- **hyps_mask** – dictionary used for multi-group hyperparmeters

Returns hyp_mat, ky_mat

flare.gp_algebra.**get_like_from_mats**(*ky_mat*, *l_mat*, *alpha*, *name*)
compute the likelihood from the covariance matrix

Parameters **ky_mat** – the covariance matrix

Returns float, likelihood

flare.gp_algebra.**get_like_grad_from_mats**(*ky_mat*, *hyp_mat*, *name*)
compute the gradient of likelihood to hyper-parameters from covariance matrix and its gradient

Parameters

- **ky_mat** (*np.array*) – covariance matrix
- **hyp_mat** (*np.array*) – dky/d(hyper parameter) matrix
- **name** – name of the gp instance.

Returns float, list. the likelihood and its gradients

flare.gp_algebra.**get_neg_like_grad**(*hyps: <sphinx.ext.autodoc.importer._MockObject object at 0x7fda804ec240>*, *name: str*, *kernel_grad*, *logger=None*, *cutoffs=None*, *hyps_mask=None*, *n_cpus=1*, *n_sample=100*)
compute the log likelihood and its gradients

Parameters

- **hyps** (*np.ndarray*) – list of hyper-parameters
- **name** – name of the gp instance.
- **kernel_grad** – function object of the kernel gradient
- **output** (*logger*) – Output object for dumping every hyper-parameter sets computed
- **cutoffs** (*list of 2 float numbers*) – The cutoff values used for the atomic environments
- **hyps_mask** – dictionary used for multi-group hyperparmeters
- **n_cpus** – number of cpus to use.
- **n_sample** – the size of block for matrix to compute

Returns float, np.array

flare.gp_algebra.**obtain_noise_len**(*hyps*, *hyps_mask*)
obtain the noise parameter from hyps and mask

flare.gp_algebra.**partition_force_energy_block**(*n_sample: int*, *size1: int*, *size2: int*, *n_cpus: int*)
Special partition method for the force/energy block. Because the number of environments in a structure can vary, we only split up the environment list, which has length size1.

Note that two sizes need to be specified: the size of the envionment list and the size of the structure list.

Parameters

- **n_sample** (*int*) – Number of environments per processor.
- **size1** (*int*) – Size of the environment list.
- **size2** (*int*) – Size of the structure list.
- **n_cpus** (*int*) – Number of cpus.

flare.gp_algebra.**partition_matrix**(*n_sample*, *size*, *n_cpus*)
partition the training data for matrix calculation the number of blocks are close to n_cpus since mp.Process does not allow to change the thread number

flare.gp_algebra.**partition_matrix_custom**(*n_sample: int*, *start1*, *end1*, *start2*, *end2*, *n_cpus*)
Partition a specified portion of a matrix.

flare.gp_algebra.**partition_vector**(*n_sample*, *size*, *n_cpus*)
partition the training data for vector calculation the number of blocks are the same as n_cpus since mp.Process does not allow to change the thread number

flare.gp_algebra.**queue_wrapper**(*result_queue*, *wid*, *func*, *args*)
wrapper function for multiprocessing queue

1.3.1.6 Output

Class which contains various methods to print the output of different ways of using FLARE, such as training a GP from an AIMD run, or running an MD simulation updated on-the-fly.

class flare.output.**Output**(*basename: str = 'otf_run'*, *verbose: str = 'INFO'*, *always_flush: bool = False*)
This is an I/O class that hosts the log files for OTF and Trajectories class. It is also used in get_neg_like_grad and get_neg_likelihood in gp_algebra to print intermediate results.

It opens and print files with the basename prefix and different suffixes corresponding to different kinds of output data.

Parameters

- **basename** (*str, optional*) – Base output file name, suffixes will be added
- **verbose** (*str, optional*) – print level. The same as logging level. It can be CRITICAL, ERROR, WARNING, INFO, DEBUG, NOTSET
- **always_flush** – Always write to file instantly

conclude_run()
destruction function that closes all files

open_new_log(*filetype: str*, *suffix: str*, *verbose='info'*)
Open files. If files with the same name are exist, they are backed up with a suffix "-bak".

Parameters

- **filetype** – the key name for logging
- **suffix** – the suffix of the file to be opened
- **verbose** – the verbose level for the logger

write_gp_dft_comparison(*curr_step*, *frame*, *start_time*, *dft_forces*, *error*, *local_energies=None*, *KE=None*, *mgp=False*)
Write the comparison to logfile.

Parameters

- **curr_step** – current timestep

- **frame** – Structure object that contains the current GP calculation results.
- **start_time** – start time for time profiling
- **dft_forces** – list of forces computed by DFT
- **error** – list of force differences between DFT and GP prediction
- **local_energies** – local atomic energy
- **KE** – total kinetic energy

Returns

write_header(*gp_str: str*, *dt: float = None*, *Nsteps: int = None*, *structure: <sphinx.ext.autodoc.importer._MockObject object at 0x7fda8018f860> = None*, *std_tolerance: Union[float, int] = None*, *optional: dict = None*)

TO DO: this should be replace by the string method of GP and OTF, GPFA

Write header to the log function. Designed for Trajectory Trainer and OTF runs and can take flexible input for both.

Parameters

- **gp_str** – string representation of the GP
- **dt** – timestep for OTF MD
- **Nsteps** – total number of steps for OTF MD
- **structure** – initial structure
- **std_tolerance** – tolarence for active learning
- **optional** – a dictionary of all the other parameters

write_hyps(*hyp_labels*, *hyps*, *start_time*, *like*, *like_grad*, *name='log'*, *hyps_mask=None*)

write hyperparameters to logfile

Parameters

- **name** –
- **hyp_labels** – labels for hyper-parameters. can be None
- **hyps** – list of hyper-parameters
- **start_time** – start time for time profiling
- **like** – likelihood
- **like_grad** – gradient of likelihood

Returns

write_md_config(*dt*, *curr_step*, *structure*, *temperature*, *KE*, *start_time*, *dft_step*, *velocities*)

write md configuration in log file

Parameters

- **dt** – timestemp of OTF MD
- **curr_step** – current timestep of OTF MD
- **structure** – atomic structure
- **temperature** – current temperature
- **KE** – current total kinetic energy

- **local_energies** – local energy
- **start_time** – starting time for time profiling
- **dft_step** – # of DFT calls
- **velocities** – list of velocities

Returns

write_to_log(*logstring: str*, *name: str = 'log'*, *flush: bool = False*)
Write any string to logfile

Parameters

- **logstring** – the string to write
- **name** – the key name of the file to logger named 'log'
- **flush** – whether it should be flushed

write_xyz(*curr_step: int*, *pos: <sphinx.ext.autodoc.importer._MockObject object at 0x7fda8056f128>*, *species: list*, *filename: str*, *header=''*, *forces: <sphinx.ext.autodoc.importer._MockObject object at 0x7fda8056f898> = None*, *stds: <sphinx.ext.autodoc.importer._MockObject object at 0x7fda804ec208> = None*, *forces_2: <sphinx.ext.autodoc.importer._MockObject object at 0x7fda804ece48> = None*)
write atomic configuration in xyz file

Parameters

- **curr_step** – Int, number of frames to note in the comment line
- **pos** – nx3 matrix of forces, positions, or nything
- **species** – n element list of symbols
- **filename** – key of logger
- **header** – header in comments
- **forces** – list of forces on atoms predicted by GP
- **stds** – uncertainties predicted by GP
- **forces_2** – true forces from ab initio source

write_xyz_config(*curr_step*, *structure*, *dft_step*, *forces: <sphinx.ext.autodoc.importer._MockObject object at 0x7fda804ecbe0> = None*, *stds: <sphinx.ext.autodoc.importer._MockObject object at 0x7fda804ec128> = None*, *forces_2: <sphinx.ext.autodoc.importer._MockObject object at 0x7fda804ecd68> = None*)
write atomic configuration in xyz file

Parameters

- **curr_step** – Int, number of frames to note in the comment line
- **structure** – Structure, contain positions and forces
- **dft_step** – Boolean, whether this is a DFT call.
- **forces** – Optional list of forces to xyz file
- **stds** – Optional list of uncertanties to xyz file
- **forces_2** – Optional second list of forces (e.g. DFT forces)

Returns

flare.output.**add_file**(*logger: logging.Logger*, *filename: str*, *verbose: str = 'info'*)
set up file handler to the logger with handlers

Parameters

- **logger** – the logger
- **filename** (*str*) – name of the logfile
- **verbose** (*str*) – verbose level

flare.output.**add_stream**(*logger: logging.Logger*, *verbose: str = 'info'*)
set up screen sctream handler to the logger with handlers

Parameters

- **logger** – the logger
- **verbose** (*str*) – verbose level

flare.output.**set_logger**(*name: str*, *stream: bool*, *fileout_name: str = None*, *verbose: str = 'info'*)
set up a logger with handlers

Parameters

- **name** (*str*) – unique name of the logger in logging module
- **stream** (*bool*) – if True, set up a screen output
- **fileout_name** (*str*) – name for log file
- **verbose** (*str*) – verbose level

class flare.gp.**GaussianProcess**(*kernels: List[str] = ['two', 'three']*, *component: str = 'mc'*, *hyps: ndarray = None*, *cutoffs={}*, *hyps_mask: dict = {}*, *hyp_labels: List[T] = None*, *opt_algorithm: str = 'L-BFGS-B'*, *maxiter: int = 10*, *parallel: bool = False*, *per_atom_par: bool = True*, *n_cpus: int = 1*, *n_sample: int = 100*, *output: flare.output.Output = None*, *name='default_gp'*, *energy_noise: float = 0.01*, ***kwargs*)
Gaussian process force field. Implementation is based on Algorithm 2.1 (pg. 19) of "Gaussian Processes for Machine Learning" by Rasmussen and Williams.

Parameters

- **kernels** (*list, optional*) – Determine the type of kernels. Example: ['2', '3'], ['2', '3', 'mb'], ['2']. Defaults to ['2', '3']
- **component** (*str, optional*) – Determine single- ("sc") or multi- component ("mc") kernel to use. Defaults to "mc"
- **hyps** (*np.ndarray, optional*) – Hyperparameters of the GP.
- **cutoffs** (*Dict, optional*) – Cutoffs of the GP kernel.
- **hyp_labels** (*List, optional*) – List of hyperparameter labels. Defaults to None.
- **opt_algorithm** (*str, optional*) – Hyperparameter optimization algorithm. Defaults to 'L-BFGS-B'.
- **maxiter** (*int, optional*) – Maximum number of iterations of the hyperparameter optimization algorithm. Defaults to 10.
- **parallel** (*bool, optional*) – If True, the covariance matrix K of the GP is computed in parallel. Defaults to False.
- **n_cpus** (*int, optional*) – Number of cpus used for parallel calculations. Defaults to 1 (serial)

- **n_sample** (*int, optional*) – Size of submatrix to use when parallelizing predictions.
- **output** (*Output, optional*) – Output object used to dump hyperparameters during optimization. Defaults to None.
- **hyps_mask** (*dict, optional*) – hyps_mask can set up which hyper parameter is used for what interaction. Details see kernels/mc_sephyps.py
- **name** (*str, optional*) – Name for the GP instance.

add_one_env(*env: flare.env.AtomicEnvironment*, *force*, *train: bool = False*, ***kwargs*)
Add a single local environment to the training set of the GP.

Parameters

- **env** (*AtomicEnvironment*) – Local environment to be added to the training set of the GP.
- **force** (*np.ndarray*) – Force on the central atom of the local environment in the form of a 3-component Numpy array containing the x, y, and z components.
- **train** (*bool*) – If True, the GP is trained after the local environment is added.

adjust_cutoffs(*new_cutoffs: Union[list, tuple, np.ndarray], reset_L_alpha=True, train=True, new_hyps_mask=None*)
Loop through atomic environment objects stored in the training data, and re-compute cutoffs for each. Useful if you want to gauge the impact of cutoffs given a certain training set! Unless you know *exactly* what you are doing for some development or test purpose, it is **highly** suggested that you call set_L_alpha and re-optimize your hyperparameters afterwards as is default here.

Parameters **new_cutoffs** –

Returns

as_dict()
Dictionary representation of the GP model.

static backward_arguments(*kwargs*, *new_args={}*)
update the initialize arguments that were renamed

static backward_attributes(*dictionary*)
add new attributes to old instance or update attribute types

check_L_alpha()
Check that the alpha vector is up to date with the training set. If not, update_L_alpha is called.

check_instantiation()
Runs a series of checks to ensure that the user has not supplied contradictory arguments which will result in undefined behavior with multiple hyperparameters. :return:

compute_matrices()
When covariance matrix is known, reconstruct other matrices. Used in re-loading large GPs. :return:

static from_dict(*dictionary*)
Create GP object from dictionary representation.

static from_file(*filename: str*, *format: str = ''*)
One-line convenience method to load a GP from a file stored using write_file

Parameters

- **filename** (*str*) – path to GP model
- **format** (*str*) – json or pickle if format is not in filename

Returns

par
Backwards compability attribute :return:

predict(*x_t: flare.env.AtomicEnvironment*, *d: int*) → [<class 'float'>, <class 'float'>]
Predict a force component of the central atom of a local environment.

Parameters

- **x_t** (*AtomicEnvironment*) – Input local environment.
- **d** (*int*) – Force component to be predicted (1 is x, 2 is y, and 3 is z).

Returns Mean and epistemic variance of the prediction.

Return type (float, float)

predict_efs(*x_t: flare.env.AtomicEnvironment*)
Predict the local energy, forces, and partial stresses of an atomic environment and their predictive variances.

predict_local_energy(*x_t: flare.env.AtomicEnvironment*) → float
Predict the local energy of a local environment.

Parameters **x_t** (*AtomicEnvironment*) – Input local environment.

Returns Local energy predicted by the GP.

Return type float

predict_local_energy_and_var(*x_t: flare.env.AtomicEnvironment*)
Predict the local energy of a local environment and its uncertainty.

Parameters **x_t** (*AtomicEnvironment*) – Input local environment.

Returns Mean and predictive variance predicted by the GP.

Return type (float, float)

remove_force_data(*indexes: Union[int, List[int]]*, *update_matrices: bool = True*) → Tuple[List[<sphinx.ext.autodoc.importer._MockObject object at 0x7fda8018f860>], List[ndarray]]
Remove force components from the model. Convenience function which deletes individual data points.

Matrices should *always* be updated if you intend to use the GP to make predictions afterwards. This might be time consuming for large GPs, so, it is provided as an option, but, only do so with extreme caution. (Undefined behavior may result if you try to make predictions and/or add to the training set afterwards).

Returns training data which was removed akin to a pop method, in order of lowest to highest index passed in.

Parameters

- **indexes** – Indexes of envs in training data to remove.
- **update_matrices** – If false, will not update the GP's matrices afterwards (which can be time consuming for large models). This should essentially always be true except for niche development applications.

Returns

set_L_alpha()
Invert the covariance matrix, setting L (a lower triangular matrix s.t. L L^T = (K + sig_n^2 I)) and alpha, the inverse covariance matrix multiplied by the vector of training labels. The forces and variances are later obtained using alpha.

train(*logger=None*, *custom_bounds=None*, *grad_tol: float = 0.0001*, *x_tol: float = 1e-05*, *line_steps: int = 20*, *print_progress: bool = False*)

Train Gaussian Process model on training data. Tunes the hyperparameters to maximize the likelihood, then computes L and alpha (related to the covariance matrix of the training set).

Parameters

- **logger** (*logging.logger*) – logger object specifying where to write the progress of the optimization.
- **custom_bounds** (*np.ndarray*) – Custom bounds on the hyperparameters.
- **grad_tol** (*float*) – Tolerance of the hyperparameter gradient that determines when hyperparameter optimization is terminated.
- **x_tol** (*float*) – Tolerance on the x values used to decide when Nelder-Mead hyperparameter optimization is terminated.
- **line_steps** (*int*) – Maximum number of line steps for L-BFGS hyperparameter optimization.

training_statistics

Return a dictionary with statistics about the current training data. Useful for quickly summarizing info about the GP. :return:

update_L_alpha()

Update the GP's L matrix and alpha vector without recalculating the entire covariance matrix K.

update_db(*struc: <sphinx.ext.autodoc.importer._MockObject object at 0x7fda8018f860>*, *forces: List[T]*, *custom_range: List[int] = ()*, *energy: float = None*)

Given a structure and forces, add local environments from the structure to the training set of the GP. If energy is given, add the entire structure to the training set.

Parameters

- **struc** (*Structure*) – Input structure. Local environments of atoms in this structure will be added to the training set of the GP.
- **forces** (*np.ndarray*) – Forces on atoms in the structure.
- **custom_range** (*List[int]*) – Indices of atoms whose local environments will be added to the training set of the GP.
- **energy** (*float*) – Energy of the structure.

write_model(*name: str*, *format: str = None*, *split_matrix_size_cutoff: int = 5000*)

Write model in a variety of formats to a file for later re-use. JSON files are open to visual inspection and are easier to use across different versions of FLARE or GP implementations. However, they are larger and loading them in takes longer (by setting up a new GP from the specifications). Pickled files can be faster to read & write, and they take up less memory.

Parameters

- **name** (*str*) – Output name.
- **format** (*str*) – Output format.
- **split_matrix_size_cutoff** (*int*) – If there are more than this
- **number of training points in the set, save the matrices seperately.**

1.3.2 On-the-Fly Training

1.3.2.1 DFT Interface

Quantum Espresso

This module is used to call Quantum Espresso simulation and parse its output The user need to supply a complete input script with single-point scf calculation, CELL_PARAMETERS, ATOMIC_POSITIONS, nat, ATOMIC_SPECIES arguments. It is case sensitive. and the nat line should be the first argument of the line it appears. The user can also opt to the ASE interface instead.

This module will copy the input template to a new file with "_run" suffix, edit the atomic coordination in the ATOMIC_POSITIONS block and run the similation with the parallel set up given.

`flare.dft_interface.qe_util.`**`dft_input_to_structure`**(*dft_input: str*)
Parses a qe input and returns the atoms in the file as a Structure object

Parameters **`dft_input`** – QE Input file to parse

Returns class Structure

`flare.dft_interface.qe_util.`**`edit_dft_input_positions`**(*dft_input: str*, *structure*)
Write the current configuration of the OTF structure to the qe input file

Parameters

- **`dft_input`** – dft input file name
- **`structure`** – atomic structure to compute

Returns the name of the edited file

`flare.dft_interface.qe_util.`**`parse_dft_forces`**(*outfile: str*)
Get forces from a pwscf file in eV/A

Parameters **`outfile`** – str, Path to pwscf output file

Returns list[nparray] , List of forces acting on atoms

`flare.dft_interface.qe_util.`**`parse_dft_forces_and_energy`**(*outfile: str*)
Get forces from a pwscf file in eV/A

Parameters **`outfile`** – str, Path to pwscf output file

Returns list[nparray] , List of forces acting on atoms

`flare.dft_interface.qe_util.`**`parse_dft_input`**(*dft_input: str*)
parse the input to get information of atomic configuration

Parameters **`dft_input`** – input file name

Returns positions, species, cell, masses

`flare.dft_interface.qe_util.`**`run_dft_en_par`**(*dft_input*, *structure*, *dft_loc*, *n_cpus*)
run DFT calculation with given input template and atomic configurations. This function is not used atm

if n_cpus == 1, it executes serial run.

Parameters

- **`dft_input`** – input template file name
- **`structure`** – atomic configuration
- **`dft_loc`** – relative/absolute executable of the DFT code

- **n_cpus** – # of CPU for mpi
- **dft_out** – output file name
- **npool** – not used
- **mpi** – not used
- ****dft_wargs** – not used

Returns forces, energy

flare.dft_interface.qe_util.**run_dft_par**(*dft_input*, *structure*, *dft_loc*, *n_cpus=1*, *dft_out='pwscf.out'*, *npool=None*, *mpi='mpi'*, ***dft_kwargs*)

run DFT calculation with given input template and atomic configurations. if n_cpus == 1, it executes serial run.

Parameters

- **dft_input** – input template file name
- **structure** – atomic configuration
- **dft_loc** – relative/absolute executable of the DFT code
- **n_cpus** – # of CPU for mpi
- **dft_out** – output file name
- **npool** – not used
- **mpi** – not used
- ****dft_wargs** – not used

Returns forces

CP2K

This module is used to call CP2K simulation and parse its output The user need to supply a complete input script with ENERGY_FORCE or ENERGY runtype, and CELL, COORD blocks. Example scripts can be found in tests/test_files/cp2k_input...

The module will copy the input template to a new file with "_run" suffix, edit the atomic coordination in the COORD blocks and run the similation with the parallel set up given.

We note that, if the CP2K executable is only for serial run, using it along with MPI setting can lead to repeating output in the output file, wrong number of forces and error in the other modules.

flare.dft_interface.cp2k_util.**dft_input_to_structure**(*dft_input: str*)

Parses a qe input and returns the atoms in the file as a Structure object :param dft_input: input file to parse :return: atomic structure

flare.dft_interface.cp2k_util.**edit_dft_input_positions**(*dft_input: str*, *structure*)

Write the current configuration of the OTF structure to the qe input file

Parameters

- **dft_input** – intput file name
- **structure** (*class Structure*) – structure to print

Return newfilename the name of the edited intput file. with "_run" suffix

flare.dft_interface.cp2k_util.**parse_dft_forces**(*outfile: str*)

Get forces from a pwscf file in eV/A

Parameters **outfile** – str, Path to dft.output file

Returns list[nparray] , List of forces acting on atoms

flare.dft_interface.cp2k_util.**parse_dft_forces_and_energy**(*outfile: str*)
Get forces from a pwscf file in eV/A the input run type to be ENERGY_FORCE

Parameters **outfile** – str, Path to dft.output file

Returns list[nparray] , List of forces acting on atoms

Returns float, total potential energy

flare.dft_interface.cp2k_util.**parse_dft_input**(*dft_input: str*)
Parse CP2K input file prepared by the user the parser is very limited. The user have to define things in a good format. It requires the "CELL", "COORD" blocks

Parameters **dft_input** – file name

Returns positions, species, cell, masses

flare.dft_interface.cp2k_util.**run_dft_en_par**(*dft_input: str*, *structure*, *dft_loc: str*, *ncpus: int*, *dft_out: str = 'dft.out'*, *npool: int = None*, *mpi: str = 'mpi'*, ***dft_kwargs*)
run DFT calculation with given input template and atomic configurations. This function is not used atm.

Parameters

- **dft_input** – input template file name
- **structure** – atomic configuration
- **dft_loc** – relative/absolute executable of the DFT code
- **ncpus** – # of CPU for mpi
- **dft_out** – output file name
- **npool** – not used
- **mpi** – not used
- ****dft_wargs** – not used

Returns forces, energy

flare.dft_interface.cp2k_util.**run_dft_par**(*dft_input*, *structure*, *dft_loc*, *ncpus=1*, *dft_out='dft.out'*, *npool=None*, *mpi='mpi'*, ***dft_kwargs*)
run DFT calculation with given input template and atomic configurations. if ncpus == 1, it executes serial run.

Parameters

- **dft_input** – input template file name
- **structure** – atomic configuration
- **dft_loc** – relative/absolute executable of the DFT code
- **ncpus** – # of CPU for mpi
- **dft_out** – output file name
- **npool** – not used
- **mpi** – not used
- ****dft_wargs** – not used

Returns forces

VASP

flare.dft_interface.vasp_util.**check_vasprun**(*vasprun: Union[str, <sphinx.ext.autodoc.importer._MockObject object at 0x7fda804f1710>], vasprun_kwargs: dict = {}*) → <sphinx.ext.autodoc.importer._MockObject object at 0x7fda804f1710>

Helper utility to take a vasprun file name or Vasprun object and return a vasprun object. :param vasprun: vasprun filename or object

flare.dft_interface.vasp_util.**dft_input_to_structure**(*poscar: str*)

Parse the DFT input in a directory. :param vasp_input: directory of vasp input

flare.dft_interface.vasp_util.**edit_dft_input_positions**(*output_name: str, structure: <sphinx.ext.autodoc.importer._MockObject object at 0x7fda8018f860>*)

Writes a VASP POSCAR file from structure with the name poscar . WARNING: Destructively replaces the file with the name specified by poscar :param poscar: Name of file :param structure: structure to write to file

flare.dft_interface.vasp_util.**md_trajectory_from_vasprun**(*vasprun: Union[str, <sphinx.ext.autodoc.importer._MockObject object at 0x7fda804f1710>], ionic_step_skips=1, vasprun_kwargs: dict = {}*)

Returns a list of flare Structure objects decorated with forces, stress, and total energy from a MD trajectory performed in VASP. :param vasprun: pymatgen Vasprun object or vasprun filename :param ionic_step_skips: if True, only samples the configuration every

ionic_skip_steps steps.

flare.dft_interface.vasp_util.**parse_dft_forces**(*vasprun: Union[str, <sphinx.ext.autodoc.importer._MockObject object at 0x7fda804f1710>]*)

Parses the DFT forces from the last ionic step of a VASP vasprun.xml file :param vasprun: pymatgen Vasprun object or vasprun filename

flare.dft_interface.vasp_util.**parse_dft_forces_and_energy**(*vasprun: Union[str, <sphinx.ext.autodoc.importer._MockObject object at 0x7fda804f1710>]*)

Parses the DFT forces and energy from a VASP vasprun.xml file :param vasprun: pymatgen Vasprun object or vasprun filename

flare.dft_interface.vasp_util.**parse_dft_input**(*input: str*)

Returns the positions, species, and cell of a POSCAR file. Outputs are specced for OTF module.

Parameters **input** – POSCAR file input

Returns

flare.dft_interface.vasp_util.**run_dft**(*calc_dir: str, dft_loc: str, structure: <sphinx.ext.autodoc.importer._MockObject object at 0x7fda8018f860> = None, en: bool = False, vasp_cmd='{}'*)

Run a VASP calculation. :param calc_dir: Name of directory to perform calculation in :param dft_loc: Name of VASP command (i.e. executable location) :param structure: flare structure object :param en: whether to return the final energy along with the forces :param vasp_cmd: Command to run VASP (leave a formatter "{}" to insert dft_loc);

this can be used to specify mpirun, srun, etc.

Returns forces on each atom (and energy if en=True)

any DFT interface can be added here, as long as two functions listed below are implemented

- parse_dft_input(dft_input)
- dft_module.run_dft_par(dft_input, structure, dft_loc, no_cpus)

class flare.otf.**OTF**(*dt: float, number_of_steps: int, prev_pos_init: ndarray = None, rescale_steps: List[int] = [], rescale_temps: List[int] = [], gp: flare.gp.GaussianProcess = None, calculate_energy: bool = False, calculate_efs: bool = False, write_model: int = 0, std_tolerance_factor: float = 1, skip: int = 0, init_atoms: List[int] = None, output_name: str = 'otf_run', max_atoms_added: int = 1, freeze_hyps: int = 10, force_source: str = 'qe', npool: int = None, mpi: str = 'srun', dft_loc: str = None, dft_input: str = None, dft_output='dft.out', dft_kwargs=None, store_dft_output: Tuple[Union[str, List[str]], str] = None, n_cpus: int = 1*)

Trains a Gaussian process force field on the fly during molecular dynamics.

Parameters

- **dt** (*float*) – MD timestep.
- **number_of_steps** (*int*) – Number of timesteps in the training simulation.
- **prev_pos_init** (*[type], optional*) – Previous positions. Defaults to None.
- **rescale_steps** (*List[int], optional*) – List of frames for which the velocities of the atoms are rescaled. Defaults to [].
- **rescale_temps** (*List[int], optional*) – List of rescaled temperatures. Defaults to [].
- **gp** (*gp.GaussianProcess*) – Initial GP model.
- **calculate_energy** (*bool, optional*) – If True, the energy of each frame is calculated with the GP. Defaults to False.
- **calculate_efs** (*bool, optional*) – If True, the energy and stress of each frame is calculated with the GP. Defaults to False.
- **write_model** (*int, optional*) – If 0, write never. If 1, write at end of run. If 2, write after each training and end of run. If 3, write after each time atoms are added and end of run.
- **std_tolerance_factor** (*float, optional*) – Threshold that determines when DFT is called. Specifies a multiple of the current noise hyperparameter. If the epistemic uncertainty on a force component exceeds this value, DFT is called. Defaults to 1.
- **skip** (*int, optional*) – Number of frames that are skipped when dumping to the output file. Defaults to 0.
- **init_atoms** (*List[int], optional*) – List of atoms from the input structure whose local environments and force components are used to train the initial GP model. If None is specified, all atoms are used to train the initial GP. Defaults to None.
- **output_name** (*str, optional*) – Name of the output file. Defaults to 'otf_run'.

- **max_atoms_added** (*int, optional*) – Number of atoms added each time DFT is called. Defaults to 1.
- **freeze_hyps** (*int, optional*) – Specifies the number of times the hyperparameters of the GP are optimized. After this many updates to the GP, the hyperparameters are frozen. Defaults to 10.
- **force_source** (*Union[str, object], optional*) – DFT code used to calculate ab initio forces during training. A custom module can be used here in place of the DFT modules available in the FLARE package. The module must contain two functions: parse_dft_input, which takes a file name (in string format) as input and returns the positions, species, cell, and masses of a structure of atoms; and run_dft_par, which takes a number of DFT related inputs and returns the forces on all atoms. Defaults to "qe".
- **npool** (*int, optional*) – Number of k-point pools for DFT calculations. Defaults to None.
- **mpi** (*str, optional*) – Determines how mpi is called. Defaults to "srun".
- **dft_loc** (*str*) – Location of DFT executable.
- **dft_input** (*str*) – Input file.
- **dft_output** (*str*) – Output file.
- **dft_kwargs** (*[type], optional*) – Additional arguments which are passed when DFT is called; keyword arguments vary based on the program (e.g. ESPRESSO vs. VASP). Defaults to None.
- **store_dft_output** (*Tuple[Union[str,List[str]],str], optional*) – After DFT calculations are called, copy the file or files specified in the first element of the tuple to a directory specified as the second element of the tuple. Useful when DFT calculations are expensive and want to be kept for later use. The first element of the tuple can either be a single file name, or a list of several. Copied files will be prepended with the date and time with the format 'Year.Month.Day:Hour:Minute:Second:'.
- **n_cpus** (*int, optional*) – Number of cpus used during training. Defaults to 1.

compute_properties()
In ASE-OTF, it will be replaced by subclass method

md_step()
Take an MD step. This updates the positions of the structure.

run()
Performs an on-the-fly training run.

If OTF has store_dft_output set, then the specified DFT files will be copied with the current date and time prepended in the format 'Year.Month.Day:Hour:Minute:Second:'.

run_dft()
Calculates DFT forces on atoms in the current structure.

If OTF has store_dft_output set, then the specified DFT files will be copied with the current date and time prepended in the format 'Year.Month.Day:Hour:Minute:Second:'.

Calculates DFT forces on atoms in the current structure.

train_gp()
Optimizes the hyperparameters of the current GP model.

update_gp(*train_atoms: List[int], dft_frcs: ndarray*)
Updates the current GP model.

Parameters

- **train_atoms** (*List[int]*) – List of atoms whose local environments will be added to the training set.
- **dft_frcs** (*np.ndarray*) – DFT forces on all atoms in the structure.

update_positions(*new_pos: ndarray*)
Performs a Verlet update of the atomic positions.

Parameters **new_pos** (*np.ndarray*) – Positions of atoms in the next MD frame.

update_temperature()
Updates the instantaneous temperatures of the system.

Parameters **new_pos** (*np.ndarray*) – Positions of atoms in the next MD frame.

1.3.3 Mapped Gaussian Process

1.3.3.1 Splines Methods

Cubic spline functions used for interpolation.

class flare.mgp.splines_methods.**CubicSpline**(*a*, *b*, *orders*, *values=None*)
Forked from Github repository: https://github.com/EconForge/interpolation.py. High-level API for cubic splines. Class representing a cubic spline interpolator on a regular cartesian grid.

Creates a cubic spline interpolator on a regular cartesian grid.

Parameters

- **a** (*numpy array of size d (float)*) – Lower bounds of the cartesian grid.
- **b** (*numpy array of size d (float)*) – Upper bounds of the cartesian grid.
- **orders** (*numpy array of size d (int)*) – Number of nodes along each dimension (=(n1,. . . ,nd))

Other Parameters **values** (*numpy array (float)*) – (optional, (n1 x . . . x nd) array). Values on the nodes of the function to interpolate.

grid
Cartesian enumeration of all nodes.

interpolate(*points*, *values=None*, *with_derivatives=False*)
Interpolate spline at a list of points.

Parameters

- **points** – (array-like) list of point where the spline is evaluated.
- **values** – (optional) container for inplace computation.

Return values (array-like) list of point where the spline is evaluated.

set_values(*values*)
Set values on the nodes for the function to interpolate.

class flare.mgp.splines_methods.**PCASplines**(*l_bounds*, *u_bounds*, *orders*, *svd_rank*)
Build splines for PCA decomposition, mainly used for the mapping of the variance

Parameters

- **l_bounds** (*numpy array*) – lower bound for the interpolation. E.g. 1-d for two-body, 3-d for three-body.
- **u_bounds** (*numpy array*) – upper bound for the interpolation.
- **orders** (*numpy array*) – grid numbers in each dimension. E.g, 1-d for two-body, 3-d for three-body, should be positive integers.
- **svd_rank** (*int*) – rank for decomposition of variance matrix, also equal to the number of mappings constructed for mapping variance. For two-body *svd_rank<=min(grid_num, train_size*3)*, for three-body *svd_rank<=min(grid_num_in_cube, train_size*3)*

flare.mgp.splines_methods.**vec_eval_cubic_spline**(*a*, *b*, *orders*, *coefs*, *points*, *values=None*)

Forked from Github repository: https://github.com/EconForge/interpolation.py. Evaluates a cubic spline at many points

Parameters

- **a** (*numpy array of size d (float)*) – Lower bounds of the cartesian grid.
- **b** (*numpy array of size d (float)*) – Upper bounds of the cartesian grid.
- **orders** (*numpy array of size d (int)*) – Number of nodes along each dimension (=(n1,. . . ,nd))
- **coefs** (*array of dimension d, and size (n1+2, .., nd+2)*) – Filtered coefficients.
- **point** (*array of size N x d*) – List of points where the splines must be interpolated.
- **values** (*array of size N*) – (optional) If not None, contains the result.

Return value Interpolated values. values[i] contains spline evaluated at point points[i,:].

MappedGaussianProcess uses splines to build up interpolationfunction of the low-dimensional decomposition of Gaussian Process, with little loss of accuracy. Refer to Vandermause et al., Glielmo et al.

class flare.mgp.mgp.**MappedGaussianProcess**(*grid_params: dict*, *unique_species: list = []*, *map_force: bool = False*, *GP: flare.gp.GaussianProcess = None*, *mean_only: bool = True*, *container_only: bool = True*, *lmp_file_name: str = 'lmp.mgp'*, *n_cpus: int = None*, *n_sample: int = 100*)

Build Mapped Gaussian Process (MGP) and automatically save coefficients for LAMMPS pair style.

Parameters

- **grid_params** (*dict*) – Parameters for the mapping itself, such as grid size of spline fit, etc. As described below.
- **unique_species** (*dict*) – List of all the (unique) species included during the training that need to be mapped
- **map_force** (*bool*) – if True, do force mapping; otherwise do energy mapping, default is False
- **GP** (*GaussianProcess*) – None or a GaussianProcess object. If a GP is input, and container_only is False, automatically build a mapping corresponding to the GaussianProcess.
- **mean_only** (*bool*) – if True: only build mapping for mean (force)
- **container_only** (*bool*) – if True: only build splines container (with no coefficients); if False: Attempt to build map immediately

- **lmp_file_name** (*str*) – LAMMPS coefficient file name
- **n_cpus** (*int*) – Default None. Set to the number of cores needed for parallelization. Used in the construction of the map.
- **n_sample** (*int*) – Default 100. The batch size for building map. Not used now.

Examples:

```
>>> # build 2 + 3 body map
>>> grid_params = {'twobody': {'grid_num': [64]},
...                'threebody': {'grid_num': [64, 64, 64]}}
```

For *grid_params*, the following keys and values are allowed

Parameters

- **'two_body'** (*dict, optional*) – if 2-body is present, set as a dictionary of parameters for 2-body mapping. Parameters see below.
- **'three_body'** (*dict, optional*) – if 3-body is present, set as a dictionary of parameters for 3-body mapping. Parameters see below.
- **'load_grid'** (*str, optional*) – Default None. the path to the directory where the previously generated grids (`grid_*.npy`) are stored. If no path is specified, MGP will construct grids from scratch.
- **'lower_bound_relax'** (*float, optional*) – Default 0.1. if 'lower_bound' is set to 'auto' this value will be used as a relaxation of lower bound. (see below the description of 'lower_bound')

For two/three body parameter dictionary, the following keys and values are allowed

Parameters

- **'grid_num'** (*list*) – a list of integers, the number of grid points for interpolation. The larger the number, the better the approximation of MGP is compared with GP.
- **'lower_bound'** (*str or list, optional*) – Default 'auto', the lower bound of the spline interpolation will be searched. First, search the training set of GP and find the minimal interatomic distance r_min. Then, the `lower_bound = r_min - lower_bound_relax`. The user can set their own lower_bound, of the same shape as 'grid_num'. E.g. for threebody, the customized lower bound can be set as [1.2, 1.2, 1.2].
- **'upper_bound'** (*str or list, optional*) – Default 'auto', the upper bound of the spline interpolation will be the cutoffs of GP. The user can set their own upper_bound, of the same shape as 'grid_num'. E.g. for threebody, the customized lower bound can be set as [3.5, 3.5, 3.5].
- **'svd_rank'** (*int, optional*) – Default 'auto'. If the variance mapping is needed, it is set as the rank of the mapping. 'auto' uses full rank, which is the smaller one between the total number of grid points and training set size. i.e. `full_rank = min(np.prod(grid_num), 3 * N_train)`

as_dict() → dict
Dictionary representation of the MGP model.

static from_dict(*dictionary: dict*)
Create MGP object from dictionary representation.

predict(*atom_env: flare.env.AtomicEnvironment*, *mean_only: bool = True*) -> (<class 'float'>, 'ndarray', 'ndarray', <class 'float'>)
predict force, variance, stress and local energy for given atomic environment

Parameters

- **atom_env** – atomic environment (with a center atom and its neighbors)
- **mean_only** – if True: only predict force (variance is always 0)

Returns 3d array of atomic force variance: 3d array of the predictive variance stress: 6d array of the virial stress energy: the local energy (atomic energy)

Return type force

write_lmp_file(*lammps_name*)
write the coefficients to a file that can be used by lammps pair style

write_model(*name: str*, *format='json'*)
Write everything necessary to re-load and re-use the model :param model_name: :return:

1.3.4 ASE Interface

We provide an interface to do the OTF training coupling with ASE. Including wrapping the FLARE's GaussianProcess and MappedGaussianProcess into an ASE calculator: `FLARE_Calculator`,

1.3.4.1 FLARE ASE Calculator

FLARE_Calculator is a calculator compatible with *ASE*. You can build up *ASE Atoms* for your atomic structure, and use *get_forces*, *get_potential_energy* as general *ASE Calculators*, and use it in *ASE Molecular Dynamics* and our *ASE OTF* training module. For the usage users can refer to ASE Calculator module and ASE Calculator tutorial.

class flare.ase.calculator.**FLARE_Calculator**(*gp_model*, *mgp_model=None*, *par=False*, *use_mapping=False*)
Build FLARE as an ASE Calculator, which is compatible with ASE Atoms and Molecular Dynamics. :Parameters: * **gp_model** (*GaussianProcess*) – FLARE's Gaussian process object

- **mgp_model** (*MappedGaussianProcess*) – FLARE's Mapped Gaussian Process object. *None* by default. MGP will only be used if *use_mapping* is set to True.
- **par** (*Bool*) – set to *True* if parallelize the prediction. *False* by default.
- **use_mapping** (*Bool*) – set to *True* if use MGP for prediction. *False* by default.

calculate(*atoms*, *structure*)

Calculate properties including: energy, local energies, forces, stress, uncertainties.

Parameters **atoms** (*Atoms*) – ASE Atoms object

1.3.4.2 On-the-fly training

`OTF` is the on-the-fly training module for ASE, WITHOUT molecular dynamics engine. It needs to be used adjointly with ASE MD engine.

class flare.ase.otf.**ASE_OTF**(*atoms*, *timestep*, *number_of_steps*, *dft_calc*, *md_engine*, *md_kwargs*, *trajectory=None*, ***otf_kwargs*)
On-the-fly training module using ASE MD engine, a subclass of OTF.

Parameters

- **atoms** (*ASE Atoms*) – the ASE Atoms object for the on-the-fly MD run, with calculator set as FLARE_Calculator.

- **timestep** – the timestep in MD. Please use ASE units, e.g. if the timestep is 1 fs, then set *timestep = 1 * units.fs*
- **number_of_steps** (*int*) – the total number of steps for MD.
- **dft_calc** (*ASE Calculator*) – any ASE calculator is supported, e.g. Espresso, VASP etc.
- **md_engine** (*str*) – the name of MD thermostat, only *VelocityVerlet*, *NVTBerendsen*, *NPTBerendsen*, *NPT* and *Langevin* are supported.
- **md_kwargs** (*dict*) – Specify the args for MD as a dictionary, the args are as required by the ASE MD modules consistent with the *md_engine*.
- **trajectory** (*ASE Trajectory*) – default *None*, not recommended, currently in experiment.

The following arguments are for on-the-fly training, the user can also refer to `OTF`

Parameters

- **prev_pos_init** (*[type], optional*) – Previous positions. Defaults to None.
- **rescale_steps** (*List[int], optional*) – List of frames for which the velocities of the atoms are rescaled. Defaults to [].
- **rescale_temps** (*List[int], optional*) – List of rescaled temperatures. Defaults to [].
- **calculate_energy** (*bool, optional*) – If True, the energy of each frame is calculated with the GP. Defaults to False.
- **write_model** (*int, optional*) – If 0, write never. If 1, write at end of run. If 2, write after each training and end of run. If 3, write after each time atoms are added and end of run.
- **std_tolerance_factor** (*float, optional*) – Threshold that determines when DFT is called. Specifies a multiple of the current noise hyperparameter. If the epistemic uncertainty on a force component exceeds this value, DFT is called. Defaults to 1.
- **skip** (*int, optional*) – Number of frames that are skipped when dumping to the output file. Defaults to 0.
- **init_atoms** (*List[int], optional*) – List of atoms from the input structure whose local environments and force components are used to train the initial GP model. If None is specified, all atoms are used to train the initial GP. Defaults to None.
- **output_name** (*str, optional*) – Name of the output file. Defaults to 'otf_run'.
- **max_atoms_added** (*int, optional*) – Number of atoms added each time DFT is called. Defaults to 1.
- **freeze_hyps** (*int, optional*) – Specifies the number of times the hyperparameters of the GP are optimized. After this many updates to the GP, the hyperparameters are frozen. Defaults to 10.
- **n_cpus** (*int, optional*) – Number of cpus used during training. Defaults to 1.

compute_properties()

Compute energies, forces, stresses, and their uncertainties with the FLARE ASE calcuator, and write the results to the OTF structure object.

md_step()

Get new position in molecular dynamics based on the forces predicted by FLARE_Calculator or DFT calculator

update_gp(*train_atoms*, *dft_frcs*)

Updates the current GP model.

Parameters

- **train_atoms** (*List[int]*) – List of atoms whose local environments will be added to the training set.
- **dft_frcs** (*np.ndarray*) – DFT forces on all atoms in the structure.

update_positions(*new_pos*)
Performs a Verlet update of the atomic positions.

Parameters **new_pos** (*np.ndarray*) – Positions of atoms in the next MD frame.

update_temperature()
Updates the instantaneous temperatures of the system.

Parameters **new_pos** (*np.ndarray*) – Positions of atoms in the next MD frame.

1.3.5 GP From AIMD

Tool to enable the development of a GP model based on an AIMD trajectory with many customizable options for fine control of training. Contains methods to transfer the model to an OTF run or MD engine run.

1.3.5.1 Seed frames

The various parameters in the `TrajectoryTrainer` class related to "Seed frames" are to help you train a model which does not yet have a training set. Uncertainty- and force-error driven training will go better with a somewhat populated training set, as force and uncertainty estimates are better behaveed with more data.

You may pass in a set of seed frames or atomic environments. All seed environments will be added to the GP model; seed frames will be iterated through and atoms will be added at random. There are a few reasons why you would want to pay special attention to an individual species.

If you are studying a system where the dynamics of one species are particularly important and so you want a good representation in the training set, then you would want to include as many as possible in the training set during the seed part of the training.

Inversely, if a system has high representation of a species well-described by a simple 2+3 body kernel, you may want it to be less well represented in the seeded training set.

By specifying the pre_train_atoms_per_element, you can limit the number of atoms of a given species which are added in. You can also limit the number of atoms which are added from a given seed frame.

flare.gp_from_aimd.**parse_trajectory_trainer_output**(*file: str*, *return_gp_data: bool = False*) → Union[List[dict], Tuple[List[dict], dict]]
Reads output of a TrajectoryTrainer run by frame. return_gp_data returns data about GP model growth useful for visualizing progress of model training.

Parameters

- **file** – filename of output
- **return_gp_data** – flag for returning extra GP data

Returns List of dictionaries with keys 'species', 'positions', 'gp_forces', 'dft_forces', 'gp_stds', 'added_atoms', and 'maes_by_species', optionally, gp_data dictionary

1.3.6 Utility

1.3.6.1 Conversion between atomic numbers and element symbols

Utility functions for various tasks.

class flare.utils.element_coder.**NumpyEncoder**(*, *skipkeys=False*, *ensure_ascii=True*, *check_circular=True*, *allow_nan=True*, *sort_keys=False*, *indent=None*, *separators=None*, *default=None*)

Special json encoder for numpy types for serialization use as

json.loads(... cls = NumpyEncoder)

or:

json.dumps(... cls = NumpyEncoder)

Thanks to StackOverflow users karlB and fnunnari, who contributed this from: *https://stackoverflow.com/a/47626762*

default(*obj*)

flare.utils.element_coder.**Z_to_element**(*Z: int*) → str

Maps atomic numbers Z to element name, e.g. 1->"H".

Parameters **Z** – Atomic number corresponding to element.

Returns One or two-letter name of element.

flare.utils.element_coder.**element_to_Z**(*element: str*) → int

Returns the atomic number Z associated with an elements 1-2 letter name. Returns the same integer if an integer is passed in.

Parameters **element** –

Returns

1.3.6.2 Conditions to add training data

Utility functions for various tasks.

flare.utils.learner.**get_max_cutoff**(*cell: <sphinx.ext.autodoc.importer._MockObject object at 0x7fda8021c7b8>*) → float

Compute the maximum cutoff compatible with a 3x3x3 supercell of a structure. Called in the Structure constructor when setting the max_cutoff attribute, which is used to create local environments with arbitrarily large cutoff radii.

Parameters **cell** (*np.ndarray*) – Bravais lattice vectors of the structure stored as rows of a 3x3 Numpy array.

Returns

Maximum cutoff compatible with a 3x3x3 supercell of the structure.

Return type float

flare.utils.learner.**is_force_in_bound_per_species**(*abs_force_tolerance: float, predicted_forces: ndarray, label_forces: ndarray, structure, max_atoms_added: int = inf, max_by_species: dict = {}, max_force_error: float = inf) -> (<class 'bool'>, typing.List[int]*)

Checks the forces of GP prediction assigned to the structure against a DFT calculation, and return a list of atoms which meet an absolute threshold abs_force_tolerance.

Can limit the total number of target atoms via max_atoms_added, and limit per species by max_by_species.

The max_atoms_added argument will 'overrule' the max by species; e.g. if max_atoms_added is 2 and max_by_species is {"H":3}, then at most two atoms total will be added.

Because adding atoms which are in configurations which are far outside of the potential energy surface may not always be desirable, a maximum force error can be passed in; atoms with

Parameters

- **abs_force_tolerance** – If error exceeds this value, then return atom index
- **predicted_forces** – Force predictions made by GP model
- **label_forces** – "True" forces computed by DFT
- **structure** – FLARE Structure
- **max_atoms_added** – Maximum atoms to return
- **max_by_species** – Limit to a maximum number of atoms by species
- **max_force_error** – In order to avoid counting in highly unlikely configurations, if the error exceeds this, do not add atom

Returns Bool indicating if any atoms exceeded the error threshold, and a list of indices of atoms which did sorted by their error.

flare.utils.learner.**is_std_in_bound**(*std_tolerance: float, noise: float, structure: flare.struc.Structure, max_atoms_added: int = inf) -> (<class 'bool'>, typing.List[int]*)

Given an uncertainty tolerance and a structure decorated with atoms, species, and associated uncertainties, return those which are above a given threshold, agnostic to species.

If std_tolerance is negative, then the threshold used is the absolute value of std_tolerance.

If std_tolerance is positive, then the threshold used is std_tolerance * noise.

If std_tolerance is 0, then do not check.

Parameters

- **std_tolerance** – If positive, multiply by noise to get cutoff. If negative, use absolute value of std_tolerance as cutoff.
- **noise** – Noise variance parameter
- **structure** (*FLARE Structure*) – Input structure
- **max_atoms_added** – Maximum # of atoms to add

Returns (True,[-1]) if no atoms are above cutoff, (False,[...]) of the top *max_atoms_added* uncertainties

flare.utils.learner.**is_std_in_bound_per_species**(*rel_std_tolerance: float, abs_std_tolerance: float, noise: float, structure: flare.struc.Structure, max_atoms_added: int = inf, max_by_species: dict = {}*) -> (<class 'bool'>, typing.List[int])

Checks the stds of GP prediction assigned to the structure, returns a list of atoms which either meet an absolute threshold or a relative threshold defined by rel_std_tolerance * noise. Can limit the total number of target atoms via max_atoms_added, and limit per species by max_by_species.

The max_atoms_added argument will 'overrule' the max by species; e.g. if max_atoms_added is 2 and max_by_species is {"H":3}, then at most two atoms will be added.

Parameters

- **rel_std_tolerance** – Multiplied by noise to get a lower bound for the uncertainty threshold defined relative to the model.
- **abs_std_tolerance** – Used as an absolute lower bound for the uncertainty threshold.
- **noise** – Noise hyperparameter for model, used to define relative uncertainty cutoff.
- **structure** – FLARE structure decorated with uncertainties in structure.stds.
- **max_atoms_added** – Maximum number of atoms to return from structure.
- **max_by_species** – Dictionary describing maximum number of atoms to return by species (e.g. {'H':1,'He':2} will return at most 1 H and 2 He atoms.)

Returns Bool indicating if any atoms exceeded the uncertainty threshold, and a list of indices of atoms which did, sorted by their uncertainty.

flare.utils.learner.**subset_of_frame_by_element**(*frame: flare.Structure, predict_atoms_per_element: dict*) → List[int]

Given a structure and a dictionary formatted as {"Symbol":int, ..} describing a number of atoms per element, return a sorted list of indices corresponding to a random subset of atoms by species :param frame: :param predict_atoms_by_species: :return:

1.3.6.3 Advanced Hyperparameters Set Up

For multi-component systems, the configurational space can be highly complicated. One may want to use different hyper-parameters and cutoffs for different interactions, or do constraint optimisation for hyper-parameters.

To use more hyper-parameters, we need special kernel function that can differentiate different pairs, triplets and other descriptors and determine which number to use for what interaction.

This kernel can be enabled by using the `hyps_mask` argument of the GaussianProcess class. It contains multiple arrays to describe how to break down the array of hyper-parameters and apply them when computing the kernel. Detail descriptions of this argument can be seen in kernel/mc_sephyps.py.

The ParameterHelper class is to generate the hyps_mask with a more human readable interface.

Example:

```
>>> pm = ParameterHelper(species=['C', 'H', 'O'],
...                      kernels={'twobody':[['*', '*'], ['O','O']],
...                      'threebody':[['*', '*', '*'],
...                          ['O','O', 'O']]},
...                      parameters={'twobody0':[1, 0.5, 1], 'twobody1':[2, 0.2, 2],
```

(continues on next page)

(continued from previous page)

```
...                              'threebody0':[1, 0.5], 'threebody1':[2, 0.2],
...                              'cutoff_threebody':1},
...                         constraints={'twobody0':[False, True]})
>>> hm = pm.hyps_mask
>>> hyps = hm['hyps']
>>> cutoffs = hm['cutoffs']
>>> kernels = hm['kernels']
>>> gp_model = GaussianProcess(kernels=kernels, cutoffs=cutoffs,
...                              hyps=hyps, hyps_mask=hm)
```

In this example, four atomic species are involved. There are many kinds of twobodys and threebodys. But we only want to use eight different signal variance and length-scales.

In order to do so, we first define all the twobodys to be group "twobody0", by listing "-" as the first element in the twobody argument. The second element O-O is then defined to be group "twobody1". Note that the order matters here. The later element overrides the ealier one. If twobodys=[['O', 'O'], ['*', '*']], then all twobodys belong to group "twobody1".

Similarly, O-O-O is defined as threebody1, while all remaining ones are left as threebody0.

The hyperpameters for each group is listed in the order of [sig, ls, cutoff] in the parameters argument. So in this example, O-O interaction will use [2, 0.2, 2] as its sigma, length scale, and cutoff.

For threebody, the parameter arrays only come with two elements. So there is no cutoff associated with threebody0 or threebody1; instead, a universal cutoff is used, which is defined as 'cutoff_threebody'.

The constraints argument define which hyper-parameters will be optimized. True for optimized and false for being fixed.

Here are a couple more simple examples.

Define a 5-parameter 2+3 kernel (1, 0.5, 1, 0.5, 0.05)

```
>>> pm = ParameterHelper(kernels=['twobody', 'threebody'],
...                      parameters={'sigma': 1,
...                                  'lengthscale': 0.5,
...                                  'cutoff_twobody': 2,
...                                  'cutoff_threebody': 1,
...                                  'noise': 0.05})
```

Define a 5-parameter 2+3 kernel (1, 1, 1, 1, 0.05)

```
>>> pm = ParameterHelper(kernels=['twobody', 'threebody'],
...                      parameters={'cutoff_twobody': 2,
...                                  'cutoff_threebody': 1,
...                                  'noise': 0.05},
...                      ones=ones,
...                      random=not ones)
```

Define a 9-parameter 2+3 kernel

```
>>> pm = ParameterHelper()
>>> pm.define_group('specie', 'O', ['O'])
>>> pm.define_group('specie', 'rest', ['C', 'H'])
>>> pm.define_group('twobody', '**', ['*', '*'])
>>> pm.define_group('twobody', 'OO', ['O', 'O'])
>>> pm.define_group('threebody', '***', ['*', '*', '*'])
>>> pm.define_group('threebody', 'Oall', ['O', 'O', 'O'])
```

(continues on next page)

(continued from previous page)

```
>>> pm.set_parameters('**', [1, 0.5])
>>> pm.set_parameters('OO', [1, 0.5])
>>> pm.set_parameters('Oall', [1, 0.5])
>>> pm.set_parameters('***', [1, 0.5])
>>> pm.set_parameters('cutoff_twobody', 5)
>>> pm.set_parameters('cutoff_threebody', 4)
```

See more examples in functions `ParameterHelper.define_group` , `ParameterHelper.set_parameters`, and in the tests `tests/test_parameters.py`

class flare.utils.parameter_helper.**ParameterHelper**(*hyps_mask=None*, *species=None*, *kernels={}*, *cutoff_groups={}*, *parameters=None*, *constraints={}*, *allseparate=False*, *random=False*, *ones=False*, *verbose='WARNING'*)

A helper class to construct the hyps_mask dictionary for AtomicEnvironment , GaussianProcess and MappedGaussianProcess

Parameters

- **hyps_mask** (*dict*) – Not implemented yet
- **species** (*dict, list*) – Define specie groups
- **kernels** (*dict, list*) – Define kernels and groups for the kernels
- **cutoff_groups** (*dict*) – Define different cutoffs for different species
- **parameters** (*dict*) – Define signal variance, length scales, and cutoffs
- **constraints** (*dict*) – If listed as False, the cooresponding hyperparmeters will not be trained
- **allseparate** (*bool*) – If True, define each type pair/triplet into a separate group.
- **random** (*bool*) – If True, randomized all signal variances and lengthscales
- **one** (*bool*) – If True, set all signal variances and lengthscales to one
- **verbose** (*str*) – Level to print with "ERROR", "WARNING", "INFO", "DEBUG"

- the `species` is an optional input. It can be left as None if the user only wants to set up one group of hyper-parameters for each kernel.
- the `kernels` can be defined along with or without groups. But the later mode is not compatible with the `allseparate` flag.

```
>>> kernels=['twobody', 'threebody'],
```

or

```
>>> kernels={'twobody':[['*', '*'], ['O','O']],
...          'threebody':[['*', '*', '*'],
...                       ['O','O', 'O']]},
```

Current options for the kernels are twobody, threebody and manybody (based on coordination number).

- See format of `species`, `kernels` (dict), and `cutoff_groups` in `list_groups()` function.
- See format of `parameters` and `constraints` in `list_parameters()` function.

all_separate_groups(*group_type*)
Separate all possible types of twobodys, threebodys, manybody. One type per group.

Parameters **group_type** (*str*) – “specie”, “twobody”, “threebody”, “cut3b”, “manybody”

as_dict()
Dictionary representation of the mask. The output can be used for AtomicEnvironment or the GaussianProcess

as_object()
Object representation of the mask. The output can be used for AtomicEnvironment or the GaussianProcess.

define_group(*group_type*, *name*, *element_list*, *parameters=None*, *atomic_str=False*)
Define specie/twobody/threebody/3b cutoff/manybody group

Parameters

- **group_type** (*str*) – “specie”, “twobody”, “threebody”, “cut3b”, “manybody”
- **name** (*str*) – the name use for indexing. can be anything but “*”
- **element_list** (*list*) – list of elements
- **parameters** (*list*) – corresponding parameters for this group
- **atomic_str** (*bool*) – whether the elements in element_list are specified by group names or periodic table element names.

The function is helped to define different groups for specie/twobody/threebody /3b cutoff/manybody terms. This function can be used for many times. The later one always overrides the former one.

The name of the group has to be unique string (but not “*”), that define a group of species or twobodys, etc. If the same name is used, in two function calls, the definitions of the group will be merged. Both calls will be effective.

element_list has to be a list of atomic elements, or a list of specie group names (which should be defined in previous calls), or “*”. “*” will loop the function over all previously defined species. It has to be two elements for twobody/3b cutoff/manybody term, or three elements for threebody. For specie group definition, it can be as many elements as you want.

If multiple define_group calls have conflict with element, the later one has higher priority. For example, twobody 1-2 are defined as group1 in the first call, and as group2 in the second call. In the end, the twobody will be left as group2.

Example 1:

```
>>> define_group('specie', 'water', ['H', 'O'])
>>> define_group('specie', 'salt', ['Cl', 'Na'])
```

They define H and O to be group water, and Na and Cl to be group salt.

Example 2.1:

```
>>> define_group('twobody', 'in-water', ['H', 'H'], atomic_str=True)
>>> define_group('twobody', 'in-water', ['H', 'O'], atomic_str=True)
>>> define_group('twobody', 'in-water', ['O', 'O'], atomic_str=True)
```

Example 2.2:

```
>>> define_group('twobody', 'in-water', ['water', 'water'])
```

The 2.1 is equivalent to 2.2.

Example 3.1:

```
>>> define_group('specie', '1', ['H'])
>>> define_group('specie', '2', ['O'])
>>> define_group('twobody', 'Hgroup', ['H', 'H'], atomic_str=True)
>>> define_group('twobody', 'Hgroup', ['H', 'O'], atomic_str=True)
>>> define_group('twobody', 'OO', ['O', 'O'], atomic_str=True)
```

Example 3.2:

```
>>> define_group('specie', '1', ['H'])
>>> define_group('specie', '2', ['O'])
>>> define_group('twobody', 'Hgroup', ['H', '*'], atomic_str=True)
>>> define_group('twobody', 'OO', ['O', 'O'], atomic_str=True)
```

Example 3.3:

```
>>> list_groups('specie', ['H', 'O'])
>>> define_group('twobody', 'Hgroup', ['H', '*'])
>>> define_group('twobody', 'OO', ['O', 'O'])
```

Example 3.4:

```
>>> list_groups('specie', ['H', 'O'])
>>> define_group('twobody', 'OO', ['*', '*'])
>>> define_group('twobody', 'Hgroup', ['H', '*'])
```

3.1 to 3.4 are all equivalent.

fill_in_parameters(*group_type*, *random=False*, *ones=False*, *universal=False*)
Separate all possible types of twobodys, threebodys, manybody. One type per group. And fill in either universal ls and sigma from pre-defined parameters from set_parameters("sigma", ..) and set_parameters("ls", ..) or random parameters if random is True.

Parameters

- **group_type** (*str*) – "specie", "twobody", "threebody", "cut3b", "manybody"
- **definition_list** (*list, dict*) – list of elements

find_group(*group_type*, *element_list*, *atomic_str=False*)
find the group that contains the input pair

Parameters

- **group_type** (*str*) – species, twobody, threebody, cut3b, manybody
- **element_list** (*list*) – list of elements for a pair/triplet/coordination-pair
- **atomic_str** (*bool*) – whether the elements in element_list are specified by group names or periodic table element names.

Returns

Return type name (str)

static from_dict(*hyps_mask*, *verbose=False*, *init_spec=[]*)
convert dictionary mask to HM instance This function is not tested yet

list_groups(*group_type*, *definition_list*)
define groups in batches.

Parameters

- **group_type** (*str*) – "specie", "twobody", "threebody", "cut3b", "manybody"

- **definition_list** (*list, dict*) – list of elements

This function runs define_group in batch. Please first read the manual of define_group.

If the definition_list is a list, it is equivalent to executing define_group through the definition_list.

```
>>> for all terms in the list:
>>>     define_group(group_type, group_type+'n', the nth term in the list)
```

So the first twobody defined will be group twobody0, second one will be group twobody1. For specie, it will define all the listed elements as groups with only one element with their original name.

If the definition_list is a dictionary, it is equivalent to

```
>>> for k, v in the dict:
>>>     define_group(group_type, k, v)
```

It is not recommended to use the dictionary mode, especially when the group definitions are conflicting with each other. There is no guarantee that the priority order is the same as you want.

Unlike ParameterHelper.define_group(), it can only be called once for each group_type, and not after any ParameterHelper.define_group() calls.

list_parameters (*parameter_dict: dict*, *constraints: dict = {}*)

Define many groups of parameters

Parameters

- **parameter_dict** (*dict*) – dictionary of all parameters
- **constraints** (*dict*) – dictionary of all constraints

Example:

```
>>> parameter_dict={"group_name":[sig, ls, cutoffs], ...}
>>> constraints={"group_name":[True, False, False], ...}
```

The name of parameters can be the group name previously defined in define_group or list_groups function. Aside from the group name, `noise`, `cutoff_twobody`, `cutoff_threebody`, and `cutoff_manybody` are reserved for noise parmater and universal cutoffs, while `sigma` and `lengthscale` are reserved for universal signal variances and length scales.

For non-reserved keys, the value should be a list of 2 to 3 elements, corresponding to the sigma, lengthscale and cutoff (if the third one is defined). For reserved keys, the value should be a float number.

The parameter_dict and constraints should use the same set of keys. If a key in constraints is not used in parameter_dict, it will be ignored.

The value in the constraints can be either a single bool, which apply to all parameters, or list of bools that apply to each parameter.

set_constraints (*name*, *opt*)

Set the parameters for certain group

Parameters

- **name** (*str*) – name of the patermeters
- **opt** (*bool, list*) – whether to optimize the parameter or not

The name of parameters can be the group name previously defined in define_group or list_groups function. Aside from the group name, `noise`, `cutoff_twobody`, `cutoff_threebody`, and `cutoff_manybody` are reserved for noise parmater and universal cutoffs, while `sigma` and `lengthscale` are reserved for universal signal variances and length scales.

The optimization flag can be a single bool, which apply to all parameters under that name, or list of bools that apply to each parameter.

set_parameters (*name*, *parameters*, *opt=True*)
Set the parameters for certain group

Parameters

- **name** (*str*) – name of the patermeters
- **parameters** (*list*) – the sigma, lengthscale, and cutoff of each group.
- **opt** (*bool, list*) – whether to optimize the parameter or not

The name of parameters can be the group name previously defined in define_group or list_groups function. Aside from the group name, `noise`, `cutoff_twobody`, `cutoff_threebody`, and `cutoff_manybody` are reserved for noise parmater and universal cutoffs, while `sigma` and `lengthscale` are reserved for universal signal variances and length scales.

The parameter should be a list of 2-3 elements, for sigma, lengthscale (and cutoff if the third one is defined).

The optimization flag can be a single bool, which apply to all parameters, or list of bools that apply to each parameter.

summarize_group (*group_type*)
Sort and combine all the previous definition to internal varialbes

Parameters **group_type** (*str*) – species, twobody, threebody, cut3b, manybody

1.3.6.4 Construct Atomic Environment

flare.utils.env_getarray.**get_2_body_arrays_jit** (*positions*, *atom: int*, *cell*, *r_cut*, *cutoff_2*, *species*, *sweep*, *nspecie*, *specie_mask*, *twobody_mask*)
Returns distances, coordinates, species of atoms, and indices of neighbors in the 2-body local environment. This method is implemented outside the AtomicEnvironment class to allow for njit acceleration with Numba.

Parameters

- **positions** (*np.ndarray*) – Positions of atoms in the structure.
- **atom** (*int*) – Index of the central atom of the local environment.
- **cell** (*np.ndarray*) – 3x3 array whose rows are the Bravais lattice vectors of the cell.
- **cutoff_2** (*np.ndarray*) – 2-body cutoff radius.
- **species** (*np.ndarray*) – Numpy array of species represented by their atomic numbers.
- **nspecie** – number of atom types to define bonds
- **specie_mask** – mapping from atomic number to atom types
- **twobody_mask** – mapping from the types of end atoms to bond types

Type int

Type np.ndarray

Type np.ndarray

Returns

Tuple of arrays describing pairs of atoms in the 2-body local environment.

bond_array_2: Array containing the distances and relative coordinates of atoms in the 2-body local environment. First column contains distances, remaining columns contain Cartesian coordinates divided by the distance (with the origin defined as the position of the central atom). The rows are sorted by distance from the central atom.

bond_positions_2: Coordinates of atoms in the 2-body local environment.

etypes: Species of atoms in the 2-body local environment represented by their atomic number.

bond_indices: Structure indices of atoms in the local environment.

Return type np.ndarray, np.ndarray, np.ndarray, np.ndarray

flare.utils.env_getarray.**get_3_body_arrays_jit**(*bond_array_2*, *bond_positions_2*, *ctype*, *etypes*, *r_cut*, *cutoff_3*, *nspecie*, *specie_mask*, *cut3b_mask*)

Returns distances and coordinates of triplets of atoms in the 3-body local environment.

Parameters

- **bond_array_2** (*np.ndarray*) – 2-body bond array.
- **bond_positions_2** (*np.ndarray*) – Coordinates of atoms in the 2-body local environment.
- **ctype** – atomic number of the center atom
- **cutoff_3** (*np.ndarray*) – 3-body cutoff radius.
- **nspecie** – number of atom types to define bonds
- **specie_mask** – mapping from atomic number to atom types
- **cut3b_mask** – mapping from the types of end atoms to bond types

Type int

Type int

Type np.ndarray

Type np.ndarray

Returns

Tuple of 4 arrays describing triplets of atoms in the 3-body local environment.

bond_array_3: Array containing the distances and relative coordinates of atoms in the 3-body local environment. First column contains distances, remaining columns contain Cartesian coordinates divided by the distance (with the origin defined as the position of the central atom). The rows are sorted by distance from the central atom.

cross_bond_inds: Two dimensional array whose row m contains the indices of atoms n > m that are within a distance cutoff_3 of both atom n and the central atom.

cross_bond_dists: Two dimensional array whose row m contains the distances from atom m of atoms n > m that are within a distance cutoff_3 of both atom n and the central atom.

triplet_counts: One dimensional array of integers whose entry m is the number of atoms that are within a distance cutoff_3 of atom m.

Return type (np.ndarray, np.ndarray, np.ndarray, np.ndarray)

flare.utils.env_getarray.**get_m2_body_arrays_jit**(*positions*, *atom: int*, *cell*, *r_cut*, *manybody_cutoff_list*, *species*, *sweep: <sphinx.ext.autodoc.importer._MockObject object at 0x7fda8023b550>*, *nspec*, *spec_mask*, *manybody_mask*, *cutoff_func=<function quadratic_cutoff>*)

Parameters

- **positions** (*np.ndarray*) – Positions of atoms in the structure.
- **atom** (*int*) – Index of the central atom of the local environment.
- **cell** (*np.ndarray*) – 3x3 array whose rows are the Bravais lattice vectors of the cell.
- **manybody_cutoff_list** (*float*) – 2-body cutoff radius.
- **species** (*np.ndarray*) – Numpy array of species represented by their atomic numbers.

Returns Tuple of arrays describing pairs of atoms in the 2-body local environment.

flare.utils.env_getarray.**get_m3_body_arrays**(*positions*, *atom: int*, *cell*, *cutoff: float*, *species*, *sweep*, *cutoff_func=<function quadratic_cutoff>*)

Note: here we assume the cutoff is not too large, i.e., 2 * cutoff < cell_size

flare.utils.env_getarray.**q3_value_mc**(*distances*, *cross_bond_inds*, *cross_bond_dists*, *triplets*, *r_cut*, *species_list*, *etypes*, *cutoff_func*, *q_func=<function coordination_number>*)

Compute value of many-body many components descriptor based on distances of atoms in the local many-body environment.

Parameters

- **distances** (*np.ndarray*) – distances between atoms i and j
- **r_cut** (*float*) – cutoff hyperparameter
- **ref_species** (*int*) – species to consider to compute the contribution
- **etypes** (*np.ndarray*) – atomic species of neighbours
- **cutoff_func** (*callable*) – cutoff function
- **q_func** (*callable*) – many-body pairwise descrptor function

Returns the value of the many-body descriptor

Return type float

1.3.6.5 Utilities for Molecular Dynamics

Utility functions for various tasks.

flare.utils.md_helper.**get_random_velocities**(*noa: int*, *temperature: float*, *mass: float*)

Draw velocities from the Maxwell-Boltzmann distribution, assuming a fixed mass for all particles in amu.

Parameters

- **noa** (*int*) – Number of atoms in the system.
- **temperature** (*float*) – Temperature of the system.
- **mass** (*float*) – Mass of each particle in amu.

Returns Particle velocities, corrected to give zero center of mass motion.

Return type np.ndarray

flare.utils.md_helper.**get_supercell_positions**(*sc_size: int*, *cell: <sphinx.ext.autodoc.importer._MockObject object at 0x7fda80038240>*, *positions: <sphinx.ext.autodoc.importer._MockObject object at 0x7fda80038a20>*)

Returns the positions of a supercell of atoms, with the number of cells in each direction fixed.

Parameters

- **sc_size** (*int*) – Size of the supercell.
- **cell** (*np.ndarray*) – 3x3 array of cell vectors.
- **positions** (*np.ndarray*) – Positions of atoms in the unit cell.

Returns Positions of atoms in the supercell.

Return type np.ndarray

flare.utils.md_helper.**multicomponent_velocities**(*temperature: float*, *masses: List[float]*)

Draw velocities from the Maxwell-Boltzmann distribution for particles of varying mass.

Parameters

- **temperature** (*float*) – Temperature of the system.
- **masses** (*List[float]*) – Particle masses in amu.

Returns Particle velocities, corrected to give zero center of mass motion.

Return type np.ndarray

flare.utils.md_helper.**supercell_custom**(*cell: <sphinx.ext.autodoc.importer._MockObject object at 0x7fda800389b0>*, *positions: <sphinx.ext.autodoc.importer._MockObject object at 0x7fda80038908>*, *size1: int*, *size2: int*, *size3: int*)

Returns the positions of a supercell of atoms with a chosen number of cells in each direction.

Parameters

- **cell** (*np.ndarray*) – 3x3 array of cell vectors.
- **positions** (*np.ndarray*) – Positions of atoms in the unit cell.
- **size1** (*int*) – Number of cells along the first cell vector.
- **size2** (*int*) – Number of cells along the second cell vector.
- **size3** (*int*) – Number of cells along the third cell vector.

Returns Positions of atoms in the supercell.

Return type np.ndarray

1.3.6.6 I/O for trajectories

flare.utils.flare_io.**md_trajectory_from_file**(*filename: str*)

Read a list of structures from a json file, formatted as in md_trajectory_to_file. :param filename:

flare.utils.flare_io.**md_trajectory_to_file**(*filename: str, structures: List[<sphinx.ext.autodoc.importer._MockObject object at 0x7fda8018f860>]*)

Take a list of structures and write them to a json file. :param filename: :param structures:

1.4 C++ Extension

send_message(*sender*, *recipient*, *message_body*[, *priority=1*])

Send a message to a recipient

Parameters

- **sender** (*str*) – The person sending the message
- **recipient** (*str*) – The recipient of the message
- **message_body** (*str*) – The body of the message
- **priority** (*integer or None*) – The priority of the message, can be a number 1-5

Returns the message id

Return type int

Raises

- **ValueError** – if the message_body exceeds 160 characters
- **TypeError** – if the message_body is not a basestring

1.5 Frequently Asked Questions

1.5.1 Frequently Asked Questions

1.5.1.1 Installation and Packages

1. **What numba version will I need?** >= 0.43.0

Note: If you get errors with *numba* or get *C/C++*-type errors, very possibly it's the problem of the *numba* version.

2. **Can I accelerate my calculation using parallelization?** See the section in the Installation section.

1.5.1.2 Gaussian Processes and OTF

1. **I'm confused about how Gaussian Processes work.** Gaussian Processes enjoy a long history of study and there are many excellent resources out there we can recommend. One such resource (that some of the authors consult quite frequently!) is the textbook Gaussian Processes for Machine Learning, by Rasmussen and Williams , with Chapter 2 in particular being a great help.
2. **How should I choose my cutoffs?** The right cutoff depends on the system you're studying: ionic systems often do better with a larger 2-body cutoff, while dense systems like diamond require smaller cutoffs. We recommend you try a range of cutoff values and examine the model error, optimized noise parameter, and model likelihood as a function of the cutoff.

3. **What is a good strategy for hyperparameter optimization?** The hyperparameter optimization is important for obtaining a good model. However, the optimization of hyperparameters will get slower when more training data are collected. There are a few parameters to notice:

   In *GaussianProcess*,

   - *maxiter*: maximal number of iterations, usually set to ~10 to prevent training for too long.
   - *parallel*: if *True*, then parallelization is used in optimization. The serial version could be very slow.
   - *output* (in *train* function): set up an output file for monitoring optimization iterations.
   - *grad_tol*, *x_tol*, *line_steps* (in *train* function): can be changed for iterations.

   In *OTF*,

   - *freeze_hyps*: the hyperparameter will only be optimized for *freeze_hyps* times. Can be set to a small number if optimization is too expensive with a large data set.

1.5.1.3 GPFA

1. **My models are adding too many atoms from each frame, causing a serious slowdown without much gain in model accurac**
   In order to 'govern' the rate at which the model adds atoms, we suggest using the `pre_train_atoms_per_element` and `train_atoms_per_element` arguments, which can limit the number of atoms added from each seed frame and training frame respectively. You can pass in a dictionary like `{'H':1, 'Cu':2}` to limit the number of H atoms to 1 and Cu atoms to 2 from any given frame. You can also use `max_atoms_per_frame` for the same functionality.
2. **The uncertainty seems low on my force predictions, but the true errors in the forces are high.** This could be happening for a few reasons. One reason could be that your hyperparameters aren't at an optimum (check that the gradient of the likelihood with respect to the hyperparameters is small). Another is that your model, such as 2-body or 2+3 body, may not be of sufficient complexity to handle the system (in other words, many-body effects could be important).

1.5.1.4 MGP

1. **How does the grid number affect my mapping?**
   - The lower cutoff is better set to be a bit smaller than the minimal interatomic distance.
   - The upper cutoff should be consistent with GP's cutoff.
   - For three-body, the grid is 3-D, with lower cutoffs *[a, a, cos(pi)]* and upper cutoffs *[b, b, cos(0)]*.
   - You can try different grid numbers and compare the force prediction of MGP and GP on the same testing structure. Choose the grid number of satisfying efficiency and accuracy. A reference is *grid_num=64* should be safe for *a=2.5*, *b=5*.

1.6 Applications

If you use FLARE in your research, please let us know. We will list the applications of FLARE here.

1.7 How To Contribute

1.7.1 Git Workflow

To contribute to the FLARE source code, please follow the guidelines in this section. If any of the git commands are unfamiliar, check out Chapters 3-5 of the Pro Git book.

1.7.1.1 General workflow

Development should follow this pattern:

1. Create an issue on Github describing what you want to contribute.
2. Create a topic branch addressing the issue. (See the sections below on how to push branches directly or from a forked repository.)
3. Merge with the development branch when finished and close the issue.

1.7.1.2 Master, development, and topic branches

The FLARE repository has a three-tiered structure: there is the master branch, which is only for battle-tested code that is both documented and unit tested; the development branch, which is used to push new features; and topic branches, which focus on specific issues and are deleted once the issue is addressed.

You can create local copies of branches from the remote repository as follows:

```
$ git checkout -b <local branch name> origin/<remote branch name>
```

1.7.1.3 Pushing changes to the MIR repo directly

If you have write access to the MIR version of FLARE, you can make edits directly to the source code. Here are the steps you should follow:

1. Go into the development branch.
2. Create a new topic branch with a name describing what you're up to:

   ```
   $ git checkout -b <feature branch name>
   ```

3. Commit your changes periodically, and when you're done working, push the branch upstream:

   ```
   $ git push -u origin <feature branch name>
   ```

4. Create a Pull Request that gives a helpful description of what you've done. You can now merge and delete the branch.

1.7.1.4 Pushing changes from a forked repo

1. Fork the FLARE repository.
2. Set FLARE as an upstream remote:

   ```
   $ git remote add upstream https://github.com/mir-group/flare
   ```

Before branching off, make sure that your forked copy of the master branch is up to date:

```
$ git fetch upstream
$ git merge upstream/master
```

If for some reason there were changes made on the master branch of your forked repo, you can always force a reset:

```
$ git reset --hard upstream/master
```

3. Create a new branch with a name that describes the specific feature you want to work on:

```
$ git checkout -b <feature branch name>
```

4. While you're working, commit your changes periodically, and when you're done, commit a final time and then push up the branch:

```
$ git push -u origin <feature branch name>
```

5. When you go to Github, you'll now see an option to open a Pull Request for the topic branch you just pushed. Write a helpful description of the changes you made, and then create the Pull Request.

1.7.2 Code Standards

Before pushing code to the development branch, please make sure your changes respect the following code standards.

1.7.2.1 PEP 8

Run your code through pylint to check that you're in compliance with PEP 8:

```
$ pylint <file name>
```

1.7.2.2 Docstrings

All new modules, classes, and methods should have Sphinx style docstrings describing what the code does and what its inputs are. These docstrings are used to automatically generate FLARE's documentation, so please make sure they're clear and descriptive.

1.7.2.3 Tests

New features must be accompanied by unit and integration tests written using pytest. This helps ensure the code works properly and makes the code as a whole easier to maintain.

1.8 How to Cite

If you use FLARE in your research, please cite the following paper:

[1] Jonathan Vandermause, Steven B. Torrisi, Simon Batzner, Yu Xie, Lixin Sun, Alexie M. Kolpak, and Boris Kozinsky. On-the-fly active learning of interpretable Bayesian force fields for atomistic rare events. https://arxiv.org/abs/1904.02042

Thank you for using FLARE!

Python Module Index

f

flare.ase.calculator, 110
flare.ase.otf, 110
flare.dft_interface, 105
flare.dft_interface.cp2k_util, 102
flare.dft_interface.qe_util, 101
flare.dft_interface.vasp_util, 104
flare.env, 26
flare.gp, 85
flare.gp_algebra, 91
flare.gp_from_aimd, 112
flare.kernels.cutoffs, 82
flare.kernels.kernels, 83
flare.kernels.mc_3b_sepcut, 79
flare.kernels.mc_mb_sepcut, 80
flare.kernels.mc_sephyps, 61
flare.kernels.mc_simple, 41
flare.kernels.sc, 27
flare.mgp.mgp, 108
flare.mgp.splines_methods, 107
flare.otf, 105
flare.output, 94
flare.predict, 88
flare.struc, 25
flare.utils.element_coder, 113
flare.utils.env_getarray, 121
flare.utils.flare_io, 124
flare.utils.learner, 113
flare.utils.md_helper, 123
flare.utils.parameter_helper, 115

Index

A

add_file() (*in module flare.output*), 96
add_one_env() (*flare.gp.GaussianProcess method*), 86, 98
add_stream() (*in module flare.output*), 97
adjust_cutoffs() (*flare.gp.GaussianProcess method*), 86, 98
all_separate_groups() (*flare.utils.parameter_helper.ParameterHelper method*), 117
as_dict() (*flare.env.AtomicEnvironment method*), 27
as_dict() (*flare.gp.GaussianProcess method*), 86, 98
as_dict() (*flare.mgp.mgp.MappedGaussianProcess method*), 109
as_dict() (*flare.utils.parameter_helper.ParameterHelper method*), 118
as_dict() (*in module flare.struc*), 25
as_object() (*flare.utils.parameter_helper.ParameterHelper method*), 118
as_str() (*in module flare.struc*), 25
ASE_OTF (*class in flare.ase.otf*), 110
AtomicEnvironment (*class in flare.env*), 26

B

backward_arguments() (*flare.gp.GaussianProcess static method*), 86, 98
backward_attributes() (*flare.gp.GaussianProcess static method*), 86, 98

C

calculate() (*flare.ase.calculator.FLARE_Calculator method*), 110
check_instantiation() (*flare.gp.GaussianProcess method*), 86, 98
check_L_alpha() (*flare.gp.GaussianProcess method*), 86, 98
check_vasprun() (*in module flare.dft_interface.vasp_util*), 104
coded_species (*Structure attribute*), 25
compute_matrices() (*flare.gp.GaussianProcess method*), 86, 98
compute_properties() (*flare.ase.otf.ASE_OTF method*), 111
compute_properties() (*flare.otf.OTF method*), 106
conclude_run() (*flare.output.Output method*), 94
coordination_number() (*in module flare.kernels.kernels*), 83
cosine_cutoff() (*in module flare.kernels.cutoffs*), 82
cubic_cutoff() (*in module flare.kernels.cutoffs*), 82
CubicSpline (*class in flare.mgp.splines_methods*), 107

D

default() (*flare.utils.element_coder.NumpyEncoder method*), 113
define_group() (*flare.utils.parameter_helper.ParameterHelper method*), 118
dft_input_to_structure() (*in module flare.dft_interface.cp2k_util*), 102
dft_input_to_structure() (*in module flare.dft_interface.qe_util*), 101
dft_input_to_structure() (*in module flare.dft_interface.vasp_util*), 104

E

edit_dft_input_positions() (*in module flare.dft_interface.cp2k_util*), 102
edit_dft_input_positions() (*in module flare.dft_interface.qe_util*), 101
edit_dft_input_positions() (*in module flare.dft_interface.vasp_util*), 104
efs_energy_vector() (*in module flare.gp_algebra*), 91

efs_force_vector() (*in module flare.gp_algebra*), 91
element_to_Z() (*in module flare.utils.element_coder*), 113
energy_energy_vector() (*in module flare.gp_algebra*), 91
energy_energy_vector_unit() (*in module flare.gp_algebra*), 91
energy_force_vector() (*in module flare.gp_algebra*), 91
energy_force_vector_unit() (*in module flare.gp_algebra*), 91

F

fill_in_parameters() (*flare.utils.parameter_helper.ParameterHelper method*), 119
find_group() (*flare.utils.parameter_helper.ParameterHelper method*), 119
flare.ase.calculator (*module*), 110
flare.ase.otf (*module*), 110
flare.dft_interface (*module*), 105
flare.dft_interface.cp2k_util (*module*), 102
flare.dft_interface.qe_util (*module*), 101
flare.dft_interface.vasp_util (*module*), 104
flare.env (*module*), 26
flare.gp (*module*), 85, 97
flare.gp_algebra (*module*), 91
flare.gp_from_aimd (*module*), 112
flare.kernels.cutoffs (*module*), 82
flare.kernels.kernels (*module*), 83
flare.kernels.mc_3b_sepcut (*module*), 79
flare.kernels.mc_mb_sepcut (*module*), 80
flare.kernels.mc_sephyps (*module*), 61
flare.kernels.mc_simple (*module*), 41
flare.kernels.sc (*module*), 27
flare.mgp.mgp (*module*), 108
flare.mgp.splines_methods (*module*), 107
flare.otf (*module*), 105
flare.output (*module*), 94
flare.predict (*module*), 88
flare.struc (*module*), 25
flare.utils.element_coder (*module*), 113
flare.utils.env_getarray (*module*), 121
flare.utils.flare_io (*module*), 124
flare.utils.learner (*module*), 113
flare.utils.md_helper (*module*), 123
flare.utils.parameter_helper (*module*), 115
FLARE_Calculator (*class in flare.ase.calculator*), 110
force_energy_vector() (*in module flare.gp_algebra*), 91
force_energy_vector_unit() (*in module flare.gp_algebra*), 91
force_force_vector() (*in module flare.gp_algebra*), 91
force_force_vector_unit() (*in module flare.gp_algebra*), 92
force_helper() (*in module flare.kernels.kernels*), 84
from_dict() (*flare.env.AtomicEnvironment static method*), 27
from_dict() (*flare.gp.GaussianProcess static method*), 86, 98
from_dict() (*flare.mgp.mgp.MappedGaussianProcess static method*), 109
from_dict() (*flare.utils.parameter_helper.ParameterHelper static method*), 119
from_file() (*flare.gp.GaussianProcess static method*), 86, 98

G

GaussianProcess (*class in flare.gp*), 85, 97
get_2_body_arrays_jit() (*in module flare.utils.env_getarray*), 121
get_3_body_arrays_jit() (*in module flare.utils.env_getarray*), 122
get_force_block() (*in module flare.gp_algebra*), 92
get_force_block_pack() (*in module flare.gp_algebra*), 92
get_ky_and_hyp() (*in module flare.gp_algebra*), 92
get_ky_and_hyp_pack() (*in module flare.gp_algebra*), 92
get_like_from_mats() (*in module flare.gp_algebra*), 93
get_like_grad_from_mats() (*in module flare.gp_algebra*), 93
get_m2_body_arrays_jit() (*in module flare.utils.env_getarray*), 122
get_m3_body_arrays() (*in module flare.utils.env_getarray*), 123
get_max_cutoff() (*in module flare.utils.learner*), 113
get_neg_like_grad() (*in module flare.gp_algebra*), 93
get_random_velocities() (*in module flare.utils.md_helper*), 123
get_supercell_positions() (*in module flare.utils.md_helper*), 124
get_unique_species() (*in module flare.struc*), 25
grid (*flare.mgp.splines_methods.CubicSpline attribute*), 107

H

hard_cutoff() (*in module flare.kernels.cutoffs*), 83

I

indices_of_specie() (*in module flare.struc*), 25
interpolate() (*flare.mgp.splines_methods.CubicSpline method*), 107
is_force_in_bound_per_species() (*in module flare.utils.learner*), 113
is_std_in_bound() (*in module flare.utils.learner*), 114
is_std_in_bound_per_species() (*in module flare.utils.learner*), 114

K

k_sq_exp_dev() (*in module flare.kernels.kernels*), 84
k_sq_exp_double_dev() (*in module flare.kernels.kernels*), 84

L

list_groups() (*flare.utils.parameter_helper.ParameterHelper method*), 119
list_parameters() (*flare.utils.parameter_helper.ParameterHelper method*), 120

M

many_body() (*in module flare.kernels.sc*), 28
many_body_en() (*in module flare.kernels.sc*), 28
many_body_en_jit() (*in module flare.kernels.sc*), 28
many_body_force_en() (*in module flare.kernels.sc*), 29
many_body_force_en_jit() (*in module flare.kernels.sc*), 29
many_body_grad() (*in module flare.kernels.sc*), 29
many_body_grad_jit() (*in module flare.kernels.sc*), 30
many_body_jit() (*in module flare.kernels.sc*), 30
many_body_mc() (*in module flare.kernels.mc_sephyps*), 63
many_body_mc() (*in module flare.kernels.mc_simple*), 41
many_body_mc_en() (*in module flare.kernels.mc_sephyps*), 64
many_body_mc_en() (*in module flare.kernels.mc_simple*), 41
many_body_mc_en_jit() (*in module flare.kernels.mc_simple*), 42
many_body_mc_en_sepcut_jit() (*in module flare.kernels.mc_mb_sepcut*), 80
many_body_mc_force_en() (*in module flare.kernels.mc_sephyps*), 64
many_body_mc_force_en() (*in module flare.kernels.mc_simple*), 42
many_body_mc_force_en_jit() (*in module flare.kernels.mc_simple*), 43
many_body_mc_force_en_sepcut_jit() (*in module flare.kernels.mc_mb_sepcut*), 80
many_body_mc_grad() (*in module flare.kernels.mc_sephyps*), 64
many_body_mc_grad() (*in module flare.kernels.mc_simple*), 43
many_body_mc_grad_jit() (*in module flare.kernels.mc_simple*), 43
many_body_mc_grad_sepcut_jit() (*in module flare.kernels.mc_mb_sepcut*), 81
many_body_mc_jit() (*in module flare.kernels.mc_simple*), 44
many_body_mc_sepcut_jit() (*in module flare.kernels.mc_mb_sepcut*), 81
MappedGaussianProcess (*class in flare.mgp.mgp*), 108
mb_grad_helper_ls() (*in module flare.kernels.kernels*), 84
mb_grad_helper_ls_() (*in module flare.kernels.kernels*), 84
md_step() (*flare.ase.otf.ASE_OTF method*), 111
md_step() (*flare.otf.OTF method*), 106
md_trajectory_from_file() (*in module flare.utils.flare_io*), 124
md_trajectory_from_vasprun() (*in module flare.dft_interface.vasp_util*), 104
md_trajectory_to_file() (*in module flare.utils.flare_io*), 124
multicomponent_velocities() (*in module flare.utils.md_helper*), 124

N

NumpyEncoder (*class in flare.utils.element_coder*), 113

O

obtain_noise_len() (*in module flare.gp_algebra*), 93
open_new_log() (*flare.output.Output method*), 94
OTF (*class in flare.otf*), 105
Output (*class in flare.output*), 94

P

par (*flare.gp.GaussianProcess attribute*), 86, 98
ParameterHelper (*class in flare.utils.parameter_helper*), 117
parse_dft_forces() (*in module flare.dft_interface.cp2k_util*), 102
parse_dft_forces() (*in module flare.dft_interface.qe_util*), 101
parse_dft_forces() (*in module flare.dft_interface.vasp_util*), 104
parse_dft_forces_and_energy() (*in module flare.dft_interface.cp2k_util*), 103

parse_dft_forces_and_energy() (*in module flare.dft_interface.qe_util*), 101
parse_dft_forces_and_energy() (*in module flare.dft_interface.vasp_util*), 104
parse_dft_input() (*in module flare.dft_interface.cp2k_util*), 103
parse_dft_input() (*in module flare.dft_interface.qe_util*), 101
parse_dft_input() (*in module flare.dft_interface.vasp_util*), 104
parse_trajectory_trainer_output() (*in module flare.gp_from_aimd*), 112
partition_force_energy_block() (*in module flare.gp_algebra*), 93
partition_matrix() (*in module flare.gp_algebra*), 94
partition_matrix_custom() (*in module flare.gp_algebra*), 94
partition_vector() (*in module flare.gp_algebra*), 94
PCASplines (*class in flare.mgp.splines_methods*), 107
predict() (*flare.gp.GaussianProcess method*), 87, 99
predict() (*flare.mgp.mgp.MappedGaussianProcess method*), 109
predict_efs() (*flare.gp.GaussianProcess method*), 87, 99
predict_local_energy() (*flare.gp.GaussianProcess method*), 87, 99
predict_local_energy_and_var() (*flare.gp.GaussianProcess method*), 87, 99
predict_on_atom() (*in module flare.predict*), 88
predict_on_atom_efs() (*in module flare.predict*), 89
predict_on_atom_en() (*in module flare.predict*), 89
predict_on_atom_en_std() (*in module flare.predict*), 89
predict_on_structure() (*in module flare.predict*), 89
predict_on_structure_en() (*in module flare.predict*), 89
predict_on_structure_mgp() (*in module flare.predict*), 90
predict_on_structure_par() (*in module flare.predict*), 90
predict_on_structure_par_en() (*in module flare.predict*), 90

Q

q3_value_mc() (*in module flare.utils.env_getarray*), 123
q_value() (*in module flare.kernels.kernels*), 84
q_value_mc() (*in module flare.kernels.kernels*), 85
quadratic_cutoff() (*in module flare.kernels.cutoffs*), 83
quadratic_cutoff_bound() (*in module flare.kernels.cutoffs*), 83
queue_wrapper() (*in module flare.gp_algebra*), 94

R

remove_force_data() (*flare.gp.GaussianProcess method*), 87, 99
run() (*flare.otf.OTF method*), 106
run_dft() (*flare.otf.OTF method*), 106
run_dft() (*in module flare.dft_interface.vasp_util*), 104
run_dft_en_par() (*in module flare.dft_interface.cp2k_util*), 103
run_dft_en_par() (*in module flare.dft_interface.qe_util*), 101
run_dft_par() (*in module flare.dft_interface.cp2k_util*), 103
run_dft_par() (*in module flare.dft_interface.qe_util*), 102

S

send_message() (*built-in function*), 125
set_constraints() (*flare.utils.parameter_helper.ParameterHelper method*), 120
set_L_alpha() (*flare.gp.GaussianProcess method*), 87, 99
set_logger() (*in module flare.output*), 97
set_parameters() (*flare.utils.parameter_helper.ParameterHelper method*), 121
set_values() (*flare.mgp.splines_methods.CubicSpline method*), 107
Structure (*built-in class*), 24
subset_of_frame_by_element() (*in module flare.utils.learner*), 115
summarize_group() (*flare.utils.parameter_helper.ParameterHelper method*), 121
supercell_custom() (*in module flare.utils.md_helper*), 124

T

three_body() (*in module flare.kernels.sc*), 31
three_body_en() (*in module flare.kernels.sc*), 31
three_body_en_jit() (*in module flare.kernels.sc*), 32
three_body_force_en() (*in module flare.kernels.sc*), 32
three_body_force_en_jit() (*in module flare.kernels.sc*), 33
three_body_grad() (*in module flare.kernels.sc*), 33

three_body_grad_jit() (*in module flare.kernels.sc*), 34
three_body_jit() (*in module flare.kernels.sc*), 34
three_body_mc() (*in module flare.kernels.mc_sephyps*), 65
three_body_mc() (*in module flare.kernels.mc_simple*), 45
three_body_mc_en() (*in module flare.kernels.mc_sephyps*), 66
three_body_mc_en() (*in module flare.kernels.mc_simple*), 45
three_body_mc_en_jit() (*in module flare.kernels.mc_simple*), 46
three_body_mc_force_en() (*in module flare.kernels.mc_sephyps*), 67
three_body_mc_force_en() (*in module flare.kernels.mc_simple*), 46
three_body_mc_force_en_jit() (*in module flare.kernels.mc_sephyps*), 67
three_body_mc_force_en_jit() (*in module flare.kernels.mc_simple*), 47
three_body_mc_force_en_sepcut_jit() (*in module flare.kernels.mc_3b_sepcut*), 79
three_body_mc_grad() (*in module flare.kernels.mc_sephyps*), 68
three_body_mc_grad() (*in module flare.kernels.mc_simple*), 48
three_body_mc_grad_jit() (*in module flare.kernels.mc_sephyps*), 68
three_body_mc_grad_jit() (*in module flare.kernels.mc_simple*), 48
three_body_mc_grad_sepcut_jit() (*in module flare.kernels.mc_3b_sepcut*), 79
three_body_mc_jit() (*in module flare.kernels.mc_simple*), 49
three_body_se_jit() (*in module flare.kernels.mc_simple*), 50
three_body_sf_jit() (*in module flare.kernels.mc_simple*), 51
three_body_ss_jit() (*in module flare.kernels.mc_simple*), 52
to_pmg_structure() (*in module flare.struc*), 25
to_xyz() (*in module flare.struc*), 25
train() (*flare.gp.GaussianProcess method*), 87, 99
train_gp() (*flare.otf.OTF method*), 106
training_statistics (*flare.gp.GaussianProcess attribute*), 88, 100
two_body() (*in module flare.kernels.sc*), 35
two_body_en() (*in module flare.kernels.sc*), 36
two_body_en_jit() (*in module flare.kernels.sc*), 36
two_body_force_en() (*in module flare.kernels.sc*), 36
two_body_force_en_jit() (*in module flare.kernels.sc*), 36
two_body_grad() (*in module flare.kernels.sc*), 37
two_body_grad_jit() (*in module flare.kernels.sc*), 37
two_body_jit() (*in module flare.kernels.sc*), 38
two_body_mc() (*in module flare.kernels.mc_sephyps*), 69
two_body_mc() (*in module flare.kernels.mc_simple*), 52
two_body_mc_en() (*in module flare.kernels.mc_sephyps*), 69
two_body_mc_en() (*in module flare.kernels.mc_simple*), 53
two_body_mc_en_jit() (*in module flare.kernels.mc_sephyps*), 70
two_body_mc_en_jit() (*in module flare.kernels.mc_simple*), 53
two_body_mc_force_en() (*in module flare.kernels.mc_sephyps*), 70
two_body_mc_force_en() (*in module flare.kernels.mc_simple*), 53
two_body_mc_force_en_jit() (*in module flare.kernels.mc_sephyps*), 71
two_body_mc_force_en_jit() (*in module flare.kernels.mc_simple*), 54
two_body_mc_grad() (*in module flare.kernels.mc_sephyps*), 71
two_body_mc_grad() (*in module flare.kernels.mc_simple*), 54
two_body_mc_grad_jit() (*in module flare.kernels.mc_sephyps*), 72
two_body_mc_grad_jit() (*in module flare.kernels.mc_simple*), 55
two_body_mc_jit() (*in module flare.kernels.mc_sephyps*), 72
two_body_mc_jit() (*in module flare.kernels.mc_simple*), 55
two_body_mc_stress_en_jit() (*in module flare.kernels.mc_simple*), 56
two_body_mc_stress_force_jit() (*in module flare.kernels.mc_simple*), 56
two_body_mc_stress_stress_jit() (*in module flare.kernels.mc_simple*), 57
two_plus_three_body() (*in module flare.kernels.sc*), 38
two_plus_three_body_grad() (*in module flare.kernels.sc*), 38
two_plus_three_body_mc() (*in module flare.kernels.mc_sephyps*), 72
two_plus_three_body_mc() (*in module flare.kernels.mc_simple*), 57
two_plus_three_body_mc_grad() (*in module flare.kernels.mc_sephyps*), 73
two_plus_three_body_mc_grad() (*in module flare.kernels.mc_simple*), 58

two_plus_three_en() (*in module flare.kernels.sc*), 39
two_plus_three_force_en() (*in module flare.kernels.sc*), 39
two_plus_three_mc_en() (*in module flare.kernels.mc_sephyps*), 74
two_plus_three_mc_en() (*in module flare.kernels.mc_simple*), 58
two_plus_three_mc_force_en() (*in module flare.kernels.mc_sephyps*), 74
two_plus_three_mc_force_en() (*in module flare.kernels.mc_simple*), 59
two_plus_three_plus_many_body() (*in module flare.kernels.sc*), 39
two_plus_three_plus_many_body_en() (*in module flare.kernels.sc*), 40
two_plus_three_plus_many_body_force_en() (*in module flare.kernels.sc*), 40
two_plus_three_plus_many_body_grad() (*in module flare.kernels.sc*), 40
two_plus_three_plus_many_body_mc() (*in module flare.kernels.mc_simple*), 59
two_plus_three_plus_many_body_mc_en() (*in module flare.kernels.mc_simple*), 59
two_plus_three_plus_many_body_mc_force_en() (*in module flare.kernels.mc_simple*), 60
two_plus_three_plus_many_body_mc_grad() (*in module flare.kernels.mc_simple*), 60
two_three_many_body_mc() (*in module flare.kernels.mc_sephyps*), 75
two_three_many_body_mc_grad() (*in module flare.kernels.mc_sephyps*), 76
two_three_many_mc_en() (*in module flare.kernels.mc_sephyps*), 77
two_three_many_mc_force_en() (*in module flare.kernels.mc_sephyps*), 78

U

update_db() (*flare.gp.GaussianProcess method*), 88, 100
update_gp() (*flare.ase.otf.ASE_OTF method*), 111
update_gp() (*flare.otf.OTF method*), 106
update_L_alpha() (*flare.gp.GaussianProcess method*), 88, 100
update_positions() (*flare.ase.otf.ASE_OTF method*), 112
update_positions() (*flare.otf.OTF method*), 107
update_temperature() (*flare.ase.otf.ASE_OTF method*), 112
update_temperature() (*flare.otf.OTF method*), 107

V

vec_eval_cubic_spline() (*in module flare.mgp.splines_methods*), 108

W

write_gp_dft_comparison() (*flare.output.Output method*), 94
write_header() (*flare.output.Output method*), 95
write_hyps() (*flare.output.Output method*), 95
write_lmp_file() (*flare.mgp.mgp.MappedGaussianProcess method*), 110
write_md_config() (*flare.output.Output method*), 95
write_model() (*flare.gp.GaussianProcess method*), 88, 100
write_model() (*flare.mgp.mgp.MappedGaussianProcess method*), 110
write_to_log() (*flare.output.Output method*), 96
write_xyz() (*flare.output.Output method*), 96
write_xyz_config() (*flare.output.Output method*), 96

Z

Z_to_element() (*in module flare.utils.element_coder*), 113