
flare Documentation

Release 0.0.1

Jonathan Vandermause

Feb 26, 2022

1	Contents	3
1.1	Installation	3
1.1.1	Installation of FLARE	3
1.1.1.1	Requirements	3
1.1.1.2	Installation using pip	3
1.1.1.3	Manual Installation with Git	4
1.1.1.4	Acceleration with multiprocessing and MKL	4
1.1.1.5	Environment variables (optional)	4
1.1.2	Compile LAMMPS with MGP Pair Style	5
1.1.2.1	Download	5
1.1.2.2	MPI	5
1.1.2.3	MPI+OpenMP through Kokkos	5
1.1.2.4	MPI+CUDA through Kokkos	6
1.1.2.5	Notes on Newton (only relevant with Kokkos)	7
1.2	Tutorials	7
1.2.1	Introduction to FLARE: Fast Learning of Atomistic Rare Events	7
1.2.1.1	Installation	9
1.2.1.2	Machine learned force fields	9
1.2.1.3	Learning a force field on the fly	17
1.2.2	Prepare your data	26
1.2.2.1	VASP data	26
1.2.2.2	Data from Quantum Espresso, LAMMPS, etc.	26
1.2.2.3	Try building GP from data	27
1.2.3	Training a Gaussian Process from an AIMD Run	28
1.2.3.1	Roadmap Figure	28
1.2.3.2	Step 1: Setting up a Gaussian Process Object	28
1.2.3.3	Step 2 (Optional): Extracting the Frames from a previous AIMD Run	30
1.2.3.4	Step 3: Training your Gaussian Process	30
1.2.3.5	Pre-Training arguments	31
1.2.4	On-the-fly training using ASE	32
1.2.4.1	Step 1: Set up supercell with ASE	32
1.2.4.2	Step 2: Set up FLARE calculator	32
1.2.4.3	Step 3: Set up DFT calculator	33
1.2.4.4	Step 4: Set up On-The-Fly MD engine	34
1.2.4.5	Step 5 (Optional): Resume Interrupted Training	35
1.2.5	After Training	36

1.2.5.1	Parse OTF log file	36
1.2.5.2	Construct GP model from log file	36
1.2.5.3	Map the GP force field & Dump LAMMPS coefficient file	37
1.2.5.4	Run LAMMPS with MGP pair style	37
1.3	Code Documentation	39
1.3.1	Bayesian Force Fields	39
1.3.1.1	Gaussian Process Force Fields	39
1.3.1.2	Mapped Gaussian Process	46
1.3.1.3	Sparse Gaussian Process Force Fields	49
1.3.2	Bayesian Active Learning	49
1.3.2.1	On-the-Fly Training	49
1.3.2.2	GP From AIMD	51
1.3.2.3	Conditions to add training data	52
1.3.3	ASE MD Engine	54
1.3.3.1	Fake MD	54
1.3.3.2	LAMMPS Calculator and MD	54
1.3.3.3	NoseHoover (NVT Ensemble)	55
1.3.4	Descriptors	56
1.3.4.1	Atomic Environments	56
1.3.5	Kernels	56
1.3.5.1	Single-element Kernels	56
1.3.5.2	Multi-element Kernels (simple)	72
1.3.5.3	Multi-element Kernels (Separate Parameters)	93
1.3.5.4	Cutoff Functions	114
1.3.5.5	Helper Functions	115
1.3.6	File Input and Output	117
1.3.6.1	Output	117
1.3.6.2	OTF Parser	120
1.3.7	Utility	121
1.3.7.1	Advanced Hyperparameters Set Up	121
1.3.7.2	Construct Atomic Environment	127
1.4	Frequently Asked Questions	130
1.4.1	Frequently Asked Questions	130
1.4.1.1	Installation and Packages	130
1.4.1.2	Gaussian Processes	130
1.4.1.3	OTF (On-the-fly) Training	130
1.4.1.4	GPFA	131
1.4.1.5	MGP	131
1.5	Applications	132
1.6	How To Contribute	132
1.6.1	Git Workflow	132
1.6.1.1	General workflow	132
1.6.1.2	Master, development, and topic branches	133
1.6.1.3	Pushing changes to the MIR repo directly	133
1.6.1.4	Pushing changes from a forked repo	133
1.6.2	Code Standards	134
1.6.2.1	PEP 8	134
1.6.2.2	Docstrings	134
1.6.2.3	Tests	134
1.7	How to Cite	134

Python Module Index	135
----------------------------	------------

Index	137
--------------	------------

FLARE



1.1 Installation

1.1.1 Installation of FLARE

1.1.1.1 Requirements

- Python
- NumPy
- SciPy
- Memory_profiler
- Numba

Optional:

- ASE
- Pymatgen

1.1.1.2 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
```

1.1.1.3 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
```

1.1.1.4 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 parallelization 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_THREADS=2
python training.py
```

Note: The “n_cpus” and OMP_NUM_THREADS 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 OMP_NUM_THREADS>1, it is equivalent to run with OMP_NUM_THREADS * 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.

1.1.1.5 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)
```

(continues on next page)

(continued from previous page)

```
# the path and filename of LAMMPS executable
export lmp=$(which lmp_mpi)
```

1.1.2 Compile LAMMPS with MGP Pair Style

Anders Johansson, Yu Xie

1.1.2.1 Download

If you want to use MGP force field in LAMMPS, the first step is to download our MGP pair_style source code from: https://github.com/mir-group/flare/tree/master/lammps_plugins

Then, follow the instruction below to compile the LAMMPS executable.

1.1.2.2 MPI

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

Compiling

```
cp -r lammps_plugin /path/to/lammps/src/USER-MGP
cd /path/to/lammps/src
make yes-user-mgp
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.

1.1.2.3 MPI+OpenMP through Kokkos

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

Compiling

```
cp -r lammps_plugin /path/to/lammps/src/USER-MGP
cd /path/to/lammps/src
make yes-user-mgp
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.

1.1.2.4 MPI+CUDA through Kokkos

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

Compiling

```
cp -r lammps_plugin /path/to/lammps/src/USER-MGP  
cd /path/to/lammps/src  
make yes-user-mgp  
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
```

1.1.2.5 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.

1.2 Tutorials

1.2.1 Introduction to FLARE: Fast Learning of Atomistic Rare Events

Jonathan Vandermause (jonathan_vandermause@g.harvard.edu)



Learning objectives: * Train 2+3-body Gaussian process models on *ab initio* force data. * Use the uncertainties of the GP to train a force field on the fly.

Computing the properties of real materials with quantum mechanical accuracy is extremely expensive. One of the most efficient explicitly quantum mechanical tools we have for the job, density functional theory (or DFT), has a computational cost that scales cubically with the number of atoms in the system, and is therefore limited in practice to at most a few hundred atoms (with some hope of scaling further with orbital free DFT methods).

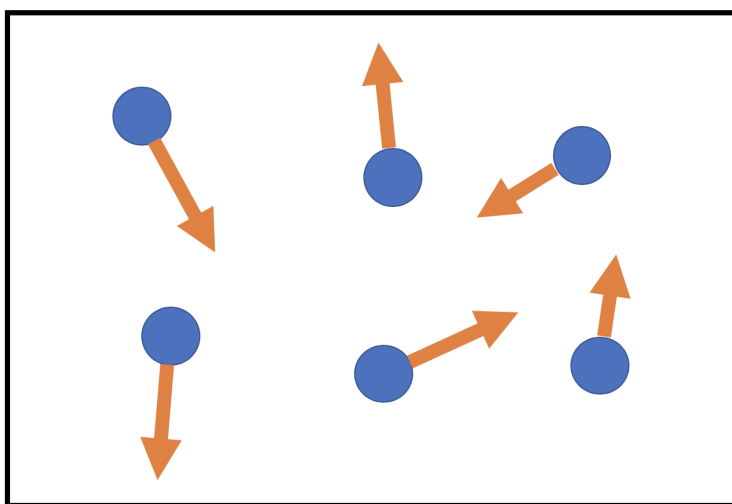
One of the most popular strategies for getting around this is to simply ignore the hard part of the problem—the electrons—and to instead try to model the system with an *empirical interatomic potential*, which expresses the potential energy as a sum over local, atom-centered contributions that depend only on atomic coordinates. The resulting model scales only linearly with the number of atoms, making it possible to simulate the behavior of hundreds of thousands or

even millions of atoms over microsecond timescales. The problem then becomes: how do you design a good potential? Is it possible to find a fast empirical potential with the accuracy of DFT?

FLARE is an open-source software that uses Bayesian machine learning to try to bridge the gap between accurate but slow quantum methods (like DFT) and fast but inaccurate classical methods (like empirical interatomic potentials). The idea is to train fast potentials on accurate DFT data, and ideally to do so in an automatic, closed loop fashion, so that a wide class of materials and material compositions can be efficiently explored. The key tool FLARE uses to accomplish this is Gaussian process (GP) regression, an elegant framework for defining probability distributions over functions. In this tutorial, we'll explore the use of GPs to model interactions between atoms in materials.

If you're interested in learning more about the FLARE code, check out our [paper](#), [GitHub page](#), and [code documentation](#).

Molecular dynamics

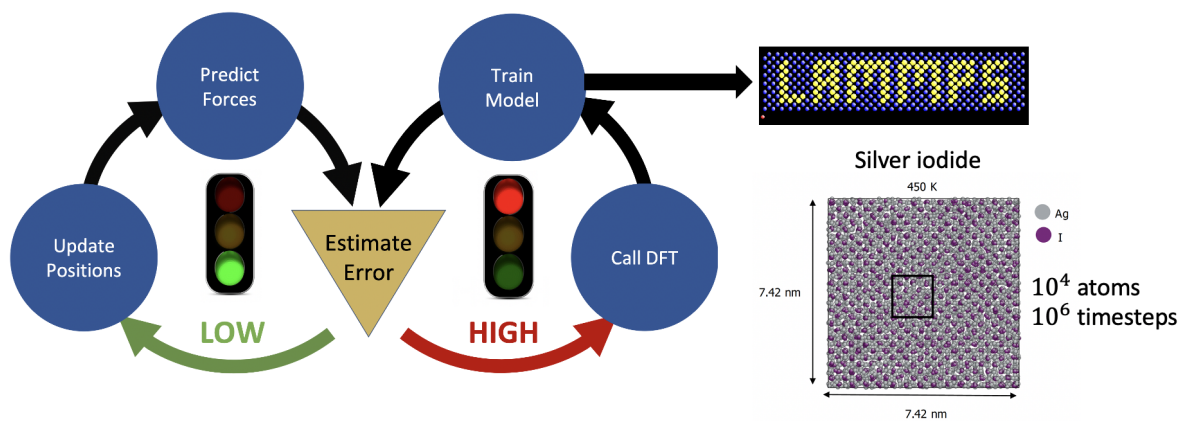


1. initialize
2. compute forces
3. update positions
4. repeat

To scale up, forces need to be computed cheaply and accurately.

Fast Learning of Atomistic Rare Events (FLARE)

- **Goal:** Automatically learn fast, accurate, multi-element force fields for large-scale MD.
- **Tool:** Bayesian inference with Gaussian process regression.



1.2.1.1 Installation

We can install FLARE and its dependencies here in Google Colab. (This will take about a minute.)

```
[ ]: ! pip install --upgrade mir-flare
```

Let's check that it worked by trying to import FLARE:

```
[2]: import flare
```

If you don't see an error, you're all set for the tutorial! Let's also go ahead and import all the modules we'll need now:

```
[3]: from flare import gp, struc, output, predict, md, otf, env,\
      otf_parser
      from flare.kernels import mc_simple
      from flare.utils import md_helper

      import numpy as np
      import time
      import matplotlib.pyplot as plt
      import pickle
      from ase.visualize import view
```

1.2.1.2 Machine learned force fields

Training data

Let's start by downloading some *ab initio* force data to play with.

```
[ ]: # download ab initio MD data
      ! wget https://zenodo.org/record/3688843/files/AgI_data.zip?download=1

      # unzip the folder
      ! unzip AgI_data.zip?download=1
```

The folder “AgI_data” contains data from an *ab initio* molecular dynamics simulation of the fast-ion conductor silver iodide (AgI), where *ab initio* means “from the beginning” or “from first principles”. In other words, we simulate how the atoms move by calculating the quantum mechanical forces on the ions at every timestep using DFT. To give you a sense of how expensive that is, the ~2500 timesteps in this simulation (about 12 picoseconds of simulation time) required 2 days of wall time on 256 cpus. (And there were only 32 atoms in the simulation!)

Let's load the positions, forces, cell, and species of the atoms in the simulation.

```
[5]: # load AIMD training data
      data_directory = 'AgI_data/'
      species = np.load(data_directory + 'species.npy') # atomic numbers of the atoms
      positions = np.load(data_directory + 'positions.npy') # in angstrom (A)
      cell = np.load(data_directory + 'cell.npy') # 3x3 array of cell vectors (in A)
      forces = np.load(data_directory + 'forces.npy') # in eV/A
```

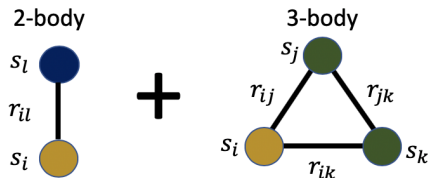
Training a GP model.

Low dimensional force fields

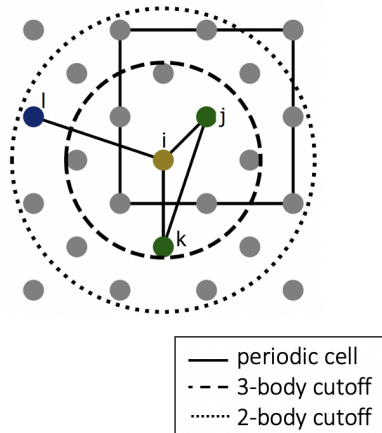
- The Born-Oppenheimer potential energy surface is high dimensional:

$$E = E(\vec{r}_1, \dots, \vec{r}_N).$$

- Simplifying assumption:** each cluster of 2 and 3 atoms in the local environment of atom i contributes to its local energy E_i .



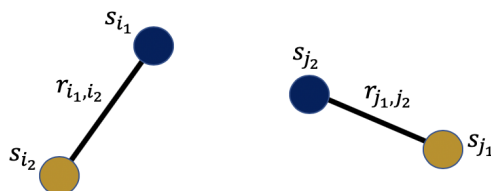
Local environment of atom i



- Regression task:** learn $\varepsilon_{s_i, s_l}(r_{il})$ and $\varepsilon_{s_i, s_j, s_k}(r_{ij}, r_{ik}, r_{jk})$ from *ab initio* force data.

N-body covariant kernels

- Compares interatomic distances of two clusters if they're of the same type:



$$k_2 = \underbrace{\sigma^2 \delta_{s_{i_1}, s_{j_1}} \delta_{s_{i_2}, s_{j_2}}}_{\text{multi-element}} \underbrace{\exp\left(-\frac{(r_{i_1, i_2} - r_{j_1, j_2})^2}{2 \ell^2}\right)}_{\text{rotationally invariant}} \underbrace{f_{\text{cut}}(r_{i_1, i_2}, r_{j_1, j_2})}_{\text{smooth}}$$

Gaussian process force fields

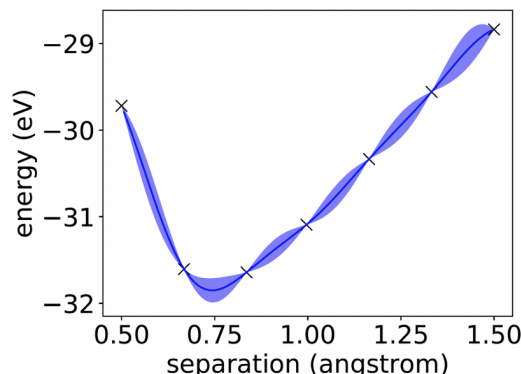
We use Gaussian processes regression to learn two- and three-body contributions $\varepsilon_{s_i, s_l}(r_{il})$ and $\varepsilon_{s_i, s_j, s_k}(r_{ij}, r_{ik}, r_{jk})$ from DFT force components.

The GP model estimates force components $f_{i\alpha}$ on all atoms and their uncertainties $\sigma_{i\alpha}$.

Force prediction:
$$f_{i\alpha} = \sum_t \underbrace{k(\rho_i, \rho_t)}_{\text{comparison to training environments}} \alpha_t$$

Uncertainty estimate:

$$\sigma_{\text{tot}, i\alpha}^2 = \underbrace{k_{\alpha, \alpha}(\rho_i, \rho_i) - \bar{k}_{i\alpha}^T (K + \sigma_n^2 \mathbf{I})^{-1} \bar{k}_{i\alpha}}_{\text{epistemic uncertainty } \sigma_{i\alpha}^2} + \underbrace{\sigma_n^2}_{\text{noise}}$$



Let's train a GP model on the data. We'll first initialize a Gaussian process object, which involves choosing a kernel function and its gradient, an initial set of hyperparameters, and the cutoff radii of local environments. We'll use a 2+3-body kernel, which compares local environments by comparing the pairs and triplets of atoms inside the environments.

```
[6]: # create a 2+3-body gaussian process object
kernels = ['twobody', 'threobody']
component = 'mc'
hyps = np.array([0.1, 0.1, 0.1, 2.0, 0.5]) # initial (bad) choice of hyps
cutoffs = {'twobody': 7.0, 'threobody': 5.5} # cutoff radii in Å
maxiter = 100 # max number of hyperparameter optimization steps

gp_model = gp.GaussianProcess(
    kernels=kernels,
    component=component,
    hyps=hyps,
    cutoffs=cutoffs,
    maxiter=50
)
```

Let's put a couple of structures into the training set.

```
[7]: # put a few snapshots in the training set
snapshots = [500, 1500]
for snapshot in snapshots:
    # create flare structure
    training_positions = positions[snapshot]
    training_forces = forces[snapshot]
    training_structure = struc.Structure(cell, species, training_positions)

    # add the structure to the training set of the GP
    gp_model.update_db(training_structure, training_forces)

gp_model.set_L_alpha()
```

The “set_L_alpha” method updates the covariance matrix of the GP, computes the alpha vector used to make predictions, and computes the log marginal likelihood of the training data. Let's see what that looks like:

```
[8]: print(gp_model.likelihood)
-491.75669961275526
```

This is too low, suggesting that our initial guess for the hyperparameters was not a good one.

Task: Find hyperparameters that give a positive log marginal likelihood.

Hint: For the 2+3-body kernel, the hyperparameters are in the following order: * Signal variance of the 2-body kernel (in eV) * Length scale of the 2-body kernel (in Å) * Signal variance of the 3-body kernel (in eV) * Length scale of the 3-body kernel (in Å) * Noise hyperparameter (in eV/Å)

What is a plausible length scale for this problem? Note that machine learned force fields typically have errors in the range 50-200 meV/Å, and the energy of a pair or triplet is much less than the total local energy assigned to an atom.

```
[ ]: # your code here

# hint: reset the hyperparameters with gp_model.hyps = *new hyps*, then use
# set_L_alpha to recompute the likelihood
```

Solution

A reasonable guess for the length scale is 1 Å, and since errors are often around 0.1 eV/Å, we'll choose that as our noise level. The signal variances require some tuning, but we should expect them to be significantly less than 1 eV, with the triplet contribution significantly smaller than the pair contribution.

```
[9]: gp_model.hyps = np.array([0.01, 1, 0.001, 1, 0.2])
gp_model.set_L_alpha()
print(gp_model.likelihood)
9.500574018895804
```

Optimizing the hyperparameters rigorously

The above was an exercise in manual hyperparameter tuning, which is tedious and should be avoided if possible. Because we can compute the gradient of the likelihood with respect to the hyperparameters, we can instead use a more principled gradient descent approach to find the best set of hyperparameters.

```
[ ]: # optimize the hyperparameters (this will take some time!)
gp_model.train(print_progress=True)
```

In case you don't want to wait, here are some good values:

```
[10]: gp_model.hyps = np.array([1.59612454e-02, 5.70104540e-01, 6.01290125e-04,
                               9.17243358e-01, 1.09666317e-01])
gp_model.set_L_alpha()
print(gp_model.likelihood)
71.16991245736048
```


Calculating the learning curve

Let's get a feel for how much data the model needs by computing the learning curve, i.e. the performance on a validation set as a function of the number of training points. Let's create training and validation structures drawn from the AIMD simulation.

```
[11]: # choose a training snapshot
training_snapshot = 500
training_positions = positions[training_snapshot]
training_forces = forces[training_snapshot]
training_structure = struc.Structure(cell, species, training_positions)

# choose a validation snapshot
validation_snapshot = 2300
validation_positions = positions[validation_snapshot]
validation_forces = forces[validation_snapshot]
validation_structure = struc.Structure(cell, species, validation_positions)
```

We now loop over the atomic environments in the training structure and add them one by one to the training set of the GP model. After adding an environment, we predict all the forces on the validation structure and compute the MAE. We'll also time the prediction step to get a feel for how the cost of GP predictions depends on the size of the training set.

```
[12]: # reset the gp with hyperparameters fixed to the optimized values
hyps_final = gp_model.hyps
gp_model = gp.GaussianProcess(
    kernels=kernels,
    component=component,
    hyps=hyps,
    cutoffs=cutoffs,
    maxiter=50
)
gp_model.hyps = hyps_final

# add atomic environments one by one to the training set
n_atoms = 32 # number of atoms in the structure
validation_errors = np.zeros(n_atoms)
prediction_times = np.zeros(n_atoms)
validation_count = 0

print('computing validation errors...')
for n, atom in enumerate(range(n_atoms)):
    # add the current atomic environment to the training set
    gp_model.update_db(training_structure, training_forces,
                      custom_range=[atom])
    gp_model.set_L_alpha()

    # predict the forces on the validation structure
    time0 = time.time()
    pred_forces, stds = \
        predict.predict_on_structure(validation_structure, gp_model)
    time1 = time.time()

    mae = np.mean(np.abs(pred_forces - validation_forces))
    validation_errors[validation_count] = mae
    prediction_times[validation_count] = time1 - time0
    validation_count += 1
```

(continues on next page)

(continued from previous page)

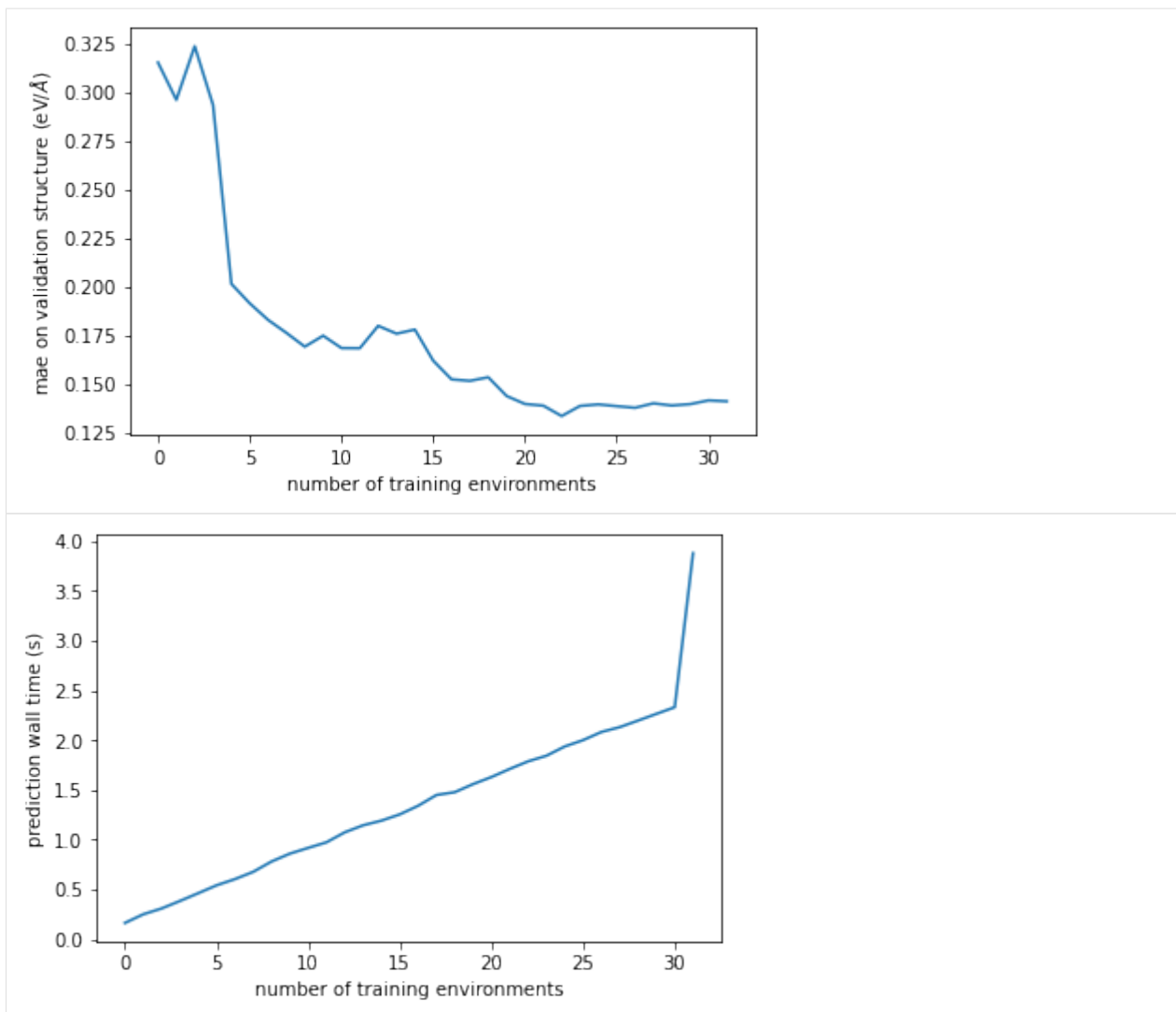
```
print(mae)

computing validation errors...
0.315073761272364
0.29588515439108337
0.3234313322939279
0.29323785059806706
0.20129532813447323
0.19131142668143372
0.1827709765647784
0.17610478136490326
0.16902120241436183
0.17466172091548085
0.16829757387913633
0.16823160886145053
0.1797266370933175
0.17570045468126091
0.1778018841113225
0.16181457527423726
0.15221658747132247
0.151463002400943
0.15330262545396417
0.14366990594229426
0.13955981466966869
0.1386958938185947
0.13337032134291107
0.13856470639127907
0.13934854695295343
0.1384439054077334
0.1376182149699538
0.1398980197468981
0.13878505117526346
0.13947592439212986
0.14140188721074384
0.14095157040791634
```

Let's make a plot of the results.

```
[13]: plt.plot(validation_errors)
plt.xlabel('number of training environments')
plt.ylabel('mae on validation structure (eV/$\AA$)')
plt.show()

plt.plot(prediction_times)
plt.xlabel('number of training environments')
plt.ylabel('prediction wall time (s)')
plt.show()
```



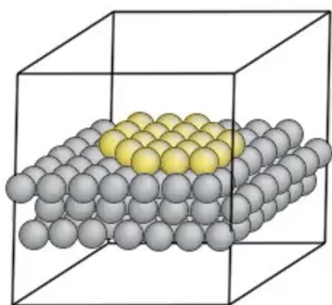
Notice that the prediction time grows roughly linearly with the number of training environments.



Toward large scale applications: Pd/Ag

- How does palladium dissolve into a silver surface?
- Relevant to the selectivity/reactivity tradeoff in bimetallic catalysts

Training simulation



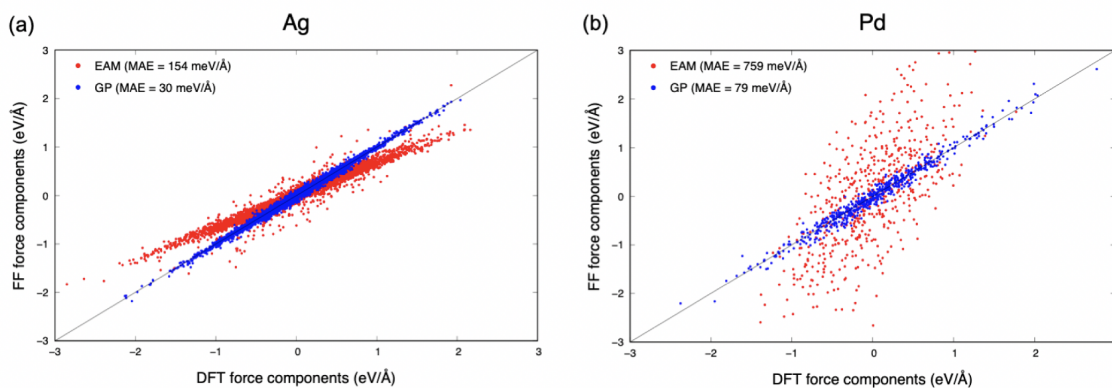
Training environments selected with automated, uncertainty driven active learning

# of Environments	# of CPUs	Wall Time
91	25	12.3 hours

J.S. Lim, J. Vandermause, M.A. von Spronsen *et al.* chemRxiv: 11811660

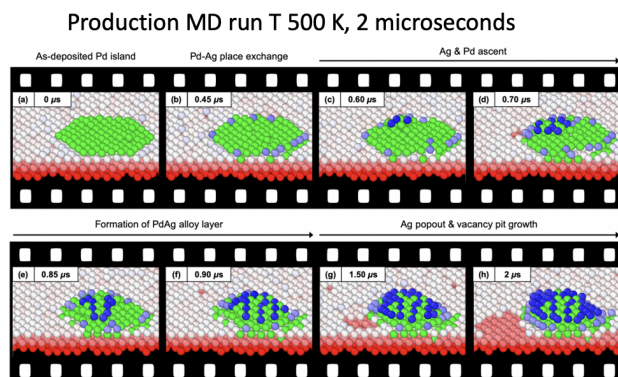
Toward large scale applications: Pd/Ag

Off-the-shelf classical force fields are inadequate for this problem.

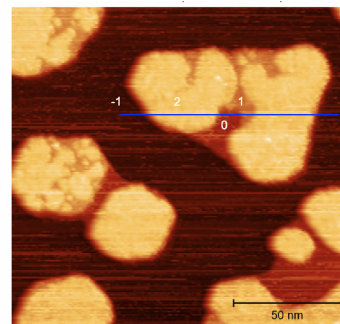


J.S. Lim, J. Vandermause, M.A. von Spronsen *et al.* chemRxiv: 11811660

Toward large scale applications: Pd/Ag

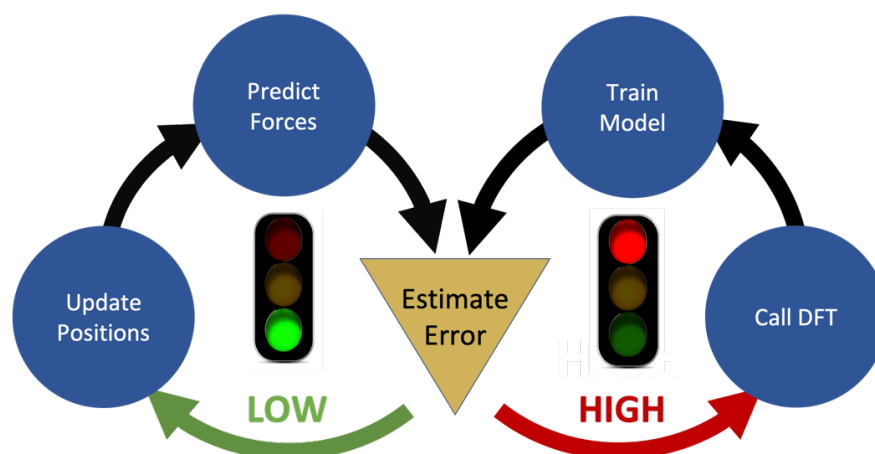


Vacancy pit growth consistent with STM images



J.S. Lim, J. Vandermause, M.A. von Spronsen *et al.* chemRxiv: 11811660

1.2.1.3 Learning a force field on the fly



The Lennard Jones Potential

In production FLARE runs, we make calls to a DFT solver like Quantum Espresso, VASP, or CP2K whenever the uncertainty on a force component is unacceptably high. In principle, the forces can come from anywhere, and FLARE allows users to write a custom solver that returns forces on an arbitrary input structure.

Here, we'll define a simple custom solver that returns Lennard Jones forces, and then try to reconstruct the potential on the fly.

The Lennard Jones potential takes the following form:

$$V_{LJ} = 4\epsilon \left[\left(\frac{\sigma}{r} \right)^{12} - \left(\frac{\sigma}{r} \right)^6 \right]. \quad (1.1)$$

Let's define our custom solver. To do this, we need to create a class with two methods:

- `parse_dft_input`: Takes the name of an input file, and returns the positions, species, cell, and masses of the atoms specified in the input file.
- `run_dft_par`: Takes a FLARE structure and an optional dictionary of keywords, and returns the forces on the atoms.

We'll make a multi-species Lennard Jones function, which allows different values of σ and ϵ for different pairs of species. We'll store these as 2-D numpy arrays, so that, e.g., the σ value assigned to species 1 and 3 is stored in element (1, 3) of this array.

The derivative of the Lennard Jones potential with respect to r is:

$$\frac{dV_{\text{LJ}}}{dr} = 4\epsilon \left[-\frac{12\sigma^{12}}{r^{13}} + \frac{6\sigma^6}{r^7} \right]. \quad (1.2)$$

```
[14]: def get_LJ_forces(structure, lj_parameters):
    """Calculate multicomponent Lennard Jones forces on a structure of atoms.
    dft_kwargs is assumed to be a dictionary containing the cutoff, an ordered
    list of species, and arrays containing epsilon and sigma values."""

    cutoff = lj_parameters['cutoff']
    epsilons = lj_parameters['epsilons']
    sigmas = lj_parameters['sigmas']
    spec_list = lj_parameters['species']

    forces = np.zeros((structure.nat, 3))
    # Loop over atoms in the structure.
    for m in range(structure.nat):
        # Create atomic environment.
        environment = env.AtomicEnvironment(structure, m, np.array([cutoff]))
        ind1 = spec_list.index(environment.ctype)

        # Loop over atoms in the environment to compute the total force.
        for n in range(len(environment.etypes)):
            ind2 = spec_list.index(environment.etypes[n])
            eps = epsilons[ind1, ind2]
            sig = sigmas[ind1, ind2]

            # Compute LJ force.
            bond_vals = environment.bond_array_2[n]
            r = bond_vals[0]

            dE_dr = 4 * eps * (-12 * sig ** 12 / (r ** 13) +
                               6 * sig ** 6 / (r ** 7))

            forces[m, 0] += dE_dr * bond_vals[1]
            forces[m, 1] += dE_dr * bond_vals[2]
            forces[m, 2] += dE_dr * bond_vals[3]

    return forces
```

We can use our Lennard Jones force function to define a custom module.

```
[15]: # create lj module
class lj_module:
    def parse_dft_input(file_name):
        """We assume the input is a pickled dictionary containing positions
```

(continues on next page)

(continued from previous page)

```
(in angstrom), species (as a list of strings), cell (as a 3x3 matrix of
cell vectors), and masses (as a dictionary assigning each species a mass
in AMU)."""
```

```
input_file = open(file_name, 'rb')
struc_dict = pickle.load(input_file)
input_file.close()
```

```
# Convert masses to MD units (energy = eV, length = A, )
masses = struc_dict['masses']
mass_convert = 0.000103642695727
for species in masses:
    masses[species] *= mass_convert
```

```
return struc_dict['positions'], struc_dict['species'], \
    struc_dict['cell'], masses
```

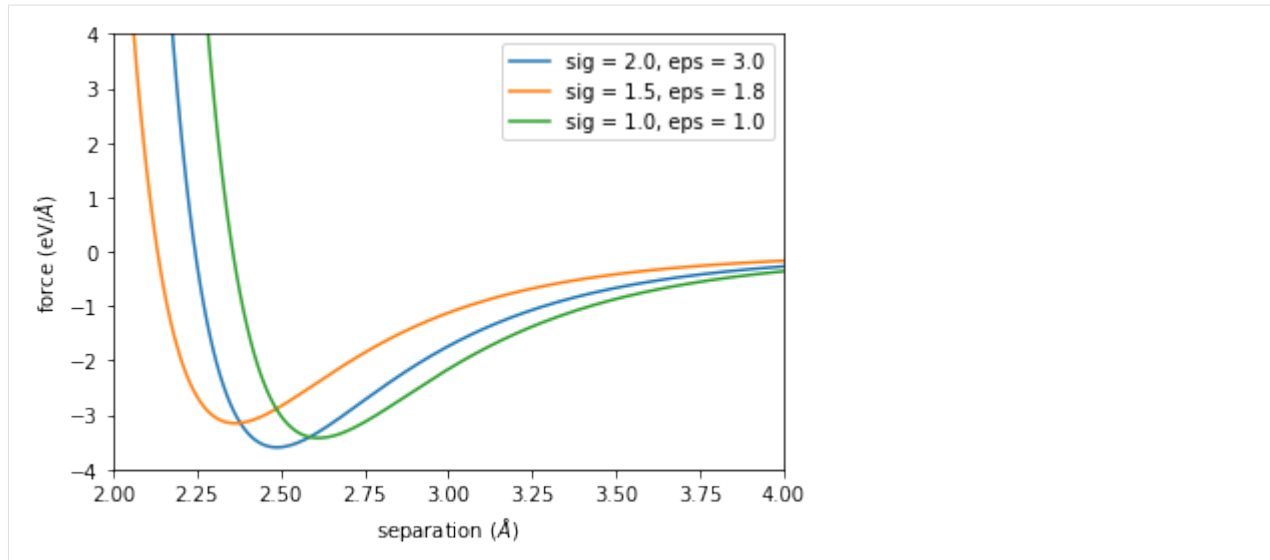
```
def run_dft_par(dft_input=None, structure=None, dft_loc=None, n_cpus=None,
               npool=None, mpi=None, dft_kwargs=None, dft_out=None):
    return get_LJ_forces(structure, dft_kwargs)
```

```
[16]: # create dictionary of lj parameters
cutoff = 5.
epsilons = np.array([[3, 2.5], [2.5, 3.]])
sigmas = np.array([[2.0, 1.9], [1.9, 2.1]])
spec_list = [47, 53] # silver and iodine atomic numbers

lj_params = {'cutoff': cutoff, 'epsilons': epsilons, 'sigmas': sigmas,
             'species': spec_list}
```

```
[17]: # create structures of 2 atoms, and plot the force on the second atom
cell = np.eye(3) * 1000
seps = np.arange(2, 5, 0.01)
specs = [[47, 47], [47, 53], [53, 53]]
store_fracs = np.zeros((3, len(seps)))
for m, sep in enumerate(seps):
    pos = np.array([[0, 0, 0], [sep, 0, 0]])
    for n, spec_curr in enumerate(specs):
        struc_curr = struc.Structure(cell, spec_curr, pos)
        fracs = \
            lj_module.run_dft_par(structure=struc_curr, dft_kwargs=lj_params)
        store_fracs[n, m] = fracs[1, 0]

plt.plot(seps, store_fracs[0], label='sig = 2.0, eps = 3.0')
plt.plot(seps, store_fracs[1], label='sig = 1.5, eps = 1.8')
plt.plot(seps, store_fracs[2], label='sig = 1.0, eps = 1.0')
plt.xlim(2, 4)
plt.ylim(-4, 4)
plt.xlabel('separation ($\AA$)')
plt.ylabel('force (eV/$\AA$)')
plt.legend()
plt.show()
```



Perform an on-the-fly training simulation.

Step 1: Set up the initial structure.

Let's create a bcc (body centered cubic) Lennard Jones crystal.

```
[18]: # create bcc unit cell
alat = 2.54
unit_cell = np.eye(3) * alat

# define bcc positions
unit_positions = np.array([[0, 0, 0],
                           [1/2, 1/2, 1/2]]) * alat

# make a supercell
sc_size = 3
positions = md_helper.get_supercell_positions(sc_size, unit_cell, unit_positions)
cell = unit_cell * sc_size

# jitter positions to give nonzero force on first frame
for atom_pos in positions:
    for coord in range(3):
        atom_pos[coord] += (2*np.random.random()-1) * 0.05

# create initial structure
species = ['Ag', 'I'] * sc_size ** 3
struc_curr = struc.Structure(cell, species, positions)

# create pseudo input file
mass_dictionary = {'Ag': 108, 'I': 127}
input_dictionary = {'positions': positions, 'species': species, 'cell': cell,
                   'masses': mass_dictionary}
input_file_name = 'lj.in'
with open(input_file_name, 'wb') as input_file:
    pickle.dump(input_dictionary, input_file)
```



```
[19]: # view the structure
view(struc_curr.to_ase_atoms(), viewer='x3d')

[19]: <IPython.core.display.HTML object>
```

Step 2: Set up a GP model.

```
[20]: # create gaussian process model
kernels = ['twobody']
component = 'mc'
hyps = np.array([0.1, 1., 0.06])
hyp_labels = ['Sigma', 'Length Scale', 'Noise']
cutoffs = {'twobody': 5.0}
maxiter = 50

gp_model = gp.GaussianProcess(
    kernels=kernels,
    component=component,
    hyps=hyps,
    hyp_labels=hyp_labels,
    cutoffs=cutoffs,
    maxiter=50
)
```

Step 3: Set up an OTF training object.

```
[21]: # create otf object
dt = 0.001 # time step (ps)
number_of_steps = 50
dft_loc = None # path to dft executable would usually go here
std_tolerance_factor = -0.01 # 10 meV/A
init_atoms = [0, 25, 50] # initial atoms added to the training set
max_atoms_added = 5 # number of atoms added when dft is called
freeze_hyps = 5 # no hyperparameter optimization after this many updates

# rescale the temperature halfway through the simulation
rescale_steps = [10] # rescale at step 10
rescale_temps = [1000]

otf_model = otf.OTF(
    # MD arguments
    dt,
    number_of_steps,
    rescale_steps=rescale_steps,
    rescale_temps=rescale_temps,
    # FLARE arguments
    gp=gp_model,
    # OTF arguments
    std_tolerance_factor=std_tolerance_factor,
    init_atoms=init_atoms,
    max_atoms_added=max_atoms_added,
    freeze_hyps=freeze_hyps,
    # DFT arguments
```

(continues on next page)

(continued from previous page)

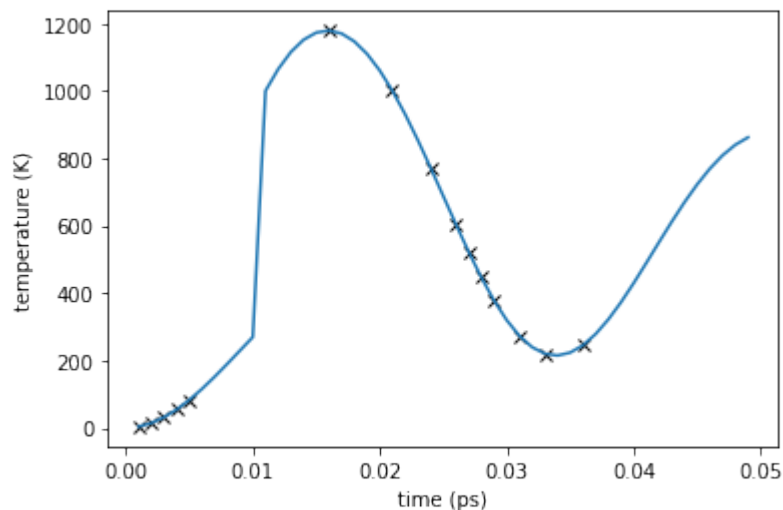
```
force_source=lj_module,  
dft_loc=dft_loc,  
dft_input=input_file_name,  
dft_kwargs=lj_params  
)
```

Step 4: Perform the simulation!

```
[22]: # perform flare run (this will take about 2.5 minutes)  
otf_model.run()
```

Parsing the OTF output file.

```
[23]: # parse output file  
output_file = 'otf_run.out'  
otf_trajectory = otf_parser.OtfAnalysis(output_file)  
  
# plot temperature vs. simulation time  
times = otf_trajectory.times  
temps = otf_trajectory.thermostat['temperature']  
dft_times = otf_trajectory.dft_times[1:] # exclude t = 0  
  
for n, dft_time in enumerate(dft_times):  
    otf_ind = times.index(dft_time)  
    plt.plot(dft_times[n], temps[otf_ind], 'kx')  
  
plt.plot(times, temps)  
plt.xlabel('time (ps)')  
plt.ylabel('temperature (K)')  
plt.show()
```



Comparing the learned model against ground truth.

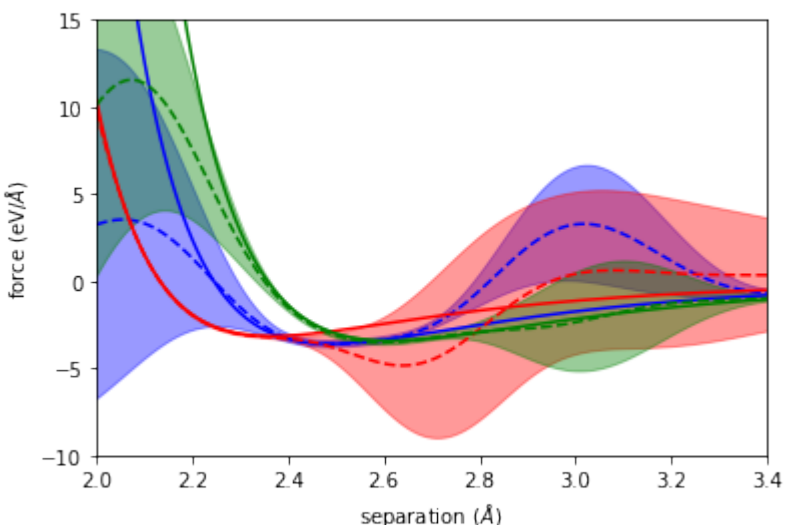
```
[24]: # create structures of 2 atoms, and plot the force on the second atom
cell = np.eye(3) * 1000
seps = np.arange(2, 5, 0.01)
specs = [[47, 47], [47, 53], [53, 53]]
store_fracs = np.zeros((3, len(seps)))
gp_fracs = np.zeros((3, len(seps)))
gp_stds = np.zeros((3, len(seps)))
for m, sep in enumerate(seps):
    pos = np.array([[0, 0, 0], [sep, 0, 0]])
    for n, spec_curr in enumerate(specs):
        struc_curr = struc.Structure(cell, spec_curr, pos)
        env_curr = env.AtomicEnvironment(struc_curr, 1, np.array([cutoff]))
        fracs = \
            lj_module.run_dft_par(structure=struc_curr, dft_kwargs=lj_params)
        store_fracs[n, m] = fracs[1, 0]

    # predict the x component of the force
    pred, var = gp_model.predict(env_curr, 1)

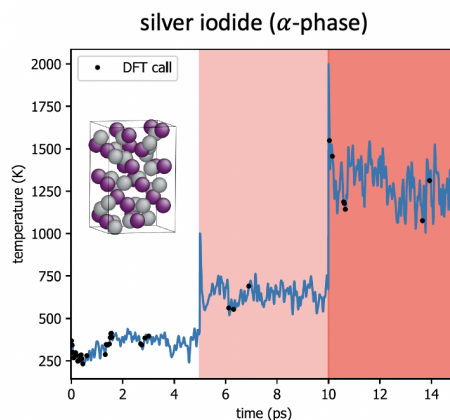
    gp_fracs[n, m] = pred
    gp_stds[n, m] = np.sqrt(var)

# plot GP predictions vs ground truth
cols = ['b', 'r', 'g']
for n in range(3):
    plt.plot(seps, store_fracs[n], color=cols[n], linestyle='-')
    plt.plot(seps, gp_fracs[n], color=cols[n], linestyle='--')
    plt.fill_between(seps, gp_fracs[n] + 3 * gp_stds[n],
                    gp_fracs[n] - 3 * gp_stds[n],
                    color=cols[n], alpha=0.4)

plt.xlabel('separation ($\AA$)')
plt.ylabel('force (eV/$\AA$)')
plt.xlim(2, 3.4)
plt.ylim(-10, 15)
plt.show()
```

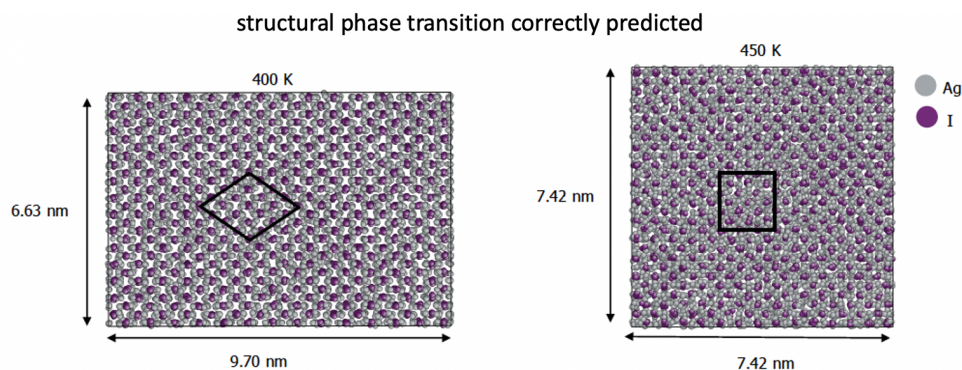


On-the-fly learning: fast-ion diffusion



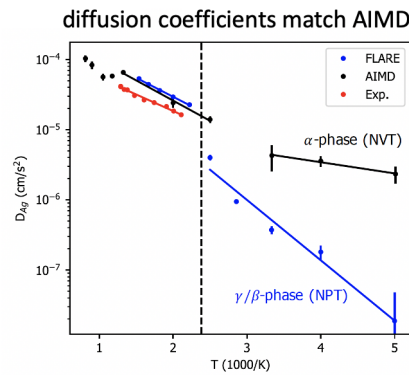
Vandermause, J., Torrisi, S. B., Batzner, S., Xie, Y., Sun, L., Kolpak, A. M., & Kozinsky, B. (2020). On-the-fly active learning of interpretable Bayesian force fields for atomistic rare events. *npj Computational Materials*, 6(1), 1-11. <https://www.nature.com/articles/s41524-020-0283-z>

On-the-fly learning: fast-ion diffusion



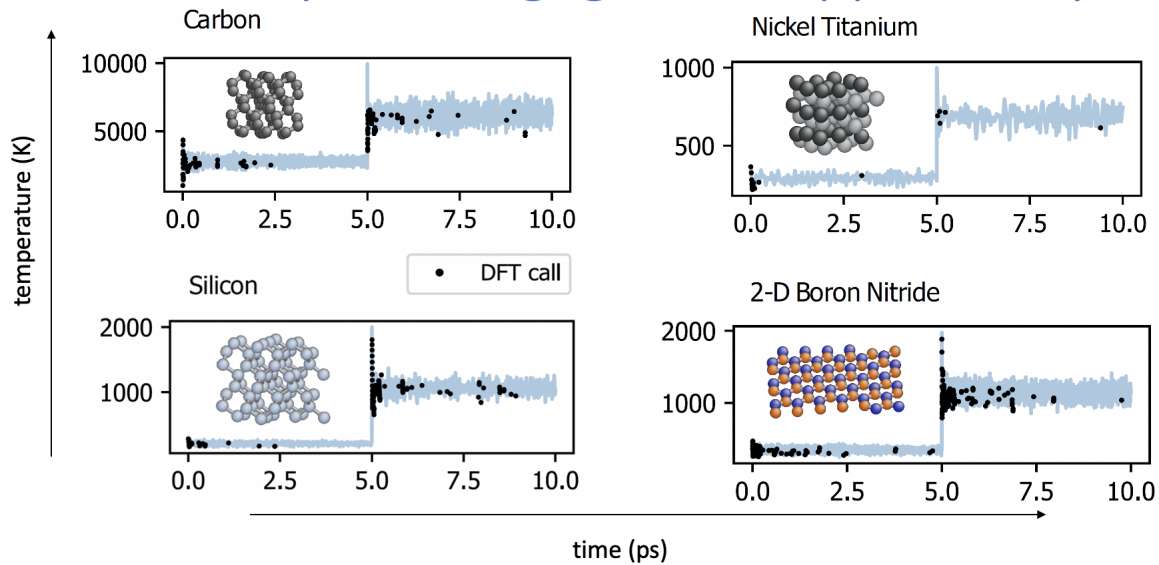
Vandermause, J., Torrisi, S. B., Batzner, S., Xie, Y., Sun, L., Kolpak, A. M., & Kozinsky, B. (2020). On-the-fly active learning of interpretable Bayesian force fields for atomistic rare events. *npj Computational Materials*, 6(1), 1-11. <https://www.nature.com/articles/s41524-020-0283-z>

On-the-fly learning: fast-ion diffusion



Vandermause, J., Torrisi, S. B., Batzner, S., Xie, Y., Sun, L., Kolpak, A. M., & Kozinsky, B. (2020). On-the-fly active learning of interpretable Bayesian force fields for atomistic rare events. *npj Computational Materials*, 6(1), 1-11. <https://www.nature.com/articles/s41524-020-0283-z>

On-the-fly learning: general applicability



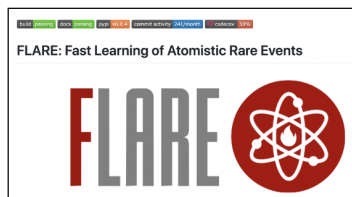
Thank you for your attention!

Paper: nature.com/articles/s41524-020-0283-z

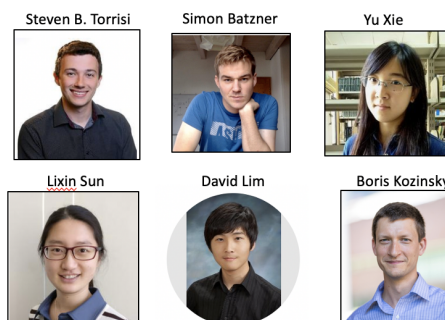
Github: github.com/mir-group/flare

Documentation: flare.readthedocs.io

Tutorial: tinyurl.com/wrq3kr7



The FLARE team



Vandermause, J., Torrisi, S. B., Batzner, S., Xie, Y., Sun, L., Kolpak, A. M., & Kozinsky, B. (2020). On-the-fly active learning of interpretable Bayesian force fields for atomistic rare events. *npj Computational Materials*, 6(1), 1-11.

[]:

1.2.2 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.

1.2.2.1 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.

1.2.2.2 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.

1.2.2.3 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,
                    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)
```

1.2.3 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.

1.2.3.1 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.

1.2.3.2 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.

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 Å) and a relatively large 3-body cutoff (7 Å), both of which will completely contain the molecule.

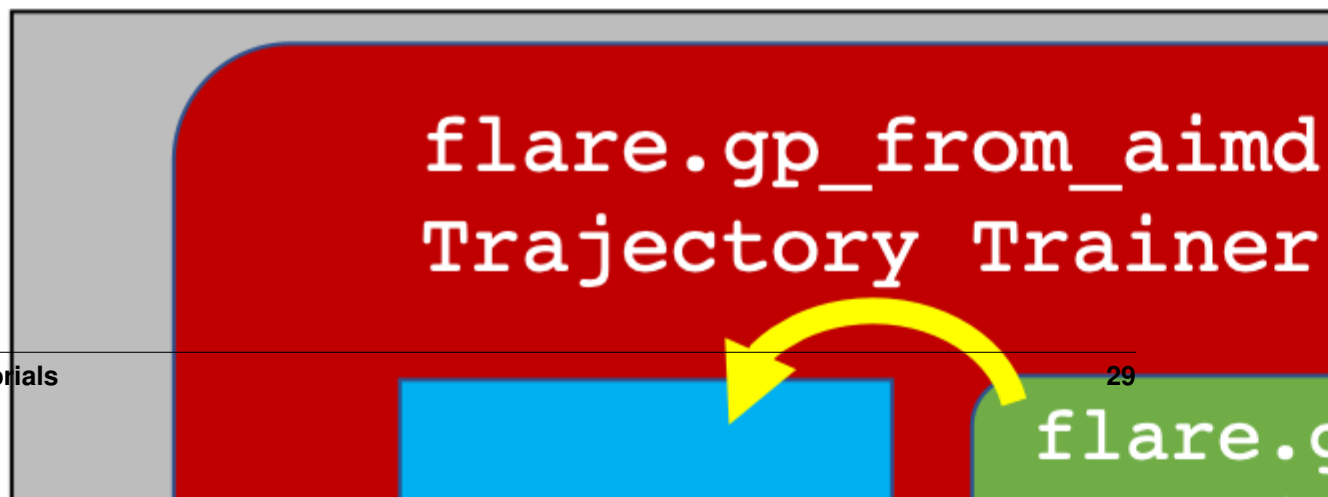
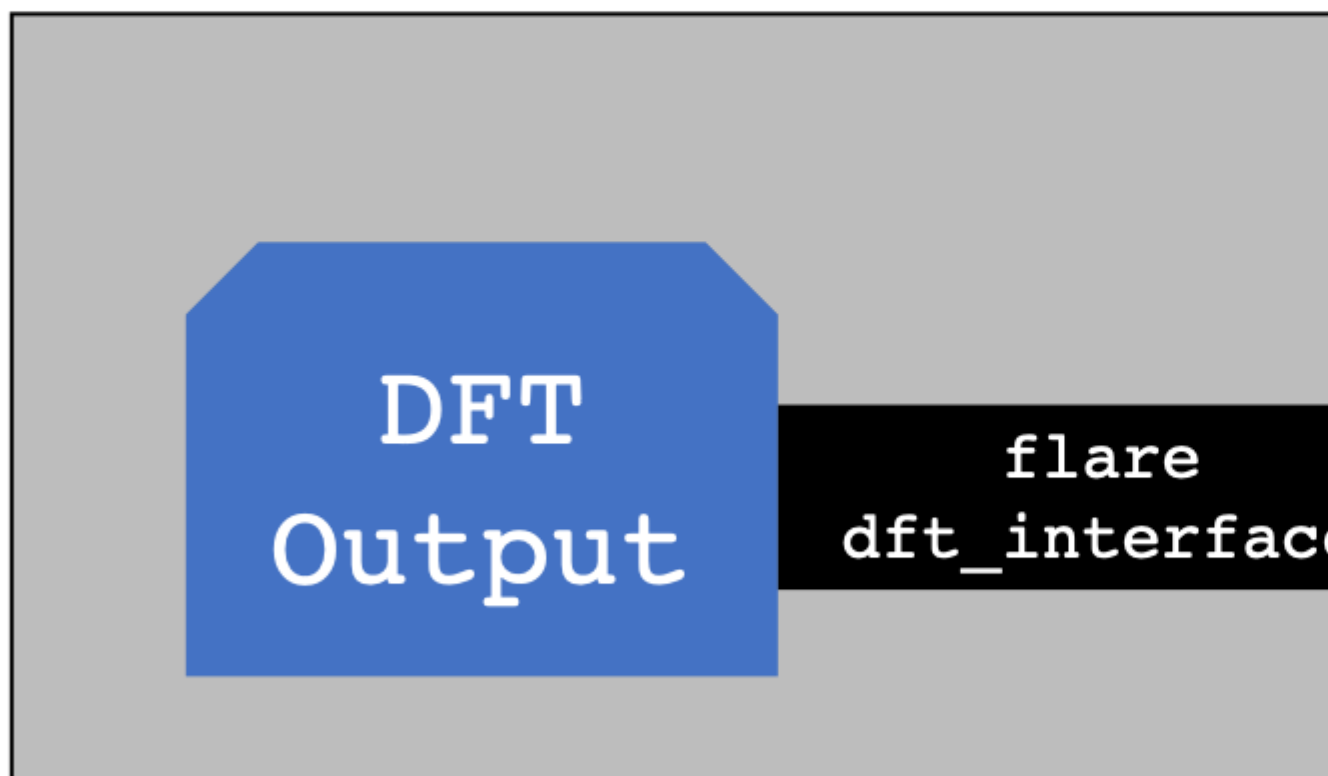
At the header of a file, include the following imports:

```
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 sufficient.
- 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.

GAUSSIAN PROCESS



```
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']
)
```

1.2.3.3 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')
```

1.2.3.4 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 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.

1.2.3.5 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).

1.2.4 On-the-fly training using ASE

Yu Xie (xiey@g.harvard.edu)

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()
      hys = hm['hyps']
      cut = hm['cutoffs']
      print('hyps', hys)

      gp_model = GaussianProcess(
          kernels = kernels,
```

(continues on next page)

(continued from previous page)

```

    component = 'mc', # If you are using ASE, please set to "mc" no matter for single-
    ↪ component or multi-component
    hyps = hyps,
    cutoffs = cut,
    hyp_labels = ['sig2', 'ls2', 'sig3', 'ls3', 'noise'],
    opt_algorithm = 'L-BFGS-B',
    n_cpus = 1
)
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]}}

mgp_model = MappedGaussianProcess(grid_params,
                                  unique_species = [6],
                                  n_cpus = 1,
                                  var_map=None)

```

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}'

## 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,
- and 1 MD engine implemented by FLARE: NoseHoover.

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 1: 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}`

Note 2: For some tricks and tips related to the on-the-fly training (e.g. how to set up temperatures, how to optimize hyperparameters), see [FAQs](#)

```
[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

Note: the ASE Trajectory is supported, but NOT recommended.
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,
              'write_model': 3} # If you will probably resume the training, please_
↪set to 3

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()
```

Then check the `*.out` file for the training log.

1.2.4.5 Step 5 (Optional): Resume Interrupted Training

At the end of each OTF training step, there will be several checkpoint files dumped

- `<output_name>_ckpt.json`: checkpoint of the current MD step of OTF. In the above example, `al_otf_qe_ckpt.json`.
- `<output_name>_flare.json`: If you've set `write_model=3`, then there will be another file saving the trained FLARE calculator, which will be loaded when restarting OTF.

- `<output_name>_atoms.json`: The ASE Atoms of the current MD step in the format of json
- `<output_name>_dft.pickle`: The DFT calculator saved in the format of .pickle.

Then, use `ASE_OTF.from_checkpoint(<output_name>_ckpt.json)` to load the OTF state, and resume the training by `run()`.

```
[ ]: new_otf = ASE_OTF.from_checkpoint("<output_name>_ckpt.json")
new_otf.run()
```

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.

```
[1]: 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.

```
[2]: 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='json')
```

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

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

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


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.

```
[4]: 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'

mgp_model = MappedGaussianProcess(grid_params, data['species'],
                                  var_map=None, lmp_file_name='AgI_Molten', n_cpus=1)
mgp_model.build_map(gp_model)

/Users/yuxie/opt/anaconda3/lib/python3.8/site-packages/flare/mgp/mapxb.py:519:
↳UserWarning: The minimal distance in training data is lower than the current lower
↳bound, will reset lower bound to 2.129780094032889
  warnings.warn(
/Users/yuxie/opt/anaconda3/lib/python3.8/site-packages/flare/mgp/mapxb.py:519:
↳UserWarning: The minimal distance in training data is lower than the current lower
↳bound, will reset lower bound to 2.129780094032889
  warnings.warn(
/Users/yuxie/opt/anaconda3/lib/python3.8/site-packages/flare/mgp/mapxb.py:519:
↳UserWarning: The minimal distance in training data is lower than the current lower
↳bound, will reset lower bound to 2.129780094032889
  warnings.warn(
```

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](#) in the *Installation* section.

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.mgp` for AgI system, with two-body (the 1st yes) together with three-body (the 2nd yes).

```
pair_coeff * * AgI_Molten.mgp Ag I yes yes
```

2. Another way is to use the ASE LAMMPS interface

```
[5]: import os
from flare.utils.element_coder import _Z_to_mass, _element_to_Z
from flare.ase.calculator import FLARE_Calculator
```

(continues on next page)

(continued from previous page)

```

from ase.calculators.lammpsrun import LAMMPS

from ase import Atoms

# create test structure
species = otf_object.gp_species_list[-1]
positions = otf_object.position_list[-1]
forces = otf_object.force_list[-1]
otf_cell = otf_object.header['cell']
structure = Atoms(symbols=species, cell=otf_cell, positions=positions)

# 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 + ".mgp"} {specie_symbol_list} _
→yes yes'],
              'mass': masses}
files = [lammps_location + ".mgp"]

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

structure.calc = lmp_calc

# To compute energy, forces and stress
# energy = structure.get_potential_energy()
# forces = structure.get_forces()
# stress = structure.get_stress()

/Users/yuxie/opt/anaconda3/lib/python3.8/site-packages/ase/calculators/lammpsrun.py:
→191: UserWarning: You are using an old syntax to set 'parameters'.
Please use LAMMPSRUN.set().
      warnings.warn(self.legacy_warn_string.format("parameters"))

```

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

```

[6]: 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]
otf_cell = otf_object.header['cell']
structure = struc.Structure(otf_cell, species, positions)

atom_types = [1, 2]

```

(continues on next page)

(continued from previous page)

```

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)

# To run lammps and get forces
# lammps_calculator.run_lammps(lammps_executable, input_file_name,
#                               output_file_name)

# lammps_forces = lammps_calculator.lammps_parser(dump_file_name)

```

1.3 Code Documentation

1.3.1 Bayesian Force Fields

1.3.1.1 Gaussian Process Force Fields

Predict

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

```

flare.bffs.gp.predict.predict_on_atom(param: Tuple[flare.atoms.FLARE_Atoms, int,
                                                flare.bffs.gp.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 (FLARE_Atoms, integer, GaussianProcess)) – tuple of FLARE FLARE_Atoms, atom index, and Gaussian Process object

Returns 3-element force array and associated uncertainties

Return type (np.ndarray, np.ndarray)

```

flare.bffs.gp.predict.predict_on_atom_efs(param)

```

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

```
flare.bffs.gp.predict.predict_on_atom_en(param: Tuple[flare.atoms.FLARE_Atoms,  
int, flare.bffs.gp.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 (FLARE_Atoms, integer, GaussianProcess)*) – tuple of FLARE FLARE_Atoms, atom index, and Gaussian Process object

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

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

```
flare.bffs.gp.predict.predict_on_atom_en_std(param)
```

Predict local energy and predictive std of a chemical environment.

```
flare.bffs.gp.predict.predict_on_structure(structure: flare.atoms.FLARE_Atoms, gp:  
flare.bffs.gp.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 forces, Nx3 numpy array of uncertainties

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

```
flare.bffs.gp.predict.predict_on_structure_en(structure: flare.atoms.FLARE_Atoms, gp:  
flare.bffs.gp.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.bffs.gp.predict.predict_on_structure_mgp(structure:
    flare.atoms.FLARE_Atoms,      mgp:
    flare.bffs.mgp.mgp.MappedGaussianProcess,
    output=None,      output_name=None,
    n_cpus: int = None, write_to_structure:
    bool = True, selective_atoms: List[int]
    = None,      skipped_atom_value:
    Union[float, int] = 0, en-
    ergy: bool = False) ->
    Union[Tuple[<sphinx.ext.autodoc.importer._MockObject
    object at 0x7f593fc1d2d0>,
    <sphinx.ext.autodoc.importer._MockObject
    object at 0x7f593fc1d290>, float], Tu-
    ple[<sphinx.ext.autodoc.importer._MockObject
    object at 0x7f593fc49a50>,
    <sphinx.ext.autodoc.importer._MockObject
    object at 0x7f593fa678d0>]]
```

Assign forces to structure based on an mgp

```
flare.bffs.gp.predict.predict_on_structure_par(structure: flare.atoms.FLARE_Atoms,
    gp: flare.bffs.gp.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.bffs.gp.predict.predict_on_structure_par_en(structure:
    flare.atoms.FLARE_Atoms,      gp:
    flare.bffs.gp.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)

Helper functions for GP

```
flare.bffs.gp.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 energy, force components, and partial stresses of a test environment and the total energy labels in the training set.

```
flare.bffs.gp.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 energy, force components, and partial stresses of a test environment and the force labels in the training set.

```
flare.bffs.gp.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.bffs.gp.gp_algebra.energy_energy_vector_unit(name, s, e, x, kernel, hyps, cutoffs=None, hyps_mask=None, d_l=None)
```

Gets part of the energy/energy vector.

```
flare.bffs.gp.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.bffs.gp.gp_algebra.energy_force_vector_unit(name, s, e, x, kernel, hyps, cutoffs=None, hyps_mask=None, d_l=None)
```

Gets part of the energy/force vector.

```
flare.bffs.gp.gp_algebra.force_energy_vector(name, kernel, x, d_l, 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.bffs.gp.gp_algebra.force_energy_vector_unit(name, s, e, x, kernel, hyps, cutoffs, hyps_mask, d_l)
```

Gets part of the force/energy vector.

```
flare.bffs.gp.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.bffs.gp.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.bffs.gp.gp_algebra.get_distance_mat_pack(hyps: <sphinx.ext.autodoc.importer._MockObject object at 0x7f593f9e60d0>, 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 hyperparameters :return: covariance matrix

```
flare.bffs.gp.gp_algebra.get_force_block(hyps: <sphinx.ext.autodoc.importer._MockObject object at 0x7f593f9e6a50>, 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 hyperparameters

Returns covariance matrix

```
flare.bffs.gp.gp_algebra.get_force_block_pack(hyps: <sphinx.ext.autodoc.importer._MockObject object at 0x7f593f9e6150>, 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 hyperparameters

Returns covariance matrix

```
flare.bffs.gp.gp_algebra.get_ky_and_hyp(hyps: <sphinx.ext.autodoc.importer._MockObject object at 0x7f593f9b7410>, 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 hyperparameters
- **n_cpus** – number of cpus to use.
- **n_sample** – the size of block for matrix to compute

Returns hyp_mat, ky_mat

flare.bffs.gp.gp_algebra.**get_ky_and_hyp_pack**(name, s1, e1, s2, e2, same: bool, hyps: *<sphinx.ext.autodoc.importer._MockObject object at 0x7f593f9b73d0>*, 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 possible 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 hyperparameters

Returns hyp_mat, ky_mat

flare.bffs.gp.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.bffs.gp.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.bffs.gp.gp_algebra.**get_neg_like**(hyps: *<sphinx.ext.autodoc.importer._MockObject object at 0x7f593f9b7450>*, name: str, force_kernel, logger_name=None, cutoffs=None, hyps_mask=None, n_cpus=1, n_sample=100)

compute the log likelihood and its gradients :param hyps: list of hyper-parameters :type hyps: np.ndarray :param name: name of the gp instance. :param kernel_grad: function object of the kernel gradient :param output: Output object for dumping every hyper-parameter

sets computed

Parameters

- **cutoffs** (*list of 2 float numbers*) – The cutoff values used for the atomic environments
- **hyps_mask** – dictionary used for multi-group hyperparameters
- **n_cpus** – number of cpus to use.
- **n_sample** – the size of block for matrix to compute

Returns float, np.array

`flare.bffs.gp.gp_algebra.get_neg_like_grad` (*hyps: <sphinx.ext.autodoc.importer.MockObject object at 0x7f593f9b7490>, name: str, kernel_grad, logger_name: str = 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
- **logger_name** (*str*) – name of logger 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 hyperparameters
- **n_cpus** – number of cpus to use.
- **n_sample** – the size of block for matrix to compute

Returns float, np.array

`flare.bffs.gp.gp_algebra.kernel_distance_mat` (*hyps: <sphinx.ext.autodoc.importer.MockObject object at 0x7f593f9e6110>, 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 hyperparameters :return: covariance matrix

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

`flare.bffs.gp.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 environment 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.bffs.gp.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.bffs.gp.gp_algebra.partition_matrix_custom` (*n_sample: int, start1, end1, start2, end2, n_cpus*)

Partition a specified portion of a matrix.

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

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.bffs.gp.calculator.FLARE_Calculator (gp_model,          mgp_model=None,
                                                  par=False,         use_mapping=False,
                                                  **kwargs)
```

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=None, properties=None, system_changes=<sphinx.ext.autodoc.importer._MockObject object>*)

Calculate properties including: energy, local energies, forces, stress, uncertainties.

Parameters **atoms** (*FLARE_Atoms*) – FLARE_Atoms object

1.3.1.2 Mapped Gaussian Process

Splines Methods

Cubic spline functions used for interpolation.

```
class flare.bffs.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.bffs.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 $svd_rank \leq \min(grid_num, train_size*3)$, for three-body $svd_rank \leq \min(grid_num_in_cube, train_size*3)$

flare.bffs.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,:].

Formulation of Mapped Gaussian Process

We take 2-body kernel as an example, for the simplicity of explanation. The local environment to be predicted is denoted as ρ_i , which consists of a center atom i whose force is to be predicted, and all its neighbors within a certain cutoff.

Energy

The atomic energy of ρ_i from Gaussian Process regression is

$$E(\rho_i) = \sum_{j=1}^N \sum_{p \in \rho_i} \sum_{q \in \rho_j} \tilde{k}(p, q) \alpha_j = \sum_{p \in \rho_i} \epsilon(p)$$

where in 2-body, p is a bond in ρ_i , and q is a bond in training data ρ_j , $\tilde{k}(p, q) = \sigma_2 \exp(-\frac{\|p-q\|^2}{l_2^2})$ is the kernel function. $\alpha = \Sigma^{-1} \mathbf{y}$ is the inverse kernel matrix multiplied by the label vector (forces of the training set).

In the equation above, we can see that the atomic energy can be decomposed into the summation of the bond energies $\epsilon(p)$, which is a low dimension function, for n-body kernel, its dimensionality of domain is $\frac{n(n-1)}{2}$. Therefore, we use a cubic spline to interpolate this function, with a GP model with fixed training data and hyperparameters. The forces can be derived from the derivative of the above function.

While in GP, the triple summations are needed in prediction, after mapping to the cubic spline, the two summations w.r.t. the training data set $\sum_{j=1}^N \sum_{q \in \rho_j}$ are removed. The computational cost in prediction of GP scales linearly w.r.t. the training set size N due to the two summations, and MGP is independent of N .

Uncertainty

The atomic uncertainty of ρ from Gaussian Process regression is

$$V(\rho) = \sum_{p \in \rho, q \in \rho} V(p, q)$$

$$V(p, q) = \tilde{k}(p, q) - \sum_{i,j} \sum_{u \in \rho_i, v \in \rho_j} \tilde{k}(p, u) (\Sigma^{-1})_{ij} \tilde{k}(v, q)$$

Since $V(p, q)$ has twice the dimensionality of domain compared with $\epsilon(p)$, it will be expensive both in computational cost and memory if we want to interpolate $V(p, q)$ directly with spline function, because a large number of grid points are needed for interpolation. To solve this problem, we observe that the 2nd term expressed in vector form is:

$$\tilde{\mathbf{k}}^\top(p) \Sigma^{-1} \tilde{\mathbf{k}}(q) = \left(\tilde{\mathbf{k}}^\top(p) L^{-\top} \right) \left(L^{-1} \tilde{\mathbf{k}}(q) \right) := \boldsymbol{\psi}(p)^\top \boldsymbol{\psi}(q)$$

L : Cholesky decomposition of matrix Σ , and define $\boldsymbol{\psi}(p)^\top := \left(L^{-1} \tilde{\mathbf{k}}(p) \right)$. The GP variance

$$V(p, q) = \tilde{k}(p, q) - \boldsymbol{\psi}(p)^\top \boldsymbol{\psi}(q)$$

Then we use splines to map the vector function $\boldsymbol{\psi}(p)$. Notice $\boldsymbol{\psi}(p)$ has $3N_{train} \sim O(10^3)$ components. Have to build $3N_{train}$ maps \rightarrow too many!

Dimension reduction

Can we find m -component vector function $\phi(p)$ (m is small), such that

$$\min_{\phi} \|\phi^\top(p) \phi(q) - \boldsymbol{\psi}(p)^\top \boldsymbol{\psi}(q)\|^2, \quad \forall p, q$$

Then we can estimate the variance as

$$V(p, q) = \tilde{k}(p, q) - \phi(p)^\top \phi(q)$$

and only build m maps instead of $3N_{train}$

- **Principle component analysis (PCA)** is what we want
- We can pick up any m by picking up the rank in PCA, the higher rank, the better estimation

References

- [1] Xie, Yu, et al. “Fast Bayesian Force Fields from Active Learning: Study of Inter-Dimensional Transformation of Stanene.” *arXiv preprint arXiv:2008.11796* (2020).
- [2] Glielmo, Aldo, Claudio Zeni, and Alessandro De Vita. “Efficient nonparametric n-body force fields from machine learning.” *Physical Review B* 97.18 (2018): 184307.
- [3] Glielmo, Aldo, et al. “Building nonparametric n-body force fields using gaussian process regression.” *Machine Learning Meets Quantum Physics*. Springer, Cham, 2020. 67-98.
- [4] Vandermause, Jonathan, et al. “On-the-fly active learning of interpretable Bayesian force fields for atomistic rare events.” *npj Computational Materials* 6.1 (2020): 1-11.

1.3.1.3 Sparse Gaussian Process Force Fields

1.3.2 Bayesian Active Learning

1.3.2.1 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.learners.otf.OTF(atoms, dt, number_of_steps, dft_calc, md_engine, md_kwargs,
                             flare_calc=None, trajectory=None, prev_pos_init: ndarray =
                             None, rescale_steps: List[int] = [], rescale_temps: List[int]
                             = [], write_model: int = 0, force_only: bool = True,
                             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, min_steps_with_model: int = 0,
                             update_style: str = 'add_n', update_threshold: float = None,
                             dft_kwargs=None, store_dft_output: Tuple[Union[str, List[str]],
                             str] = None, build_mode='bayesian', **kwargs)
```

Trains a Gaussian process force field on the fly during molecular dynamics.

Parameters

- **atoms** (*ASE Atoms*) – the ASE Atoms object for the on-the-fly MD run.
- **flare_calc** – ASE calculator. Must have “get_uncertainties” method implemented.
- **dt** – the timestep in MD, in the units of pico-second.
- **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*, *NPT-Berendsen*, *NPT* and *Langevin*, *NoseHoover* 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 `flare.otf.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 [].
- **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. If 4, write after each training and end of run, and back up after each write.
- **force_only** (*bool, optional*) – If True, only use forces for training. Default to False, use forces, energy and stress for training.
- **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.
- **min_steps_with_model** (*int, optional*) – Minimum number of steps the model takes in between calls to DFT. Defaults to 0.
- **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:’.
- **build_mode** (*str*) – default “bayesian”, run on-the-fly training. “direct” mode constructs GP model from a given list of frames, with *FakeMD* and *FakeDFT*. Each frame needs to have a global property called “target_atoms” specifying a list of atomic environments added to the GP model.

compute_properties ()

Compute energies, forces, stresses, and their uncertainties with the FLARE ASE calculator, 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

rescale_temperature (*new_pos: ndarray*)

Change the previous positions to update the temperature

Parameters *new_pos* (*np.ndarray*) – Positions of atoms in the next MD frame.

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_fracs: ndarray*, *dft_energy: float = None*, *dft_stress: ndarray = None*)

Updates the current GP model.

Parameters

- **train_atoms** (*List[int]*) – List of atoms whose local environments will be added to the training set.
- **dft_fracs** (*np.ndarray*) – DFT forces on all atoms in the structure.

update_temperature ()

Updates the instantaneous temperatures of the system.

1.3.2.2 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.

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 behaved 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.learners.gp_from_aimd.parse_trajectory_trainer_output (file: str, return_gp_data: bool = False, compute_errors: bool = True) → 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
- **compute_errors** – Compute deviation from GP and DFT forces.

Returns List of dictionaries with keys ‘species’, ‘positions’, ‘gp_forces’, ‘dft_forces’, ‘gp_stds’, ‘added_atoms’, and ‘maes_by_species’, optionally, `gp_data` dictionary

```
flare.learners.gp_from_aimd.structures_from_gpfa_output (frame_dictionaries: List[dict]) → List[flare.atoms.FLARE_Atoms]
```

Takes as input the first output from the `parse_trajectory_trainer_output` function and turns it into a series of FLARE structures, with DFT forces mapped onto the structures.

Parameters **frame_dictionaries** – The list of dictionaries which describe each GPFA frame.

Returns

1.3.2.3 Conditions to add training data

Utility functions for various tasks.

```
flare.learners.utils.get_max_cutoff (cell: <sphinx.ext.autodoc.importer._MockObject object at 0x7f593fa458d0>) → 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.learners.utils.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 ‘override’ 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.learners.utils.is_std_in_bound(std_tolerance: float, noise: float, structure:
                                     FLARE_Atoms, max_atoms_added: int = inf, up-
                                     date_style: str = 'add_n', update_threshold: float =
                                     None) -> (<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
- **update_style** – A string specifying the desired strategy for adding atoms to the training set. Current options are “add_n”, which adds the `n = max_atoms_added` highest-uncertainty atoms, and “threshold”, which adds all atoms with uncertainty greater than `update_threshold`.
- **update_threshold** – A float specifying the update threshold. Ignored if `update_style` is not set to “threshold”.

Returns (True,[-1]) if no atoms are above cutoff, (False,[...]) if at least one atom is above `std_tolerance`, with the list indicating which atoms have been selected for the training set.

```
flare.learners.utils.is_std_in_bound_per_species (rel_std_tolerance: float,
                                                  abs_std_tolerance: float, noise:
                                                  float, structure: FLARE_Atoms,
                                                  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 ‘override’ 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.learners.utils.subset_of_frame_by_element (frame: FLARE_Atoms, pre-
                                                  dict_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:

1.3.3 ASE MD Engine

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`,

1.3.3.1 Fake MD

```
class flare.md.fake.FakeDFT (filename=None, format=None, index=':', io_kwargs={}, **kwargs)
class flare.md.fake.FakeMD (atoms, timestep, trajectory=None, filename=None, format=None,
                             index=':', io_kwargs={}, logfile=None, loginterval=1, ap-
                             pend_trajectory=False)
```

1.3.3.2 LAMMPS Calculator and MD

```
class flare.md.lammps.LAMMPS_MD (atoms, timestep, trajectory=None, **kwargs)
```

Run MD with LAMMPS based on the `ase.md.md.MolecularDynamics`. It includes using `LAMMPS_MOD` to run multiple steps, and supports Bayesian active learning with flare.

Parameters `parameters` (*dict*) – LAMMPS input commands.

backup (*curr_trj*)

Back up the current trajectory into .xyz file. The atomic positions, velocities, forces and uncertainties are read from lammps trajectory. The step, potential energy and stress are read from thermo.txt

Parameters `curr_trj` (*list[ase.Atoms]*) – lammps trajectory of current run read by ASE.

step (*std_tolerance*, *N_steps*)

Run lammps until the uncertainty interrupts. Notice this method neither runs only a single MD step, nor finishes all the *N_steps*. The MD exits only when 1) the maximal atomic uncertainty goes beyond *std_tolerance*, or 2) all the *N_steps* are finished without uncertainty beyond *std_tolerance*.

Parameters

- **std_tolerance** (*float*) – the threshold for atomic uncertainty, above which the MD will be interrupted and DFT will be called.
- **N_steps** (*int*) – number of MD steps left to run.

class `flare.md.lammps.LAMMPS_MOD` (***kwargs*)

A modified ASE LAMMPS calculator based on `ase.lammpsrun.LAMMPS`, to allow for more flexible input parameters, including compute, fix/nvt, fix/npt etc.

Supported customized commands for LAMMPS input: - mass (set by arg *masses*) - package - atom_style, bond_style, angle_style, dihedral_style, improper_style, kspace_style - units (default: metal) - boundary - neighbor - newton - kim_interactions - pair_style (default: lj/cut 2.5) - pair_coeff (default: * * 1 1) - *compute - *region - *dump - group - fix - timestep - minimize - run

Note: - additional commands needed at the beginning can be specified in the arg *model_init* - additional commands needed after “pair_coeff” can be specified in the arg *model_post*

Non-customized input commands: - atom_modify sort 0 0.0 - read_data - fix fix_nve all nve - dump dump_all all custom trj_file (dump_period) id type x y z vx vy vz fx fy fz - thermo_style custom (thermo_args) - thermo_modify flush yes format float %23.16g - thermo 1

Customized parameters: - dump_period - thermo_args - specorder

calculate (*atoms=None*, *properties=None*, *system_changes=None*, *set_atoms=False*)

Modify parameters

`flare.md.lammps.check_sgp_match` (*atoms*, *sgp_calc*, *logger*, *specorder*, *command*)

Check if the lammps trajectory or calculator matches the SGP predictions

`flare.md.lammps.get_kinetic_stress` (*atoms*)

LAMMPS stress tensor = virial + kinetic, kinetic = $\sum(m_k * v_{ki} * v_{kj}) / V$. We subtract the kinetic term and keep only the virial term. In the calculator, results[“stress”] += kinetic_atoms gives virial.

1.3.3.3 NoseHoover (NVT Ensemble)

Custom Nose-Hoover NVT thermostat based on ASE.

This code was originally written by Jonathan Mailoa based on these notes:

<https://www2.ph.ed.ac.uk/~dmarendu/MVP/MVP03.pdf>

It was then adapted by Simon Batzner to be used within ASE. Parts of the overall outline of the class are also based on the Langevin class in ASE.

```
class flare.md.nosehoover.NoseHoover(atoms, timestep, temperature, nvt_q, trajectory=None, logfile=None, loginterval=1, append_trajectory=False)
```

Nose-Hoover (constant N, V, T) molecular dynamics.

Usage: NoseHoover(atoms, dt, temperature)

atoms The list of atoms.

timestep The time step.

temperature Target temperature of the MD run in [K * units.kB]

nvt_q Q in the Nose-Hoover equations

Example Usage:

```
    nvt_dyn = NoseHoover( atoms=atoms, timestep=0.5 * units.fs, temperature=300. * units.kB,
                          nvt_q=334.
    )
step()
    Perform a MD step.
```

1.3.4 Descriptors

1.3.4.1 Atomic Environments

1.3.5 Kernels

1.3.5.1 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 many body:
 - `two_plus_many_body`,
 - `two_plus_many_body_grad`,
 - `two_plus_many_body_en`,
 - `two_plus_many_body_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.ndarray*) – 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.ndarray*) – 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.ndarray*) – 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, cut-
                                         off_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_many_body` (*env1*: *flare.descriptors.env.AtomicEnvironment*, *env2*:
flare.descriptors.env.AtomicEnvironment, *d1*: *int*, *d2*: *int*,
hyps, *cutoffs*, *cutoff_func*=<function quadratic_cutoff>)
2+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 (sig2b, ls2b, sigmb, lsmb, 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_many_body_en` (*env1*: *flare.descriptors.env.AtomicEnvironment*,
env2: *flare.descriptors.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_many_body_force_en(env1: flare.descriptors.env.AtomicEnvironment,  
                                              env2: flare.descriptors.env.AtomicEnvironment,  
                                              d1: int, hyps, cutoffs, cut-  
                                              off_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_many_body_grad(env1: flare.descriptors.env.AtomicEnvironment,  
                                         env2: flare.descriptors.env.AtomicEnvironment,  
                                         d1: int, d2: int, hyps, cutoffs, cut-  
                                         off_func=<function quadratic_cutoff>)
```

2+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 (sig2b, ls2b, sigmb, lsmb, 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_body(env1: flare.descriptors.env.AtomicEnvironment,
                                     env2: flare.descriptors.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, cut-
                                     off_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, cut-  
off_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.descriptors.env.AtomicEnvironment,  
env2: flare.descriptors.env.AtomicEnvironment,  
d1: int, d2: int, hyps, cutoffs, cut-  
off_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.descriptors.env.AtomicEnvironment,  
env2:  
flare.descriptors.env.AtomicEnvironment,  
hyps, cutoffs, cut-  
off_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.descriptors.env.AtomicEnvironment,
                                                         env2:
                                                         flare.descriptors.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.descriptors.env.AtomicEnvironment,
                                                       env2:
                                                       flare.descriptors.env.AtomicEnvironment,
                                                       d1: int, d2: int, hyps, cut-
                                                       offs,      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

1.3.5.2 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.descriptors.env.AtomicEnvironment, env2:
                                     flare.descriptors.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.descriptors.env.AtomicEnvironment,
                                     env2: flare.descriptors.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.descriptors.env.AtomicEnvironment,  
                                           env2: flare.descriptors.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.
- **cutoff_func** (*Callable*) – Cutoff function.

Returns Value of the many-body kernel.

Return type float

```
flare.kernels.mc_simple.three_body_mc(env1: flare.descriptors.env.AtomicEnvironment, env2:
                                       flare.descriptors.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.descriptors.env.AtomicEnvironment,  
                                          env2: flare.descriptors.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.descriptors.env.AtomicEnvironment,
                                                env2: flare.descriptors.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.descriptors.env.AtomicEnvironment,  
                                           env2: flare.descriptors.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.descriptors.env.AtomicEnvironment, env2:
                                     flare.descriptors.env.AtomicEnvironment, d1: int, d2: int,
                                     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.descriptors.env.AtomicEnvironment,
                                         env2: flare.descriptors.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.descriptors.env.AtomicEnvironment,  
                                              env2: flare.descriptors.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.descriptors.env.AtomicEnvironment,
                                          env2: flare.descriptors.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_many_body_mc(env1: flare.descriptors.env.AtomicEnvironment,  
                                              env2: flare.descriptors.env.AtomicEnvironment,  
                                              d1: int, d2: int, hyps, cutoffs, cut-  
                                              off_func=<function quadratic_cutoff>)
```

2+many body kernel.

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 (sig2b, ls2b, sigmb, lsmb, sig_n).
- **cutoffs** (*np.ndarray*) – Two-element array containing the 2- and many-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_many_body_mc_en(env1:  
                                                  flare.descriptors.env.AtomicEnvironment,  
                                                  env2:  
                                                  flare.descriptors.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 (sig2b ls2b, sigmb, lsmb, 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_many_body_mc_force_en(env1:
                                                         flare.descriptors.env.AtomicEnvironment,
                                                         env2:
                                                         flare.descriptors.env.AtomicEnvironment,
                                                         d1: int, hyps, cutoffs,
                                                         cutoff_func=<function
                                                         quadratic_cutoff>)
```

2+many-body multi-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_many_body_mc_grad(env1:
                                                         flare.descriptors.env.AtomicEnvironment,
                                                         env2:
                                                         flare.descriptors.env.AtomicEnvironment,
                                                         d1: int, d2: int, hyps, cut-
                                                         offs, cutoff_func=<function
                                                         quadratic_cutoff>)
```

2+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

```
flare.kernels.mc_simple.two_plus_three_body_mc(env1: flare.descriptors.env.AtomicEnvironment,  
                                                env2: flare.descriptors.env.AtomicEnvironment,  
                                                d1: int, d2: int, hyps: ndarray, cut-  
                                                offs: 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.descriptors.env.AtomicEnvironment,  
                                                     env2:  
                                                     flare.descriptors.env.AtomicEnvironment,  
                                                     d1: int, d2: int, hyps: ndar-  
                                                     ray, cutoffs: ndarray, cut-  
                                                     off_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.descriptors.env.AtomicEnvironment,
                                             env2: flare.descriptors.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.descriptors.env.AtomicEnvironment,
                                                    env2:
                                                    flare.descriptors.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.descriptors.env.AtomicEnvironment,
                                                           env2:
                                                           flare.descriptors.env.AtomicEnvironment,
                                                           d1: int, d2: int,
                                                           hyps, cutoffs, cut-
                                                           off_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.descriptors.env.AtomicEnvironment,  
    env2:  
    flare.descriptors.env.AtomicEnvironment,  
    hyps, cutoffs, cut-  
    off_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.descriptors.env.AtomicEnvironn  
    env2:  
    flare.descriptors.env.AtomicEnvironn  
    d1: int,  
    hyps,  
    cutoffs, cut-  
    off_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.descriptors.env.AtomicEnvironment,
                                                                env2:
                                                                flare.descriptors.env.AtomicEnvironment,
                                                                d1: int, d2: int,
                                                                hys, cutoffs, cut-
                                                                off_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

1.3.5.3 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.bffs.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},
```

(continues on next page)

(continued from previous page)

```

... constraints={'twobody0':[False, True]})
>>> hyps_mask = pml.as_dict()
>>> hyps = hyps_mask.pop('hyps')
>>> cutoffs = hyps_mask.pop('cutoffs')
>>> hyp_labels = hyps_mask.pop('hyp_labels')
>>> 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 describing 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 pairings [0-0, 0-1, 1-0, 1-1]: if we wanted hyperparameter set 0 for 0-0 pairings 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 pairings 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 pairings of atoms in groups defined in spec_mask.
- **cut3b_mask**: Array of length nspec^3 , 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 hyperparameters 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 parameters 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_sephyys.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, cut-  
                                                  off_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, cut-  
                                              off_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*) – nspec³ long integer array
- **ncut3b** (*int*) – number of different 3-body cutoff sets to associate with 3-body pairings
- **cut3b_mask** (*np.ndarray*) – nspec² 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, cut-
                                                off_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, cut-
off_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, cut-
                                         off_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, cut-
                                                off_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, cut-
                                           off_mb, nspec, spec_mask, nbond, bond_mask,
                                           ntriplet, triplet_mask, ncut3b, cut3b_mask, nmb,
                                           mb_mask, sig2, ls2, sig3, ls3, sigm, lsm, cut-
                                           off_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, cut-
                                                  off_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_sephys.two_plus_three_mc_en(env1, env2, cutoff_2b, cutoff_3b, cut-  
off_mb, nspec, spec_mask, nbond,  
bond_mask, ntriplet, triplet_mask,  
ncut3b, cut3b_mask, nmb, mb_mask,  
sig2, ls2, sig3, ls3, sigm, lsm, cut-  
off_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*) – nspec² long integer array
- **ntriplet** (*int*) – number of different hyperparameter sets to associate with 3-body pairings
- **triplet_mask** (*np.ndarray*) – nspec³ long integer array
- **ncut3b** (*int*) – number of different 3-body cutoff sets to associate with 3-body pairings
- **cut3b_mask** (*np.ndarray*) – nspec² 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*) – nspec² long integer array
- **ntriplet** (*int*) – number of different hyperparameter sets to associate with 3-body pairings
- **triplet_mask** (*np.ndarray*) – nspec³ long integer array
- **ncut3b** (*int*) – number of different 3-body cutoff sets to associate with 3-body pairings
- **cut3b_mask** (*np.ndarray*) – nspec² 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, cut-
                                                    off_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

1.3.5.4 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 $r_{\text{cut}} - d$, and assigns a cosine envelope to values of r between $r_{\text{cut}} - d$ 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)

1.3.5.5 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 lengthscale 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 lengthscale 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 descriptor 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 descriptor function

Returns the value of the many-body descriptor

Return type float

For details on our 2- and 3-body kernels, please refer to [Vandermause et al.](#) and [Glielmo et al.](#)

1.3.6 File Input and Output

1.3.6.1 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.io.output.Output` (*basename: str = 'otf_run', verbose: str = 'INFO', print_as_xyz: bool = False, 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 (*extra_strings: List[str] = None*)
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*, *dft_energy*, *error*,
local_energies=None, *KE*=None, *mgp*=False, *cell*=None,
stress=None)

Write the comparison to logfile.

Parameters

- **curr_step** – current timestep
- **frame** – FLARE_Atoms object that contains the current GP calculation results.
- **start_time** – start time for time profiling
- **dft_forces** – list of forces computed by DFT
- **dft_energy** – total energy computed by DFT
- **error** – list of force differences between DFT and GP prediction
- **local_energies** – local atomic energy
- **KE** – total kinetic energy
- **cell** – print the unit cell of the structure
- **stress** – print the stress acting on the cell

Returns

write_header(*gp_str*: str, *dt*: float = None, *Nsteps*: int = None, *structure*=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** – timestep 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_config (*curr_step, structure, forces: <sphinx.ext.autodoc.importer._MockObject object at 0x7f593f9b2d90> = None, stds: <sphinx.ext.autodoc.importer._MockObject object at 0x7f593f9b2dd0> = None, dft_forces: <sphinx.ext.autodoc.importer._MockObject object at 0x7f593f9b2f50> = None, dft_energy=0, predict_energy=nan, target_atoms=None*)
 write atomic configuration in xyz file

Parameters

- **curr_step** – Int, number of frames to note in the comment line
- **structure** – FLARE_Atoms, contain positions and forces
- **forces** – Optional list of forces to xyz file
- **stds** – Optional list of uncertainties to xyz file
- **dft_forces** – Optional second list of forces (e.g. DFT forces)

Returns

flare.io.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.io.output.add_stream(logger: logging.Logger, verbose: str = 'info')`
set up screen stream handler to the logger with handlers

Parameters

- **logger** – the logger
- **verbose** (*str*) – verbose level

`flare.io.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

1.3.6.2 OTF Parser

class `flare.io.otf_parser.OtfAnalysis` (*filename, calculate_energy=False*)
Parse the OTF log file to get trajectory, training data, thermostat, and build GP model.

Parameters

- **filename** (*str*) – name of the OTF log file.
- **calculate_energy** (*bool*) – if the potential energy is computed and needs to be parsed, then set to True. Default False.

make_gp (*cell=None, call_no=None, hyps=None, init_gp=None, hyp_no=None, **kwargs*)
Build GP model from the training frames parsed from the log file. The cell, hyps and gp can be reset with customized values.

Parameters

- **cell** (*np.ndarray*) – Default None to use the cell from the log file. A customized cell can be input as a 3x3 numpy array.
- **call_no** (*int*) – Default None to use all the DFT frames as training data for building GP. If not None, then the frames 0 to *call_no* will be added to GP.
- **hyps** (*np.ndarray*) – Default None to use the hyperparameters from the log file. Customized hyps can be input as an array.
- **init_gp** (*GaussianProcess*) – Default to None to use no initial settings or training data. an initial GP can be used, and then the frames parsed in the log file will add to the initial GP. Then the final GP uses the hyps and kernels of *init_gp*, and consists of training data from *init_gp* and the data from the log file. **NOTE:** if a log file from restarted OTF is parsed, then an initial GP needs to be parsed from the prior log file as the *init_gp* of the restarted log file.
- **hyp_no** (*int*) – Default None to use the final optimized hyperparameters to build GP. If not None, then use the hyps from the ‘hyp_no’th optimization step.
- **kwargs** – if a new GP setting is needed without inputing *init_gp*, the GP initial args can be input as kwargs.

output_md_structures ()

Returns structure objects corresponding to the MD frames of an OTF run. :return:

parse_pos_otf (*blocks*)

Exclusively parses MD run information :param filename: :return:

to_xyz (*xyz_file*)

Convert OTF trajectory from log file to .xyz file. :Parameters: **xyz_file** (*str*) – the file name of the .xyz file to output

Returns A list of ASE Atoms objects.

```
flare.io.otf_parser.append_atom_lists (species_list: List[str], position_list:
                                     List[<sphinx.ext.autodoc.importer._MockObject
                                     object at 0x7f593f7b5d10>], force_list:
                                     List[<sphinx.ext.autodoc.importer._MockObject
                                     object at 0x7f593f61d4d0>], uncertainty_list:
                                     List[<sphinx.ext.autodoc.importer._MockObject
                                     object at 0x7f593f712110>], velocity_list:
                                     List[<sphinx.ext.autodoc.importer._MockObject
                                     object at 0x7f593f712810>], lines: List[str], index:
                                     int, noa: int, dft_call: bool, noh: int) → None
```

Update lists containing atom information at each snapshot.

flare.io.otf_parser.extract_gp_info (*block, mae_list, maf_list, atoms_list, hyps_list, noh*)

Exclusively parses DFT run information :param filename: :return:

flare.io.otf_parser.parse_frame_line (*frame_line*)

parse a line in otf output. :param frame_line: frame line to be parsed :type frame_line: string :return: species, position, force, uncertainty, and velocity of atom :rtype: list, np.arrays

flare.io.otf_parser.parse_header_information (*lines*) → dict

Get information about the run from the header of the file :param outfile: :return:

flare.io.otf_parser.parse_snapshot (*lines, index, noa, dft_call, noh*)

Parses snapshot of otf output file.

flare.io.otf_parser.strip_and_split (*line*)

Helper function which saves a few lines of code elsewhere :param line: :return:

1.3.7 Utility

1.3.7.1 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],
...                                   'threebody0':[1, 0.5], 'threebody1':[2, 0.2],
...                                   'cutoff_threebody':1},
...                       constraints={'twobody0':[False, True]})
>>> hm = pm.as_dict()
>>> kernels = hm['kernels']
>>> gp_model = GaussianProcess(kernels=kernels,
...                             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 hyperparameters 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'])

```

(continues on next page)

(continued from previous page)

```

>>> pm.define_group('threebody', '***', ['*', '*', '*'])
>>> pm.define_group('threebody', 'Oall', ['O', 'O', 'O'])
>>> 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`

If you want to add in a new hyperparameter set to an already-existing GP, you can perform the following steps:

```

>> hyps_mask = pm.as_dict() >> hyps = hyps_mask['hyps'] >> kernels = hyps_mask['kernels'] >>
gp_model.update_kernel(kernels, 'mc', hyps_mask) >> gp_model.hyps = hyps

```

```

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 hyperparameters 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 Gaussian-Process

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 parameter 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 parameters

- **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_threobody`, and `cutoff_manybody` are reserved for noise parameter 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 parameters
- **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_threobody`, and `cutoff_manybody` are reserved for noise parameter 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 variables

Parameters *group_type* (*str*) – species, twobody, threobody, cut3b, manybody

1.3.7.2 Construct Atomic Environment

`flare.utils.env_getarray.get_2_body_arrays` (*positions, atom: int, cell, r_cut, cutoff_2, species, sweep, nspecie, species_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
- **species_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` (*bond_array_2*, *bond_positions_2*, *ctype*,
etypes, *r_cut*, *cutoff_3*, *nspecie*, *species_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
- **species_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(positions, atom: int, cell, r_cut,
manybody_cutoff_list, species, sweep:
<sphinx.ext.autodoc.importer._MockObject
object at 0x7f593fa4f410>, nspec,
spec_mask, manybody_mask, cut-
off_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., $2 * \text{cutoff} < \text{cell_size}$

```
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 descriptor function

Returns the value of the many-body descriptor

Return type float

Utility functions for various tasks.

```
class flare.utils.NumpyEncoder(*, skipkeys=False, ensure_ascii=True, check_circular=True,
allow_nan=True, sort_keys=False, indent=None, separa-
tors=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>

1.4 Frequently Asked Questions

1.4.1 Frequently Asked Questions

1.4.1.1 Installation and Packages

1. **What numba version will I need?** $\geq 0.50.0$ The latest version is always recommended.

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.

1.4.1.2 Gaussian Processes

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 to refer to the radial distribution function of your atomic system first, then try a range of cutoff

values and examine the model error, optimized noise parameter, and model likelihood as a function of the cutoff.

- In the current implementation, the 3-body cutoff needs to be smaller than 2-body 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.

There are a few tips in the OTF training, see below.

1.4.1.3 OTF (On-the-fly) Training

1. **What is a good strategy for hyperparameter optimization?**

- `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.

- `std_tolerance_factor` : the DFT will be called when the predicted uncertainty is above the threshold, which is defined as `std_tolerance_factor * gp_hyps_noise`. The default value is 1. In general, you can set it to O(0.1)-O(1). If more DFT calls are desired, you can set it to a lower value.

2. How to set initial temperatures and rescale temperatures?

- For initial temperature, if you are using NVE ensemble, and starting from a perfect crystal lattice, please set the initial temperature to be twice the value you want the system to be in equilibrium. E.g., if you want the system to equilibrate at 1000K, set the initial temperature to 2000K starting from a perfect lattice. To do this,
 - if you are using our OTF trainer without ASE, you can set the `prev_positions` of the initial structure to be the perfect lattice shifted by one step with the velocity corresponding to the initial temperature.
 - if you are using OTF with ASE + VelocityVerlet, you can set the initial velocity as shown in our tutorial
- Similarly, if you want to rescale the system from 300K to 500K at step 1000, you should set the rescaling temperature to be higher than 500K, e.g.

```
rescale_temp = [800]
rescale_step = [1000]
```

The reason is that we only rescale the velocity of the atoms at the steps specified in `rescale_step`, at those steps not in the list, we will let the system evolve by itself. Thus, after step 1000, the system's temperature will gradually equilibrate at a lower temperature.

3. **Include small perturbation for initial structure** If you are starting from a perfect lattice, we recommend adding small random perturbations to the atomic positions, such that the symmetry of the crystal lattice is broken. E.g.

```
positions = positions + 0.01 * (2 * np.random.rand(len(positions), 3) - 1)
```

The reason is that the perfect lattice is highly symmetric, thus usually the force on each atom is zero, and the local environments all look the same. Adding these highly similar environments with close-to-zero forces might raise numerical stability issue for GP.

1.4.1.4 GPFA

1. **My models are adding too many atoms from each frame, causing a serious slowdown without much gain in model accuracy**

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).

1.4.1.5 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 $[a, a, a]$ and upper cutoffs $[b, b, b]$.
- 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 $a=2.5$, $b=5$.

1.5 Applications

If you use FLARE in your research, please let us know. We will list the applications of FLARE here.

- 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." npj Computational Materials 6.1 (2020): 1-11. ([arXiv](#)) ([published version](#))
- Jin Soo Lim, Jonathan Vandermause, Matthijs A. Van Spronsen, Albert Musaelian, Yu Xie, Lixin Sun, Christopher R. O'Connor, Tobias Egle, Nicola Molinari, Jacob Florian, Kaining Duanmu, Robert J. Madix, Philippe Sautet, Cynthia M. Friend, and Boris Kozinsky. "Evolution of Metastable Structures at Bimetallic Surfaces from Microscopy and Machine-Learning Molecular Dynamics." Journal of the American Chemical Society (2020). ([ChemRxiv](#)) ([published version](#))
- Yu Xie, Jonathan Vandermause, Lixin Sun, Andrea Cepellotti, and Boris Kozinsky. "Bayesian force fields from active learning for simulation of inter-dimensional transformation of stanene." npj Computational Materials 7, no. 1 (2021): 1-10. ([arXiv](#)) ([published version](#))
- Jonathan Vandermause, Yu Xie, Jin Soo Lim, Cameron J. Owen, and Boris Kozinsky. "Active learning of reactive Bayesian force fields: Application to heterogeneous hydrogen-platinum catalysis dynamics." arXiv preprint arXiv:2106.01949 (2021). ([arXiv](#))
- Claudio Zeni, Kevin Rossi, Theodore Pavludis, Joseph Kioseoglou, Stefano de Gironcoli, Richard Palmer, and Francesca Baletto. "Data-driven simulation and characterization of gold nanoparticles melting." Nat Commun 12, 6056 (2021). ([arXiv](#)) ([published version](#))
- Kai Xu, Lei Yan, and Bingran You. "Bayesian active learning of interatomic force field for molecular dynamics simulation of Pt/Ag(111)." ChemRxiv preprint. ([ChemRxiv](#))

1.6 How To Contribute

1.6.1 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](#).

1.6.1.1 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.

1.6.1.2 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>
```

1.6.1.3 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.

1.6.1.4 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.

1.6.2 Code Standards

Before pushing code to the development branch, please make sure your changes respect the following code standards.

1.6.2.1 PEP 8

Run your code through pylint to check that you're in compliance with [PEP 8](#):

```
$ pylint <file name>
```

1.6.2.2 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.

1.6.2.3 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.

1.7 How to Cite

If you use FLARE in your research, or any part of this repo (such as the GP implementation), please cite the following paper:

[1] Vandermause, J., Torrisi, S. B., Batzner, S., Xie, Y., Sun, L., Kolpak, A. M. & Kozinsky, B. *On-the-fly active learning of interpretable Bayesian force fields for atomistic rare events*. npj Comput Mater 6, 20 (2020). <https://doi.org/10.1038/s41524-020-0283-z>

If you use MGP or LAMMPS pair style, please cite the following paper:

[2] Xie, Y., Vandermause, J., Sun, L. et al. *Bayesian force fields from active learning for simulation of inter-dimensional transformation of stanene*. npj Comput Mater 7, 40 (2021). <https://doi.org/10.1038/s41524-021-00510-y>

Thank you for using FLARE!

f

- `flare.bffs.gp`, 46
- `flare.bffs.gp.calculator`, 46
- `flare.bffs.gp.gp_algebra`, 42
- `flare.bffs.gp.predict`, 39
- `flare.bffs.mgp`, 49
- `flare.bffs.mgp.splines_methods`, 46
- `flare.io.otf_parser`, 120
- `flare.io.output`, 117
- `flare.kernels.cutoffs`, 114
- `flare.kernels.kernels`, 115
- `flare.kernels.mc_3b_sepcut`, 111
- `flare.kernels.mc_mb_sepcut`, 112
- `flare.kernels.mc_sephyps`, 93
- `flare.kernels.mc_simple`, 72
- `flare.kernels.sc`, 56
- `flare.learners.gp_from_aimd`, 51
- `flare.learners.otf`, 49
- `flare.learners.utils`, 52
- `flare.md.fake`, 54
- `flare.md.lammps`, 54
- `flare.md.nosehoover`, 55
- `flare.utils`, 129
- `flare.utils.env_getarray`, 127
- `flare.utils.parameter_helper`, 121
- `flare.utils.parameters`, 127

A

`add_file()` (in module `flare.io.output`), 119
`add_stream()` (in module `flare.io.output`), 119
`all_separate_groups()`
 (`flare.utils.parameter_helper.ParameterHelper`
 method), 124
`append_atom_lists()` (in module
 `flare.io.otf_parser`), 121
`as_dict()` (`flare.utils.parameter_helper.ParameterHelper`
 method), 124

B

`backup()` (`flare.md.lammps.LAMMPS_MD` method),
 55

C

`calculate()` (`flare.bffs.gp.calculator.FLARE_Calculator`
 method), 46
`calculate()` (`flare.md.lammps.LAMMPS_MOD`
 method), 55
`check_sgp_match()` (in module `flare.md.lammps`),
 55
`compute_properties()` (`flare.learners.otf.OTF`
 method), 50
`conclude_run()` (`flare.io.output.Output` method),
 117
`coordination_number()` (in module
 `flare.kernels.kernels`), 115
`cosine_cutoff()` (in module `flare.kernels.cutoffs`),
 114
`cubic_cutoff()` (in module `flare.kernels.cutoffs`),
 114
`CubicSpline` (class in
 `flare.bffs.mgp.splines_methods`), 46

D

`define_group()` (`flare.utils.parameter_helper.ParameterHelper`
 method), 124

E

`efs_energy_vector()` (in module
 `flare.bffs.gp.gp_algebra`), 42
`efs_force_vector()` (in module
 `flare.bffs.gp.gp_algebra`), 42
`energy_energy_vector()` (in module
 `flare.bffs.gp.gp_algebra`), 42
`energy_energy_vector_unit()` (in module
 `flare.bffs.gp.gp_algebra`), 42
`energy_force_vector()` (in module
 `flare.bffs.gp.gp_algebra`), 42
`energy_force_vector_unit()` (in module
 `flare.bffs.gp.gp_algebra`), 42
`extract_gp_info()` (in module `flare.io.otf_parser`),
 121

F

`FakeDFT` (class in `flare.md.fake`), 54
`FakeMD` (class in `flare.md.fake`), 54
`fill_in_parameters()`
 (`flare.utils.parameter_helper.ParameterHelper`
 method), 125
`find_group()` (`flare.utils.parameter_helper.ParameterHelper`
 method), 125
`flare.bffs.gp` (module), 46
`flare.bffs.gp.calculator` (module), 46
`flare.bffs.gp.gp_algebra` (module), 42
`flare.bffs.gp.predict` (module), 39
`flare.bffs.mgp` (module), 49
`flare.bffs.mgp.splines_methods` (module),
 46
`flare.io.otf_parser` (module), 120
`flare.io.output` (module), 117
`flare.kernels.cutoffs` (module), 114
`flare.kernels.kernels` (module), 115
`flare.kernels.mc_3b_sepcut` (module), 111
`flare.kernels.mc_mb_sepcut` (module), 112
`flare.kernels.mc_sephyps` (module), 93
`flare.kernels.mc_simple` (module), 72

[flare.kernels.sc \(module\)](#), 56
[flare.learners.gp_from_aimd \(module\)](#), 51
[flare.learners.otf \(module\)](#), 49
[flare.learners.utils \(module\)](#), 52
[flare.md.fake \(module\)](#), 54
[flare.md.lammps \(module\)](#), 54
[flare.md.nosehoover \(module\)](#), 55
[flare.utils \(module\)](#), 129
[flare.utils.env_getarray \(module\)](#), 127
[flare.utils.parameter_helper \(module\)](#), 121
[flare.utils.parameters \(module\)](#), 127
[FLARE_Calculator \(class in flare.bffs.gp.calculator\)](#), 46
[force_energy_vector \(\) \(in module flare.bffs.gp.gp_algebra\)](#), 42
[force_energy_vector_unit \(\) \(in module flare.bffs.gp.gp_algebra\)](#), 42
[force_force_vector \(\) \(in module flare.bffs.gp.gp_algebra\)](#), 42
[force_force_vector_unit \(\) \(in module flare.bffs.gp.gp_algebra\)](#), 43
[force_helper \(\) \(in module flare.kernels.kernels\)](#), 116
[from_dict \(\) \(flare.utils.parameter_helper.ParameterHelper static method\)](#), 125

G

[get_2_body_arrays \(\) \(in module flare.utils.env_getarray\)](#), 127
[get_3_body_arrays \(\) \(in module flare.utils.env_getarray\)](#), 128
[get_distance_mat_pack \(\) \(in module flare.bffs.gp.gp_algebra\)](#), 43
[get_force_block \(\) \(in module flare.bffs.gp.gp_algebra\)](#), 43
[get_force_block_pack \(\) \(in module flare.bffs.gp.gp_algebra\)](#), 43
[get_kinetic_stress \(\) \(in module flare.md.lammps\)](#), 55
[get_ky_and_hyp \(\) \(in module flare.bffs.gp.gp_algebra\)](#), 43
[get_ky_and_hyp_pack \(\) \(in module flare.bffs.gp.gp_algebra\)](#), 44
[get_like_from_mats \(\) \(in module flare.bffs.gp.gp_algebra\)](#), 44
[get_like_grad_from_mats \(\) \(in module flare.bffs.gp.gp_algebra\)](#), 44
[get_m2_body_arrays \(\) \(in module flare.utils.env_getarray\)](#), 129
[get_m3_body_arrays \(\) \(in module flare.utils.env_getarray\)](#), 129
[get_max_cutoff \(\) \(in module flare.learners.utils\)](#), 52

[get_neg_like \(\) \(in module flare.bffs.gp.gp_algebra\)](#), 44
[get_neg_like_grad \(\) \(in module flare.bffs.gp.gp_algebra\)](#), 44
[grid \(flare.bffs.mgp.splines_methods.CubicSpline attribute\)](#), 46

H

[hard_cutoff \(\) \(in module flare.kernels.cutoffs\)](#), 115

I

[interpolate \(\) \(flare.bffs.mgp.splines_methods.CubicSpline method\)](#), 46
[is_force_in_bound_per_species \(\) \(in module flare.learners.utils\)](#), 52
[is_std_in_bound \(\) \(in module flare.learners.utils\)](#), 53
[is_std_in_bound_per_species \(\) \(in module flare.learners.utils\)](#), 53

K

[k_sq_exp_dev \(\) \(in module flare.kernels.kernels\)](#), 116
[k_sq_exp_double_dev \(\) \(in module flare.kernels.kernels\)](#), 116
[kernel_distance_mat \(\) \(in module flare.bffs.gp.gp_algebra\)](#), 45

L

[LAMMPS_MD \(class in flare.md.lammps\)](#), 54
[LAMMPS_MOD \(class in flare.md.lammps\)](#), 55
[list_groups \(\) \(flare.utils.parameter_helper.ParameterHelper method\)](#), 125
[list_parameters \(\) \(flare.utils.parameter_helper.ParameterHelper method\)](#), 126

M

[make_gp \(\) \(flare.io.otf_parser.OtfAnalysis method\)](#), 120
[many_body \(\) \(in module flare.kernels.sc\)](#), 57
[many_body_en \(\) \(in module flare.kernels.sc\)](#), 57
[many_body_en_jit \(\) \(in module flare.kernels.sc\)](#), 57
[many_body_force_en \(\) \(in module flare.kernels.sc\)](#), 58
[many_body_force_en_jit \(\) \(in module flare.kernels.sc\)](#), 58
[many_body_grad \(\) \(in module flare.kernels.sc\)](#), 58
[many_body_grad_jit \(\) \(in module flare.kernels.sc\)](#), 59
[many_body_jit \(\) \(in module flare.kernels.sc\)](#), 59
[many_body_mc \(\) \(in module flare.kernels.mc_sephyps\)](#), 95

`many_body_mc()` (in module `flare.kernels.mc_simple`), 72
`many_body_mc_en()` (in module `flare.kernels.mc_sephyps`), 96
`many_body_mc_en()` (in module `flare.kernels.mc_simple`), 72
`many_body_mc_en_jit()` (in module `flare.kernels.mc_simple`), 72
`many_body_mc_en_sepcut_jit()` (in module `flare.kernels.mc_mb_sepcut`), 112
`many_body_mc_force_en()` (in module `flare.kernels.mc_sephyps`), 96
`many_body_mc_force_en()` (in module `flare.kernels.mc_simple`), 73
`many_body_mc_force_en_jit()` (in module `flare.kernels.mc_simple`), 73
`many_body_mc_force_en_sepcut_jit()` (in module `flare.kernels.mc_mb_sepcut`), 112
`many_body_mc_grad()` (in module `flare.kernels.mc_sephyps`), 96
`many_body_mc_grad()` (in module `flare.kernels.mc_simple`), 73
`many_body_mc_grad_jit()` (in module `flare.kernels.mc_simple`), 74
`many_body_mc_grad_sepcut_jit()` (in module `flare.kernels.mc_mb_sepcut`), 113
`many_body_mc_jit()` (in module `flare.kernels.mc_simple`), 74
`many_body_mc_sepcut_jit()` (in module `flare.kernels.mc_mb_sepcut`), 113
`mb_grad_helper_ls()` (in module `flare.kernels.kernels`), 116
`mb_grad_helper_ls_()` (in module `flare.kernels.kernels`), 116
`md_step()` (`flare.learners.otf.OTF` method), 50

N

`NoseHoover` (class in `flare.md.nosehoover`), 55
`NumpyEncoder` (class in `flare.utils`), 129

O

`obtain_noise_len()` (in module `flare.bffs.gp.gp_algebra`), 45
`open_new_log()` (`flare.io.output.Output` method), 117
`OTF` (class in `flare.learners.otf`), 49
`OtfAnalysis` (class in `flare.io.otf_parser`), 120
`Output` (class in `flare.io.output`), 117
`output_md_structures()` (`flare.io.otf_parser.OtfAnalysis` method), 120

P

`ParameterHelper` (class

`flare.utils.parameter_helper`), 123
`parse_frame_line()` (in module `flare.io.otf_parser`), 121
`parse_header_information()` (in module `flare.io.otf_parser`), 121
`parse_pos_otf()` (`flare.io.otf_parser.OtfAnalysis` method), 121
`parse_snapshot()` (in module `flare.io.otf_parser`), 121
`parse_trajectory_trainer_output()` (in module `flare.learners.gp_from_aimd`), 51
`partition_force_energy_block()` (in module `flare.bffs.gp.gp_algebra`), 45
`partition_matrix()` (in module `flare.bffs.gp.gp_algebra`), 45
`partition_matrix_custom()` (in module `flare.bffs.gp.gp_algebra`), 45
`partition_vector()` (in module `flare.bffs.gp.gp_algebra`), 45
`PCASplines` (class in `flare.bffs.mgp.splines_methods`), 47
`predict_on_atom()` (in module `flare.bffs.gp.predict`), 39
`predict_on_atom_efs()` (in module `flare.bffs.gp.predict`), 39
`predict_on_atom_en()` (in module `flare.bffs.gp.predict`), 39
`predict_on_atom_en_std()` (in module `flare.bffs.gp.predict`), 40
`predict_on_structure()` (in module `flare.bffs.gp.predict`), 40
`predict_on_structure_en()` (in module `flare.bffs.gp.predict`), 40
`predict_on_structure_mgp()` (in module `flare.bffs.gp.predict`), 41
`predict_on_structure_par()` (in module `flare.bffs.gp.predict`), 41
`predict_on_structure_par_en()` (in module `flare.bffs.gp.predict`), 41

Q

`q3_value_mc()` (in module `flare.utils.env_getarray`), 129
`q_value()` (in module `flare.kernels.kernels`), 116
`q_value_mc()` (in module `flare.kernels.kernels`), 117
`quadratic_cutoff()` (in module `flare.kernels.cutoffs`), 115
`quadratic_cutoff_bound()` (in module `flare.kernels.cutoffs`), 115
`queue_wrapper()` (in module `flare.bffs.gp.gp_algebra`), 46

R

`rescale_temperature()` (`flare.learners.otf.OTF`

method), 50
 run() (*flare.learners.otf.OTF method*), 51
 run_dft() (*flare.learners.otf.OTF method*), 51

S

set_constraints() (*flare.utils.parameter_helper.ParameterHelper method*), 126
 set_logger() (*in module flare.io.output*), 120
 set_parameters() (*flare.utils.parameter_helper.ParameterHelper method*), 127
 set_values() (*flare.bffs.mgp.splines_methods.CubicSpline method*), 47
 step() (*flare.md.lammps.LAMMPS_MD method*), 55
 step() (*flare.md.nosehoover.NoseHoover method*), 56
 strip_and_split() (*in module flare.io.otf_parser*), 121
 structures_from_gpfa_output() (*in module flare.learners.gp_from_aimd*), 52
 subset_of_frame_by_element() (*in module flare.learners.utils*), 54
 summarize_group() (*flare.utils.parameter_helper.ParameterHelper method*), 127

T

three_body() (*in module flare.kernels.sc*), 60
 three_body_en() (*in module flare.kernels.sc*), 60
 three_body_en_jit() (*in module flare.kernels.sc*), 61
 three_body_force_en() (*in module flare.kernels.sc*), 61
 three_body_force_en_jit() (*in module flare.kernels.sc*), 62
 three_body_grad() (*in module flare.kernels.sc*), 62
 three_body_grad_jit() (*in module flare.kernels.sc*), 63
 three_body_jit() (*in module flare.kernels.sc*), 64
 three_body_mc() (*in module flare.kernels.mc_sephyps*), 97
 three_body_mc() (*in module flare.kernels.mc_simple*), 75
 three_body_mc_en() (*in module flare.kernels.mc_sephyps*), 98
 three_body_mc_en() (*in module flare.kernels.mc_simple*), 76
 three_body_mc_en_jit() (*in module flare.kernels.mc_simple*), 76
 three_body_mc_force_en() (*in module flare.kernels.mc_sephyps*), 99
 three_body_mc_force_en() (*in module flare.kernels.mc_simple*), 77
 three_body_mc_force_en_jit() (*in module flare.kernels.mc_sephyps*), 99

three_body_mc_force_en_jit() (*in module flare.kernels.mc_simple*), 77
 three_body_mc_force_en_sepcut_jit() (*in module flare.kernels.mc_3b_sepcut*), 111
 three_body_mc_grad() (*in module flare.kernels.mc_sephyps*), 100
 three_body_mc_grad() (*in module flare.kernels.mc_simple*), 78
 three_body_mc_grad_jit() (*in module flare.kernels.mc_sephyps*), 100
 three_body_mc_grad_jit() (*in module flare.kernels.mc_simple*), 79
 three_body_mc_grad_sepcut_jit() (*in module flare.kernels.mc_3b_sepcut*), 111
 three_body_mc_jit() (*in module flare.kernels.mc_simple*), 79
 three_body_se_jit() (*in module flare.kernels.mc_simple*), 80
 three_body_sf_jit() (*in module flare.kernels.mc_simple*), 81
 three_body_ss_jit() (*in module flare.kernels.mc_simple*), 82
 to_xyz() (*flare.io.otf_parser.OtfAnalysis method*), 121
 train_gp() (*flare.learners.otf.OTF method*), 51
 two_body() (*in module flare.kernels.sc*), 64
 two_body_en() (*in module flare.kernels.sc*), 65
 two_body_en_jit() (*in module flare.kernels.sc*), 65
 two_body_force_en() (*in module flare.kernels.sc*), 65
 two_body_force_en_jit() (*in module flare.kernels.sc*), 66
 two_body_grad() (*in module flare.kernels.sc*), 66
 two_body_grad_jit() (*in module flare.kernels.sc*), 66
 two_body_jit() (*in module flare.kernels.sc*), 67
 two_body_mc() (*in module flare.kernels.mc_sephyps*), 101
 two_body_mc() (*in module flare.kernels.mc_simple*), 83
 two_body_mc_en() (*in module flare.kernels.mc_sephyps*), 101
 two_body_mc_en() (*in module flare.kernels.mc_simple*), 83
 two_body_mc_en_jit() (*in module flare.kernels.mc_sephyps*), 102
 two_body_mc_en_jit() (*in module flare.kernels.mc_simple*), 84
 two_body_mc_force_en() (*in module flare.kernels.mc_sephyps*), 102
 two_body_mc_force_en() (*in module flare.kernels.mc_simple*), 84
 two_body_mc_force_en_jit() (*in module flare.kernels.mc_sephyps*), 103
 two_body_mc_force_en_jit() (*in module*

`flare.kernels.mc_simple`), 84
`two_body_mc_grad()` (in `flare.kernels.mc_sephyps`), 103
`two_body_mc_grad()` (in `flare.kernels.mc_simple`), 85
`two_body_mc_grad_jit()` (in `flare.kernels.mc_sephyps`), 104
`two_body_mc_grad_jit()` (in `flare.kernels.mc_simple`), 85
`two_body_mc_jit()` (in `flare.kernels.mc_sephyps`), 104
`two_body_mc_jit()` (in `flare.kernels.mc_simple`), 86
`two_body_mc_stress_en_jit()` (in `module flare.kernels.mc_simple`), 86
`two_body_mc_stress_force_jit()` (in `module flare.kernels.mc_simple`), 87
`two_body_mc_stress_stress_jit()` (in `module flare.kernels.mc_simple`), 87
`two_plus_many_body()` (in `module flare.kernels.sc`), 67
`two_plus_many_body_en()` (in `module flare.kernels.sc`), 67
`two_plus_many_body_force_en()` (in `module flare.kernels.sc`), 68
`two_plus_many_body_grad()` (in `module flare.kernels.sc`), 68
`two_plus_many_body_mc()` (in `module flare.kernels.mc_simple`), 88
`two_plus_many_body_mc_en()` (in `module flare.kernels.mc_simple`), 88
`two_plus_many_body_mc_force_en()` (in `module flare.kernels.mc_simple`), 89
`two_plus_many_body_mc_grad()` (in `module flare.kernels.mc_simple`), 89
`two_plus_three_body()` (in `module flare.kernels.sc`), 68
`two_plus_three_body_grad()` (in `module flare.kernels.sc`), 69
`two_plus_three_body_mc()` (in `module flare.kernels.mc_sephyps`), 104
`two_plus_three_body_mc()` (in `module flare.kernels.mc_simple`), 90
`two_plus_three_body_mc_grad()` (in `module flare.kernels.mc_sephyps`), 105
`two_plus_three_body_mc_grad()` (in `module flare.kernels.mc_simple`), 90
`two_plus_three_en()` (in `module flare.kernels.sc`), 69
`two_plus_three_force_en()` (in `module flare.kernels.sc`), 70
`two_plus_three_mc_en()` (in `module flare.kernels.mc_sephyps`), 106
`two_plus_three_mc_en()` (in `module flare.kernels.mc_simple`), 90
`two_plus_three_mc_force_en()` (in `module flare.kernels.mc_sephyps`), 106
`two_plus_three_plus_many_body()` (in `module flare.kernels.sc`), 70
`two_plus_three_plus_many_body_en()` (in `module flare.kernels.sc`), 70
`two_plus_three_plus_many_body_force_en()` (in `module flare.kernels.sc`), 71
`two_plus_three_plus_many_body_grad()` (in `module flare.kernels.sc`), 71
`two_plus_three_plus_many_body_mc()` (in `module flare.kernels.mc_simple`), 91
`two_plus_three_plus_many_body_mc_en()` (in `module flare.kernels.mc_simple`), 92
`two_plus_three_plus_many_body_mc_force_en()` (in `module flare.kernels.mc_simple`), 92
`two_plus_three_plus_many_body_mc_grad()` (in `module flare.kernels.mc_simple`), 93
`two_three_many_body_mc()` (in `module flare.kernels.mc_sephyps`), 107
`two_three_many_body_mc_grad()` (in `module flare.kernels.mc_sephyps`), 108
`two_three_many_mc_en()` (in `module flare.kernels.mc_sephyps`), 109
`two_three_many_mc_force_en()` (in `module flare.kernels.mc_sephyps`), 110

U

`update_gp()` (`flare.learners.otf.OTF method`), 51
`update_temperature()` (`flare.learners.otf.OTF method`), 51

V

`vec_eval_cubic_spline()` (in `module flare.bffs.mgp.splines_methods`), 47

W

`write_gp_dft_comparison()` (`flare.io.output.Output method`), 118
`write_header()` (`flare.io.output.Output method`), 118
`write_hyps()` (`flare.io.output.Output method`), 118
`write_md_config()` (`flare.io.output.Output method`), 119
`write_to_log()` (`flare.io.output.Output method`), 119
`write_xyz_config()` (`flare.io.output.Output method`), 119