CLEASE Documentation

Release 1.0.6

J. H. Chang, D. Kleiven, A. Tygesen

Jan 30, 2023

CONTENTS:

Cluster expansion (CE) is a widely used method for studying thermondynamic properties of disordered materials. CLEASE is a cluster expansion code which strives to be highly flexible and customizable, which also offering a wide range of useful tools, such as:

- Tools to construct a CE model
  - Semi-automatic *structure generation* for constructing training data, such as random, ground-state and probe structures.
  - Database for storing calculation results.
  - Multiple basis functions for the CE model to choose from: `Polynomial`, `Trigonometric` or `BinaryLinear`.
  - Many methods for parameterization *fitting* and evaluating the CE model, such as `Lasso` `Tikhonov`, `PhysicalRidge` and `GAFit`.
  - Tools for *easily visualizing* the accuracy of your CE model, and interact with the plots e.g. when made in a Jupyter notebook.
- Various flavors of Monte Carlo samplers where one can explore a large configurational space in a large simulation cell
  - Canonical and semi-grand canonical *Monte Carlo schemes*.
  - Flexible customization options for restricting the model during MC runs. CLEASE provides a *number of constraints*, but it is also easy to *implement custom constraints*.
  - Use one our *pre-made observers* to collect thermodynamic data about your system during an MC run, or *write your own*.

and much more. A tutorial of how to use CLEASE can be found in our *AuCu example*.

CHAPTER

ONE

GUI

Most of the standard CE routines can be performed using the graphical user interface (GUI). The CLEASE GUI is an app based on the jupyter notebook. Please remember to report any issues to the developers.

CHAPTER

TWO

INSTALLATION

A latest stable version of CLEASE can be installed using the following command

```
pip install clease
```

Installation can also be done through conda via the conda-forge project:

```
conda install -c conda-forge clease
```

Note: On Windows, we recommend installing CLEASE with conda, in order to simplify the compilation process.

Alternatively, you can install the latest development version of CLEASE by following the instructions in the README page.

CHAPTER

THREE

USING CLEASE

The method and implementation details of CLEASE are described in the following publication:

> J. Chang, D. Kleiven, M. Melander, J. Akola, J. M. Garcia-Lastra and T. Vegge
> CLEASE: A versatile and user-friendly implementation of Cluster Expansion method
> Journal of Physics: Condensed Matter

3.1 Release notes

3.1.1 1.0.6

- Now requires the c++14 compiler flag.
- Fixed updating current energy for metadynamics. See !590.

3.1.2 1.0.5

- Added *supercell_which_contains_sphere()* and *cell_wall_distances()*.
- *SGCObserver* now only tracks the singlet averages if `observe_singlets=True`. The default observer which is created by the *SGCMonteCarlo* can be controlled by the *SGCMonteCarlo* `observe_singlets` keyword.
- `reset_eci` in *get_thermodynamic_quantities()* now defaults to `False`. This keyword is likely to be removed in the future, see #321.

3.1.3 1.0.4

- Added `CleaseCacheCalculator`, as a primitive cache calculator object with no cache validation.
- Performance improvements to updating correlation functions.
- Performance improvements to calculating the translation matrix, so the first calculation of the clusters should be faster.
- Performance improvements to the *LowestEnergyStructure*. Correlation functions are also no longer tracked by default, but can be enabled with the `track_cf` key.
- The default SGC observer in *SGCMonteCarlo* should now be reset automatically upon changing the temperature.

3.1.4 1.0.3

- Getting thermodynamic quantities in the SGC MC now also retrieves averages from observers.
- Added *interactive* option to *plot_eci()*
- Added *get_cluster_corresponding_to_cf_name()*.
- Minor performance improvements to SGC MC.
- Added *set_normalization()* for adjusting what elements to normalize by. Default is to normalize by everything.

3.1.5 1.0.2

- *insert_structure()* returns both the initial and final ID if both an initial and final structure was inserted.
- Fixes a bug with writing the Clease calculator to a DB row.

3.1.6 1.0.1

- Added the `ignore_sizes` keyword to *plot_eci()*
- Changing the maximum cluster diameter will now clear any cached clusters, and requires a new build.
- Calling observers in canonical MC can now be disabled with the `call_observers` keyword for performing burn-in, without executing observers.

3.1.7 1.0.0

- 21 June 2022 - CLEASE is no longer considered beta.
- *Evaluate* can now properly support fitting with custom LinearRegression schemes, even if they don't support alpha cross-validation.
- *Evaluate* now required explicit calls to *fit()*. Calls to *get_eci()* and *get_eci_dict()* can no longer implicitly do fitting. This un-does a change introduced in version 0.11.6.
- Added the *current_accept_rate* property, and export the current accept rate in the thermodynamic quantities dictionary under the `accept_rate` key.
- Removed a series of deprecated things:
  - Removed the `clease.concentration` module.
  - Removed the `clease.new_struct` module.
  - Removed old regression imports. Regression classes must now be imported from the `clease.regression` module.
  - Removed the `clease.structure_generator` module.
  - Removed the `max_cluster_size` settings argument.

3.1.8 0.11.6

- Some small performance optimizations.
- Added a *warn_on_skip* parameter to the *insert_structure()* method.
- *Evaluate* should now correctly remember if it doesn't need to re-fit the ECI's (see the new *fit()* and *fit_required()* methods).
- Introduced *load_eci()* for loading stored ECI values, which is convenient for subsequent plotting.
- Added `get_attempt_freq()` which allows for more flexible customization of the attempt frequencies.
- Added the `interactive` keyword to *plot_fit()*.
- Added an experimental parallelization feature. See *Parallelization*.
- Added `clease info` to the CLI to display some information about the installation.

3.1.9 0.11.5

- Fixed a bug with interactive plotting and convex hulls.
- Added the *max_sphere_dia_in_cell()* for calculating sphere diameters within the given cell boundaries.
- Changing the temperature of the *Montecarlo* object will now reset the internal energy averagers. Also, *BaseMC* now requires a temperature, and the temperature property has been renamed `temperature`. The old `T` attribute name is still accessible for backwards compatibility.

  For more information, see #302.
- Added *iter_reconfigure_db_entries()*.

3.1.10 0.11.4

- Fixed an issue where `attach_calculator` would incorrectly try to snap the atoms onto a grid.
- Typo in the axis labeling in `plot_fit`.

3.1.11 0.11.3

- *MCStep* and `SystemChange` instances are now savable to json via the `jsonio` module.
- Fixed a bug which prevented the primitive to have more than 255 atoms.
- The primitive cell is now always wrapped in the settings object.
- Changing `db_name` will check if the primitive exists in the new DB, and write it if it's missing.
- `size` and `supercell_factor` are now stored and managed by the `TemplateAtoms` object.
- Made some adjustments to the compilation process.
- Removed the `include_background_atoms` setter in the settings object. This value must now be set explicitly in the constructor for consistency reasons. For more information, see #292.
- Fixes a bug with the *ConcentrationObserver*.

3.1.12 0.11.2

- Introduces a new `TransMatrix` dataclass for the translation matrix.
- Temporarily restricts ASE to `<3.23`, until we resolve issues with current ASE master.
- Montecarlo will no longer consider background indices in the default swap move generator, if background is ignored.
- Added a new *irun()* method, for iteratively running MC calculations.
- MC observers can now override *observe_step()* instead, which takes a *MCStep* object.
- Added a new MC observer: *clease.montecarlo.observers.MoveObserver*.

3.1.13 0.11.1

- Fixed a bug in the `FixedIndices` constraint class.
- Greatly improved speed of supercell generation - this mostly affects performance concerning large supercells.
- Improved performance of the trans matrix generation.

3.1.14 0.11.0

- Python 3.7+ is now required.
- Removed old deprecated functions and classes.
- Some performance improvements.

3.1.15 0.10.9

- Now caches the CF names if requesting every CF name. Chops off some of the computation time during a full reconfigure.
- Introduces a new *FinalStructPropertyGetter*, which can be used to get arbitrary properties stored as key-value pairs in the database. Use the `prop` keyword in the *Evaluate* class to use this feature.
- Added the `check_db` keyword to *NewStructures*
- Some minor optimizations

3.1.16 0.10.8

- Fixes an issue with the coefficients generated by the Lasso method.
- Fixes an issue with the interactive convex hull plot.
- No longer opens an extra unnecessary GUI window with interactive plots.
- Fixed a bug with the fingerprint grouping, where the relative tolerance would reduce the numerical sensitivity too much.
- Now uses the `packaging` package for managing version numbers and comparisons. Removes usage of the deprecated distutils version comparisons.

3.1.17 0.10.7

- Fixed *view_clusters()*, which broke in 0.10.6.
- Adds *ensure_clusters_exist()* and *get_all_figures_as_atoms()*. *ensure_clusters_exist()* can be used to ensure that the `cluster_list` and `trans_matrix` are constructed, but will not cause a reconstruction if they are cached.
- Fixed a deprecation warning of `normalize=True` in sklearn's `Lasso` method.
- Added a benchmarking suite in the tests directory.

3.1.18 0.10.6

- Fixed a bug in the `clease.convexhull.ConvexHull` where multiple end-points wouldn't always find the correct minimum energy structure for that end-point.
- Added *MCEvaluator*.
- The settings class should now be much faster to construct, since the construction of the translation matrix and cluster list is defered until requested.
- The built in GUI (based on Kivy) has been removed, in favor of the new Jupyter based clease-gui package.
- Deprecated the use of `max_cluster_size` for specifying clusters in *ClusterExpansionSettings*. Clusters should now be specified only though `max_cluster_dia`, where the size of the cluster is infered from the length of the list. The index 0 corresponds to 2-body clusters, index 1 to 3-body etc., i.e. `max_cluster_dia = [5, 4, 3]` would result in clusters of up to diameter 5 Å for 2-body clisters, 4 Å for 3-body and 3 Å for 4-body.

3.1.19 0.10.5

- Added `clease.logging_utils.log_stream()` and `clease.logging_utils.log_stream_context()` functions to simplify printing the CLEASE logs to a file. The global CLEASE logger can be retreived with `clease.logging_utils.get_root_clease_logger()`.

3.1.20 0.10.4

- Fixed a bug with sorting the figures in `ClusterList` would cause a de-synchronization of the indices, and crashing any further usage.
- Now supports clusters of arbitrary size. Used to be limited to 2-, 3- and 4-body clusters.

3.1.21 0.10.3

- Added convex hull plot, *clease.plot_post_process.plot_convex_hull()*
- Fixed a bug in `clease.structgen.NewStructures.generate_gs_structures()` where passing multiple atoms objects was failing
- Structure generation of pure elements should now be using the smallest possible cell.
- Alpha and CV values are now stored in the *clease.evaluate.Evaluate* class after running the *clease.evaluate.Evaluate.alpha_CV()* function.
- Added *doc* as an `extras_require` in `setup.py`.
- Other minor bugfixes

3.1.22 0.10.2

- *clease.montecarlo.SSTEBarrier* renamed to *clease.montecarlo.BEPBarrier*
- Added release notes
- Added the `clease.jsonio` module. This has been applied to the *clease.settings.ClusterExpansionSettings*, `clease.settings.Concentration` and *clease.basis_function.BasisFunction* classes, providing them with `save()` and `load()` functions.
- Tests now automatically run in the pytest temporary directory.
- Moved `new_struct` and `structure_generator` into the `structgen` module. These should now be imported from here, instead.
- Fixed a bug, where the current step counter in the `clease.montecarlo.Montecarlo` class would not be reset upon starting a new run.

3.2 Au-Cu alloy example

3.2.1 Constructing your CE model

Specify the concentration ranges of species

The first step in setting up CE in ASE is to specify the types of elements occupying each basis and their concentration ranges using *Concentration* class. For AuCu alloys, we consider the entire composition range of Au_xCu_{1-x} where $0 \leq x \leq 1$. The *Concentration* object can be created simply as

```
>>> from clease.settings import Concentration
>>> conc = Concentration(basis_elements=[['Au', 'Cu']])
```

because there is no restriction imposed on the concentration range. Note that a nested list is passed for the `basis_elements` argument because the consituting elements are specified per basis and FCC (crystal structure of Au_xCu_sub:*1*_-_*x* for all $0 \leq x \leq 1$) has only one basis. The initialization automatically creates a linear algebra representation of the default concentration range constraints. The equality condition of

$$A_{\mathrm{eq}} = \begin{bmatrix}\begin{bmatrix}1 & 1\end{bmatrix}\end{bmatrix}$$

and

$$b_{\mathrm{eq}} = \begin{bmatrix}1\end{bmatrix}$$

as well as the lower bound conditions of

$$A_{\mathrm{lb}} = \begin{bmatrix}\begin{bmatrix}1 & 0\end{bmatrix}, \begin{bmatrix}0 & 1\end{bmatrix}\end{bmatrix}$$

and

$$b_{\mathrm{lb}} = \begin{bmatrix}0 & 0\end{bmatrix}$$

are created automatically. The conditions represents the linear equations

$$A_{\mathrm{eq}} c_{\mathrm{species}} = b_{\mathrm{eq}}$$

and

$$A_{\mathrm{lb}} c_{\mathrm{species}} \geq b_{\mathrm{lb}},$$

where the concentration list, c_{species}, is defined as

$$c_{\text{species}} = \begin{bmatrix} c_{\text{Au}} & c_{\text{Cu}} \end{bmatrix}.$$

The equality condition is then expressed as

$$c_{\text{Au}} + c_{\text{Cu}} = 1,$$

which specifies that elements Au and Cu constitute the entire basis (only one basis in this case). The lower bound conditions are expressed as

$$c_{\text{Au}} \geq 0$$

and

$$c_{\text{Cu}} \geq 0,$$

which speicifies that the concentrations of Au and Cu must be greater than or equal to zero.

The AuCu system presented in this tutorial does not impose any concentration constraints. However, we demonstrate how one can impose extra constraints by using an example case where the concentration of interest is Au_xCu_{1-x} where $0 \leq x \leq 0.5$. The extra concentration constraint can be specified in one of three ways.

The first method is to specify the extra constraint using `A_eq`, `b_eq`, `A_lb` and `b_lb`. For this particular case, the extra constraint is specified using `A_lb` and `b_lb` arguments as

```
>>> from clease.settings import Concentration
>>> conc = Concentration(basis_elements=[['Au', 'Cu']], A_lb=[[2, 0]], b_lb=[1])
```

A list of many examples on how linear systems equations can be used, is found *here*.

The second method is to specify the concentration range using formula unit strings. The *Concentration* class contains *set_conc_formula_unit()* method which accepts formula strings and variable range, which can be invoked as

```
>>> from clease.settings import Concentration
>>> conc = Concentration(basis_elements=[['Au', 'Cu']])
>>> conc.set_conc_formula_unit(formulas=["Au<x>Cu<1-x>"], variable_range={"x": (0, 0.5)})
```

The last method is to specify the concentration range each constituting species using *set_conc_ranges()* method in *Concentration* class. The lower and upper bound of species are specified in a nested list in the same order as the `basis_elements` as

```
>>> from clease.settings import Concentration
>>> conc = Concentration(basis_elements=[['Au', 'Cu']])
>>> conc.set_conc_ranges(ranges=[[(0, 0.5), (0.5, 1)]])
```

The above three methods yields the same results where x is constrained to $0 \leq x \leq 0.5$.

class clease.settings.concentration.**Concentration**(*basis_elements=None*, *grouped_basis=None*, *A_lb=None*, *b_lb=None*, *A_eq=None*, *b_eq=None*)

Specify concentration ranges of consituting elements for cluster expansion. Concentration range can be specified in three different ways.

1. specifying the equality and lower bound conditions by specifying `A_lb`, `b_lb`, `A_eq` and `b_eq` during initialization, 2. using `set_conc_formula_unit()` method, and 3. using `set_conc_ranges()` method.

Parameters:

basis_elements: list
List of chemical symbols of elements to occupy each basis. Even for the cases where there is only one basis (e.g., fcc, bcc, sc), a list of symbols should be grouped by basis as in [['Cu', 'Au']] (note the nested list form).

grouped_basis: list (optional, only used when basis are grouped)
Indices of basis_elements that are considered to be equivalent when specifying concentration (e.g., useful when two basis are shared by the same set of elements and no distinctions are made between them). As an example consider a structure with three sublattices A, B and C. If sublattice A and C should be occupied by the same elements and B is occupied by a different set of elements. We can group lattice A and C by passing [(0, 2), (1,)].

A_lb: list (optional, only used for linear algebra representation)
A two-dimention matrix (or nested list) used to specify the lower bounds of the concentration ranges.

b_lb: list (optional, only used for linear algebra representation)
A list used tp specify the lower bounds of the concentration ranges.

A_eq: list (optional, only used for linear algegra representation)
A two-dimention matrix (or nested list) used to specify the equality conditions of the concentration ranges.

b_eq: list (optional, only used for linear algegra representation)
A list used tp specify the equality condisitons of the concentration ranges.

Example I: Single sublattice

```
>>> conc = Concentration(basis_elements=[['Au', 'Cu']])
```

Example II: Two sublattices >>> conc = Concentration(basis_elements=[['Au', 'Cu'], ['Au', 'Cu', 'X']])

Example III: Three sublattices where the first and third are grouped >>> conc = Concentration(basis_elements=[['Au', 'Cu'], ['Au', 'Cu', 'X'], ['Au', 'Cu']], ... grouped_basis=[(0, 2), (1,)])

set_conc_formula_unit(*formulas=None*, *variable_range=None*)

Set concentration based on formula unit strings.

Parameters:

formulas: list
List constaining formula strings (e.g., ["Li<x>Ru<1>X<2-x>", "O<3-y>X<y>"], ['Al<4-4x>Mg<3x>Si<x>"]) 1. formula string should be provided per basis. 2. formula string can only have integer numbers. 3. only one dvariable is allowed per basis. 4. each variable should have at least one instance of 'clean' representation (e.g., <x>, <y>)

variable_range: dict
Range of each variable used in formulas. key is a string, and the value should be int or float e.g., {"x": (0, 2), "y": (0, 0.7)}, {'x': (0., 1.)}

set_conc_ranges(*ranges*)

Set concentration range based on lower and upper bounds of each element.

Parameters:

ranges: list
Nested list of tuples with the same shape as basis_elements. If basis_elements is [["Li", "Ru", "X"], ["O", "X"]], ranges coulde be [[(0, 1), (0.2, 0.5), (0, 1)], [(0, 0.8), (0, 0.2)]]

Stoichiometric Constraints

The most flexible method of imposing stoichiometric constraints in CLEASE is to use linear systems of equations. Here, you can find a list of examples of how different constraints can be imposed. In CLEASE a linear system of equations with the structure shown below

The number of sublattice concentration is simply the length of the *flattened* version of *basis_elements* that is passed to the *Concentration* class. Therefore, if `basis_element = [['Au', 'Cu'], ['Cu', 'X]]` there will be to Cu concentrations you can restrict; one for each sublattice. The total number of sublattice concentrations in the example above is 4. Hence, all rows of the matrix has 4 columns. CLEASE has two types of constraints: **equality** and **lower bound**. Equality constraints are passed via `A_eq` and `b_eq` arguments in the *Concentration* class, and lower bound constraints are passed via `A_lb` and `b_lb`. For lower bound constraints, the equality sign in the figure is replaced by a *larger or equal than*-symbol. Note that upper bound constraints can trivially be converted to a lower bound constraint by multiplying the equation by -1. Finally, the example below shows how you can generate random concentrations **satisfying** your constraints. The list passed to the function is the number of sites in each sublattice.

```
>>> import numpy as np
>>> np.random.seed(0)  # Set a seed for consistent tests
>>> from clease.settings import Concentration
```

Binary System With One Basis

```
>>> basis_elements = [['Au', 'Cu']]
```

This is a system where we have the `basis_elements=[['Au', 'Cu']]`.

1. Force the Au concentration to be equal to the Cu concentration

   ```
   >>> A_eq = [[1.0, -1.0]]
   >>> b_eq = [0.0]
   >>> conc = Concentration(basis_elements=basis_elements, A_eq=A_eq, b_eq=b_eq)
   >>> for i in range(10):
   ...     x = conc.get_random_concentration([20])
   ...     assert np.abs(x[0] - x[1]) < 1e-10
   ```

2. Force number of Au atoms to be larger than 12

   ```
   >>> A_lb = [[20, 0.0]]
   >>> b_lb = [12]
   >>> conc = Concentration(basis_elements=basis_elements, A_lb=A_lb, b_lb=b_lb)
   >>> for i in range(10):
   ...    x = conc.get_random_concentration([20])
   ...    assert round(20*x[0]) >= 12
   ```

Two sublattices

```
>>> basis_elements = [['Li', 'V'], ['O', 'F']]
```

1. Force the concentration of O to be twice the concentration of F

```
>>> A_eq = [[0.0, 0.0, -1.0, 2.0]]
>>> b_eq = [0.0]
>>> conc = Concentration(basis_elements=basis_elements, A_eq=A_eq, b_eq=b_eq)
>>> for i in range(10):
...     x = conc.get_random_concentration([18, 18])
...     assert abs(x[2] - 2*x[3]) < 1e-10
```

2. Li concentration larger than 0.2 and O concentration smaller than 0.7

```
>>> A_lb = [[1.0, 0.0, 0.0, 0.0], [0.0, 0.0, -1.0, 0.0]]
>>> b_lb = [0.2, -0.7]
>>> conc = Concentration(basis_elements=basis_elements, A_lb=A_lb, b_lb=b_lb)
>>> for i in range(10):
...    x = conc.get_random_concentration([18, 18])
...    assert x[0] >= 0.2 and x[2] < 0.7
```

Specify CE settings

The next step is to specify the settings in which the CE model is constructed. One of *CEBulk* or *CECrystal* classes is used to specify the settings. *CEBulk* class is used when the crystal structure is one of "sc", "fcc", "bcc", "hcp", "diamond", "zincblende", "rocksalt", "cesiumchloride", "fluorite" or "wurtzite".

Here is how to specify the settings for performing CE on Au_xCu_{1-x} for all $0 \leq x \leq 1$ on FCC lattice with a lattice constant of 3.8 Å

```
>>> from clease.settings import CEBulk
>>> settings = CEBulk(crystalstructure='fcc',
...                   a=3.8,
...                   supercell_factor=64,
...                   concentration=conc,
...                   db_name="aucu.db",
...                   max_cluster_dia=[6.0, 4.5, 4.5])
```

CEBulk internally calls `ase.build.bulk()` function to generate a unit cell. Arguments `crystalstructure`, `a`, `c`, `covera`, `u`, `orthorhombic` and `cubic` are passed to `ase.build.bulk()` function to generate a unit cell from which the supercells are generated. In case where one prefers to perform CE on a single, fixed size supercell, `size` parameter can be set by passing a list of three integer values (e.g., [3, 3, 3] for a $3 \times 3 \times 3$ supercell). More generally, a `supercell_factor` argument is specified to set a threshold on the maximum size of the supercell.

The maximum size of clusters (i.e., number of atoms in a given cluster) and their maximum diameters are specified using `max_cluster_dia`. As empty and one-body clusters do not need diamters in specifying the clusters, maximum diameters of clusters starting from two-body clusters are specified in `max_cluster_dia` in ascending order.

Note: Several entries are generated in the database file with their names assigned as "templates". These templates are used to generate new structures and also to calculate their correlation functions.

There are several flavors of cluter expansion formalism in specifying the basis function for setting the site variable. Three types of basis functions are currently supported in ASE. The type of basis function can be selected by passing one of "polynomial", "trigonometric" and "binary_linear" to `basis_function` argument. More information on each basis function can be found in the following articles.

"polynomial":

> Sanchez, J. M., Ducastelle, F. and Gratias, D. (1984)
> Generalized cluster description of multicomponent systems
> Physica A: Statistical Mechanics and Its Applications, 128(1-2), 334-350.

"trigonometric":

> van de Walle, A. (2009)
> Multicomponent multisublattice alloys, nonconfigurational entropy and other additions to the Alloy Theoretic Automated Toolkit
> Calphad, 33(2), 266-278.

"binary_linear":

> Zhang, X. and Sluiter M. (2016)
> Cluster expansions for thermodynamics and kinetics of multicomponent alloys.
> Journal of Phase Equilibria and Diffusion 37(1), 44-52.

One can alternatively use *CECrystal* class to specify the unit cell of the system. *CECrystal* takes a more general approach where the unit cell is specified based on its space group and the positions of unique sites.

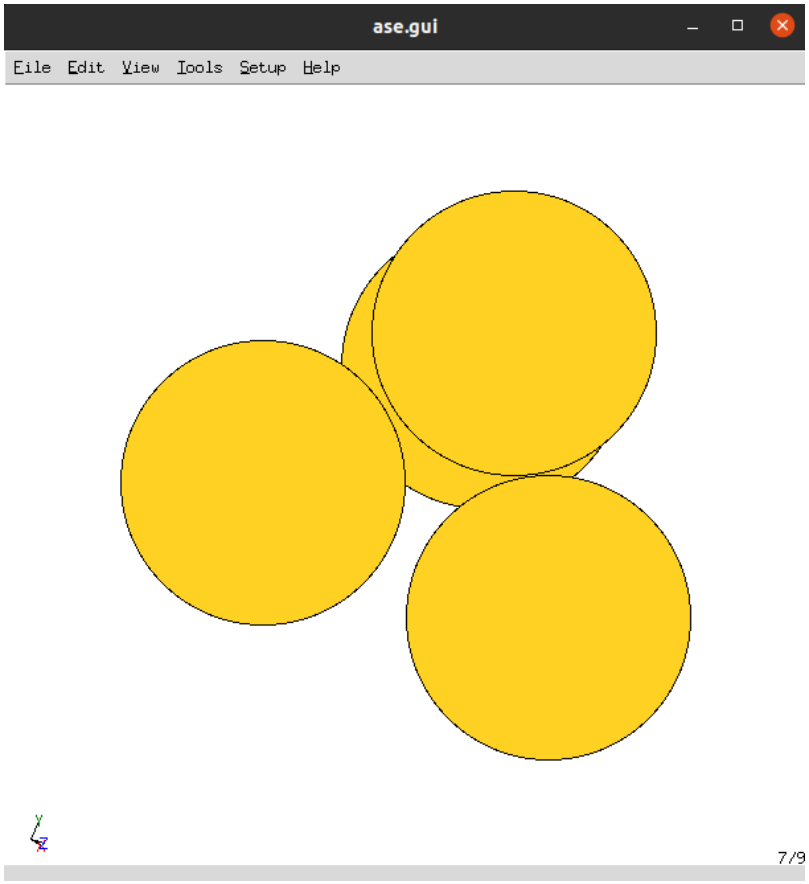
Verify your structures

After you created your templates, it may be a good idea to inspect the possible template structures and clusters, to verify that it looks like you would expect.

The `template` refers to all possible supercells that can be generated from your settings class, and `clusters` are the basic clusters found by CLEASE. Templates can be visualized with the *view_templates()*, and the clusters with *view_clusters()* of your *ClusterExpansionSettings* instance.

```
>>> settings.view_clusters()
```

This will open a new instance of the ASE GUI, which should look something like this, which is an example of a 4-body cluster:



And similarly, *view_templates()* will open the templates in the ASE GUI as well.

Generating initial structures

Generating initial pool of structures

After the cluster expansion settings is specified, the next step is to generate initial structures to start training the CE model. New structures for training CE model are generated using *NewStructures* class, which contains several methods for generating structures. The initial pool of structures is generated using *generate_initial_pool()* method as

```
>>> from clease.structgen import NewStructures
>>> ns = NewStructures(settings, generation_number=0, struct_per_gen=10)
>>> ns.generate_initial_pool()
```

The *generate_initial_pool()* method generates one structure per concentration where the number of each constituing element is at maximum/minimum. In the case of AuCu alloy, there are two extrema: Au and Cu. Consequently, *generate_initial_pool()* generates two structures for training.

Note:

- `generation_number` is used to track at which point you generated the structures.
- `struct_per_gen` specifies the maximum number of structures to be generated for that generation number.

The generated structures are automatically stored in the database with several key-value pairs specifying their features. The genereated keys are:

key	description
`gen`	generation number
`struct_type`	''initial" for input structures, "final" for converged structures after calculation
`size`	size of the supercell
`formula_unit`	reduced formula unit representation independent of the cell size
`name`	name of the structure (`formula_unit` followed by a number)
`converged`	Boolean value indicating whether the calculation of the structure is converged
`queued`	Boolean value indicating whether the calculation is queued in the workload manager
`started`	Boolean value indicating whether the calculation has started

Generating random pool of structures

As we have generated only two structures for training, we can generate more random structures using *generate_random_structures()* method by altering the above script with

```
>>> from clease.structgen import NewStructures
>>> ns = NewStructures(settings, generation_number=0,
...                    struct_per_gen=10)
>>> ns.generate_random_structures()
```

The script generates 8 additional random structures such that there are 10 structures in generation 0. By default, *generate_random_structures()* method generates a structure with both random size and concentration. If you prefer to generate random structures with a specific cell size, you can pass template atoms with desired size. For example, you can force the new structures to be $3 \times 3 \times 3$ supercell by using

```
>>> from ase.db import connect
>>> ns = NewStructures(settings, generation_number=0,
...                    struct_per_gen=10)
>>>
>>> # get template with the cell size = 3x3x3
>>> atoms = connect('aucu.db').get(id=10).toatoms()
>>>
>>> ns.generate_random_structures(atoms)
```

Running calculations on generated structures

For this tutorial, we use `EMT` calculator to demonstrate how one can run calculations on the structures generated using CLEAES and update database with the calculation results for further evaluation of the CE model. Here is a simple example script that runs the calculations for all structures that are not yet converged

```
>>> from ase.calculators.emt import EMT
>>> from ase.db import connect
>>> from clease.tools import update_db
>>> calc = EMT()
>>> db_name = "aucu.db"
>>> db = connect(db_name)
>>>
```

(continues on next page)

(continued from previous page)

```
>>> # Run calculations for all structures that are not converged.
>>> for row in db.select(converged=False):
...   atoms = row.toatoms()
...   atoms.calc = calc
...   atoms.get_potential_energy()
...   update_db(uid_initial=row.id, final_struct=atoms, db_name=db_name)
```

CLEASE has *update_db()* function to update the database entry with the calculation results. It automatically updates the intial structure entry and generates a new entry for the final structure. The key-value pairs of the initial structure entry are updated as:

key	description
converged	True
started	*empty*
queued	*empty*
final_struct_id	ID of the DB entry containing the final converged structure

clease.tools.**update_db**(*uid_initial=None*, *final_struct=None*, *db_name=None*, *custom_kvp_init: dict | None = None*, *custom_kvp_final: dict | None = None*)

Update the database.

Parameters:

uid_initial: int
entry ID of the initial structure in the database

final_struct: Atoms
Atoms object with the final structure with a physical quantity that needs to be modeled (e.g., DFT energy)

db_name: str
Database name

custom_kvp_init: dict (optional)
If desired, one can pass additional key-value-pairs for the entry containing the initial structure

custom_kvp_final: dict (optional)
If desired, one can pass additional key-value-pairs for the entry containing the final structure

Evaluating the CE model

We are now ready to evaluate a CE model constructed from the initial 10 calculations. The evaluation of the CE model is performed using `CEBulk` class, and it supports 3 different linear regression schemes: Bayesian Compressive Sensing (BCS), ℓ_1 and ℓ_2 regularization. We will be trying out ℓ_1 and ℓ_2 regularization schemes to see how they perform using the script below. The script is written to use ℓ_1 regularization as a fitting scheme (i.e., fitting_scheme='l1'), and you can change the fitting scheme to ℓ_2 simply by changing it to 'l2'.

For this tutorial, we use `EMT` calculator to demonstrate how one can run calculations on the structures generated using CLEASE and update database with the calculation results for further evaluation of the CE model. Here is a simple example script that runs the calculations for all structures that are not yet converged

```
>>> from clease import Evaluate
>>> import clease.plot_post_process as pp
>>> import matplotlib.pyplot as plt
```

(continues on next page)

(continued from previous page)

```
>>>
>>> eva = Evaluate(settings=settings, scoring_scheme='k-fold', nsplits=10)
>>> # scan different values of alpha and return the value of alpha that yields
>>> # the lowest CV score
>>> eva.set_fitting_scheme(fitting_scheme='l1')
>>> alpha = eva.plot_CV(alpha_min=1E-7, alpha_max=1.0, num_alpha=50)
>>>
>>> # set the alpha value with the one found above, and fit data using it.
>>> eva.set_fitting_scheme(fitting_scheme='l1', alpha=alpha)
>>> eva.fit()  # Run the fit with these settings.
>>>
>>> fig = pp.plot_fit(eva)
>>> plt.show()
>>>
>>> # plot ECI values
>>> fig = pp.plot_eci(eva)
>>> plt.show()
>>> # save a dictionary containing cluster names and their ECIs
>>> eva.save_eci(fname='eci_l1')
```

For more information, see *Evaluate*.

Generating structures for further training

You have now seen the initial cross validation (CV) score using 10 initial training structures. We can further train the CE model using more training structures to make it more robust.

CLEASE supports 3 ways to generate more strucures. The first (and most obvious) is generating random structures as you have already done. The second method is to generate so called "probe structures" which differ the most from the existing training structures. The third method is to generate ground-state structures predicted based on current CE model.

Generate probe structures

You can generate probe structures using the following script. Note that it internally uses simulated annealing algorithm which uses fictitious temperature values to maximize the difference in correlation function of the new structure.

```
>>> from clease import NewStructures
>>> ns = NewStructures(settings, generation_number=1, struct_per_gen=10)
>>> ns.generate_probe_structure()
```

Once 10 additional structures are generated, you can re-run the script in "Running calculations on generated structures" section to calculate their energies. You should also run the script in "Evaluation of the CE model" section to evaluate the CV score of the model.It is likely that the CV score of the model is sufficiently low (few meV/atom or less) at this point.

Generate ground-state structures

You can now genereate ground-state structures to construct convex-hull plot of formation energy. The script below generates ground-state structures with a cell size of $4 \times 4 \times 4$ at random compositions based on current CE model.

```
>>> from ase.db import connect
>>> import json
>>>
>>> # get template with the cell size = 4x4x4
>>> template = connect('aucu.db').get(id=17).toatoms()
>>>
>>> # import dictionary containing cluster names and their ECIs
>>> with open('eci_l1.json') as f:
...     eci = json.load(f)
>>>
>>> ns = NewStructures(settings, generation_number=2, struct_per_gen=10)
>>>
>>> ns.generate_gs_structure(atoms=template, init_temp=2000,
...                          final_temp=1, num_temp=10,
...                          num_steps_per_temp=5000,
...                          eci=eci, random_composition=True)
```

You should re-run the scripts in "Running calculations on generated structures" and "Evaluating the CE model" sections to see the convex-hull plot and the latest CV score of the model. If you observe that the CV score is high (more than ~5 meV/atom), you may want to repeat running the script for generating ground-state structures.

3.2.2 After constructing the CE model

Monte Carlo Sampling

CLEASE currently support two ensembles for Monte Carlo sampling: canonical and semi-grand canonical ensembles. A canonical ensemble has a fixed number of atoms, concentration and temperature while a semi-grand canonical ensemble has a fixed number of atoms, temperature and chemical potential. To use a fitted CE model to run MC sampling we first initialise *small* cell holding the nessecary information about the lattice and the clusters

```
from clease.settings import CEBulk, Concentration
conc = Concentration(basis_elements=[['Au', 'Cu']])
settings = CEBulk(crystalstructure='fcc',
                  a=3.8,
                  supercell_factor=27,
                  concentration=conc,
                  db_name="aucu.db",
                  max_cluster_dia=[6.0, 5.0])
```

Next, we need to specify a set if ECIs. These can for instance be loaded from a file, but here we hard code them for simplicity

```
eci = {'c0': -1.0, 'c1_0': 0.1, 'c2_d0000_0_00': -0.2}
```

For efficient initialisation of large cells, CLEASE comes with a convenient helper function called *attach_calculator*. We create our MC cell by repeating the *atoms* object of the settings.

```
from clease.calculator import attach_calculator
atoms = settings.atoms.copy()*(5, 5, 5)
atoms = attach_calculator(settings, atoms=atoms, eci=eci)
```

Let's insert a few *Cu* atoms

```
atoms[0].symbol = 'Cu'
atoms[1].symbol = 'Cu'
atoms[2].symbol = 'Cu'
```

We are now ready to run a MC calculation

```
from clease.montecarlo import Montecarlo
T = 500
mc = Montecarlo(atoms, T)
mc.run(steps=1000)
```

After a MC run, you can retrieve internal energy, heat capacity etc. by calling

```
thermo = mc.get_thermodynamic_quantities()
```

Monitoring a MC run

In many cases it is useful to be able to monitor the evolution of parameters during a run, and not simply getting the quantities after the run is finished. A good example can be to monitor the evolution of the energy in order to determine whether the system has reached equilibrium. CLEASE comes with a special set of classes called *MCObservers* for this task. As an example, we can store a value for the energy every 100 iteration by

```
from clease.montecarlo.observers import EnergyEvolution
obs = EnergyEvolution(mc)
mc.attach(obs, interval=100)
mc.run(steps=1000)
energies = obs.energies
```

Another useful observer is the *Snapshot* observer. This observers takes snaptshots of the configuration at regular intervals and stores them in a trajectory file.

```
from clease.montecarlo.observers import Snapshot
snap = Snapshot(atoms, fname='snapshot')
mc.attach(snap, interval=200)
mc.run(steps=1000)
```

There are many more observers distributes with CLEASE, for a complete list check the API documentation.

Constraining the MC sampling

In some cases you might want to prevent certain moves to occur. That can for instance be that certain elements should remain fixed. CLEASE offers the possibility to impose arbitrary constraint via its *MCConstraint* functionality. *MC-Constraints* can be added in a very similar fashion as the observers. To fix one element

```
from clease.montecarlo.constraints import FixedElement
cnst = FixedElement('Cu')
mc.generator.add_constraint(cnst)
```

Note, that the usage of a constraint in this system is a bit weird as it has only two elements. Hence, fixing one prevents any move from happening. But the point here is just to illustrate how a constraint can be attached.

Note: If your system has multiple basis, you most likely want to add a *ConstrainSwapByBasis* constraint object, in order to avoid swaps happening across different basis sites. The Montecarlo object will not automatically avoid cross-basis swaps.

Implementing Your Own Observer

You can implement your own observer and monitor whatever quantity you might be interested in. To to so you can create your own class that inherits from the base *MCObserver* class. To illustrate the usage, let's create an observers that monitor how many *Cu* atoms there are on average in each (100) layer!

Before we initialise this monitor we need to make sure that the tag of each atom represents the corresponding layer.

```
from clease.montecarlo.observers import MCObserver
from ase.geometry import get_layers
class LayerMonitor(MCObserver):
    def __init__(self, atoms):
        self.layers, _ = get_layers(atoms, [1, 0, 0])
        self.layer_average = [0 for _ in set(self.layers)]
        self.num_calls = 1
        # Initialise the structure
        for atom in atoms:
            if atom.symbol == 'Cu':
                self.layer_average[self.layers[atom.index]] += 1

    def observe_step(self, step):
        self.num_calls += 1
        system_changes = step.last_change
        for change in system_changes:
            layer = self.layers[change[0]]
            if change[2] == 'Cu':
                self.layer_average[layer] += 1
            if change[1] == 'Cu':
                self.layer_average[layer] -= 1

    def get_averages(self):
        return {'layer{}'.format(i): x/self.num_calls for i, x in enumerate(self.layer_
↪average)}
```

When this observer is attached, the *observe_step* method will be executed on every Monte Carlo step. The call signature takes in a *MCStep* instance. The *system_changes* variable here is a list of the following form *[(10, Au, Cu), (34, Cu,*

Au)] which means that the symbol on site 10 changes from Au to Cu and the symbol on site 34 changes from Cu to Au. Hence, in the update algorithm above we check if the last element of a single change is equal to Cu, if so we know that there is one additional Cu atom in the new layer. And if the middle element of a change is equal to Cu, there is one less atom in the corresponding layer. Note that if a MC move is rejected the *system_changes* will typically be *[(10, Au, Au), (34, Cu, Cu)]*. The *get_averages* function returns a dictionary. This method is optinal to implement, but if it is implemented the result will automatically be added to the result of *get_thermodynamic_quantities*

To use this observer in our calculation

```
monitor = LayerMonitor(atoms)
mc = Montecarlo(atoms, T)
mc.attach(monitor, interval=1)
mc.run(steps=1000)
```

There are a few other methods that can be useful to implement. First, the *reset* method. This method can be invoked if the *reset* method of the mc calculation is called.

Implementing Your Own Constraints

If you want to have custom constraints on MC moves, CLEASE lets you implement your own. The idea is to create a class that inherits from the base *MCConstraint* class and has a function *__call__** that returns *True* if a move is valid and *False* if a move is not valid. To illustrate this, let's say that we want the atoms on sites less that 25 to remain fixed. The reason for doing so, can be that you have a set of indices that you know constitutes a surface and you want to keep them fixed.

```
from clease.montecarlo.constraints import MCConstraint
class FixedIndices(MCConstraint):
   def __call__(self, system_changes):
       for change in system_changes:
           if change.index <= 25:
               return False
       return True
```

To use this constrain in our calculation

```
cnst = FixedIndices()
mc.generator.add_constraint(cnst)
mc.run(steps=1000)
```

Sampling the SGC Ensemble

CLEASE also gives the possibility to perform MC sampling in the semi grand canonical ensemble. Everything that has to do with observers and constraints mentioned above can also be used together with this class. To run a calcualtion in the SGC ensemble

```
from clease.montecarlo import SGCMonteCarlo
sgc_mc = SGCMonteCarlo(atoms, T, symbols=['Au', 'Cu'])
sgc_mc.run(steps=1000, chem_pot={'c1_0': -0.15})
```

The *chem_pot* parameter sets the chemical potentials. It is possible to set one chemical potential for each singlet correlation function (i.e. ECIs that starts with *c1*).

3.3 Metadynamics sampling

CLEASE offers the possibility to calculate free energies as a function of arbitrary collective variables via *metadynamics* sampling. A common collective variable is the concentration of species, but variable you can think of will work. In short we want to calculate curves as shown below

Let's move on to the details of how the sampling algorithm works. We start with the definition of the free energy

$$\exp(-F/kT) = \sum_{\sigma} \exp(-E(\sigma)/kT) = Z$$

where *k* is the Boltzmann constant, *T* is the temperature, *E* is the energy of a configuration, *Z* is the partition function and σ denotes an atomic configuration. Hence, the sum runs over all possible configurations. Furthermore, the probability that the system is in a state σ is given by

$$P(\sigma) = \frac{\exp(-E(\sigma)/kT)}{Z}$$

The free energy at a given value for a general collective variable *q* is defined by

$$\exp(-F(q)/kT) = \sum_{\sigma} \delta(f(\sigma) - q) \exp(-E(\sigma)/kT)$$

the function $f(\sigma)$ is a mapping from an atomic configuration to the sought collective variable. It might for instance return the concentration of the atomic arrangement. δ is a function that is 1 when $q = f(\sigma)$ and zero otherwise. Thus, the difference is now that contributions from configurations that has a different value of the collective variable is cancelled out. Since, we now summed over all configurations that satisfy $f(\sigma) = q$, the probability of finding the system in any state that satisfy $f(\sigma) = q$ can be obtained by dividing by *Z*

$$P(q) = \frac{\exp(-F(q)/kT)}{Z}$$

Now, let's see how the probabilities changes if we subtract an artificial potential *V(q)* that is only a function of the collective variable. First, we note that this potential can go inside the sum since the sum as only over configurations that has the same value for *q*. A new free energy *F'* can therefore be defined as follows

$$\exp(-F'(q)/kT) = \sum_{\sigma} \delta(f(\sigma) - q) \exp(-(E(\sigma) - V(q))/kT)$$

by comparison it follows that the relation between the two free energies is

$$F'(q) = F(q) - V(q)$$

Similarly, the probability of occupying any configuration with $f(\sigma) = q$ in the presence of an artificial potential is

$$P'(q) = \frac{\exp(-F'(q)/kT)}{Z} = \frac{\exp(-(F(q) - V(q))/kT)}{Z}$$

from the above equation, we note that if we are able to select a potential that is such that it is exactly equal to the original free energy, the probability of being in a state satisfying $f(\sigma) = q$ is

$$P'(q) = \frac{1}{Z}$$

which is constant for all values of q! Hence, if we partition the domain of possible q values into bins, monitor how often the MC sampler visits each bin and adaptively tune the artificial potential *V(q)* until we visit all bins equally often, we know that we have found the free energy.

3.3.1 Carrying out a metadynamics calculation in practice

As before, we first need to define the settings. Let's once again use our favorite example: AuCu!

```
>>> from clease.settings import CEBulk, Concentration
>>> conc = Concentration(basis_elements=[['Au', 'Cu']])
>>> settings = CEBulk(crystalstructure='fcc',
...                   a=3.8,
...                   supercell_factor=27,
...                   concentration=conc,
...                   db_name="aucu_metadyn.db",
...                   max_cluster_dia=[4.0])
```

The next thing we need to do is to load the ECIs and attach the calculator

```
>>> eci = {'c0': -1.0, 'c1_0': 0.1, 'c2_d0000_0_00': -0.2}
>>> atoms = settings.atoms.copy()*(5, 5, 5)
>>> from clease.calculator import attach_calculator
>>> atoms = attach_calculator(settings, atoms=atoms, eci=eci)
```

In pratice, the collective variables are calculated via one of the observers in CLEASE. If you plan to implement your own observers to use here, please note that there are certain requirements that needs to be satisfied if an observer should be applicable for metadynamics calculations.

- The *__call__* method needs to support a *peak* key word. Which is used to check what the collective variable is after a move, without actually performing the move

```
>>> def __call__(self, system_changes, peak=False):
...     pass
```

- It needs to have a method *calculate_from_scratch* that takes an atoms object as the only argument. This method is used to calculate the collective variable from scratch without making use of fast updates when the *system_changes* is known

```
>>> def calculate_from_scratch(self, atoms):
...     pass
```

In this example we are going to use the concentration observer to track the concentration of Au

```
>>> from clease.montecarlo.observers import ConcentrationObserver
>>> obs = ConcentrationObserver(atoms, element='Au')
```

Next, we need to define a sampler. Since, the nature of the problem requires that the concentration can change, we will use the Semi-Grand Canonical ensemble

```
>>> from clease.montecarlo import SGCMonteCarlo
>>> mc = SGCMonteCarlo(atoms, 600, symbols=['Au', 'Cu'])
```

Then we need to define the artificial bias potential. Here, we are going to use a binned potential, which is a potential that is defined via values on a grid.

```
>>> from clease.montecarlo import BinnedBiasPotential
>>> bias = BinnedBiasPotential(xmin=0.0, xmax=1.0, nbins=60, getter=obs)
```

Here, the minimum concentration is set to 0 and the maximum concentration is set to 1, and the domain is partitioned into 60 bins. At last, we pass everything to the metadynamics sampler

```
>>> from clease.montecarlo import MetaDynamicsSampler
>>> meta_dyn = MetaDynamicsSampler(mc=mc, bias=bias, flat_limit=0.8, mod_factor=0.01,
...                                fname='aucu_metadyn.json')
>>> meta_dyn.run(max_sweeps=1)
```

The parameter *flat_limit* is a threshold used to determine if we have visited all the bins equally likely. In the above example, the algorithm will say that all bins have been visited equally likely if the bins with the fewest visits is visited at least 80% of the average.

The *mod_factor* tunes how much we should modify the artificial potential when the sampler visits a bin. It is given in units of *kT*, hence the artifial potential is altered by *0.01*kT* everytime the sampler visits a bin. Finally, when we run we set here that the maximum number of sweeps is 1. This is only to avoid that the trial example takes too long running. This number should be much higher. If you set it *None*, the algorithm will run until it converges.

When you have managed to converge a calculation, you should reload the previous estimate, lower the modification factor and run again. Continue to lower the modification factor until the estimated free energy curve no longer changes.

To load an existing estimate, call this prior to passing the binned potential to the metadynamics sampler

```
>>> import json
>>> with open('aucu_metadyn.json', 'r') as f:
...     data = json.load(f)
>>> bias.from_dict(data['bias_pot'])
```

3.4 CLEASE Command Line Interface

The CLEASE package comes with a convenient command line tools, that can be used for various things.

1. Listing all tables in your database that contains correlation functions

   ```
   $ clease db mydb.db --show tab
   ```

2. Listing all the names of the correlation functions stored in your database

   ```
   $ clease db mydb.db --show names
   ```

3. Listing all the correlation functions of a particular entry

   ```
   $ clease db mydb.db --show cf --id 1
   ```

3.5 Importing Structures

If you have DFT data that are stored in different place/format than the CLEASE database (databases, trajectory files, xyz files, etc.), CLEASE offers the possibility of importing those structures. The only thing that needs to be provided is the **initial** (e.g. non-relaxed structure where all atoms are on ideal sites) and the total energy associated with it. Note that the total energy can be one of the relaxed structure. To show how this feature can be used we generate an example dataset using ASE's EMT calculator and store them in a trajectory file.

```
:options: +SKIP
>>> from ase.calculators.emt import EMT
>>> from ase.build import bulk
```

(continues on next page)

(continued from previous page)

```
>>> from ase.io.trajectory import TrajectoryWriter
>>> writer_initial = TrajectoryWriter("initial.traj")
>>> writer_final = TrajectoryWriter("final.traj")
>>> for i in range(10):
...     atoms = bulk("Au", a=4.05)*(3, 3, 3)
...     writer_initial.write(atoms)
...     calc = EMT()
...     atoms.calc = calc
...     en = atoms.get_potential_energy()
...     writer_final.write(atoms)
```

Next, we want to import these data into CLEASE. First, we create the settings

```
:options: +SKIP
>>> from clease.structgen import NewStructures
>>> from clease.settings import CEBulk, Concentration
>>> settings = CEBulk(
...     Concentration(basis_elements=[['Au', 'Cu']]),
...     crystalstructure='fcc', a=4.05, db_name="imported.db")
>>> new_struct = NewStructures(settings)

Next, we load our structures

>>> from ase.io.trajectory import TrajectoryReader
>>> reader_init = TrajectoryReader("initial.traj")
>>> reader_final = TrajectoryReader("final.traj")
>>> for i in range(len(reader_init)):
...     initial = reader_init[i]
...     final = reader_final[i]
...     ini_id = new_struct.insert_structure(init_struct=initial, final_struct=final)
```

Note that it is important that the final structure has energy. In case you have stored the structures in a way that the energy is not added to the structures when it is loaded, add the energy to the final structure via a **SinglePointCalculator**. Furthermore, if you only have the initial structure (and not the final), you can perfectly fine just replace the final structure with a copy of the initial.

3.6 API Documentation

3.6.1 Cluster Expansion Settings

clease.settings.**CEBulk**(*concentration:* Concentration, *crystalstructure='sc'*, *a=None*, *c=None*, *covera=None*, *u=None*, ***kwargs*)

Specify cluster expansion settings for bulk materials defined based on crystal structures.

Parameters

- **concentration** (*Union[Concentration, dict]*) – Concentration object or dictionary specifying the basis elements and concentration range of constituting species
- **crystalstructure** (*str*) – Must be one of sc, fcc, bcc, hcp, diamond, zincblende, rocksalt, cesiumchloride, fluorite or wurtzite.

- **a** (*float*) – Lattice constant.
- **c** (*float*) – Lattice constant.
- **covera** (*float*) – c/a ratio used for hcp. Default is ideal ratio: sqrt(8/3).
- **u** (*float*) – Internal coordinate for Wurtzite structure.

For more kwargs, see docstring of *clease.settings.ClusterExpansionSettings*.

clease.settings.**CECrystal**(*concentration: Concentration*, *spacegroup=1*, *basis=None*, *cell=None*, *cellpar=None*, *ab_normal=(0, 0, 1)*, *crystal_kwargs=None*, ***kwargs*)

Store CE settings on bulk materials defined based on space group.

Parameters

- **concentration** (*Union[Concentration, dict]*) – Concentration object or dictionary specifying the basis elements and concentration range of constituting species
- **spacegroup** (*int | string | Spacegroup instance*) – Space group given either as its number in International Tables or as its Hermann-Mauguin symbol.
- **basis** (*List[float]*) – List of scaled coordinates. Positions of the unique sites corresponding to symbols given either as scaled positions or through an atoms instance.
- **cell** (*3x3 matrix*) – Unit cell vectors.
- **cellpar** (*[a, b, c, alpha, beta, gamma]*) – Cell parameters with angles in degree. Is not used when *cell* is given.
- **ab_normal** (*vector*) – Is used to define the orientation of the unit cell relative to the Cartesian system when *cell* is not given. It is the normal vector of the plane spanned by a and b.
- **crystal_kwargs** (*dict | None*) – Extra kwargs to be passed into the ase.spacegroup.crystal function. Nothing additional is added if None. Defaults to None.

For more kwargs, see docstring of *clease.settings.ClusterExpansionSettings*.

class clease.settings.**ClusterExpansionSettings**(*prim: Atoms*, *concentration: Concentration | dict*, *size: List[int] | None = None*, *supercell_factor: int | None = 27*, *db_name: str = 'clease.db'*, *max_cluster_dia: Sequence[float] = (5.0, 5.0, 5.0)*, *include_background_atoms: bool = False*, *basis_func_type='polynomial'*)

Base class for all Cluster Expansion settings.

Parameters

- **prim** (*Atoms*) – The primitive atoms object.
- **concentration** (*Union[Concentration, dict]*) – Concentration object or dictionary specifying the basis elements and concentration range of constituting species.
- **size** (*List[int] | None, optional*) – Size of the supercell (e.g., [2, 2, 2] for 2x2x2 cell). `supercell_factor` is ignored if both `size` and `supercell_factor` are specified. Defaults to None.
- **supercell_factor** (*int, optional*) – Maximum multipilicity factor for limiting the size of supercell created from the primitive cell. `supercell_factor` is ignored if both *size* and *supercell_factor* are specified. Defaults to 27.
- **db_name** (*str, optional*) – Name of the database file. Defaults to `'clease.db'`.

- **max_cluster_dia** (*Sequence[float], optional*) – A list of int or float containing the maximum diameter of clusters (in Å). Defaults to `(5., 5., 5.)`, i.e. a 5 Å cutoff for 2-, 3-, and 4-body clusters.
- **include_background_atoms** (*bool, optional*) – Whether background elements are to be included. An element is considered to be a background element, if there is only 1 possible species which be ever be placed in a given basis. Defaults to False.
- **basis_func_type** (*str, optional*) – Type of basis function to use. Defaults to 'polynomial'.

property atomic_concentration_ratio: float

Ratio between true concentration (normalised to atoms) and the internal concentration used. For example, if one of the two basis is fully occupied, and hence ignored internally, the internal concentration is half of the actual atomic concentration.

property atoms: Atoms

The currently active template.

property background_indices: List[int]

Get indices of the background atoms.

clear_cache() → None

Clear the cached objects, due to a change e.g. in the template atoms

property cluster_list: ClusterList

Get the cluster list, will be created upon request

clusters_table() → str

String with information about the clusters

connect(***kwargs*) → Database

Return the ASE connection object to the internal database.

create_cluster_list_and_trans_matrix()

Prepares the internal cache objects by calculating cluster related properties

property db_name: str

Name of the underlaying data base.

ensure_clusters_exist() → None

Ensure the cluster list and trans matrix has been populated. They are not calculated upon creaton of the settings instance, for performance reasons. They will be constructed if required. Nothing is done if the cache exists.

classmethod from_dict(*dct: Dict[str, Any]*) → *ClusterExpansionSettings*

Load a new ClusterExpansionSettings class from a dictionary representation.

Example

```
>>> from clease.settings import CEBulk, Concentration, ClusterExpansionSettings
>>> conc = Concentration([['Au', 'Cu']])
>>> settings = CEBulk(conc, crystalstructure='fcc', a=4.1)
>>> dct = settings.todict()  # Get the dictionary representation
>>> # Remove the existing settings, perhaps due to being in a new environment
>>> del settings
>>> # Load in the settins from the dictionary representation
>>> settings = ClusterExpansionSettings.from_dict(dct)
```

get_active_sublattices() → List[bool]

List of booleans indicating if a (grouped) sublattice is active

get_all_figures_as_atoms() → List[Atoms]

Get the list of all possible figures, in their ASE Atoms representation.

get_all_templates()

Return a list with all template atoms

get_bg_syms() → Set[str]

Return the symbols in the basis where there is only one element

get_cluster_corresponding_to_cf_name(*cf_name: str*) → Cluster

Find the Cluster object which corresponds to a CF name. The cluster will not be specialized to the decoration number if such exists in the cf name.

Example

```
>>> from clease.settings import CEBulk, Concentration
>>> conc = Concentration([['Au', 'Cu']])
>>> settings = CEBulk(conc, crystalstructure='fcc', a=4.1)
>>> cluster = settings.get_cluster_corresponding_to_cf_name("c1_0")
>>> cluster.size
1
```

get_prim_cell_id(*write_if_missing=False*) → int

Retrieve the ID of the primitive cell in the database. Raises a PrimitiveCellNotFound error if it is not found and write_if_missing is False. If `write_if_missing` is True a primitive cell is written to the database if it is missing.

Returns the ID (an integer) of the row which corresponds to the primitive cell.

get_sublattice_site_ratios() → ndarray

Return the ratios of number of sites per (grouped) sublattice

property ignored_species_and_conc: Dict[str, float]

Return the ignored species and their concentrations normalised to the total number of atoms.

classmethod load(*fd*, ***kwargs*)

Method for loading class object from JSON

property max_cluster_dia: ndarray

The maximum cluster diameter, expressed in a NumPy array starting from 2-body clusters at index 0. Diameters are given in units of Ångstrom.

property multiplicity_factor: Dict[str, float]

Return the multiplicity factor of each cluster.

property non_background_indices: List[int]

Indices of sites which are not background

property num_active_sublattices: int

Number of active sublattices

property num_cf: int

Return the number of correlation functions.

prepare_new_active_template(*template*)

Prepare necessary data structures when setting new template.

property prim_cell: Atoms

The primitive atoms object of the model.

requires_build() → bool

Check if the cluster list and trans matrix exist. Returns True the cluster list and trans matrix needs to be built.

save(*fd*)

Method for writing class object to a JSON file.

set_active_template(*atoms=None*)

Set a new template atoms object.

todict() → Dict

Return a dictionary representation of the settings class.

Example

```
>>> from clease.settings import CEBulk, Concentration
>>> conc = Concentration([['Au', 'Cu']])
>>> settings = CEBulk(conc, crystalstructure='fcc', a=4.1)
>>> dct = settings.todict()  # Get the dictionary representation
```

property trans_matrix: TransMatrix

Get the translation matrix, will be created upon request

unique_element_without_background()

Remove background elements.

view_clusters() → None

Display all clusters along with their names.

view_templates()

Display all templates in the ASE GUi

clease.settings.**settings_from_json**(*fname*) → *ClusterExpansionSettings*

Initialize settings from JSON.

Exists due to compatibility. You should instead use *ClusterExpansionSettings.load(fname)*

Parameters:

fname: str
JSON file where settings are stored

3.6.2 Structure Generation

Module for generating new structures for training.

class clease.structgen.new_struct.**NewStructures**(*settings:* ClusterExpansionSettings, *generation_number: int | None = None*, *struct_per_gen: int = 5*, *check_db: bool = True*)

Generate new structure in ASE Atoms object format.

Parameters

- **settings** – Cluster expansion settings.
- **generation_number** – Generation number to be assigned to the newly generated structure
- **struct_per_gen** – Number of structures to generate per generation
- **check_db** – Should a new structure which is being inserted into the database be checked against pre-existing structures? Should only be disabled if you know what you are doing. Default is `True`.

connect(***kwargs*)

Short-cut to access the settings connection.

generate_conc_extrema() → None

Generate initial pool of structures with max/min concentration.

generate_gs_structure(*atoms: Atoms | List[Atoms]*, *eci: Dict[str, float]*, *init_temp: float = 2000.0*, *final_temp: float = 1.0*, *num_temp: int = 10*, *num_steps_per_temp: int = 1000*, *random_composition: bool = False*) → None

Generate ground-state structures based on cell sizes and shapes of the passed ASE Atoms.

Parameters

- **atoms** – Atoms object with the desired size and composition of the new structure. A list of Atoms with different size and/or compositions can be passed. Compositions of the supplied Atoms object(s) are ignored when random_composition=True.
- **eci** – cluster names and their ECI values
- **init_temp** – Initial temperature (does not represent *physical* temperature)
- **final_temp** – Final temperature (does not represent *physical* temperature)
- **num_temp** – Number of temperatures to use in simulated annealing
- **num_steps_per_temp** – Number of steps in simulated annealing
- **random_composition** – Whether or not to fix the composition of the generated structure.
  1. ***False* and atoms = Atoms object: One ground-state structure with** matching size and composition of the supplied Atoms object is generated
  2. ***False* and atoms = list: The same number of ground-state** structures that matches the length of the list is generated
     - **Note 1: num_struct_per_gen is ignored and all of the generated** structures have the same generation number

- **Note 2: each GS structure will have matching size and** composition of the suplied Atoms objects

3. ***True* and atoms = Atoms object: GS structure(s) with a** matching size of the Atoms object is generated at a random composition (within the composition range specified in Concentration class)
   - **Note 1: This will generate GS structures until the number of** structures with the current generation number equals num_struct_per_gen
   - **Note 2: A check is performed to ensure that none of the newly** generated GS structures have the same composition
4. ***True* and atoms = list: The same number of GS structures that** matches the length of the list is generated
   - **Note 1: num_struct_per_gen is ignored and all of the generated** structures have the same generation number
   - **Note 2: each GS structure will have matching sizes of the** supplied Atoms objects but with a random composition
   - **Note 3: No check is performed to ensure that all new GS** structures have unique composition

`generate_gs_structure_multiple_templates`(*eci: Dict[str, float]*, *num_templates: int = 20*, *num_prim_cells: int = 2*, *init_temp: float = 2000.0*, *final_temp: float = 1.0*, *num_temp: int = 10*, *num_steps_per_temp: int = 1000*) → None

Generate ground-state structures using multiple templates (rather than using fixed cell size and shape). Structures are generated until the number of structures with the current *generation_number* in database reaches *struct_per_gen*.

Parameters

- `num_templates` – Number of templates to search in. Simmulated annealing is done in each cell and the one with the lowest energy is taken as the ground state.
- `num_prim_cells` – Number of primitive cells to use when constructing templates. The volume of all the templates used will be num_prim_cells*vol_primitive, where vol_primitive is the volume of the primitive cell.

See docstring of *generate_gs_structure* for the rest of the arguments.

`generate_initial_pool`(*atoms: Atoms | None = None*) → None

Generate initial pool of structures.

Initial pool of structures are generated, in sequence, using

1. generate_conc_extrema(): structures at concentration where the number of consituting elements is at its max/min.
2. generate_random_structures(): random structures are random concentration.

Structures are genereated until the number of structures reaches *struct_per_gen*.

Parameters

`atoms` – If Atoms object is passed, the size and shape of its cell will be used for all the random structures. If None, a randome size and shape will be chosen for each structure.

`generate_metropolis_trajectory`(*atoms: Atoms | None = None*, *random_comp: bool = True*) → None

Generate a set of structures consists of single atom swaps

Parameters

- `atoms` – ASE Atoms object that will be used as a template for the trajectory

- **random_comp** – If 'True' the passed atoms object will be initialised with a random composition. Otherwise, the trajectory will start from the passed Atoms object.

generate_one_random_structure(*atoms: Atoms | None = None*) → bool

Generate and insert a random structure to database if a unique structure is found.

Returns `True` if unique structure is found and inserted in DB, `False` otherwise.

Parameters

atoms – If Atoms object is passed, the passed object will be used as a template for all the random structures being generated. If None, a random template will be chosen. (different for each structure)

generate_probe_structure(*atoms: Atoms | None = None*, *init_temp: float | None = None*, *final_temp: float | None = None*, *num_temp: int = 5*, *num_steps_per_temp: int = 1000*, *approx_mean_var: bool = True*, *num_samples_var: int = 10000*) → None

Generate a probe structure according to PRB 80, 165122 (2009).

Parameters

- **atoms** – ASE Atoms object with the desired cell size and shape of the new structure.
- **init_temp** – initial temperature (does not represent *physical* temperature)
- **final_temp** – final temperature (does not represent *physical* temperature)
- **num_temp** – number of temperatures to be used in simulated annealing
- **num_steps_per_temp** – number of steps in simulated annealing
- **approx_mean_var** – whether or not to use a spherical and isotropical distribution approximation scheme for determining the mean variance.

  -*True*: Assume a spherical and isotropical distribution of
  structures in the configurational space. Corresponds to eq.4 in PRB 80, 165122 (2009)

  -*False*: Use sigma and mu of eq.3 in PRB 80, 165122 (2009)
  to characterize the distribution of structures in population. Requires pre-sampling of random structures before generating probe structures. sigma and mu are generated and stored in 'probe_structure-sigma_mu.npz' file.
- **num_samples_var** – number of samples to be used in determining signam and mu. Only used when *approx_mean_var* is *True*.

Note: init_temp and final_temp are automatically generated if either
one of the two is not specified.

generate_random_structures(*atoms: Atoms | None = None*) → None

Generate random structures until the number of structures with *generation_number* equals *struct_per_gen*.

Parameters

atoms – If Atoms object is passed, the passed object will be used as a template for all the random structures being generated. If None, a random template will be chosen. (different for each structure)

insert_structure(*init_struct: Atoms | str*, *final_struct: Atoms | str | None = None*, *name: str | None = None*, *cf: Dict[str, float] | None = None*, *meta: Dict[str, Any] | None = None*, *warn_on_skip: bool = True*) → int | Tuple[int, int] | None

Insert a structure to the database.

Returns the ID of the initial structure which was inserted into the database. If a row for the final structure is also inserted, a tuple of (initial_id, final_id) is returned. If no structure was inserted, one is returned, instead.

Parameters

- **init_struct** – Unrelaxed initial structure. If a string is passed, it should be the file name with .xyz, .cif or .traj extension.
- **final_struct** – (Optional) final structure that contains energy. It can be either ASE Atoms object or file name readable by ASE.
- **name** – (Optional) name of the DB entry if a custom name is to be used. If *None*, default naming convention will be used.
- **cf** – (Optional) full correlation function of the initial structure (correlation functions with zero values should also be included). If cf is given, the preprocessing of the init_structure is bypassed and the given cf is inserted in DB.
- **meta** – (Optional) Extra information which will be added to the key-value pair entries in the database.
- **warn_on_skip** – (Bool, optional) Toggle emitting a warning if a structure was not inserted due to having a symmetrically equivalent structure in the database. Defaults to true.

insert_structures(*traj_init: str*, *traj_final: str | None = None*, *cb=<function NewStructures.<lambda>>*) → None

Insert a sequence of initial and final structures from their trajectory files.

Parameters

- **traj_init** – Name of a trajectory file with initial structures
- **traj_final** – Name of a trajectory file with the final structures
- **cb** – Callback function that is called every time a structure is inserted (or rejected because it exists before). The signature of the function is cb(num, tot) where num is the number of inserted structure and tot is the total number of structures that should be inserted

3.6.3 Basis Functions

Each cluster is defined on a set of cluster functions, which is expanded on a set of single-site basis functions. The basis function obeys the orthogonality condition

$$\frac{1}{M}\sum_{s_i=-m}^{m}\Theta_n(s_i)\Theta_{n'}(s_i)=\delta_{nn'}$$

For more information, please see the CLEASE paper. CLEASE implements three different basis functions: *Polynomial*, *Trigonometric* and *BinaryLinear*.

class clease.basis_function.**Polynomial**(*unique_elements: Sequence[str]*)

Pseudospin and basis function from Sanchez et al.

Sanchez, J. M., Ducastelle, F. and Gratias, D. (1984). Generalized cluster description of multicomponent systems. Physica A: Statistical Mechanics and Its Applications, 128(1-2), 334-350.

get_basis_functions() → List[Dict[str, float]]

Create basis functions to guarantee the orthonormality.

get_spin_dict() → Dict[str, int]

Define pseudospins for all consistuting elements.

class clease.basis_function.**Trigonometric**(*unique_elements: Sequence[str]*)

Pseudospin and basis function from van de Walle.

van de Walle, A. (2009). Multicomponent multisublattice alloys, nonconfigurational entropy and other additions to the Alloy Theoretic Automated Toolkit. Calphad, 33(2), 266-278.

get_basis_functions() → List[Dict[str, float]]

Create basis functions to guarantee the orthonormality.

get_spin_dict() → Dict[str, int]

Define pseudospins for all consistuting elements.

class clease.basis_function.**BinaryLinear**(*unique_elements: List[str]*, *redundant_element: str | None = 'auto'*)

Pseudospin and basis function from Zhang and Sluiter. The `redunant_element` parameter can be used to select which element is not explicitly defined by the ECI values. If it is not set, the element will be chosen as the first element in alphabetical order.

Zhang, X. and Sluiter M. Cluster expansions for thermodynamics and kinetics of multicomponent alloys. Journal of Phase Equilibria and Diffusion 37(1) 44-52.

customize_full_cluster_name(*full_cluster_name: str*) → str

Translate the decoration number to element names.

get_basis_functions() → List[Dict[str, float]]

Create orthonormal basis functions.

Due to the constraint that any site is occupied by exactly one element, we only need to track N-1 species if there are N species. Hence, the first element specified is redundant, and will not have a basis function.

get_spin_dict() → Dict[str, int]

Define pseudospins for all consistuting elements.

todict() → dict

Creates a dictionary representation of the class

All three basis functions inherit from the same base abstract base interface:

class clease.basis_function.**BasisFunction**(*unique_elements: Sequence[str]*)

Base class for all Basis Functions.

property basis_functions: List[Dict[str, float]]

Property access to *get_basis_functions()*.

customize_full_cluster_name(*full_cluster_name: str*) → str

Customize the full cluster names. Default is to do nothing.

abstract get_basis_functions()

Create basis functions which guarantees the orthonormality condition.

abstract get_spin_dict()

Get spin dictionary.

classmethod load(*fd*, ***kwargs*)

Method for loading class object from JSON

save(*fd*)

Method for writing class object to a JSON file.

todict() → dict

Create a dictionary representation of the basis function class

3.6.4 Correlation Functions

Module for calculating correlation functions.

class clease.corr_func.**CorrFunction**(*settings:* ClusterExpansionSettings)

Class for calculating the correlation functions.

Parameters

settings (ClusterExpansionSettings) – The settings object which defines the cluster expansion parameters.

property cf_table_name: str

Name of the table which holds the correlation functions.

check_consistency_of_cf_table_entries()

Get IDs of the structures with inconsistent correlation functions.

Note: consisent structures have the exactly the same list of cluster names as stored in settings.cf_names.

clear_cf_table() → None

Delete the external table which holds the correlation functions.

get_cf(*atoms*) → Dict[str, float]

Calculate correlation functions for all possible clusters and return them in a dictionary format.

Parameters

atoms (*Atoms*) – The atoms object

get_cf_by_names(*atoms*, *cf_names*) → Dict[str, float]

Calculate correlation functions of the specified clusters and return them in a dictionary format.

Parameters

- **atoms** – Atoms object
- **cf_names** – list names of correlation functions that will be calculated for the structure provided in atoms

iter_reconfigure_db_entries(*select_cond=None*) → Iterator[Tuple[int, int, int]]

Iterator which reconfigures the correlation function values in the DB, which yields after each reconfiguration and reports on the progress.

For more information, see *reconfigure_db_entries()*.

Yields

Tuple[int, int, int] – (row_id, count, total) A tuple containing the ID of the row which was just reconfigured, current count which has been reconfigured, as well as the total number of reconfigurations which will be performed. The percentage-wise progress is thus (count / total) * 100.

reconfigure_db_entries(*select_cond=None*, *verbose=False*)

Reconfigure the correlation function values of the entries in DB.

Parameters

- **select_cond** – One of either:
  - None (default): select every item in DB with `struct_type='initial'`

– Select based on the condictions provided (`struct_type='initial'` is not automatically included)

- **verbose** (*bool*) – print the progress of reconfiguration if set to *True*

reconfigure_inconsistent_cf_table_entries()

Find and correct inconsistent correlation functions in table.

reconfigure_single_db_entry(*row_id: int*) → None

Reconfigure a single DB entry. Assumes this is the initial structure, and will not check that.

Parameters

row_id – int The ID of the row to be reconfigured.

set_template(*atoms: Atoms*) → None

Check the size of provided cell and set as the currently active template in the settings object.

Parameters

atoms (*Atoms*) – Unrelaxed structure

3.6.5 Fitting ECIs

Table of Contents

The Evaluate Class

class clease.evaluate.**Evaluate**(*settings*, *prop='energy'*, *cf_names=None*, *select_cond=None*, *parallel=False*, *num_core='all'*, *fitting_scheme='ridge'*, *alpha=1e-05*, *max_cluster_size=None*, *max_cluster_dia=None*, *scoring_scheme='loocv'*, *min_weight=1.0*, *nsplits=10*, *num_repetitions=1*, *normalization_symbols: Sequence[str] | None = None*)

Evaluate RMSE/MAE of the fit and CV scores.

Parameters

- **settings** – ClusterExpansionSettings object
- **prop** – str User defined property for the fit. The property should exist in database as key-value pairs. Default is `energy`.
- **cf_names** – list Names of clusters to include in the evalutation. If None, all of the possible clusters are included.
- **select_cond** – tuple or list of tuples (optional) Custom selection condition specified by user. Default only includes "converged=True" and "struct_type='initial'".
- **max_cluster_size** – int maximum number of atoms in the cluster to include in the fit. If `None`, no restriction on the number of atoms will be imposed.
- **max_cluster_dia** – float or int maximum diameter of the cluster (in angstrom) to include in the fit. If `None`, no restriction on the diameter. Note that this diameter of the circumscribed sphere, which is slightly different from the meaning of max_cluster_dia

in *ClusterExpansionSettings* where it refers to the the maximum internal distance between any of the atoms in the cluster.

- **scoring_scheme** – str should be one of 'loocv', 'loocv_fast' or 'k-fold'
- **min_weight** – float Weight given to the data point furthest away from any structure on the convex hull. An exponential weighting function is used and the decay rate is calculated as

  decay = log(min_weight)/min(sim_measure)

  where sim_measure is a similarity measure used to asses how different the structure is from structures on the convex hull.
- **nsplits** – int Number of splits to use when partitioning the dataset into training and validation data. Only used when scoring_scheme='k-fold'
- **num_repetitions** – int Number of repetitions used to use when calculating k-fold cross validation. The partitioning is repeated num_repetitions times and the resulting value is the average of the k-fold cross validation score obtained in each of the runs.

alpha_CV(*alpha_min=1e-07*, *alpha_max=1.0*, *num_alpha=10*, *scale='log'*, *logfile=None*, *fitting_schemes=None*)

Calculate CV for a given range of alpha.

In addition to calculating CV with respect to alpha, a logfile can be used to extend the range of alpha or to add more alpha values in a given range.

Returns a list of alpha values, and a list of CV scores.

Parameters:

alpha_min: int or float
minimum value of regularization parameter alpha.

alpha_max: int or float
maximum value of regularization parameter alpha.

num_alpha: int
number of alpha values to be used in the plot.

scale: str

- 'log'(default): alpha values are evenly spaced on a log scale.
- 'linear': alpha values are evenly spaced on a linear scale.

logfile: file object, str or None.

- None: logging is disabled
- str: a file with that name will be opened. If '-', stdout used.
- file object: use the file object for logging

fitting_schemes: None or array of instance of LinearRegression.

Note: If the file with the same name exists, it first checks if the
alpha value already exists in the logfile and evalutes the CV of the alpha values that are absent. The newly evaluated CVs are appended to the existing file.

property atomic_concentrations

The actual atomic concentration (including background lattices) normalised against the total number of atoms

property concentrations

The internal concentrations normalised against the 'active' sublattices

`cv_for_alpha`(*alphas: List[float]*) → None

Calculate the CV scores for alphas using the fitting scheme specified in the Evaluate object.

Parameters

alphas – List of alpha values to get CV scores

`export_dataset`(*fname*)

Export the dataset used to fit a model y = Xc where y is typically the DFT energy per atom and c is the unknown ECIs. This function exports the data to a csv file with the following format

ECIname_1, ECIname_2, . . . , ECIname_n, E_DFT 0.1, 0.4, . . . , -0.6, -2.0 0.3, 0.2, . . . , -0.9, -2.3

thus each row in the file contains the correlation function values and the corresponding DFT energy value.

Parameter:
fname: str
Filename to write to. Typically this should end with .csv

`fit`() → None

Determine the ECI with the given regressor.

This will always calculate a new fit.

`fit_required`() → bool

Check whether we need to calculate the ECI values.

`generalization_error`(*validation_id: List[int]*)

Estimate the generalization error to new datapoints

Parameters

validation_ids – List with IDs to leave out of the dataset

`get_cv_score`()

Calculate the CV score according to the selected scheme

`get_eci`() → ndarray

Determine and return ECIs for a given alpha. Raises a ValueError if no fit has been performed yet.

Returns

A 1D array of floats with all ECI values.

Return type

np.ndarray

`get_eci_by_size`() → Dict[str, Dict[str, list]]

Classify distance, eci and cf_name according to cluster body size

Returns

Dictionary which contains

- Key: body size of cluster
- Value: A dictionary with the following entries:
  - ”distance” : distance of the cluster
  - ”eci” : eci of the cluster
  - ”name” : name of the cluster
  - ”radius” : Radius of the cluster in Ångstrom.

get_eci_dict(*cutoff_tol: float = 1e-14*) → Dict[str, float]

Determine cluster names and their corresponding ECI value and return them in a dictionary format.

Parameters

cutoff_tol (*float, optional*) – Cutoff value below which the absolute ECI value is considered to be 0. Defaults to 1e-14.

Returns

Dictionary with the CF names and the corresponding ECI value.

Return type

Dict[str, float]

get_energy_predict(*normalize: bool = True*) → ndarray

Perform matrix multiplication of eci and cf_matrix

Returns

Energy predicted using ECIs

k_fold_cv()

Determine the k-fold cross validation.

load_eci(*fname='eci.json'*) → None

Read in ECI values stored to a json file.

Note: this doesn't load the scheme or the alpha value, so it will not prevent a new fit to be performed if requested, as it may be incompatible with the current fitting scheme.

load_eci_dict(*eci_dict: Dict[str, float]*) → None

Load the ECI's from a dictionary. Any ECI's which are missing from the internal cf_names list are assumed to be 0.

Note: this doesn't load the scheme or the alpha value, so it will not prevent a new fit to be performed if requested, as it may be incompatible with the current fitting scheme.

loocv()

Determine the CV score for the Leave-One-Out case.

loocv_fast()

CV score based on the method in J. Phase Equilib. 23, 348 (2002).

This method has a computational complexity of order n^1.

mae()

Calculate mean absolute error (MAE) of the fit.

plot_CV(*alpha_min=1e-07*, *alpha_max=1.0*, *num_alpha=10*, *scale='log'*, *logfile=None*, *fitting_schemes=None*, *savefig=False*, *fname=None*)

Plot CV for a given range of alpha.

In addition to plotting CV with respect to alpha, logfile can be used to extend the range of alpha or add more alpha values in a given range. Returns an alpha value that leads to the minimum CV score within the pool of evaluated alpha values.

Parameters:

alpha_min: int or float
minimum value of regularization parameter alpha.

alpha_max: int or float
maximum value of regularization parameter alpha.

num_alpha: int
number of alpha values to be used in the plot.
scale: str

- 'log'(default): alpha values are evenly spaced on a log scale.
- 'linear': alpha values are evenly spaced on a linear scale.

logfile: file object, str or None

- None: logging is disabled
- str: a file with that name will be opened. If '-', stdout used.
- file object: use the file object for logging

fitting_schemes: None or array of instance of LinearRegression
savefig: bool

- **True: Save the plot with a file name specified in 'fname'. This** option does not display figure.
- False: Display figure without saving.

fname: str
file name of the figure (only used when savefig = True)
Note: If the file with the same name exists, it first checks if the alpha value already exists in the logfile and evalutes the CV of the alpha values that are absent. The newly evaluated CVs are appended to the existing file.

`plot_ECI`(*ignore_sizes=(0,)*, *interactive=True*)

Plot the all the ECI.

Parameters:
ignore_sizes: list of ints
Sizes listed in this list will not be plotted. Default is to ignore the emptry cluster.
interactive: bool
If `True`, one can interact with the plot using mouse.

`plot_fit`(*interactive=False*, *savefig=False*, *fname=None*, *show_hull=True*)

Plot calculated (DFT) and predicted energies for a given alpha.

Paramters:
alpha: int or float
regularization parameter.
savefig: bool

- **True: Save the plot with a file name specified in 'fname'.** Only works when interactive=False. This option does not display figure.
- False: Display figure without saving.

fname: str
file name of the figure (only used when savefig = True)
show_hull: bool
whether or not to show convex hull.

`print_coverage_report`(*file=<_io.TextIOWrapper name='<stdout>' mode='w' encoding='utf-8'>*) → None

Prints a report of how large fraction of the possible variation in each cluster is covered by the dataset
Parameters
`file` – a file-like object (stream); defaults to the current sys.stdout.

rmse()

Calculate root-mean-square error (RMSE) of the fit.

save_eci(*fname='eci.json'*, ***kwargs*)

Save a dictionary of cluster names and their corresponding ECI value in JSON file format.

Parameters:
fname: str
json filename. If no extension if given, .json is added
kwargs:
Extra keywords are passed on to the *get_eci_dict()* method.

set_normalization(*normalization_symbols: Sequence[str] | None = None*) → None

Set the energy normalization factor, e.g. to normalize the final energy reports in energy per metal atom, rather than energy per atom (i.e. every atom).

Parameters

normalization_symbols – A list of symbols which should be included in the counting. If this is None, then the default of normalizing to energy per every atom is maintained.

Fitting ECI's to Non-Energy Properties

Note: It is currently only possible to fit to values stored as key-value pairs in the database, i.e. it cannot be the default built-in `fmax` or similar properties, yet. To get around this, store the desired property as a key-value pair with a (slightly) different name.

Note: The desired target property should be stored in the row belonging to the **final** structure.

It is possible to fit ECI's to non-energy properties, and instead use values stored as key-value pairs. To do this, use the `prop` keyword in the evalutate class. As an example, say we already have a database of completed DFT calculations, and we wanted to fit to the average magnetic moment (why would want to do that you ask? In this case, for the sake of demonstration!).

Let's assume that this quantity has not already been calculated from our database, so we first loop through our final structures, find the average magnetic moment, and insert that quantity back in the database as a key-value pair.

```
from ase.db import connect
import numpy as np

db = connect("clease.db")  # We assume our database is called 'clease.db'
# Select all the final structures
for row in db.select(struct_type="final"):
    atoms = row.toatoms()
    avg_magmom = np.mean(atoms.get_magnetic_moments())
    # Insert the new quantity as a key-value pair.
    db.update(row.id, avg_magmom=avg_magmom)
```

Now we calculated the average magnetic moment of all our final structures. We can now do a fit on this new property with our evaluate class, `Evalutate(..., prop='avg_magmom')` and then proceeding as normal.

3.6.6 Fitting Schemes

class clease.regression.**LinearRegression**

fit(*X: ndarray*, *y: ndarray*) → ndarray

Fit a linear model by performing ordinary least squares

y = Xc

Parameters

- **X** – Design matrix (NxM)
- **y** – Data points (vector of length N)

class clease.regression.**Tikhonov**(*alpha: float | ndarray = 1e-05*, *penalize_bias_term: bool = False*, *normalize: bool = True*)

Ridge regularization.

Parameters

- **alpha** – regularization term
  - **float: A single regularization coefficient is used for all features.**
    Tikhonov matrix is T = alpha * I (I = identity matrix).
  - **1D array: Regularization coefficient is defined for each feature.**
    Tikhonov matrix is T = diag(alpha) (the alpha values are put on the diagonal). The length of array should match the number of features.
  - **2D array: Full Tikhonov matrix supplied by a user.**
    The dimensions of the matrix should be M * M where M is the number of features.
- **normalize** – If True each feature will be normalized to before fitting

fit(*X: ndarray*, *y: ndarray*) → ndarray

Fit coefficients based on Ridge regularizeation.

precision_matrix(*X: ndarray*) → ndarray

Calculate the presicion matrix.

class clease.regression.**Lasso**(*alpha: float = 1e-05*, *max_iter: int = 1000000*)

LASSO regularization.

Parameters

- **alpha** – regularization coefficient
- **max_iter** – (int) Maximum number of iterations.

fit(*X: ndarray*, *y: ndarray*) → ndarray

Fit coefficients based on LASSO regularizeation.

class clease.regression.ga_fit.**GAFit**(*cf_matrix*, *e_dft*, *mutation_prob=0.001*, *elitism=1*, *fname='ga_fit.csv'*, *num_individuals='auto'*, *max_num_in_init_pool=None*, *cost_func='aicc'*)

Genetic Algorithm for selecting relevant clusters.

Parameters:

cf_matrix: np.ndarray
Design matrix of the linear regression (nxm) where n is the number of data points and m is the number of features

e_dft: list
Array of length n with DFT energies

elitism: int
Number of best structures that will be passed unaltered on to the next generation

fname: str
File name used to backup the population. If this file exists, the next run will load the population from the file and start from there. Another file named 'fname'_cf_names.txt is created to store the names of selected clusters.

num_individuals: int or str
Integer with the number of inidivuals or it is equal to "auto", in which case 10 times the number of candidate clusters is used

max_num_in_init_pool: int
If given the maximum clusters included in the initial population is given by this number. If max_num_in_init_pool=150, then solution with maximum 150 will be present in the initial pool.

cost_func: str
Use the inverse as fitness measure. Possible cost functions: bic - Bayes Information Criterion aic - Afaike Information Criterion aicc - Modified Afaikes Information Criterion (tend to avoid overfitting better than aic)

`check_valid()`
Check that the current population is valid.

`create_new_generation()`
Create a new generation.

`design_matrix(individual)`
Return the corresponding design matrix.

`evaluate_fitness()`
Evaluate fitness of all species.

`static flip_one_mutation(individual)`
Apply mutation where one bit flips.

`get_eci(individual)`
Calculate the LOOCV for the current individual.

`index_of_selected_clusters(individual)`
Return the indices of the selected clusters

Parameters:
individual: int
Index of the individual

`static make_valid(individual)`
Make sure that there is at least two active ECIs.

`mutate()`
Introduce mutations.

`plot_evolution()`
Create a plot of the evolution.

`population_diversity()`
Check the diversity of the population.

`run(gen_without_change=100, min_change=0.01, save_interval=100)`
Run the genetic algorithm.

Return a list consisting of the names of selected clusters at the end of the run.

Parameters:

gen_without_change: int
Terminate if gen_without_change are created without sufficient improvement
min_change: float
Changes a larger than this value is considered "sufficient" improvement
save_interval: int
Rate at which all the populations are backed up in a file

class clease.regression.physical_ridge.**PhysicalRidge**(*lamb_size: float = 1e-06*, *lamb_dia: float = 1e-06*, *size_decay: str | Callable[[int], float] = 'linear'*, *dia_decay: str | Callable[[int], float] = 'linear'*, *normalize: bool = True*, *cf_names: List[str] | None = None*)

Physical Ridge is a special ridge regression scheme that enforces a convergent series. The physical motivation behind the choice of prior distributions is motivated by the fact that one expects that interactions strengths decays with both the number of atoms in the cluster and the diameter of the cluster. See for instance

Cao, L., Li, C. and Mueller, T., 2018. The use of cluster expansions to predict the structures and properties of surfaces and nanostructured materials. Journal of chemical information and modeling, 58(12), pp.2401-2413.

This fitting scheme uses Gaussian priors on the coefficients of the model

P(M) = P_size(M)*P_dia(M), where

P_size(M) = prod_i exp(-lamb_size*size_decay(size)*coeff_i^2) P_dia(M) = prod_i exp(-lamb_dia*dia_decay(dia)*coeff_i^2)

where size_decay and dia_decay is a monotonically increasing function of the size and diameter respectively. The product goes over all coefficients in the model M.

Parameters

- **lamb_size** – Prefactor in front of the size penalization
- **lamb_dia** – Prefactor in fron the the diameter penalization
- **size_decay** – The size_decay function in the priors explained above. It can be one of ['linear', 'exponential', 'polyN'], where N is any integer, or a callable function with the signature f(size), where size is the number of atoms in the cluster. If polyN is given the penalization is proportional to size**N
- **dia_decay** – The dia_decay function in the priors explained above. It can be one of ['linear', 'exponential', 'polyN'] where N is any integer, of a callable function with the signature f(dia) where dia is the diameter. If polyN is given the penalization is proportional to dia**N
- **normalize** – If True the data will be normalized to unit variance and zero mean before fitting.

  NOTE: Normalization works only when the first column in X corresponds to a constant. If the X matrix contains several simultaneous fits (e.g. energy, pressure, bulk moduli) there will typically be different columns that corresponds to the bias term for the different groups. It is recommended to put normalize=False for such cases.
- **cf_names** – List of strings, used to initialize the size and diameters which will be used.

add_constraint(*A: ndarray*, *c: ndarray*) → None

Adds a constraint that the coefficients (ECI) has to obey, A.dot(coeff) = c

Parameters

- **A** – Matrix describing the linear constraint
- **c** – Vector representing the right hand side of constraint equations

diameters_from_names(*names: List[str]*) → None

Extract the diameters from a list of correltion function names

Parameters

names – List of cluster names. The length of the list has to match the number of columns in the X matrix passed to the fit method. Ex: [‘c0’, ‘c1_1’, ‘c2_d0000_0_00’]

fit(*X: ndarray*, *y: ndarray*) → ndarray

Fit ECIs

Parameters

- **X** – Design matrix with correlation functions. The shape is N x M, where N is the number of data points and M is the number of correlation functions
- **y** – Vector with target values. The length of this vector is N (e.g. equal to the number of rows in X)

fit_data(*X: ndarray*, *y: ndarray*) → Tuple[ndarray, ndarray]

If normalize is True, a normalized version of the passed data is returned. Otherwise, X and y is returned as they are passed.

Parameters

- **X** – Design matrix
- **y** – Target data

sizes_from_names(*names: List[str]*) → None

Extract the sizes from a list of correlation function names

Parameters

names – List of cluster names. The length of the list has to match the number of columns in the X matrix passed to the fit method. Ex: [‘c0’, ‘c1_1’, ‘c2_d0000_0_00’]

class clease.regression.bayesian_compressive_sensing.**BayesianCompressiveSensing**(*shape_var=0.5*, *rate_var=0.5*, *shape_lamb=0.5*, *lamb_opt_start=200*, *variance_opt_start=100*, *fname='bayes_compr_sens.jso*
maxiter=100000, *output_rate_sec=2*, *select_strategy='max_increase'*, *noise=0.1*, *init_lamb=0.0*, *penalty=1e-08*)

Fit a sparse CE model to data. Based on the method described in

Babacan, S. Derin, Rafael Molina, and Aggelos K. Katsaggelos. “Bayesian compressive sensing using Laplace priors.” IEEE Transactions on Image Processing 19.1 (2010): 53-63.

Different values has different priors.

1. **For the ECIs a normal distribution is assumed**
   (the i-th eci is: eci_i – N(J | 0, var_i)=
2. **The inverce variance of each ECI is gamma distributed**
   (i.e. 1/var_i – gamma(x | 1, lambda/2))

3. **The lambda parameter above is also gamma distributed**
   (i.e. lamb – gamma(x | shape_lamb/2, shape_lamb/2))
4. **The noise parameter is uniformly distributed on the**
   positive axis (i.e. noise – uniform(x | 0, inf)

Parameters:

shape_var: float
Shape parameter for the gamma distribution for the inverse variance (1/var – gamma(x | shape_var/2, rate_var/2))

rate_var: float
Rate parameter for the gamma distribution for the inverse variance (1/var – gamma(x | shape_var/2, rate_var/2))

shape_lamb: float
Shape parameter for gamma distribution for the lambda parameter (lambda – gamma(x | 1, shape_lamb))

variance_opt_start: int
Optimization of inverse variance starts after this amount of iterations

lamb_opt_start: int
Optimization of lambda and shape_lamb starts after this amount of iterations. If this number is set very high, lambda will be kept at zero, making the algorithm efficitively a Relvance Vector Machine (RVM)

fname: str
Backup file for parameters

maxiter: int
Maximum number of iterations

output_rate_sec: int
Interval in seconds between status messages

select_strategy: str
Strategy for selecting new correlation function for each iteration. If 'max_increase' it will select the basis function that leads to the largest increase in likelihood value. If 'random' correlation functions are selected at random

noise: float
Initial estimate of the noise in the data

init_lamb: float
Initial value for the lambda parameter

penalty: float
Penalization value added to the diagonal of matrice to avoid singular matrices

`estimate_loocv()`
Return an estimate of the LOOCV.

`fit`(*X*, *y*)
Fit ECIs to the data

Parameters:

X: np.ndarray
Design matrix (NxM: N number of datapoints, M number of correlation functions)

y: np.ndarray
Array of length N with the energies

`get_basis_function_index`(*select_strategy*) → int
Select a new correlation function.

`log_likelihood_for_each_gamma`(*gammas*)
Log likelihood value for all gammas.

Parameters
gammas (*np.ndarray*) – Value for all the gammas

`mu()`
Calculate the expectation value for the ECIs

optimal_gamma(*indx*)

Return the gamma value that maximize the likelihood

Parameters:
indx: int
Index of the selected correlation function

optimal_inv_variance()

Calculate the optimal value for the inverse variance

optimal_lamb()

Calculate the optimal value for the lambda parameter.

optimal_shape_lamb()

Calculate the optimal value for the shape paremeter for lambda.

precision_matrix(*X*)

Return the precision matrix needed by the Evaluate class. Only contributions from the correlation functions with gamma > 0 are included.

rmse()

Return root mean square error.

save()

Save the results from file.

show_shape_parameter()

Show a plot of the transient equation for the optimal shape parameter for lambda.

todict()

Convert all parameters to a dictionary.

update_quantities()

Update helper parameters needed for the next iteration.

update_sigma_mu()

Update sigma and mu.

class clease.regression.sequential_cluster_ridge.**SequentialClusterRidge**(*min_alpha=1e-10*, *max_alpha=10.0*, *num_alpha=20*, *verbose: bool = False*)

SequentialClusterRidge is a fit method that optimizes the LOOCV over the regularization parameter as well as the cluster support. The method adds features in the design matrix X (see *fit* method) by including column by column. For each set of columns it performs a fit to a logspaced set of regularization parameters. The returned coefficients are the one from the model that has the smallest LOOCV.

Parameters:
alpha_min: float
Minimum value of the regularization parameter alpha
alpha_max: float
Maximum value of the regularization parameter alpha
num_alpha: int
Number of alpha values
verbose: bool
Print information about fit after completion

fit(*X*, *y*)

Performs the fitting

Parameters:
X: np.ndarray
Design matrix of size (N x M). During the CV optimization columns of X will be added one by one starting with a model consisting of the two first columns.
y: np.ndarray
Vector of length N

3.6.7 Monte Carlo

Table of Contents

Canonical MC

The canonical Monte Carlo class has the following API:

class clease.montecarlo.montecarlo.**Montecarlo**(*system: Atoms* | MCEvaluator, *temp: float*, *generator:* TrialMoveGenerator | *None = None*)

Class for running Monte Carlo at a fixed composition (canonical). For more information, also see the documentation of the parent class *BaseMC*.

Parameters

- **system** (*Union[ase.Atoms, MCEvaluator]*) – Either an ASE Atoms object with an attached calculator, or a pre-initialized *MCEvaluator* object.
- **temp** (*float*) – Temperature of Monte Carlo simulation in Kelvin
- **generator** (*TrialMoveGenerator, optional*) – A *TrialMoveGenerator* object that produces trial moves. Defaults to None.

add_bias(*potential: BiasPotential*)

Add a new bias potential.

Parameters:
potential:
Potential to be added

attach(*obs:* MCObserver, *interval: int = 1*)

Attach observers to be called on a given MC step interval.

Parameters:
obs: MCObserver
Observer to be added
interval: int
How often the observer should be called

count_atoms() → Dict[str, int]

Count the number of each element.

property current_accept_rate: float

Return the current accept rate as a value between 0 and 1.

get_thermodynamic_quantities() → Dict[str, Any]

Compute thermodynamic quantities.

initialize_run()

Prepare MC object for a new run.

irun(*steps: int*, *call_observers: bool = True*) → Iterator[*MCStep*]

Run Monte Carlo simulation as an iterator. Can be used to inspect the MC after each step, for example, to print the energy every 5 steps, one could do:

```
>>> mc = Montecarlo(...)
>>> for mc_step in mc.irun(500):
...     if mc_step.step % 5 == 0:
...         print(f"Current energy: {mc_step.energy:.2f} eV")
```

The iterator yields individual instances of *MCStep* for each step which is taken.

Parameters:

steps: int
Number of steps in the MC simulation

call_observers: bool
Should the observers be called during this run? Can be turned off for running burn-ins. The energy averagers will still be updated, even if this flag is disabled. Defaults to True.

iter_observers() → Iterator[*MCObserver*]

Directly iterate the attached observers without also getting information about the interval.

property meta_info: Dict[str, str]

Return dict with meta info.

reset() → None

Reset all member variables to their original values.

reset_averagers() → None

Reset the energy averagers.

run(*steps: int = 100*, *call_observers: bool = True*) → None

Run Monte Carlo simulation.

Parameters:

steps: int
Number of steps in the MC simulation

call_observers: bool
Should the observers be called during this run? Can be turned off for running burn-ins. The energy averagers will still be updated, even if this flag is disabled. Defaults to True.

Semi-grand canonical MC

The semi-grand canonical (SGC) Monte Carlo class:

class clease.montecarlo.sgc_montecarlo.**SGCMonteCarlo**(*atoms: Atoms*, *temp: float*, *symbols: Sequence[str] = ()*, *generator: TrialMoveGenerator | None = None*, *observe_singlets: bool = False*)

Class for running Monte Carlo in the Semi-Grand Canonical Ensebmle (i.e., fixed number of atoms, but varying composition)

See the docstring of `clease.montecarlo.Montecarlo`

Parameters

- **atoms** – Atoms object (with CLEASE calculator attached!)
- **temp** – Temperature in kelvin
- **symbols** – Possible symbols to be used in swaps
- **generator** – Generator that produces trial moves

get_thermodynamic_quantities(*reset_eci: bool = False*) → Dict[str, Any]

Compute thermodynamic quantities.

Parameters:
reset_eci: bool
If True, the chemical potential will be removed from the ECIs.

reset()

Reset the simulation object

reset_averagers() → None

Reset the energy averagers, including the internal SGC Observer

reset_eci()

Return the ECIs.

run(*steps: int = 10*, *call_observers: bool = True*, *chem_pot: Dict[str, float] | None = None*)

Run Monte Carlo simulation. See *run()*

Parameters:
chem_pot: dict
Chemical potentials. The keys should correspond to one of the singlet terms. A typical form of this is {"c1_0":-1.0,c1_1_1.0}

singlet2composition(*avg_singlets: Dict[str, float]*)

Convert singlets to composition.

Related Objects

All MC classes inherit from the *BaseMC* interface, which adds the following methods:

class clease.montecarlo.base.**BaseMC**(*system: Atoms | MCEvaluator*, *temp: float*)

Base Monte Carlo Class. Initializes the internal atoms and evaluator objects.

Parameters

- **system** (*Union[Atoms, MCEvaluator]*) – Either an ASE Atoms object with an attached calculator, or a pre-initialized *MCEvaluator* object.
- **temp** (*float*) – Temperature of Monte Carlo simulation in Kelvin

property T: float

Alias for the temperature variable. This variable name is deprecated in favor of *temperature*.

Type
float

property atoms: Atoms

The internal Atoms object.

Type
ase.atoms.Atoms

property evaluator: *MCEvaluator*

The internal evaluator object.

Getter
Returns the internal *MCEvaluator* object.

Setter
Sets the internal evaluator object. Can either accept an atoms object, or a pre-initialized evaluator object. See `system` in the docstring of the class constructor.

Type
MCEvaluator

property temperature: float

Property for getting and setting the temperature of the MC object.

Type
float

Individual steps from montecarlo iterations return *MCStep* objects:

class clease.datastructures.mc_step.**MCStep**(*step: int*, *energy: float*, *move_accepted: bool*, *last_move: Sequence[SystemChange]*, *other: Dict[str, Any] = _Nothing.NOTHING*)

Container with information about a single MC step. No validation checks are made in this class for performance reasons.

classmethod load(*fd*, ***kwargs*)

Method for loading class object from JSON

save(*fd*)

Method for writing class object to a JSON file.

Below are some related objects, which may be useful in your Monte Carlo endeavours.

Monte Carlo Constraints

class clease.montecarlo.constraints.**CollectiveVariableConstraint**(*xmin=0.0*, *xmax=1.0*, *getter=None*)

Constraint that ensures that the collective variable defined by the getter stays within certain bounds

Parameters:
xmin: float
Minimum value for the collective variable
xmax: float
Maximum value for the collective variable
getter: MCObserver
MCObsrever that support peak keyword that returns the collective variable after the proposed move

class clease.montecarlo.constraints.**ConstrainElementInserts**(*atoms*, *index_by_basis*, *element_by_basis*)

Constrain inserting the elements by basis. This constraint is intended to be used together with SGCMonteCarlo

atoms: Atoms object
ASE Atoms object used in the MC simulation

index_by_basis: list
Indices ordered by basis (same as *index_by_basis* parameter in *ClusterExpansionSettings*). If an Atoms object has 10 sites where the first 4 belongs to the first basis, the next 3 belongs to the next basis and the last 3 belongs to the last basis, the *index_by_basis* would be [[0, 1, 2, 3], [4, 5, 6], [7, 8, 9]]

element_by_basis: list
List specifying which elements are allowed in each basis. If there are two basis where Si and O are allowed in the fist basis while Si and C are allowed in the second basis, the argument would be [['Si', 'O'], ['Si', 'C']]

class clease.montecarlo.constraints.**ConstrainSwapByBasis**(*atoms: Atoms*, *index_by_basis: Sequence[Sequence[int]]*)

Constraint that restricts swaps of atoms within a given basis. This constraint is intended to be used together with canonical Monte Carlo calculations where the trial moves consist of swapping two atoms.

Parameters:

atoms: Atoms object
ASE Atoms object used in the MC simulation

index_by_basis: List[List[int]]
Indices ordered by basis (same as `index_by_basis` parameter in the *ClusterExpansionSettings* settings object.). If an Atoms object has 10 sites where the first 4 belongs to the first basis, the next 3 belongs to the next basis and the last 3 belongs to the last basis, the `index_by_basis` would be [[0, 1, 2, 3], [4, 5, 6], [7, 8, 9]].

Note: swaps are only allowed within each basis, not across two basis.

class clease.montecarlo.constraints.**FixedElement**(*element*)

Class for forcing an element of a certiain type to stay fixed.

Parameters:

element: str
Name of the element that is supposed to stay fixed

class clease.montecarlo.constraints.**FixedIndices**(*fixed_indices: Sequence[int]*)

Constrain a given set of indices during an MC run. Any suggested system changes by the MC algorithm are rejected if they invovle an index in the fixed indices.

Parameters:

fixed_indices: sequence of integers
The indices of the atoms object which are to be fixed.

class clease.montecarlo.constraints.**MCConstraint**

Class for that prevents the MC sampler to run certain moves

class clease.montecarlo.constraints.**PairConstraint**(*elements*, *pair_cluster*, *trans_matrix*, *atoms*)

Pair constraint is a constraint that prevents two species from being part of a pair cluster

Parameters:

elements: list
List of symbols (e.g. [Al, X] or [X, X])

pair_cluster: Cluster
Instance of a the Cluster class. An instance of a cluster class can for instance be obtained from a ClusterExpansionSettings object via settings.cluster_list.get_by_name("c2_d000_0")[0]

trans_matrix: list of dicts
Translation matrix for indices. This can be obtained from the trans_matrix attribute of the ClusterExpansionSettings object
atoms: Atoms object
Atoms object used for MC calculations

Monte Carlo Observers

class clease.montecarlo.observers.**AcceptanceRate**

Observer that tracks the fraction of monte carlo steps that is accepted

get_averages() → Dict[str, float]

Return dictionary with the rate such that it is added to thermodynaic quantities

property rate: float

Acceptance rate

reset()

Reset the observer

class clease.montecarlo.observers.**ConcentrationObserver**(*atoms: Atoms*, *element: str*)

Observer that can be attached to a MC run, to track the concenctration of a particular element. This observer has to be executed on every MC step.

Parameters:
atoms: Atoms object
Atoms object used for MC
element: str
The element that should be tracked

calculate_from_scratch(*atoms: Atoms*) → float

Calculate the concentration of the element in the atoms object.

get_averages() → Dict[str, float]

Return averages in the form of a dictionary.

interval_ok(*interval: int*) → bool

Every step must be observed, as otherwise we'd miss updates, and the concentration becomes incorerct.

new_concentration(*system_changes: Sequence[SystemChange]*) → float

Calculate the new consentration after the changes.

observe_step(*mc_step:* MCStep, *peak: bool = False*) → float

Observe on a MCStep object. Defaults to __call__(system_changes) for compatibility reasons. Child classes overriding this function should therefore not call the super() version.

reset() → None

Reset the observer

class clease.montecarlo.observers.**CorrelationFunctionObserver**(*calc*, *names=None*)

Track the history of the correlation function.

Parameters:
calc: *clease.calculators.Clease*
Clease calculator
names: list
List with correlation functions to track. If None, all correlation functions are tracked.

get_averages()

Return averages in the form of a dictionary.

reset()

Reset all values of the MC observer

class clease.montecarlo.observers.**DiffractionObserver**(*atoms=None*, *k_vector=()*, *active_symbols=()*, *all_symbols=()*, *name='reflection1'*)

Trace the reflection intensity.

Parameters:

atoms: Atoms

Atoms object used in Monte Carlo

k_vector: list

Fourier reflection to be traced

active_symbols: list

List of symbols that reflects

all_symbols: list

List of all symbols in the simulation

name: str

Name of the DiffractionObserver (users are given the freedom to set names because they can attach multiple DiffractionObserver instances)

Example:

Consider a system where Al, Mg and Si occupy FCC lattice sites. We want to trace the occurence of Mg layers that are separated by a distance 3*a where *a* is the lattice parameter. We further assume that the *y*-axis is normal to the planes we want to trace. In that case, we specify the variables as

```
>>> from ase.build import bulk
>>> import numpy as np
>>> a = 4.05
>>> atoms = bulk('Al', crystalstructure='fcc', a=a)
>>> k_vector = [0, 2.0*np.pi/(3*a), 0]
>>> active_elements = ['Mg']
>>> all_symbols = ['Al', 'Mg', 'Si']
```

If we do not wish to distinguish Mi and Si (we do not distiguish Mg layer, Si layer or a mixture of the two) the *active_elements* is changed to

```
>>> active_elements = ['Mg', 'Si']
```

get_averages()

Return averages in the form of a dictionary.

interval_ok(*interval*)

Check if the interval specified on attach is ok. Default is that all intervals are OK

Parameters

interval – Interval controlling how often a MC observer will be called.

reset()

Reset all values of the MC observer

class clease.montecarlo.observers.**EnergyEvolution**(*mc*, *ignore_reset=False*)

Trace the evolution of energy.

reset()

Reset the history.

save(*fname: str = 'energy_evolution.csv'*) → None

Save the energy evolution in .csv file.

Parameters

fname – File name of .csv file. Adds extension if none is given.

class clease.montecarlo.observers.**EnergyPlotUpdater**(*energy_obs=None*, *graph=None*, *mean_plot=None*)

class clease.montecarlo.observers.**EntropyProductionRate**(*buffer_length: int = 10000*, *logfile: str | Path = 'epr.txt'*)

Tracks entropy production rate (EPR) using a Gallavotti-Cohen functional.

EPR = 1/N sum_{i=0}^N ln P(i -> j)/P(j -> i)

N is the number of steps of a path and P(i -> j) is the probability of going from state i to state j. The expression is exact in the limit N -> infty. However, this class tracks the terms inside the sum and write them to file. To calculate the time evolution of EPR one can use a windowed average of the resulting data.

References:

[1] Gourgoulias, Konstantinos, Markos A. Katsoulakis, and Luc Rey-Bellet.
"Information criteria for quantifying loss of reversibility in parallelized KMC." Journal of Computational Physics 328 (2017): 438-454.

Parameters

- **buffer_length** – Length of buffer used to temporarily store the terms in the sum in memory. When the buffer is full, it is flushed to a text file.
- **logfile** – Filename of the file used when the buffer is flushed

reset()

Clear all information stored

update(*current: int*, *choice: int*, *cumulative_rates: ndarray*, *swaps: List[int]*)

Update the buffer

Parameters

- **current** – Current position of the vacancy
- **choice** – Index into cumulative_rates that is chosen
- **cumulative_rates** – Cumulative sum of the rates
- **swaps** – Possible swaps

class clease.montecarlo.observers.**LowestEnergyStructure**(*atoms: Atoms*, *track_cf: bool = False*, *verbose: bool = False*)

Track the lowest energy state visited during an MC run.

atoms: Atoms object

Atoms object used in Monte Carlo

track_cf: bool

Whether to keep a copy of the correlation functions for the emin structure. If enabled, this will be stored in the `lowest_energy_cf` variable. Defaults to False.

verbose: bool

If *True*, progress messages will be printed

property emin_results: dict

The results dictionary of the lowest energy atoms

property energy

The energy of the current atoms object (not the emin energy)

observe_step(*mc_step:* MCStep) → None

Check if the current state has lower energy and store the current state if it has a lower energy than the previous state.

mc_step: MCStep

Instance of MCStep with information on the latest step.

reset() → None

Reset all values of the MC observer

class clease.montecarlo.observers.**MCObserver**

Base class for all MC observers.

Child observers should override the observe_step() method.

calculate_from_scratch(*atoms: Atoms*) → None

Method for calculating the tracked value from scratch (i.e. without using fast update methods)

get_averages() → dict

Return averages in the form of a dictionary.

interval_ok(*interval: int*) → bool

Check if the interval specified on attach is ok. Default is that all intervals are OK

Parameters

interval – Interval controlling how often a MC observer will be called.

observe_step(*mc_step:* MCStep) → None

Observe on a MCStep object. Defaults to __call__(system_changes) for compatibility reasons. Child classes overriding this function should therefore not call the super() version.

reset() → None

Reset all values of the MC observer

class clease.montecarlo.observers.**MoveObserver**(*base_atoms: Atoms*, *only_accept: bool = False*)

Store each step from an MC run to reconstruct the individual atoms objects later.

The interval must be set to 1 when attaching this observer, as otherwise steps may be lost and the reconstruction may end up being incorrect.

Parameters

- **base_atoms** (*ase.Atoms*) – The base atoms object which is run in the MC.
- **only_accept** (*bool, optional*) – Selects whether the only accepted moves or all the attempted moves are saved. If False, every move will be saved. Defaults to False.

interval_ok(*interval: int*) → bool

Missing steps will result in incorrect reconstructions

observe_step(*mc_step:* MCStep) → None

Observe a single step

reconstruct() → List[Atoms]

Rebuild the atoms objects as defined by the observed changes.

reconstruct_iter() → Iterator[Atoms]

Iterator which builds the atoms objects 1-by-1.

reset() → None

Reset all values of the MC observer

class clease.montecarlo.observers.**MultiStateSGCConcObserver**(*ref_state: SGCState*, *thermo_states: List[SGCState]*, *calc: Clease*)

Observer that tracks the concentration at severl different temperatures and/or chemical potentials. The observer utilizes the following results. Let A be an observable, $\beta = \frac{1}{kT}$, μ the chemical potential and n the number of atoms of one of the species in a binary alloy. The average value of the observable is given by

$$\langle A \rangle = \frac{\sum_{conf} A \exp\left(\beta\mu n - \beta E\right)}{Z(\beta, \mu)}$$

where Z is the partition function. At a different chemical potential μ and inverse temperature β',

$$\langle A' \rangle' = \frac{\sum_{conf} A \exp\left(\beta'\mu' n - \beta' E\right)}{Z(\beta', \mu')}$$

After some algabraic manipulation one arrives at

$$\langle A' \rangle' = \frac{\langle A \exp\left((\beta'\mu' - \beta\mu)n - (\beta - \beta')E\right) \rangle}{\langle \exp\left((\beta'\mu' - \beta\mu)n - (\beta - \beta')E\right) \rangle}$$

where the averages should be taken at inverse temperature β and chemical potential μ. It should be noted that the predicted value will not be accurate if μ' or β' is very different from the reference values μ and β.

Parameters

- **ref_state** – Reference state
- **thermo_states** – List of SGCStates where the concentration should be tracked
- **calc** – Reference to the calculator attached to the atoms object used in the Monte Carlo simulation

get_averages() → Dict[str, float]

Return a dictionary with the calculated averages

reset() → None

Resets the observers to its initial state

class clease.montecarlo.observers.**SGCObserver**(*calc: Clease*, *observe_singlets: bool = False*)

Observer mainly intended to track additional quantities needed when running SGC Monte Carlo. This observer has to be executed on every MC step.

Parameters:

calc: *clease.calculators.Clease*

Clease calculator

observe_singlets: bool

Whether the singlet values of the calculator are measured during each observation. Measuring singlets is slightly more expensive, so this is disabled by default.

get_current_energy() → float

Return the current energy of the attached calculator object.

interval_ok(*interval*)

Check if the interval specified on attach is ok. Default is that all intervals are OK

Parameters

interval – Interval controlling how often a MC observer will be called.

observe_step(*mc_step:* MCStep)

Update all SGC parameters.

reset()

Reset all variables to zero.

class clease.montecarlo.observers.**SGCState**(*temp: float*, *chem_pot: Dict[str, float]*)

Represent a thermodynamic state in the semi-grand-canonical ensemble.

Parameters

- **temp** – Temperature in kelvin
- **chem_pot** – Chemical potentials of the form {c1_0: -0.2, c1_1: 0.3}. The function *clease.tools.species_chempot2eci* is useful to convert chemical potentials given for each species to chemical potentials for each singlet.

property prefix: str

Construct a prefix based on the chemical potentials and the temperature

class clease.montecarlo.observers.**SiteOrderParameter**(*atoms*)

Detect phase transitions by monitoring the average number of sites that are occupied by a different element from the initial structure. This observer has to be executed on every MC step.

Parameters:

atoms: Atoms object

Atoms object use for Monte Carlo

get_averages()

Get the average and standard deviation of the number of sites that are different from the initial state.

interval_ok(*interval*)

Check if the interval specified on attach is ok. Default is that all intervals are OK

Parameters

interval – Interval controlling how often a MC observer will be called.

reset()

Resets the tracked data. (Not the original symbols array).

class clease.montecarlo.observers.**Snapshot**(*atoms: Atoms*, *fname: str = 'snapshot.traj'*, *mode: str = 'w'*)

Store a snapshot in a trajectory file.

Parameters

- **atoms** – Instance of the atoms objected modofied by the MC object
- **fname** – Name of the trajectory file. Adds extension '.traj' if none is given.
- **mode** – IO mode used by the ASE TrajectoryWriter (must be w or a)

close()

Close the trajectory file.

Monte Carlo Evaluator

For standard Monte Carlo (MC) runs using the standard `clease.calculator.clease.Clease` cluster expansion (CE) calculator, this is generally not required. However, it is possible to use the *clease.montecarlo.montecarlo.Montecarlo* class without the CLEASE calculator, and use a different calculator instead.

In general, if the atoms object has a generic calculator attached, which is not a CLEASE calculator, it will assume it is an ASE calcualtor, and simply use the `get_potential_energy` method of the calculator object. This will also cause a complete re-evaluation of the entire system whenever a change is proposed in the MC algorithm, which may or may not be desired. The specifics of how to deal with local changes in the energy evaluation is up to the individual cases, but let's take a look at how to use the ASE EMT calculator with the MC class, using the *MCEvaluator* class.

An Example

Let's assume we have a system comprised of `Au`, `Cu` and vacancies (in ASE denoted as `X`). The EMT calculator is unable to evaluate an atom which is `X`, however we need to keep track of them anyway in the Monte Carlo run. We can then create a new MC evaluator, which changes the rules for how we get the energy, by removing vacancies from the atoms object prior to evaluating the energy.

```
>>> from clease.montecarlo import MCEvaluator
>>> from ase.calculators.emt import EMT
>>> class MyEvaluator(MCEvaluator):
...     def __init__(self, atoms):
...         super().__init__(atoms)
...         # Have a pre-made calculator instance ready
...         self.calc = EMT()
...     def get_energy(self, applied_changes = None) -> float:
...         # Make a copy of the atoms, and remove all vacancies.
...         atoms_cpy = self.atoms.copy()
...         mask = [atom.index for atom in atoms_cpy if atom.symbol != 'X']
...         atoms_cpy = atoms_cpy[mask]
...
...         atoms_cpy.calc = self.calc
...         return atoms_cpy.get_potential_energy()
```

Note that we overwrite the `get_energy` method of the *MCEvaluator*, in order to have custom rules for the energy evaluation. Let's create an example system to run the MC on:

```
>>> from ase.build import bulk
>>> atoms = bulk('Au') * (5, 5, 5)
>>> atoms.symbols[:10] = 'Cu'
>>> atoms.symbols[10:20] = 'X'
>>> print(atoms.symbols)
Cu10X10Au105
```

We can now run our Monte Carlo:

```
>>> from clease.montecarlo import Montecarlo
>>> temp = 300  # 300 kelvin
>>> evaluator = MyEvaluator(atoms)
>>> mc = Montecarlo(evaluator, temp)
>>> mc.run(steps=10)
```

Which successfully now runs our MC simulation on an atoms object using custom energy evaluation rules. You can write your own custom evaluators to do more complex things, such as utilizing the `applied_changes` keyword, to make energy evaluations only consider local changes to the atoms object.

The API

class clease.montecarlo.mc_evaluator.**MCEvaluator**(*atoms: Atoms*)

A Montecarlo evaluator class, used to perform the energy evaluations within a Montecarlo run.

Parameters

atoms (*ase.Atoms*) – ASE Atoms object to be used for the evaluation. This atoms object may be mutated.

apply_system_changes(*system_changes: Sequence[SystemChange]*, *keep=False*) → None

Mutate the atoms object to reflect the system change.

Parameters

- **system_changes** (*SystemChanges*) – Sequence of changes to be applied.
- **keep** (*bool, optional*) – Whether to call *keep_system_changes()* after applying changes. Defaults to False.

get_energy(*applied_changes: Sequence[SystemChange] | None = None*) → float

Evaluate the energy of a system. If a change is sufficiently local/small, it there, in some situations, may be other ways of evaluating the energy than a full calculation. Must return the energy of the new configuration.

The applied changes only reflect what has already been applied to the system.

Parameters

applied_changes (*SystemChanges, optional*) – A list of changes which has been applied to the atoms object. This change has already been applied, and is only for bookkeeping purposes, if evaluation schemas want to make decisions based on what has changed. Defaults to None.

Returns

Energy of the atoms object.

Return type

float

get_energy_given_change(*system_changes: Sequence[SystemChange]*) → float

Calculate the energy of a set of changes, and undo any changes.

Parameters

- **atoms** (*Atoms*) – Atoms object to be mutated.
- **system_changes** (*SystemChanges*) – Sequence of changes to be applied.

Returns

The resulting energy from a call to *get_energy()*.

Return type

float

keep_system_changes(*system_changes: Sequence[SystemChange] | None = None*) → None

A set of system changes are to be kept. Perform necessary actions to prepare for a new evaluation.

reset() → None

Perform a reset on the evaluator and/or on the atoms

synchronize() → None

Ensure the calculator and atoms objects are synchronized.

undo_system_changes(*system_changes: Sequence[SystemChange]*) → None

Mutate the atoms object to undo the system change.

Parameters

- **atoms** (*Atoms*) – Atoms object to be mutated.
- **system_changes** (*SystemChanges*) – Sequence of changes to be applied.

Trial Move Generators

Trial moves used in Monte Carlo (MC) sampling in Clease are provided from *clease.montecarlo.trial_move_generator.TrialMoveGenerator* classes.

The API

class clease.montecarlo.trial_move_generator.**TrialMoveGenerator**(*max_attempts: int = 10000*)

Class for producing trial moves.

Parameters

max_attempts – Maximum number of attempts to try to find a move that passes the constraints. If not constraints are added, this has no effect.

add_constraint(*cnst:* MCConstraint)

Add a constraint to the generator

Parameters

cnst – Constraint that must be satisfied for all trial moves

abstract get_single_trial_move() → Sequence[SystemChange]

Return a single trial move, must be implemented in sub-classes

get_trial_move() → Sequence[SystemChange]

Produce a trial move that is consistent with all cosntraints

initialize(*atoms: Atoms*) → None

Initialize the generator.

Parameters

atoms – Atoms object used in the simulation

on_move_accepted(*changes: Sequence[SystemChange]*) → None

Callback that is called by Monte Carlo after each accepted move

param change

Seqeunce of trial moves performed

on_move_rejected(*changes: Sequence[SystemChange]*) → None

Callback that is called after a move is rejected

Parameters

change – Seqeunce of trial moves performed

remove_constraints() → None

Remove all constraints

class clease.montecarlo.trial_move_generator.**SingleTrialMoveGenerator**(***kwargs*)

Interface class for generators that return only one type of trial moves

made_changes(*changes: Sequence[SystemChange]*) → List[SystemChange]

Extract the subset system changes made by an instance of itself. This method can be overrided in sublcasses, but the default behavior is to extract the subset of changes where the name matches.

name_matches(*change: SystemChange*) → bool

Return true of the name of the passed system change matches the CHANGE_NAME attribute.

Parameters

change – a system change

class clease.montecarlo.trial_move_generator.**RandomFlip**(*symbols: Set[str]*, *atoms: Atoms*, *indices: List[int] | None = None*, ***kwargs*)

Generate trial moves where the symbol at a given site is flipped

Parameters

- **symbols** – Set with all symbols considered in a move
- **atoms** – Atoms object for the simulation
- **indices** – List with all indices that should be considered. If None, all indices are considered

get_single_trial_move() → List[SystemChange]

Get a random flip of an included site into a different element.

class clease.montecarlo.trial_move_generator.**RandomSwap**(*atoms: Atoms*, *indices: List[int] | None = None*, ***kwargs*)

Produce random swaps

Parameters

- **atoms** – Atoms object in the MC simulation
- **indices** – List with indices that can be chosen from. If None, all indices can be chosen.

get_single_trial_move() → List[SystemChange]

Create a swap move

is_tracked(*index: int*) → bool

Check if a given index is being tracked.

on_move_accepted(*changes: Sequence[SystemChange]*)

Callback that is called by Monte Carlo after each accepted move

param change

Seqeunce of trial moves performed

class clease.montecarlo.trial_move_generator.**MixedSwapFlip**(*atoms: Atoms*, *swap_indices: Sequence[int]*, *flip_indices: Sequence[int]*, *flip_symbols: Sequence[str]*, *flip_prob: float = 0.5*, ***kwargs*)

Class for generating trial moves in a mixed ensemble. A subset of the sites should maintain a constant concentrations, and a subset should maintain constant chemical potential. Thus, for the subset of sites where the concentration should be fixed, swap moves are proposed and for the subset that should have constant chemical potentia, flip moves are probosed (e.g. switching symbol type on a site)

Parameters

- **atoms** – Atoms object used in the simulation
- **swap_indices** – List of indices that constitue the sub-lattice that should have fixed concentration

- **flip_indices** – List of indices that constitute the sub-lattice that should have fixed chemical potential
- **flip_symbols** – List of possible symbols that can be substituted on the lattice with fixed chemical potential.
- **flip_prob** – Probability of returning a flip move. The probability of returning a swap move is then 1 - flip_prob.

get_single_trial_move() → Sequence[SystemChange]

Produce a single trial move. Return a swap move with probability

initialize(*atoms: Atoms*) → None

Initialize the trial move generator

on_move_accepted(*changes: Sequence[SystemChange]*)

Callback triggered when a move have been accepted.

on_move_rejected(*changes: Sequence[SystemChange]*) → None

Callback triggered when a move have been accepted.

property weights: Tuple[float]

The probability weights for each generator

class clease.montecarlo.trial_move_generator.**RandomFlipWithinBasis**(*symbols: Sequence[Sequence[str]]*, *atoms: Atoms*, *indices: Sequence[Sequence[int]] | None = None*, ***kwargs*)

Produce trial moves consisting of flips within each basis. Each basis is defined by a list of indices.

Parameters

- **symbols** – Sequence allowed symbols in each basis
- **atoms** – Atoms object to be used in the simulation for which the trial moves are produced
- **indices** – Sequence of sets of indices where each set specify the indices of a basis. Note len(symbols) == len(indices)

Example:

Create a generator for a rocksalt structure with two basis

```
>>> from ase.build import bulk
>>> from clease.montecarlo import RandomFlipWithinBasis
>>> atoms = bulk("LiO", crystalstructure="rocksalt", a=3.9)*(3, 3, 3)
>>> basis1 = [a.index for a in atoms if a.symbol == "Li"]
>>> basis2 = [a.index for a in atoms if a.symbol == "O"]
>>> generator = RandomFlipWithinBasis([["Li", "X"], ["O", "V"]], atoms, [basis1,␣
↪basis2])
```

get_single_trial_move() → Sequence[SystemChange]

Produce a trial move by choosing a random flipper

3.6.8 Getting Data From Database

CLEASE retrieves data from ASE databases, via the generic *DataManager* class. For convenience, concrete implementations of the *DataManager* is provided for the most common applications.

class clease.data_manager.**CorrFuncEnergyDataManager**(*db_name: str*, *tab_name: str*, *cf_names: List[str] | None = None*, *order: int = 1*)

CorrFuncFinalEnergyDataManager is a convenience class provided to handle the standard case where the features are correlation functions and the target is the DFT energy per atom

Parameters

- **db_name** – Name of the database being passed
- **cf_names** – List with the correlation function names to extract
- **tab_name** – Name of the table where the correlation functions are stored
- **order** – Order of the correlation function. Default 1.

get_data(*select_cond: List[tuple]*) → Tuple[ndarray, ndarray]

Return X and y, where X is the design matrix containing correlation functions and y is the DFT energy per atom.

Parameters

select_cond – List with select conditions for the database (e.g. [('converged', '=', True)])

class clease.data_manager.**CorrFuncVolumeDataManager**(*db_name: str*, *tab_name: str*, *cf_names: List[str] | None = None*, *order: int = 1*)

CorrFuncVolumeDataManager is a convenience class provided to handle the standard case where the features are correlation functions and the target is the volume of the relaxed cell

Parameters

- **db_name** – Name of the database being passed
- **tab_name** – Name of the table where the correlation functions are stored
- **cf_names** – List with the correlation function names to extract. If None, all correlation functions in the database will be extracted.
- **order** – Order of the correlation functions. Default 1.

get_data(*select_cond: List[tuple]*) → Tuple[ndarray, ndarray]

Return X and y, where X is the design matrix containing correlation functions and y is the volume per atom.

Parameters:

select_cond: list

List with select conditions for the database (e.g. [('converged', '=', True)])

class clease.data_manager.**CorrelationFunctionGetter**(*db_name: str*, *tab_name: str*, *cf_names: List[str] | None = None*, *order: int = 1*)

CorrelationFunctionGetter is a class that extracts the correlation functions from an AtomsRow object

Parameters

- **db_name** – Name of the database
- **tab_name** – Name of the external table where the correlation functions are stored
- **cf_names** – List with the names of the correlation functions. If None, all correlation functions in the database will be extracted
- **order** – Order of the correlation function. Default is 1.

get_property(*ids: Sequence[int]*) → ndarray

Extracts the design matrix associated with the database IDs. The first row in the matrix corresponds to the first item in ids, the second row corresponds to the second item in ids etc. If cf_names was None, all correlation functions in the database will be extracted. cf_names will be updated such that it reflects the names of the correlation functions that were extracted.

Parameters

ids – Database IDs of initial structures

property names

Return a name of each column

class clease.data_manager.**CorrelationFunctionGetterVolDepECI**(*db_name: str*, *tab_name: str*, *cf_names: List[str]*, *order: int | None = 0*, *properties: Tuple[str] = ('energy', 'pressure')*, *cf_order: int = 1*)

Extracts correlation functions, multiplied with a power of the volume per atom. The feature names are named according to the correlation function names in the database, but a suffix of _Vd is appended. d is an integer inticading the power. Thus, if the name is for example c2_d0000_0_00_V2, it means that the column contains the correlation function c2_d0000_0_00, multiplied by V^2, where V is the volume per atom.

Parameters

- **db_name** – Name of the database
- **tab_name** – Name of the table where correlation functions are stored
- **cf_names** – Name of the correlation functions that should be extracted
- **order** – Each ECI will be a polynomial in the volume of the passed order (default: 0)
- **properties** – List of properties that should be used in fitting. Can be energy, pressure, bulk_mod. (default: ['energy', 'pressure']). The pressure is always assumed to be zero (e.g. the energies passed are for relaxed structures.). All entries in the database are expected to have an energy. The remaining properties (e.g. bulk_mod) is not required for all structures. In class will pick up and the material property for the structures where it is present.
- **cf_order** – The energy is expanded up and (inluding) this order in the correlation function. Default is 1.

build(*ids: List[int]*) → ndarray

Construct the design matrix and the target value required to fit a cluster expansion model to all material properties in self.properties.

Parameters

ids – List of ids to take into account

get_data(*select_cond: List[tuple]*) → Tuple[ndarray, ndarray]

Return the design matrix and the target values for the entries corresponding to select_cond.

Parameters

select_cond – ASE select condition. The design matrix and the target vector will be extracted for rows matching the passed condition.

groups() → List[int]

Return the group of each rows.

class clease.data_manager.**DataManager**(*db_name: str*)

DataManager is a class for extracting data from CLEASE databases to be used to fit ECIs

Parameters

db_name – Name of the database

get_cols(*names: List[str]*) → ndarray

Get all columns corresponding to the names

Pram names

List of names (e.g. [‘c0’, ‘c1_1’])

abstract get_data(*select_cond: List[tuple]*) → Tuple[ndarray, ndarray]

Return the design matrix X and the target data y

get_matching_names(*pattern: str*) → List[str]

Get names that matches pattern

Parameters

pattern – Pattern which the string should contain.

Example:

If the names are [‘abc’, ‘def’, ‘gbcr’] and the passed pattern is ‘bc’, then [‘abc’, ‘gbcr’] will be returned

groups() → List[int]

Returns the group of each item in the X matrix. In the top-level DataManager it is assumed that each row in the X matrix constitutes its own group. But this method may be overrided in child classes.

to_csv(*fname: str*)

Export the dataset used to fit a model y = Xc where y is typically the DFT energy per atom and c is the unknown ECIs. This function exports the data to a csv file with the following format

ECIname_1, ECIname_2, . . . , ECIname_n, E_DFT 0.1, 0.4, . . . , -0.6, -2.0 0.3, 0.2, . . . , -0.9, -2.3

thus each row in the file contains the correlation function values and the corresponding DFT energy value.

Parameters

fname – Filename to write to. Typically this should end with .csv

class clease.data_manager.**FinalStructPropertyGetter**(*db_name: str*, *prop: str*)

FinalStructPropertyGetter is a class that returns the user defined property value corresponding to the passed AtomsRow object. The user defined property should be located in the *final* atoms row.

Parameters

db_name – Name of the database

get_property(*ids: Sequence[int]*) → ndarray

Extract the property of the ids passed.

Parameters

ids – Database ids of initial structures

property name

Return the name of the target property

exception clease.data_manager.**InconsistentDataError**

Data is inconsistent

clease.data_manager.**make_corr_func_data_manager**(*prop: str*, *db_name: str*, *tab_name: str*, *cf_names: Sequence[str]*, ***kwargs*) → *DataManager*

Helper function for creating a correlation function data manager

3.6.9 Geometry Tools

Module for tools pertaining to geometry of atoms and cells.

clease.geometry.**cell_wall_distances**(*cell: ndarray*) → ndarray

Get the distances from each cell wall to the opposite cell wall of the cell. Returns the distances in the order of the distances between the (b, c), (a, c) and (a, b) planes, such that the shorest vector corresponds to being limited by the a, b or c vector, respectively.

Parameters

cell (*np.ndarray*) – A (3 x 3) matrix which defines the cell parameters. Raises a ValueError if the cell shape is wrong.

Returns

The distance to each of the three cell walls.

Return type

np.ndarray

clease.geometry.**max_sphere_dia_in_cell**(*cell: ndarray*) → float

Find the diameter of the largest possible sphere which can be placed inside a cell.

For example, how large of a Death Star could be built inside a given atoms object?

Parameters

cell (*np.ndarray*) – A (3 x 3) matrix which defines the cell parameters. Raises a ValueError if the cell shape is wrong.

Returns

The diameter of the largest sphere which can fit within the cell.

Return type

float

clease.geometry.**supercell_which_contains_sphere**(*atoms: Atoms*, *diameter: float*) → Atoms

Find the smallest supercell of an atoms object which can contain a sphere, using only repetitions of (nx, ny, nz) (i.e. a diagonal P matrix).

The number of repetitions is stored in the info dictionary of the supercell under the "repeats" keyword, i.e. sc.info["repeats"] is the number of times the supercell was repeated.

Parameters

- **atoms** (*Atoms*) – The atoms object to be enlarged.
- **diameter** (*float*) – The diameter of the sphere which should be contained within the supercell.

Returns

The supercell which can contain a sphere of the given diameter.

Return type

Atoms

3.6.10 Post Process Plotting

clease.plot_post_process.**plot_convex_hull**(*evaluate: Evaluate*, *interactive: bool = False*) → Figure

Plot the convex hull of an evaluate object.

Parameters

- **evaluate** (*Evaluate*) – The Evaluate object to draw the convex hull from.
- **interactive** (*bool, optional*) – Plot as an interactive figure?. Defaults to False.

clease.plot_post_process.**plot_cv**(*evaluate:* Evaluate, *plot_args: dict | None = None*) → Figure

Figure object of CV values according to alpha values If the plot_args dictionary contains keys, return figure object to relate plot_args keys

Parameters

- **evaluate** – Use the evaluate object to define the plot argument.
- **plot_args** – plot_args dictionary contains:
  - "xlabel": x-axis label
  - "ylabel": y-axis label
  - "title": title of plot

Returns

Figure instance of plot

clease.plot_post_process.**plot_eci**(*evaluate:* Evaluate, *plot_args: dict | None = None*, *ignore_sizes=()*, *interactive: bool = False*) → Figure

Figure object of ECI value according to cluster diameter If the plot_args dictionary contains keys, return figure object to relate plot_args keys

Parameters

- **evaluate** – Use the evaluate object to define the plot argument.
- **plot_args** – plot_args dictionary contains:
  - "xlabel": x-axis label
  - "ylabel": y-axis label
  - "title": title of plot
  - "sizes": list of int to include n-body cluster in plot
- **ignore_sizes** – list of ints Sizes listed in this list will not be plotted. E.g. `ignore_sizes=[0]` will exclude the 0-body cluster. Default is to not ignore any clusters.
- **interactive** – Add interactive elements to the plot.

Returns

Figure instance of plot

clease.plot_post_process.**plot_fit**(*evaluate:* Evaluate, *plot_args: dict | None = None*, *interactive: bool = False*) → Figure

Figure object calculated (DFT) and predicted energies. If the plot_args dictionary contains keys, return figure object to relate plot_args keys

Parameters

- **evaluate** – Use the evaluate object to define the plot argument.
- **plot_args** – plot_args dictionary contains:
  - "xlabel": x-axis label
  - "ylabel": y-axis label
  - "title": title of plot
- **interactive** – Add interactive elements to the plot.

Returns

Figure instance of plot

clease.plot_post_process.**plot_fit_residual**(*evaluate:* Evaluate, *plot_args: dict | None = None*, *interactive: bool = False*) → Figure

Figure object subtracted (DFT) and predicted energies. If the plot_args dictionary contains keys, return figure object to relate plot_args keys

Parameters

- **evaluate** – Use the evaluate object to define the plot argument.
- **plot_args** – plot_args dictionary contains:
  - "xlabel": x-axis label
  - "ylabel": y-axis label
  - "title": title of plot
- **interactive** – Add interactive elements to the plot.

Returns

Figure instance of plot

3.7 Parallelization

Clease has an **expermental** support for OpenMP when calculating the correlation functions. It currently only kicks in when updating correlation functions, e.g. during Monte Carlo simulations.

Note: This feature is experimental, and only tested on Unix systems.

Also, the API may be subject to change.

Note: Initial testing hasn't shown much improvement past 2-4 threads, as the load balancing is very uneven, since parallelization is done across each ECI value.

Models with few ECI values will therefore also gain less from parallelization. Remember to do your own testing.

3.7.1 Installation

In order to use the OpenMP feature, CLEASE needs to be compiled with the OpenMP flag enabled in your compiler. For most systems, the compiler will be `gcc`, where the flag is `-fopenmp`. This flag can be enabled by setting the `CLEASE_OMP` environment variable when installing with pip.

```
$ CLEASE_OMP=-fopenmp pip install clease --no-cache-dir --no-binary=clease
```

Note: This only works when installing CLEASE via pip.

3.7.2 Testing the install

Once you have installed, check your CLEASE installation with the command

```
$ clease info
```

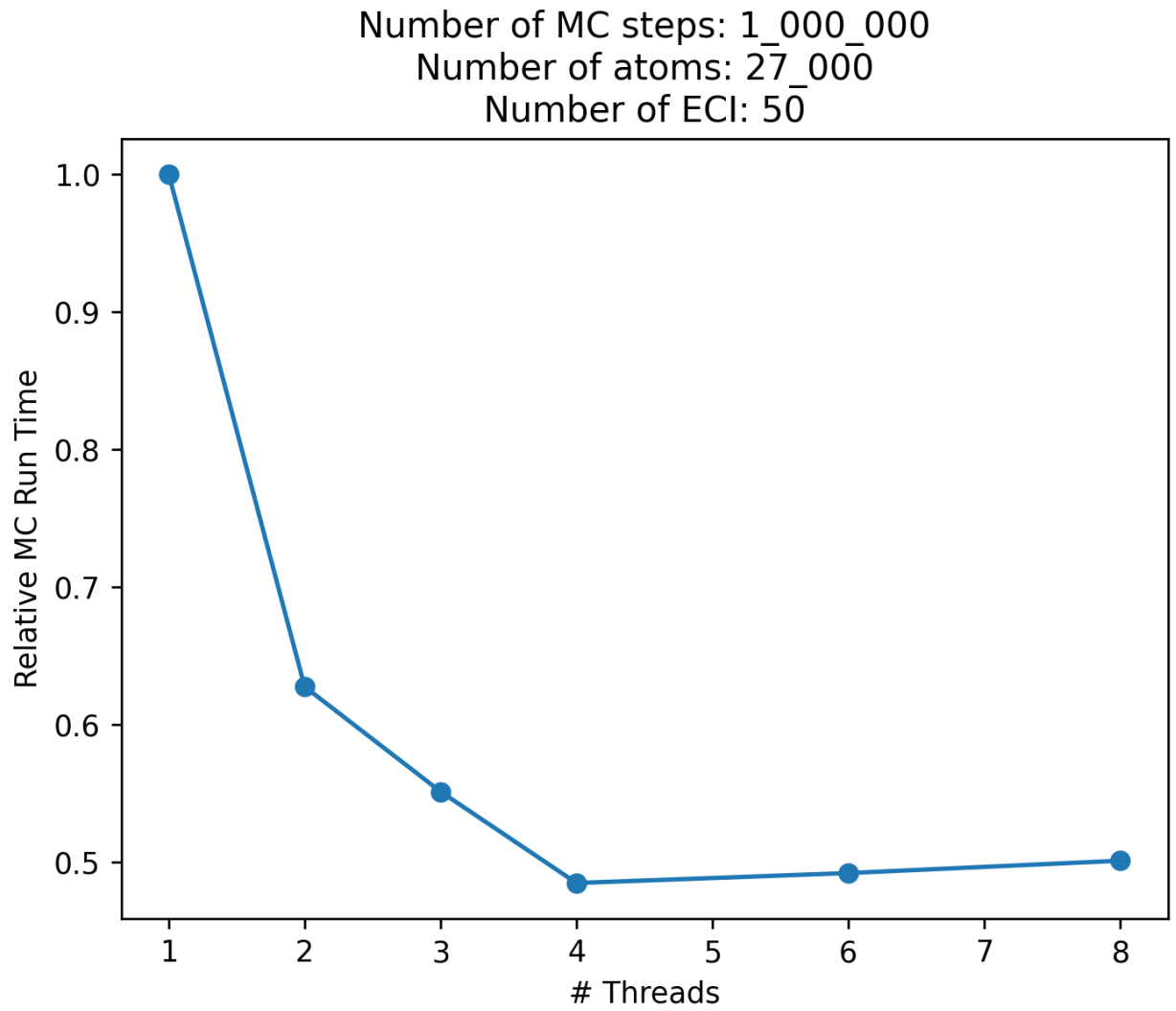
If the line `C++ OpenMP` says `True`, your install should be configured correctly.

3.7.3 Running in parallel

If the above command returns `True`, you can run with thread parallelization. Simply run your Monte Carlo simulations as normal, but adjust `num_threads` parameter in your `Clease` class, e.g. via the `attach_calculator` helper function.

```
>>> atoms = attach_calculator(settings, atoms, eci, num_threads=2) # Use 2 threads
>>> mc = Montecarlo(atoms, 5000)
>>> mc.run(1_000)
```

Below are some timings for an AuCu cell normalized to the runtime for 1 thread (lower is better):



3.8 Benchmarking

The CLEASE test suite contains a benchmarking sub-suite as well, which can be useful for testing new code. If the adjusted code is not tested in the benchmarking suite yet, and is performance sensitive, please remember to add a new benchmark test.

Note: Running the benchmarks requires the extra requirements the `test` installation, e.g. from a `pip install .[test]` install.

The benchmarks must be enabled via pytest, and is run via the pytest-benchmark extension. Running a normal `pytest` command will simply skip the benchmarking tests, as they are assumed to be more expensive to run. Therefore, the recommended way to execute the benchmarks, is to tell pytest to only execute tests marked for benchmarking, e.g. from the CLEASE root directory:

```
pytest --fig --benchmark-only --benchmark-autosave tests/
```

The `fig` command allows a test which constructs a plots to output figures. Alternatively, *tox* can be used to execute the benchmarks, which is roughly equivalent to the above command:

```
tox -e benchmark
```

The `--benchmark-autosave` option saves a benchmark run to the `.benchmarks/` folder in the root directory. Two runs from the benchmark can be compared, for example doing

```
pytest-benchmark compare --histogram benchmark 0001 0002
```

would generate a histogram file called `benchmark.svg`. The compare also generates a text output, and example is shown `here`. You can omit the `--histogram benchmark` flag to just get the text comparison. The run ID's are granted automatically, so in this example the first and second run were automatically named `0001` and `0002`. Omit the numbers to simply compare every previous benchmark run. For more details on how to compare benchmarks, please see the pytest-benchmark docs. The following is an example of what this histogram can look like:

Note: Running benchmarks is highly sensitive to the machine, and to other processes running on the machine. So to ensure a fair comparison, always compare results from the same machine under as similar loads as possible.

3.9 Publications Using CLEASE

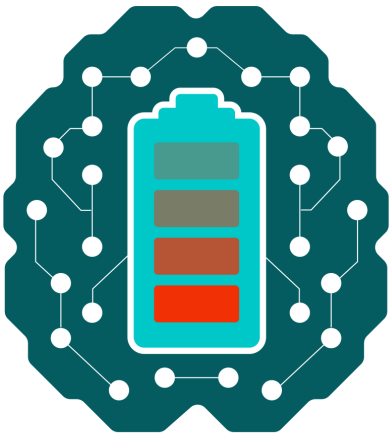
If you found CLEASE to be a useful tool and have used it in a publication, we invite you to add your work to the list below. This makes it easy for other CLEASE users to see your work as well as showing examples of research topics where cluster expansion techniques are useful. The list is sorted by publication date, where the latest publications appear on top. In order to add a contribution to the list, you can open a merge request to the CLEASE repository.

- **Chable, J., Baur, C., Chang, J.H., Wenzel, S., García-Lastra, J.M. and Vegge, T., 2019.**
  From Trigonal to Cubic LiVO2: A High-Energy Phase Transition Towards Disordered Rock Salt Materials. *The Journal of Physical Chemistry C.*
- **Tranås, R.A., 2019.**
  Atomistic simulations of thermodynamics and dissolution of iron-silicon phases in iron encasements *(Master's thesis, NTNU).*

- **Chang, J.H., Kleiven, D., Melander, M., Akola, J., Garcia-Lastra, J.M. and Vegge, T., 2019.** CLEASE: A versatile and user-friendly implementation of Cluster Expansion method. *Journal of Physics: Condensed Matter*, 31(32), p.325901.
- **Kleiven, D., Ødegård, O.L., Laasonen, K. and Akola, J., 2019.** Atomistic simulations of early stage clusters in AlMg alloys. *Acta Materialia*, 166, pp.484-492.
- **Tygesen, A.S., Chang, J.H., Vegge, T., García-Lastra, J.M., 2020.** Computational framework for a systematic investigation of anionic redox process in Li-rich compounds. *npj Computational Materials*, 6(1), [65].

3.10 Partners and Support

Development of CLEASE was supported by LiRichFCC (H2020, #766581) and BIG-MAP (H2020, #957189).



BIG MAP

PYTHON MODULE INDEX

C

INDEX

D

E

F

G

M

N

O

P

R

S

T

U

V

W