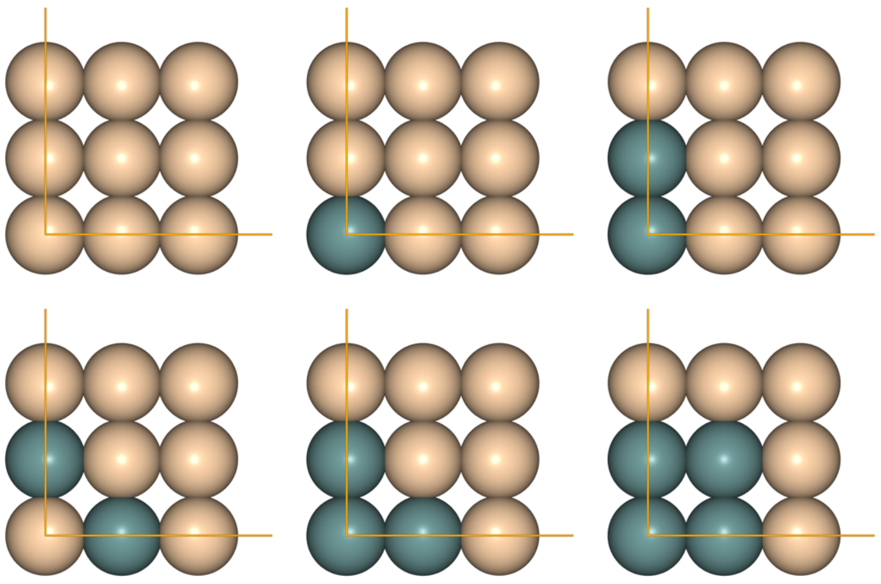
Hands-on Workshop
Density-Functional Theory and Beyond:
Frontiers of Advanced Electronic Structure
and Molecular Dynamics Methods
Beijing, China
July 30 – August 10, 2018



Tutorial V: Cluster Expansion with `CELL`
Manuscript for Exercise Problems

Prepared by Santiago Rigamonti, Maria Troppenz, and Martin Kuban
Humboldt-Universität zu Berlin, SOL-group Prof. Draxl
Beijing, August 6th, 2018

Introduction

Achieving a realistic physical description of alloy materials is a very challenging task. One of the main difficulties resides on the dependence of the materials properties on the specific atomic configuration of the different species in the crystal lattice. Taking the example of a Si_xGe_{1-x} alloy, its properties (energy of formation, band-gap, specific heat, etc.) will depend on the actual positions of Si atoms which replace, or substitute, Ge atoms in the lattice, and also on the concentration x of Si atoms. Moreover, at finite temperatures, the properties will be determined by an ensemble of atomic configurations compatible with the macroscopic conditions (e.g. temperature, pressure, etc.). Therefore, an *ab initio* physical description of this material would require performing first-principle density-functional theory (DFT) simulations for a large portion of the possible configurations of the system. If we consider the number of configurations for even moderate system sizes (e.g. 30 atoms), we soon realize the presence of a combinatorial explosion which precludes such an approach. Thus, one must resort to alternative descriptions, as for instance those provided by mean-field approaches (e.g. CPA). These have the advantage of being simple and efficient, but their validity is limited by constraining assumptions, as for instance the lack of structural relaxations and the absence of short-range order (i.e. disorder is assumed to be uncorrelated).

An alternative to mean-field approaches is provided by the cluster expansion method (CE) [1]. This method allows for predicting the configuration-dependent properties of materials in an accurate and efficient manner. In contrast, the CE fully accounts for the effects of structural relaxations and short-range order on physical properties, which have been shown to play an essential role in, for instance, the determination of the electronic properties of materials for energy harvesting applications [2].

In this tutorial, you will learn the basic concepts of the cluster expansion method, and you will learn the basic usage of **`CELL`** (also known as **`clusterX`**) [2], a python package for cluster expansion with a focus on complex alloys. In particular, you will learn to set up and perform a cluster expansion for bulk and surface systems, and how to optimize a cluster expansion model for accurate and efficient predictions of configuration-dependent properties.

Cluster expansion in a nutshell

Consider a crystal defined by a two-dimensional rectangular lattice as shown in Figure 1, left panel. The periodicity of the pristine crystal is indicated by the primitive lattice vectors. Now, suppose that a number of atoms of type A are substituted by atoms of type B, as shown in the right panel. In this case the translational invariance of the primitive lattice is broken and we have to resort to a supercell description. In the following analysis, we will consider supercells of size 5x5 (referred to the primitive lattice) as in Figure 1, although arbitrary supercell sizes and shapes could be considered as well.

Considering that on every site of the supercell there are two possible atomic occupations, either A or B, a simple combinatorial analysis tells us that the number of possible configurations or *decorations* of the lattice is 2^N, with N the number of sites in the supercell. In this case N=25, and $2^N \approx 3.3 \times 10^7$. This estimate gives an upper limit, since it does not account for the symmetries of the crystal, which would reduce this value. However, the number of symmetries of any given configuration cannot be higher than that of the primitive lattice. Thus, this reduction by symmetry considerations will not avoid the combinatorial explosion presented by the exponential behavior 2^N. In view of this result, a direct *ab-initio* treatment of the thermodynamical properties of this systems is obviously out of question.

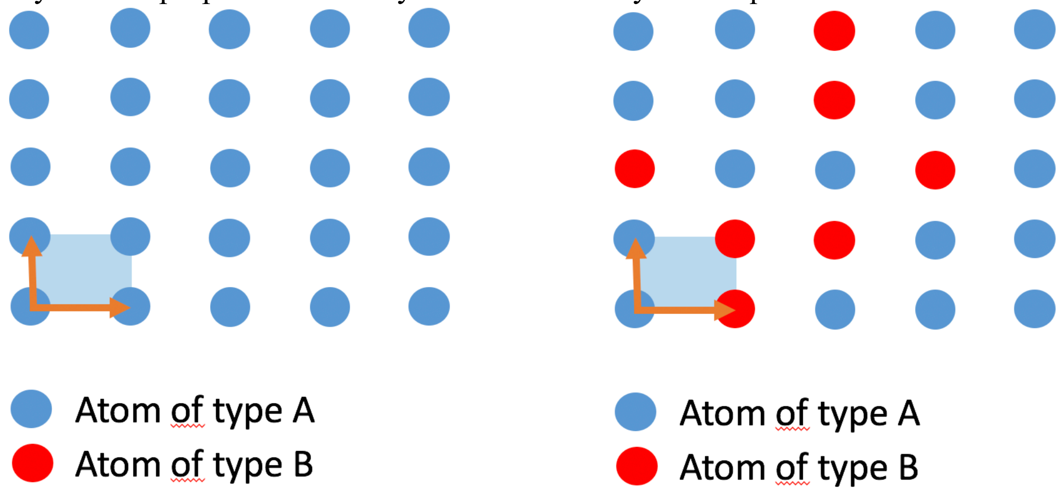


Figure 1: Left: Pristine two-dimensional rectangular lattice. The parent lattice is indicated with a shaded area and arrows indicate primitive lattice vectors. Right: partially substituted lattice.

Now, we will show how to build a simple model that will allow us to predict the energy of arbitrary configurations like that on the right panel of Figure 1 in a computationally very efficient manner. The construction of such a model will require only a few first principles simulations for some selected structures. A possible selection of such structures is depicted in panels *a)* to *f)* of Figure 2. Now, suppose that we have obtained the total energies E_a, E_b, …, E_f through accurate *ab initio* simulations (including atomic relaxations). Then, proceed with the following steps:

1. Obtain parameter J_0 through: $E_a = J_0$

2. Obtain parameter J_1 through: $E_b = J_0 + J_1$, i.e. we equate the calculated energy E_b to the sum of J_0 (which represents the energy of the pristine lattice) plus an additional *interaction* J_1 which represents the change in energy due to replacing species A by B.

3. Obtain parameter J_2 through: $E_c = J_0 + 2J_1 + J_2$, thus, the interaction J_2 represents the change in energy due to having two neighbor atoms of type B along the x-axis.

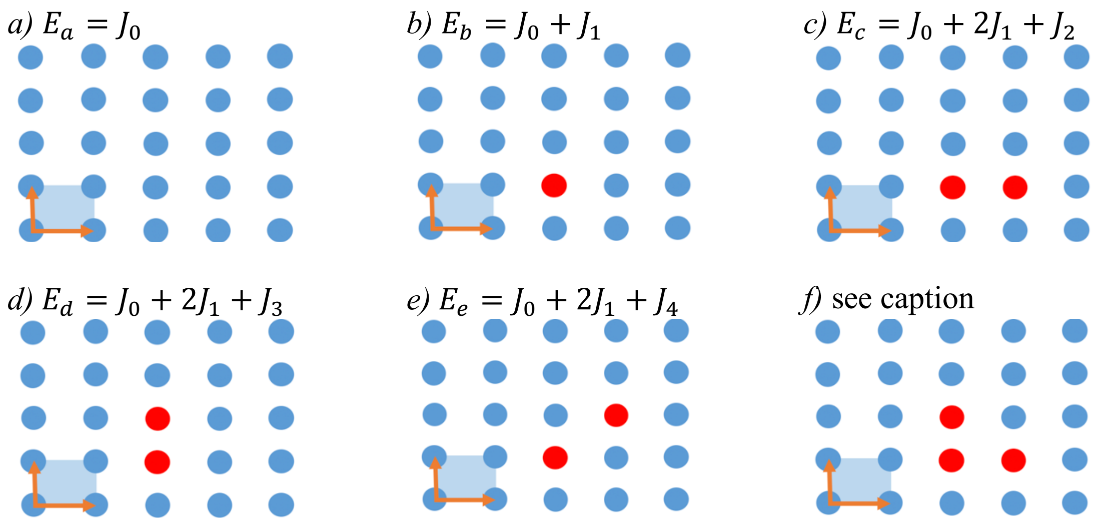
4. Obtain parameter J_3 through: $E_d = J_0 + 2J_1 + J_3$, thus, the interaction J_3 represents the change in energy due to having two neighbor atoms of type B along the y-axis.

5. Obtain parameter J_4 through: $E_e = J_0 + 2J_1 + J_4$, thus, the interaction J_4 represents the change in energy due to having two neighbor atoms of type B along the diagonal.

6. Obtain parameter J_5 through: $E_f = J_0 + 3J_1 + J_2 + J_3 + J_4 + J_5$, thus, the interaction J_4 represents the change in energy due to having three neighbor atoms of type B.

Finally, once the parameters J_α, $\alpha = 0, \ldots, 5$ have been obtained, we can make predictions of the energy of arbitrary crystal decorations. This can be done just by counting how many interactions J_α of each kind are present in the structure and summing them up. For instance, you may easily verify that, using the interactions obtained above, our prediction for the energy of the structure on the right panel of Figure 1 is:

$$E = J_0 + 7J_1 + J_2 + 2J_3 + 4J_4 + J_5 \tag{1}$$



a) $E_a = J_0$
b) $E_b = J_0 + J_1$
c) $E_c = J_0 + 2J_1 + J_2$
d) $E_d = J_0 + 2J_1 + J_3$
e) $E_e = J_0 + 2J_1 + J_4$
f) see caption

Figure 2: Some selected structures to perform a CE *by hand*. For panel *f)* $E_f = J_0 + 3J_1 + J_2 + J_3 + J_4 + J_5$.

The structural patterns in Figure 2 represent *clusters*. Clusters are the sets of crystal sites corresponding to those patterns (there is a more general definition in the context of multicomponent systems -Sanchez2008-, but for the present example the simpler definition given here is correct).

Having introduced the clusters, the expression for the energy in Eq.(1) can be conveniently generalized as by using the cluster correlations $X_{\sigma\alpha}$. These can be defined, for the case of a binary alloy in this example, as:

$$X_{\sigma\alpha} = \frac{1}{m_\alpha} \sum_{\beta \equiv \alpha} \prod_{i \in \beta} \sigma_i \tag{2}$$

Here, m_α denotes the multiplicity of cluster α, i.e. the number of clusters (β in the sum above) which are symmetrically equivalent to cluster α. σ is a vector which represents the atomic configuration of the crystal: the component $\sigma_i = 1$ if the crystal site i is occupied with species B and 0 otherwise. Defined in this way, the correlation of cluster α in the structure with configuration σ, $X_{\sigma\alpha}$, quantifies the relative frequency of the pattern α (and its accompanying interaction J_α), in the structure σ. With this definition, you can verify that the expression of Eq.(1) for the energy can be written as:

$$E_\sigma = \sum_\alpha m_\alpha J_\alpha X_{\sigma\alpha} \tag{3}$$

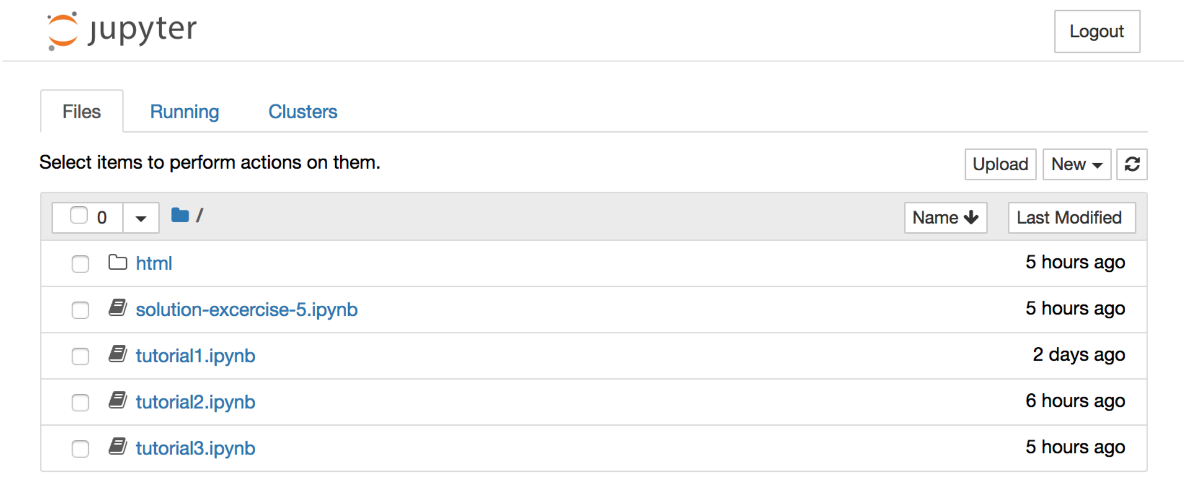
In this way, we obtain an expansion of the energy in terms of clusters, with J_α the expansion coefficients. These coefficients are called *effective cluster interctions* (ECI).

Tutorials

The tutorials and exercises are formatted to run the code **CELL** in jupyter notebooks. For this, you must copy the tutorial folder and execute the following commands ($> indicates the command prompt):

```
$> source activate py36
$> cp -r /public/hands-on-2018-tutorials/tutorial-ce ~
$> cd tutorial-ce
$> jupyter-nbextension enable nglview --py -user
$> jupyter notebook
```

This will open the web browser with a screen as shown in Fig. 3.

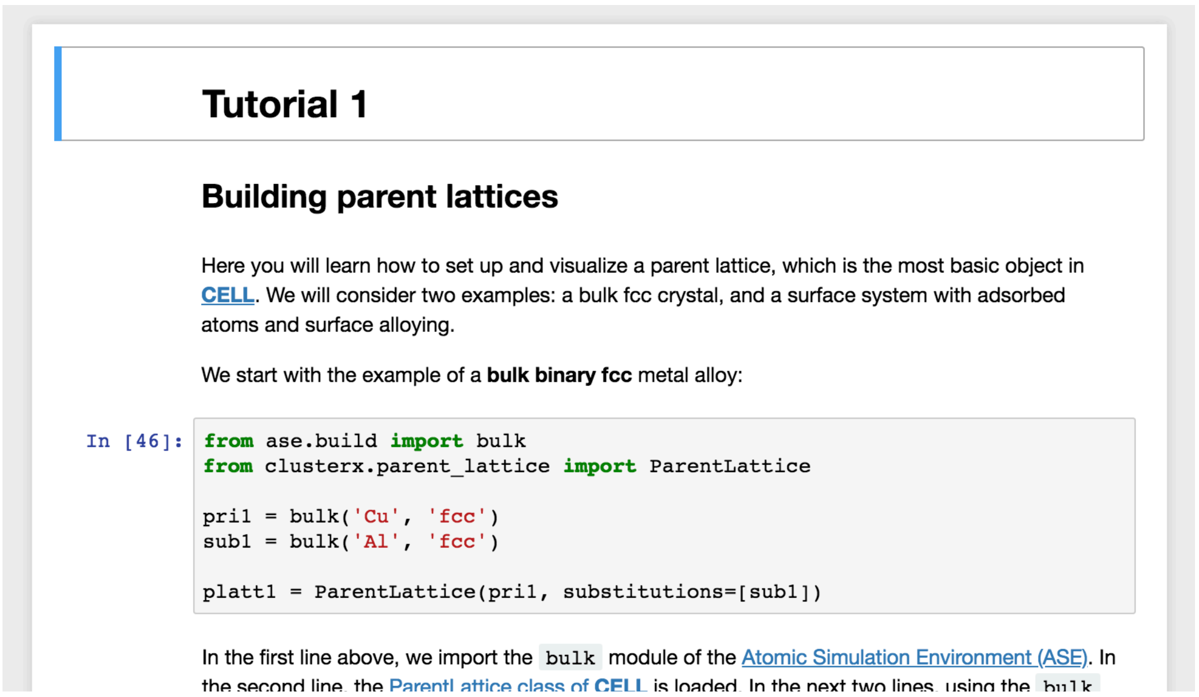


Select items to perform actions on them.

```
html                            5 hours ago
solution-excercise-5.ipynb      5 hours ago
tutorial1.ipynb                 2 days ago
tutorial2.ipynb                 6 hours ago
tutorial3.ipynb                 5 hours ago
```

Figure 3: jupyter document-tree view.

Then, click on tutorial1.ipynb. This will open another window with the actual interactive tutorial which should look like Fig. 4.



Tutorial 1

Building parent lattices

Here you will learn how to set up and visualize a parent lattice, which is the most basic object in CELL. We will consider two examples: a bulk fcc crystal, and a surface system with adsorbed atoms and surface alloying.

We start with the example of a **bulk binary fcc** metal alloy:

```
In [46]: from ase.build import bulk
         from clusterx.parent_lattice import ParentLattice

         pri1 = bulk('Cu', 'fcc')
         sub1 = bulk('Al', 'fcc')

         platt1 = ParentLattice(pri1, substitutions=[sub1])
```

In the first line above, we import the `bulk` module of the Atomic Simulation Environment (ASE). In the second line, the ParentLattice class of CELL is loaded. In the next two lines, using the `bulk`

Figure 4: Interactive jupyter browser window.

Jupyter is an interactive programming and development tool. It provides *cells*, which can contain e.g. *code* or *formatted text* in your web browser. Within these cells, you can *edit* and *execute* the code (by typing the key combination Shift+Enter), allowing you to immediately see the results of your calculations and store variables for later use.

Now we will go through tutorials 1 to 3 (which you can run by opening the corresponding files tutorial1.ipynb, tutorial2.ipynb, and tutorial3.ipynb). Tutorial 1 explains how to build parent lattices, supercells and structures sets. Tutorial 2, is dedicated to the construction of clusters pools, visualization of cluster orbits and the calculation of cluster correlations. Finally, Tutorial 3 is dedicated to model optimization.

Tutorial 1

Building parent lattices

Here you will learn how to set up and visualize a parent lattice, which is the most basic object in **CELL**. We will consider two examples: a bulk fcc crystal, and a surface system with adsorbed atoms and surface alloying.

We start with the example of a **bulk binary fcc** metal alloy:

```
from ase.build import bulk
from clusterx.parent_lattice import ParentLattice

pri1 = bulk('Cu', 'fcc')
sub1 = bulk('Al', 'fcc')

platt1 = ParentLattice(pri1, substitutions=[sub1])
```

In the first line above, we import the bulk module of the Atomic Simulation Environment (ASE). In the second line, the ParentLattice class of CELL is loaded. In the next two lines, using the bulk function, we define the structures for the pristine non-substituted Cu lattice pri1 and the fully substituted Al (fcc) lattice sub1. These two structures are then used to initialize the ParentLattice object (which we call platt1) in the last line.

Next, we would like to visualize the just created parent lattice. To this end, we use the juview function of the visualization module of CELL:

```
from clusterx.visualization import juview
juview(platt1)
```

The execution of this cell will produce the image of Fig. 5.

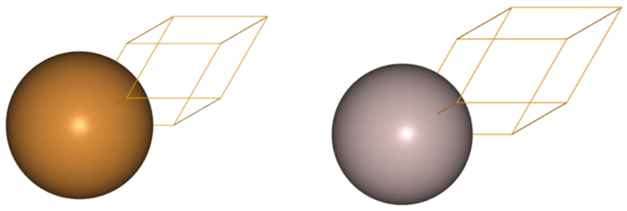


Figure 5: fcc parent lattice for a binary compound.

The left panel corresponds to the pristine non-substituted Cu fcc crystal, while the right panel represents the fully Al-substituted crystal. In general, the line of code juview(parent_lattice), will generate as many additional figures as substituents species are present in the parent lattice, as you will see for the next example of a surface system.

Now, we will set up the parent lattice for a **surface system**. It consists on a fcc(111) Al surface, with possible Na substitution on the uppermost Al layer and adsorption of oxygen atoms in "on-top" configuration. In order to build the parent lattice for such a system, we execute the following code:

```
from ase.build import fcc111, add_adsorbate

pri2 = fcc111('Al', size=(1,1,3))
add_adsorbate(pri2,'X',1.7,'ontop')
pri2.center(vacuum=10.0, axis=2)
```

In the code above, first we load some builder utilities from ASE (fcc111 and add_adsorbate). In the next three lines, we i) create an (111)-terminated fcc Al slab with three atomic layers; ii) add a vacancy (symbol X) site with "on top" configuration, and iii) add vacuum on the sides of the slab along the z -direction. In this way we have defined the pristine structure pri2. Now we would like to set up the substitutions: Na on the top-most Al layer and oxygen on the "on-top" vacancy sites. To proceed, we first need some information from the pristine structure, as shown below:

```
for i, (symbol, z_coord) in
enumerate(zip(pri2.get_chemical_symbols(),pri2.get_positions()[:,2])):
    print("atom index: ",i,"| Chemical symbol: ",symbol,"| z
coordinate: ",z_coord)
```

This will output something like:

```
atom index:  0 | Chemical symbol:  Al | z coordinate:  10.0
atom index:  1 | Chemical symbol:  Al | z coordinate:  12.338
atom index:  2 | Chemical symbol:  Al | z coordinate:  14.676
atom index:  3 | Chemical symbol:  X | z coordinate:  16.376
```

From this output, we see that the uppermost "Al" layer has atom index 2, and that the adsorbate layer has atom index 3. With this information, we initialize the parent lattice object in an alternative way, by telling which species can occupy every atom index: This is done with the site_symbols argument, which allows us to tell CELL which atomic species can occupy every atomic site:

```
platt2 =
ParentLattice(pri2,site_symbols=[['Al'],['Al'],['Al','Na'],['X','O']])

juview(platt2)
```

This will produce the plot of Fig. 6. In this way, we see that for atom indices 1 and 2 only ['Al'] is allowed, while atom index 2 admits the species in the array ['Al','Na'] and atom index 3 admits species ['X','O'], where 'X' denotes a vacancy. From left to right, the figures above denote:

pristine non-substituted lattice (vacancy sites indicated with white color), on-top vacancy site substituted by oxygen (red), and top-most Al layer substituted by Na (purple).

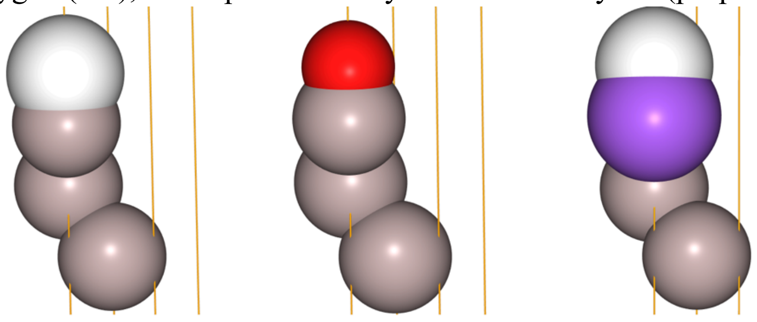


Figure 6: parent lattice for a surface with on-top adsorbates and top-layer alloying. White spheres represent vacant sites, red sphere represents oxygen, purple sphere represents Na.

Building structure sets

In order to generate *ab initio* data to be used as input to train a cluster expansion model, we need to perform calculations on super cells of the parent lattice. In CELL, super cells are represented by objects of the class SuperCell. We will take the example of the surface system above, and create a $4\times4\times1$ super cell object:

```
from clusterx.super_cell import SuperCell
import numpy as np

scell2 = SuperCell(platt2,np.diag([4,4,1]))
juview(scell2)
```

This will produce the following visualization of the created supercell

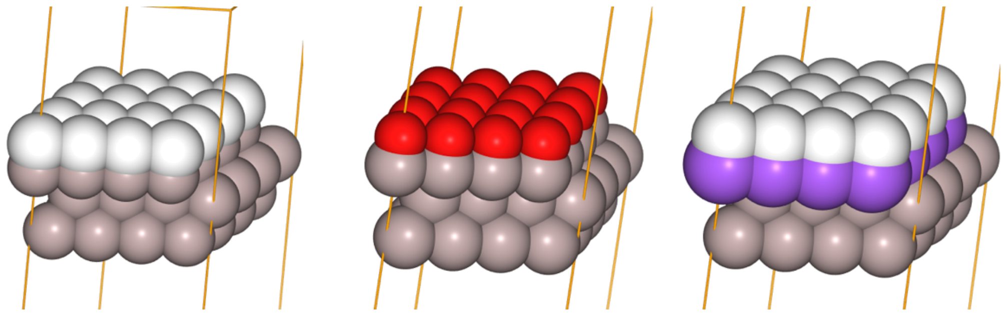


Figure 7: 4x4x1 supercell for a surface system. White spheres represent vacant sites, red spheres represents oxygen, purple sphere represents Na.

As you can see, a super cell looks very much like an enlarged parent lattice. Indeed, objects of the SuperCell class inherit from the ParentLattice class and share many properties.

Now, using the created super cell, we will generate a few random decorations of it at different concentrations. The generated structures will be collected in a StructuresSet object, that will be later used as training set for a cluster expansion. Before doing so, however, we need some information from the just created SuperCell object that we will need to define the concentrations of Na substituents and vacancies in the generation of random structures. This information is contained in the sites dictionary of the super cell, which we can access with:

```
print(scell2.get_idx_subs())
```

With the output:

```
{0: array([0, 8]), 1: array([13, 11]), 2: array([13])}
```

This tells us that the supercell has three types of sites, or sublattices, with indices 0, 1 and 2. Sites of type 0 contain species number 0 (vacancy), and can be substituted by species number 8 (oxygen) ; sites of type 1 contain species number 13 (Al) and can be substituted with species number 11 (Na); while sites of type 2 contain species number 13 (Al) and cannot be substituted.

In the code shown below, we first load the StructuresSet class and then create a structures-set object that we call sset2. Next, in three different for loops, by using the scell2's gen_random() method, we create i) two random structures with 4 on-top oxygen atoms, ii) two random structures with 4 Al → Na substitutions, and iii) 2 structures with 2 oxygen atoms and 4 Al → Na substitutions. The result is presented in Fig. 8.

```
from clusterx.structures_set import StructuresSet

sset2 = StructuresSet(platt2)

nstruc = 2

# i) Random structures with 4 on-top oxygen atoms
for i in range(nstruc):
    sset2.add_structure(scell2.gen_random({0:[4]}))

# ii) Random structures with 4 substituent Na atoms
for i in range(nstruc):
    sset2.add_structure(scell2.gen_random({1:[4]}))

# iii) Random structures with 2 on-top oxygen and 4 substituent Na atoms
for i in range(nstruc):
    sset2.add_structure(scell2.gen_random({0:[2],1:[4]}))

juview(sset2)
```

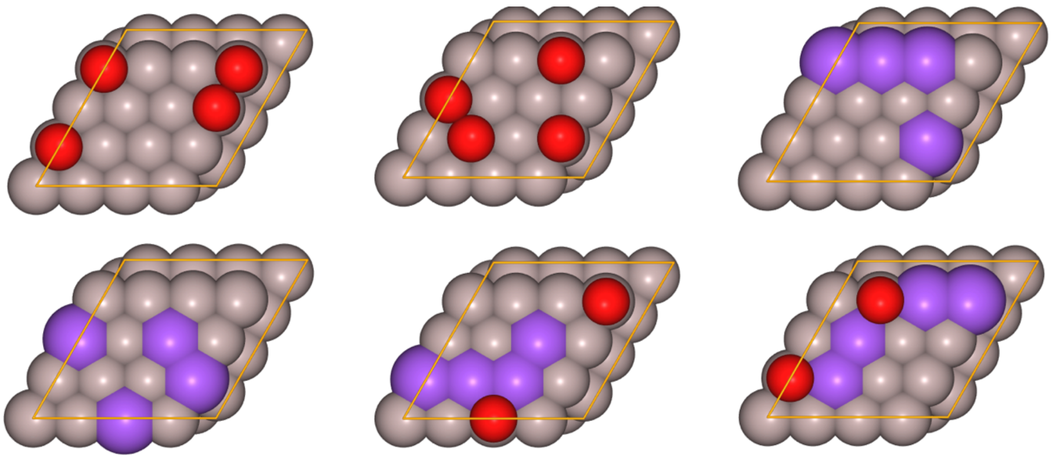


Figure 8: Random decorations of a 4x4x1 supercell. Red spheres represent oxygen atoms and purple spheres represent Na substitutions.

Exercise 1

Build the parent lattice for a two-dimensional square lattice of a binary (e.g. SiGe) material and create (and visualize) 6 random structures on a 5×55×5 super cell.

As help, you can use the following Atoms object to initialze the ParentLattice object:

```
from ase import Atoms

a=3.0
pri4 = Atoms(positions=[[0,0,0]],symbols=['Si'],
cell=[[a,0,0],[0,a,0],[0,0,2*a]],pbc=(1,1,0))
```

Exercise 2

Generate and visualize a few random structures for the fcc CuAl alloy of the first example on this tutorial. Do it so in a 3×3×3 super cell.

Tutorial 2

Building a pool of clusters

Now that you know how to build parent lattices and structures sets, the next task is to create a pool of clusters. The clusters pool defines the basis set for the expansion of configuration-dependent properties in terms of cluster functions. Although this basis is infinite (consisting of all symmetrically distinct atomic configurations of the substituent species), in practical applications one must cut off the basis. In order to fix ideas, we will start with a very simple model (corresponding to the solution of Exercise 1 of Tutorial 1) of a binary two-dimensional lattice. Afterwards, you will tackle the more complicated surface system shown in Tutorial 1 by solving Exercise 4 of this tutorial.

To start, we must define the parent lattice:

```
from ase import Atoms
from clusterx.parent_lattice import ParentLattice
from clusterx.visualization import juview

a=4.0
pri = Atoms(positions=[[0,0,0]],symbols=['Si'],
cell=[[a,0,0],[0,a,0],[0,0,2*a]],pbc=(1,1,0))

plat = ParentLattice(pri,site_symbols=[['Si','Ge']])

juview(plat)
```

With the result shown in Fig. 9.

Figure 9: Parent lattice of a two-dimensional square binary compound.

Now, we will create all possible clusters of up to three points and radius up $a\sqrt{2}$ (with a the lattice constant). This is done with the following piece of code:

```
from clusterx.clusters.clusters_pool import ClustersPool

r = 1.5
cpool =
ClustersPool(plat,npoints=[0,1,2,3,4],radii=[0,0,a*r,a*r,a*r])
```

In the first line, we load the ClustersPool class. Then, we create an object of this class (cpool). To initialize it, we use the parameters npoints and radii. npoints indicates the number of points of the clusters in the pool, and the parameter radii indicates the maximum radius corresponding to each of the number of points indicated in npoints. In this way, we have created a clusters pool containing the empty cluster, all the 1-point clusters (in this case there will be only one of this kind), and all the 2- and 3-point clusters up to radius 1.5×a.

The number of clusters just created is:

```
print("Number of clusters: ", len(cpool))
Number of clusters:  6
```

These can be visualized by first creating a clusters database:

```
cpool.write_clusters_db()
```

Now, we have at least two ways to visualize the pool of clusters: i) we can plot a few of them (e.g. n=6) on this notebook with juview by invoking the get_cpool_atoms() method of cpool:

```
juview(cpool.get_cpool_atoms(), n=6)
```

with the result shown in Fig. 10, or ii) by using the graphical user interface (gui) of ASE on a terminal. To use this second option, which is the recommended way to proceed when the clusters pool is large, first note that when creating the clusters database above with cpool.write_clusters_db(), a file called cpool.json was created in the same folder (let's call it $CWD) where this tutorial is located. This file has the proper format to be visualized with ASE's gui. Now, open a terminal and move to this folder with $>cd $CWD (the $> denotes the command prompt) and execute $>ase gui cpool.json. A number of windows will open. The relevant ones for the visualization of the clusters are shown in the screen capture in Fig. 11.

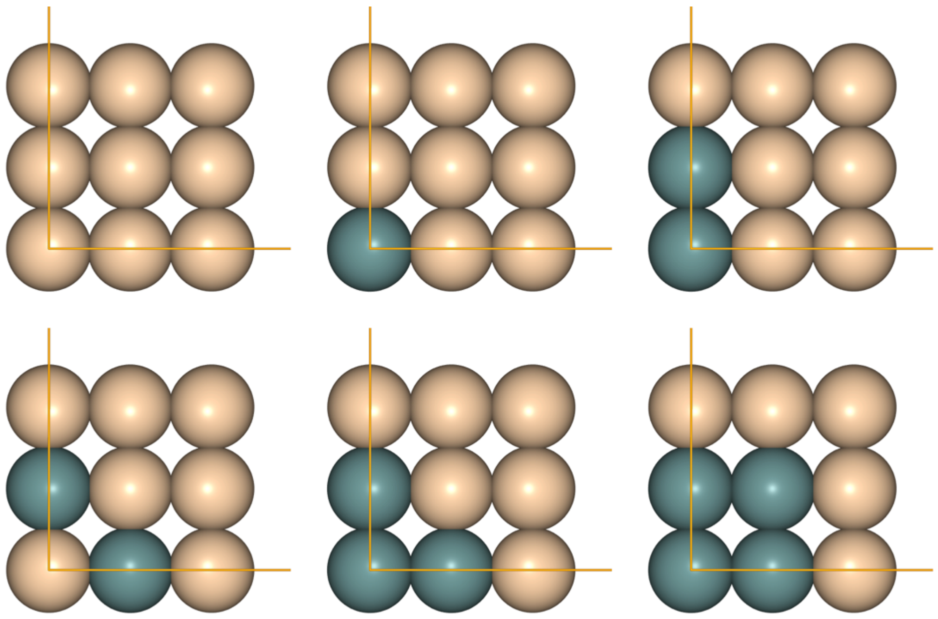


Figure 10: Small clusters pool for a square binary two-dimensional lattice.

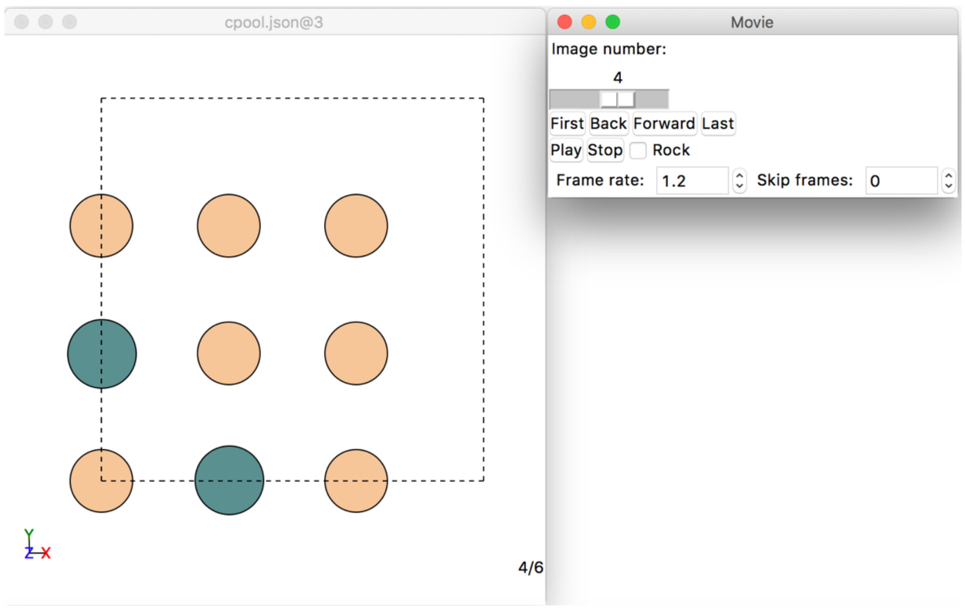


Figure 11: The graphical user interface of the Atomic Simulation Environment.

You can visualize all the clusters by clicking on the Back and Forward buttons of the Movie window. Let's see how this works for a larger pool:

```
r = 2.5
cpool = ClustersPool(plat, npoints=[0,1,2,3,4,5], radii=[0,0,a*r,a*r,a*r,a*r])
cpool.write_clusters_db()
print(len(cpool), " clusters were created.")
34  clusters were created.
```

Now, visualize these 34 clusters with ASE's gui as explained above.

Cluster orbits

The clusters obtained above are all symmetrically inequivalent. Moreover, each of them is a representative of an infinite set of symmetrically equivalent clusters. Every infinite set is called a cluster orbit. We can calculate a cluster orbit for a supercell and visualize it with ASE's gui. Let's do so for one cluster of the clusters pool cpool:

```
cluster_index = 6 # Select a cluster from the pool

# Obtain the orbit for this cluster
cluster_orbit, cluster_multiplicity =
cpool.get_cluster_orbit(cluster_index = cluster_index)

print("There are ",len(cluster_orbit),"symmetrically equivalent
representations of cluster ", cluster_index," in the supercell.")
print("The cluster multiplicity is ",cluster_multiplicity,".")
```

```
There are  100 symmetrically equivalent representations of cluster  6
in the supercell.
The cluster multiplicity is 4.
```

The cluster multiplicity is the number of symmetrically equivalent realizations of the cluster, under the symmetry operations of the parent cell, without counting the internal translations of the parent cell inside the supercell. To visualize the orbit, let’s first serialize it to a json file:

```
cpool.write_clusters_db(orbit=cluster_orbit,db_name="cluster_orbit.json")
```

Now, you may visualize the cluster orbit by executing the command $>ase gui cluster_orbit.json in a terminal.

Cluster correlations

In this section we will illustrate the calculation of the cluster correlations for the simple binary case studied here. To simplify the analysis, we will use a basis in which the site occupations are represented by a vector σ, whose components σ_i are equal to 1 if the crystal site i is ocupied with the substitutional species (Ge in this example), or zero otherwise (Si in this example).

Let's now calculate the cluster correlations defined in Eq. 2, for the same small pool of clusters obtained at the start of this tutorial:

```
from clusterx.clusters.clusters_pool import ClustersPool

r = 1.5
cpool = ClustersPool(plat, npoints=[0,1,2,3,4],
radii=[0,0,a*r,a*r,a*r])

cpool.write_clusters_db()

print(len(cpool), " clusters were created.")
6  clusters were created.
```

The created pool here is the same as in Fig. 10. To accomplish our task, we must first create a CorrelationsCalculator object:

```
from clusterx.correlations import CorrelationsCalculator
corrcal = CorrelationsCalculator("binary-linear", plat, cpool)
```

Let's now create a random structure and then obtain the cluster correlations for this structure:

```
structure = cpool.get_cpool_scell().gen_random({0:[3]})
juview(structure.get_atoms())
```

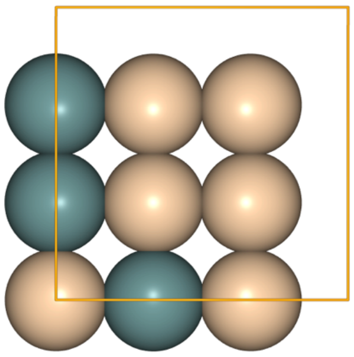


Figure 12: a random structure

Now, we calculate the correlations for this structure and display some relevant information:

```
mult = cpool.get_multiplicities()

corrs = corrcal.get_cluster_correlations(structure,
multiplicities=mult)

for i in range(len(cpool)):
    print("Cluster: ",i,"|   Correlation: ", corrs[i], "|
    Multiplicity: ", mult[i])
```

```
Cluster:  0 |   Correlation:  1.0 |   Multiplicity:  1
Cluster:  1 |   Correlation:  3.0 |   Multiplicity:  1
Cluster:  2 |   Correlation:  0.5 |   Multiplicity:  2
Cluster:  3 |   Correlation:  1.0 |   Multiplicity:  2
Cluster:  4 |   Correlation:  0.0 |   Multiplicity:  4
Cluster:  5 |   Correlation:  0.0 |   Multiplicity:  1
```

Excercise 3

With the figures of the clusters, the figure of the structure and the obtained multiplicities, calculate by hand the correlations for the random structure and verify that they are equal to what is returned by the get_cluster_correlations method. (Note: the correlation of the empty cluster, i.e. that one with npoints=0, is always 1 by definition.)

Excercise 4

Create a pool of clusters for the surface system of Tutorial 1 and visualize the generated clusters with ASE's gui.

Tutorial 3

In this tutorial, you will learn how to select the best model using different optimization criteria.

The optimal model

Once a training data set and a pool of clusters are available, the next question is how to find the optimal cluster expansion model to make actual predictions. This task requires finding the set of clusters that best describe the relevant interactions present in the system. A number of optimality criteria can be used. Here we will focus on obtaining models which are optimal in the sense of providing the best possible predictions for new data, i.e., data not included in the training set. For this purpose, the quality of the predictions can be quantified by the cross-validation score (CVS), which you will learn to calculate and interpret in this section.

Here, we will use the surface system with oxygen adsorption and Na substitution, which was used in Tutorial 1 and Tutorial 2, to find the optimal cluster expansion model. We start by generating the needed elements for the task, namely a training data set and a pool of clusters.

```
from ase.build import fcc111, add_adsorbate
from clusterx.parent_lattice import ParentLattice
from clusterx.structures_set import StructuresSet
from clusterx.visualization import juview
from clusterx.super_cell import SuperCell
import numpy as np
from random import randint

nsc = 4

pri2 = fcc111('Al', size=(1,1,3))
add_adsorbate(pri2,'X',1.7,'ontop')
pri2.center(vacuum=10.0, axis=2)

platt2 = ParentLattice(pri2,
site_symbols=[['Al'],['Al'],['Al','Na'],['X','O']])
scell2 = SuperCell(platt2,np.diag([nsc,nsc,1]))

sset2 = StructuresSet(platt2)

nstruc = 60

for i in range(nstruc):
    concentration = {0:[randint(0,nsc*nsc)],
                     1:[randint(0,nsc*nsc)]}
    sset2.add_structure(scell2.gen_random(concentration))

juview(sset2,n=3) # Plot the first 3 created random structrues
```

Sixty random structures, similar as those shown in Fig. 8, are created with this code. In contrast to the case in Fig. 8, here we create structures with all possible concentrations for the two sublattices (i.e. for the Vacancy-Oxygen and the Al-Na sublattices).

Now, we calculate the total energies for these structures using the Effective Medium Theory calculator, set up with fictitious potentials, i.e. “toy” model potentials which do not have special physical significance, but which serve the purpose of obtaining quick values to focus on learning the construction of a cluster expansion. In a “real life” calculation, you would calculate those energies with expensive first-principles calculations, using for instance the FHIaims DFT code.

```
from clusterx.calculators.emt import EMT2
sset2.set_calculator(EMT2())
energies = sset2.calculate_property()
print(energies)
```

Now we generate a clusters pool, with 3 of the 24 generated clusters shown in Fig. 13.

```
r=3.5
from clusterx.clusters.clusters_pool import ClustersPool
cpool = ClustersPool(platt2, npoints=[0,1,2,3,4], radii=[0,0,r,r,r])
cpool.write_clusters_db()
print(len(cpool)," clusters were generated.")
juview(cpool.get_cpool_atoms(),n=6),
24  clusters were generated.
```

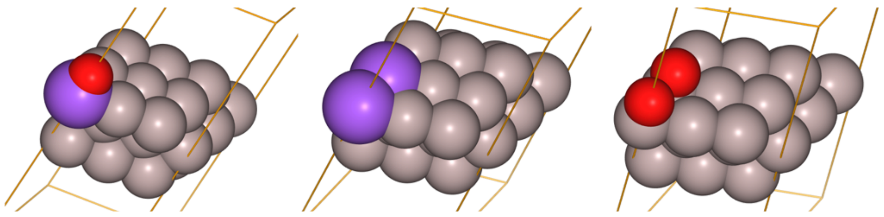


Figure 13: Three clusters from a pool of 24 clusters.

Above, only a few of the training structures and generated clusters are displayed. If you would like to visualize them all, use the ASE's gui interface as explained in Tutorial 2.

Now that we have the basic elements (i.e. training set sset2 and pool of clusters cpool), we can proceed to calculate the matrix of cluster correlations. We do so with a CorrelationsCalculator object:

```
from clusterx.correlations import CorrelationsCalculator

corrcal = CorrelationsCalculator("trigonometric", platt2, cpool)
comat = corrcal.get_correlation_matrix(sset2)
print(comat)
```

In Tutorial 2, you used the get_cluster_correlations method to obtain the cluster correlations of a single random structure. Above, we used the get_correlation_matrix method to obtain at once the correlations for all random structures in the training set and all clusters. The columns of the obtained matrix comat refer to clusters, while the rows correspond to structures. Thus the vector comat[3,:] contains the correlations of structure 3 with all clusters.

Now we are ready to perform the selection of the optimal cluster expansion model. To this end, we load the ClustersSelector class of clusterx and create an instance of it (that we call clsel):

```
from clusterx.clusters_selector import ClustersSelector

clsel = ClustersSelector('linreg', cpool, clusters_sets = "size")
```

To understand the meaning of the initialisation arguments and have a full list of them, you can inspect the documentation of **CELL** for this class. In this example, the argument 'linreg' will set-up the selector in a mode which uses cross-validation for model selection, i.e. it performs a search for the sub-pool of clusters yielding the lowest cross-validation score. This search is performed on subpools of the cpool clusters pool given in the arguments list. The parameter clusters_sets = "size" indicates that the subpools are formed by creating clusters sets of increasing size.

Next, we perform the actual cluster selection, calling the select_clusters() method of the previously created clusters selector. The select_clusters() method takes as arguments the correlation matrix and the target property values (energies in this case). The code below will also print out some relevant information regarding the optimal model, this is done with the display_info() method (of ClustersSelector and ClustersPool):

```
clsel.select_clusters(comat,energies)
print("Optimal model:")
clsel.display_info()

cpool_opt = clsel.get_optimal_cpool()
print("\nOptimal set of clusters:")
cpool_opt.display_info()
```

```
Optimal model:
CV score (LOO) for optimal model        :    0.1080
RMSE of the fit for optimal model       :    0.0693
Size of optimal clusters pool           :        17

Optimal set of clusters:
Index              |Nr. of points       |Radius
0                  |0                   |0.000
1                  |1                   |0.000
2                  |1                   |0.000
3                  |2                   |1.700
4                  |2                   |2.864
5                  |2                   |2.864
6                  |2                   |3.330
7                  |3                   |2.864
8                  |3                   |2.864
9                  |3                   |2.864
10                 |3                   |2.864
11                 |3                   |3.330
12                 |3                   |3.330
13                 |3                   |3.330
14                 |3                   |3.330
15                 |3                   |3.330
16                 |3                   |3.330
```

Probably you will not get exaclty the same result as shown above, since the training set is form ed with random structures, which will not be the same in every run of **CELL**.
From this output, we see that the cross-validation score is larger than the root mean squared er ror (RMSE) of the fit. This is as expected since the RMSE is the error from the fit to the whole data set, while the CV-score is the average prediction error. Since we are using a "toy" calcula tor (the Effective Medium Theory calculator of ASE), arb. units are used for the energy, theref ore the same for the CV-score and RMSE of the fit.

The remaining output refers to some characteristics of the optimal pool of clusters: it consists of 17 clusters, each of them with a number of points and radius as indicated above.

To better understand how this model was selected, we can plot the CV-score and RMSEs as a function of the size of the tried clusters pool in the CV procedure. This is done with the plot_optimization_vs_number_of_clusters() method of the visualization module:

```
%matplotlib inline
from clusterx.visualization import plot_optimization_vs_number_of_clusters
plot_optimization_vs_number_of_clusters(clsel)
```

The result is shown in Fig.14.

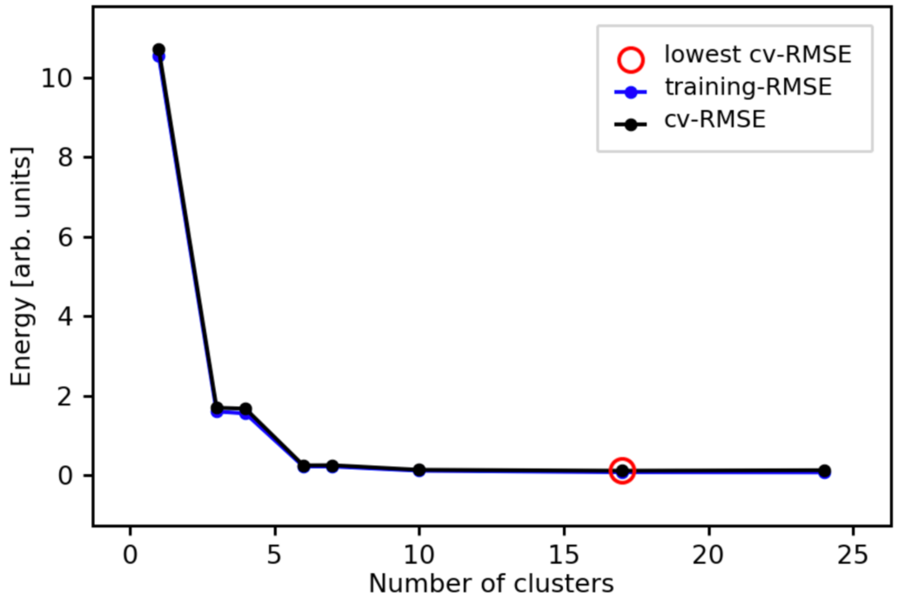


Figure 14: model selection using cross-validation.

The red circle in Fig.14 indicates the selected model (for this, you can check that the info from the plot is consistent with the output from display_info before). Also, for all tried sets, the CV-score is always larger than the RMSE of the fit.

Besides the average errors displayed above, it is also useful to inspect the errors of individual data points. This can be done with the plot_predictions() function of the visualization module:

```
from clusterx.visualization import plot_predictions
plot_predictions(clsel,energies)
```

The result is shown in Fig.15

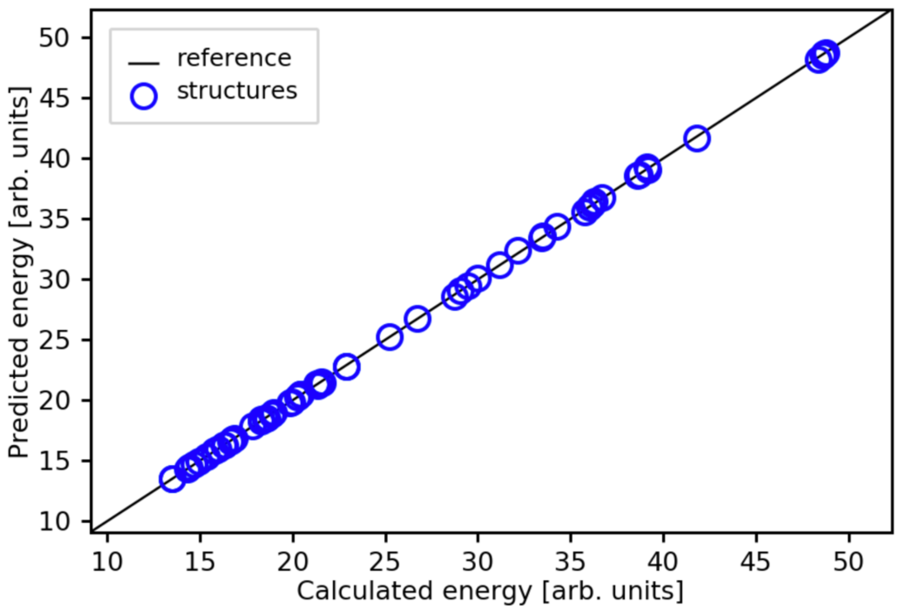


Figure 15: Calculated vs. predicted energies for structures (circles) in the training set. The solid line represents perfect predictions.

Next, we will experiment with a different optimization procedure by changing the clusters_sets parameter from "size" to "size+combinations" (and adding two additional parameters, nclmax and set0). By setting clusters_sets to "size+combinations", all the clusters pools for the CV procedure will be formed by a fixed pool with clusters up to 1 point and radius 0 (indicated by set0 = [1,0]) plus all possible cluster subsets (i.e. all possible combinations) of size nclmax (1 in the example below) from the remaining in the clusters pool. Below a minimal example of this is given:

```
clsel = ClustersSelector('linreg', cpool, clusters_sets =
"size+combinations", nclmax = 1, set0 = [1,0])

clsel.select_clusters(comat,energies)
print("Optimal model:")
clsel.display_info()

cpool_opt = clsel.get_optimal_cpool()
print("\nOptimal set of clusters:")
cpool_opt.display_info()

plot_predictions(clsel,energies)
```

```
Optimal model:
CV score (LOO) for optimal model          :    0.4141
RMSE of the fit for optimal model         :    0.3742
Size of optimal clusters pool             :         4

Optimal set of clusters:
Index                |Nr. of points       |Radius
0                    |0                   |0.000
1                    |1                   |0.000
2                    |1                   |0.000
3                    |2                   |2.864
```

The predictions for this examples are shown in Fig. 16. As you can see, the selection of small values for the parameters nclmax and set0 leads to a poor quality of the predictions.

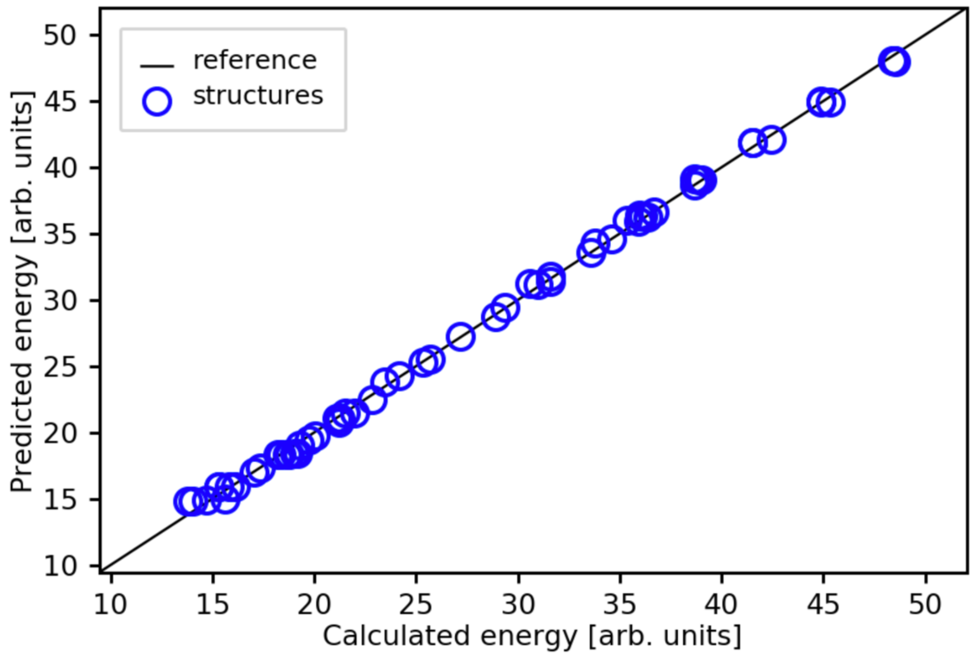


Figure 16: Calculated vs. predicted energies for structures (circles) in the training set. The solid line represents perfect predictions.

Below, larger values are set for these parameters. The combinatorial search takes more time now, but the quality of the model improves considerably:

```
clsel = ClustersSelector('linreg', cpool, clusters_sets =
"size+combinations", nclmax = 3, set0 = [2,3.5])

clsel.select_clusters(comat,energies)
print("Optimal model:")
clsel.display_info()

cpool_opt = clsel.get_optimal_cpool()
print("\nOptimal set of clusters:")
cpool_opt.display_info()

plot_optimization_vs_number_of_clusters(clsel)
```

```
Optimal model:
CV score (LOO) for optimal model         :    0.0934
RMSE of the fit for optimal model        :    0.0783
Size of optimal clusters pool            :        10

Optimal set of clusters:
Index                  |Nr. of points        |Radius
0                      |0                    |0.000
1                      |1                    |0.000
2                      |1                    |0.000
3                      |2                    |1.700
4                      |2                    |2.864
5                      |2                    |2.864
6                      |2                    |3.330
```

```
7                   |3                   |2.864
8                   |3                   |2.864
9                   |3                   |3.330
```

The thorough combinatorial search performed above is shown in Fig. 17: In contrast with the previous approaches, in this case there is a very large number of clusters subsets at every clusters set size.

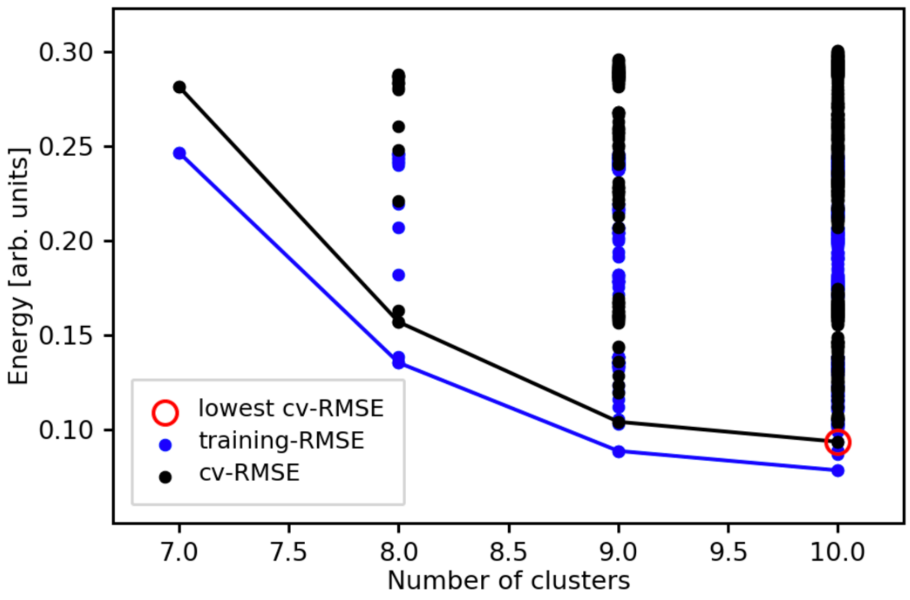


Figure 17: Model selection through cross-validation on a combinatorial search.

In the next example, we will perform cluster selection with LASSO (Least Absolute Shrinkage and Selection Operator) [3], using cross-validation for selecting the sparsity hyper-parameter. To this end, we change the first argument from "linreg" to "lasso", and we indicate the sparsity range (sparsity_max and sparsity_min) for the hyper-parameter determination through cross-validation.

```
clsel = ClustersSelector('lasso', cpool, sparsity_max=0.10,
sparsity_min=0.001)

clsel.select_clusters(comat,energies)
print("Optimal model:")
clsel.display_info()

cpool_opt = clsel.get_optimal_cpool()
print("\nOptimal set of clusters:")
cpool_opt.display_info()
```

```
Optimal model:
CV score (LOO) for optimal model        :    0.1003
RMSE of the fit for optimal model       :    0.0700
Size of optimal clusters pool           :        15

Optimal set of clusters:
Index               |Nr. of points       |Radius
0                   |0                   |0.000
1                   |1                   |0.000
2                   |1                   |0.000
```

```
3                    |2                   |1.700
4                    |2                   |2.864
5                    |2                   |2.864
6                    |2                   |3.330
7                    |3                   |2.864
8                    |3                   |2.864
9                    |3                   |2.864
10                   |3                   |3.330
11                   |3                   |3.330
12                   |3                   |3.330
13                   |4                   |3.330
14                   |4                   |3.330
```

The result of the optimization is shown in Fig. 18.

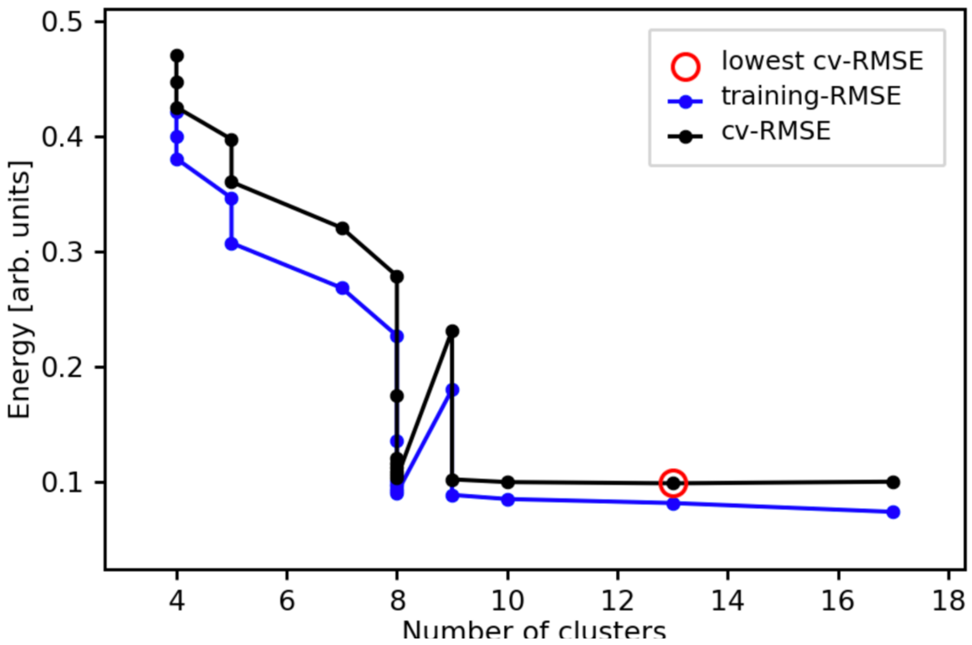


Figure 18: Model selection with LASSO, with cross-validation for the sparsity hyper-parameter.

In order to see if the selected sparsity range contains the optimal sparsity, it is very informative to plot the cross-validation procedure for the hyperparameter. This is done with the plot_optimization_vs_sparsity function of the visualization module:

```
from clusterx.visualization import plot_optimization_vs_sparsity
plot_optimization_vs_sparsity(clsel)
```

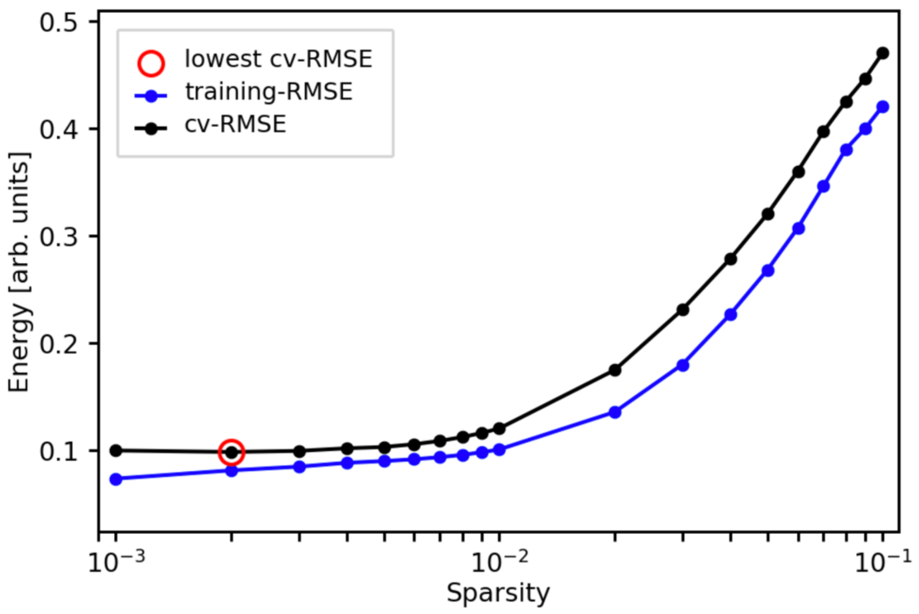


Figure 18: cross-validation procedure for the sparsity parameter in LASSO. The red circle represents the optimal sparsity value.

Acknowledgements

We are very grateful to Carsten Baldauf, Marcel Langer, and Henrik Kowalski for helping with the organization, testing, and the installation of **`CELL`** on the computer pools of the School of Physics at the Peking University. We also acknowledge the local organizers for their help.

References

[1] *J. Sanchez, F. Ducastelle, and D. Gratias.*
Generalized cluster description of multicomponent systems.
Physica A: Statistical Mechanics and its Applications, **128**(1):334 – 350, 1984.

[2] *Maria Troppenz, Santiago Rigamonti, and Claudia Draxl*
Predicting Ground-State Configurations and Electronic Properties of the Thermoelectric Clathrates $Ba_8Al_xSi_{46-x}$ and $Sr_8Al_xSi_{46-x}$
Chem. Mater., 2017, **29** (6), pp 2414–2424

[3] *L. J. Nelson, G. L. W. Hart, F. Zhou, and V. Ozolin¸ˇs.*
Compressive sensing as a paradigm for building physics models.
Phys. Rev. B, **87**:035125, Jan 2013.