# aiida-cusp

***Release 0.0.0***

**Aug 05, 2020**

# Installation

The aiida-cusp plugin is an AiiDA plugin for VASP optionally utilizing Custodian to run the VASP calculations with automated error correction. Contrary to error correction methods that may be implemented directly in AiiDA workflows that allow for error corrections on the workflows' step level, the Custodian interface of this plugin introduces error corrections for VASP on the calculation's runtime level. In addition to the error correction of VASP calculations this plugin was also designed with fast and easy pre- and post-processing in mind. Thus, all data types serving as inputs and outputs to the calculators, implemented by this plugin, are tightly connected to the Pymatgen framework maintaining direct access to the therein implemented set of atomistic tools.

# CHAPTER 1

## Installing the Plugin

It is strongly recommended to install the plugin using the *conda* package manager which will also install the required RabbitMQ and PostgreSQL services. However, you may also install the plugin using either *pip* or directly from the plugin's *source code*.

**Warning:** Be advised that the installation via `pip` or directly from source will **not** install the required RabbitMQ or PostgreSQL services automatically! While this allows you to use RabbitMQ or PostgreSQL installations already available on your system, it requires more manual work to get AiiDA setup and running. Thus, these installation routes are considered suitable only for advanced AiiDA users.

After the plugin has been installed you can check if the installation was successful by running the command

```
$ reentry scan
$ verdi plugin list aiida.calculations
Registered entry points for aiida.calculations:
* arithmetic.add
* cusp.vasp
* templatereplacer
```

The output of this command should now contain the new calculation entry point `cusp.vasp`.

**Note:** Independent of the method you chose to install the plugin never forget to run `reentry` after successful installation in order to discover the newly added entry points using the command:

```
$ reentry scan
```

If this post-installation step is skipped new entry points installed by the plugin will not be discoverable and the plugin will not work as expected.

## 1.1 Installing via Conda (recommended)

To install the plugin via the `conda` installer Anaconda needs to be installed on your machine first. Consult the conda installation guide for more information on how to install Anaconda on your system.

**Note:** Instead of installing the full Anaconda distribution you may also install the more lightweight Miniconda distribution which will take up less disk space.

Once Anaconda (or Miniconda) is installed on your system you may install the plugin using `conda` by running the command

```
$ conda create --name aiida-cusp -c conda-forge aiida-cusp aiida-core.services
```

The command shown above will create a new python environment named *aiida-cusp* and installs the plugin and all required dependencies to that environment. After the installation has finished we can activate the newly created environment and run the `reentry scan` command to discover the added entry points:

```
$ conda activate aiida-cusp
(aiida-cusp) $ reentry scan
```

If the installation was successful the new calculation entry point `cusp.vasp` should now be available from the `verdi` command line:

```
(aiida-cusp) $ verdi plugin list aiida.calculations
Registered entry points for aiida.calculations:
* arithmetic.add
* cusp.vasp
* templatereplacer
```

If the new entry point `cusp.vasp` is disovered correctly please proceed with the *next step* to finalize the plugin's installation.

## 1.2 Installing via PIP (advanced)

To install the plugin via the `pip` installer run the command

```
$ pip install aiida-cusp
```

which will install the plugin and the required dependencies using the resources available from the python package index (PyPi).

## 1.3 Installing from Source (advanced)

Alternatively to the previous installation methods, install the plugin directly from source by cloning the plugin's repository. After cloning go to the source root directory containing the project's *setup.py* and run

```
$ python setup.py install
```

# CHAPTER 2

# Getting the Plugin Ready

For the steps discussed in the following a working AiiDA installation is expected, i.e.

- AiiDA and a corresponding database has been setup already
- all services (i.e. postgresql, rabbitmq and the daemon) are up and running
- a computer is already configured and the next logical step is to add a(nother) code for it

If the above is not true for your installation, please complete the AiiDA installation before proceeding.

---

**Note:** Although the plugin installation automatically installs AiiDA to the plugin's environment you still need to manually configure and setup the new AiiDA installation. In case you are new to AiiDA please refer to the AiiDA documentation for detailed instructions on how to setup a new AiiDA installation and it's related services.

---

## 2.1 Setting up the Custodian Codes

Setting up the codes for Custodian is straightforward and does in general not differ compared to the setup of regular codes. Similar to setting up the VASP code which tells AiiDA how to load and access the corresponding `vasp` executable, Custodian codes are setup to give AiiDA the same information on the `cstdn` executable. The only reason for the Custodian code setup being covered in more detail here is the fact that it may likely not be available on your system by default and thus may required some extra steps to setup.

---

**Note:** In case Custodian is already available on your system you may skip the following steps and directly setup the code as usual. To check if Custodian is already available simply you can simply test if the `cstdn` executable is available, for instance using the command

```
$ module load anaconda
$ conda activate aiida-cusp-dev
$ which cstdn
/home/andreas/local/apps/anaconda3/2019.3/envs/aiida-cusp-dev/bin/cstdn
```

Assuming the above check has failed for you and Custodian is obviously not available on your system yet, the following example shows how to install and setup the code through anaconda.

**Note:** Keep in mind that the example shown in the following is only one possibility of installing Custodian on your system. The solution best suited for your use case, of course, depends on the available python installation and your personal preferences. Thus, consider the given example as a basic guide giving you a general idea of the required installation steps

**Warning:** Custodian has to be installed on the **target computer** that is later used to run the calculation (i.e. usually the same computer where also VASP is installed). Keep in mind that this computer may be different from the computer running your AiiDA installation!

To setup the Custodian code on a remote computer using anaconda we first create a new environment containing the Custodian installation. Using `custodian` for the environment name the required `conda` command could look like the following

```
$ conda create --name custodian python=3.6 custodian
```

After the installation is completed you may check that custodian was indeed installed. To do this simply activate the created environment and perform the above mentioned check for the `cstdn` executable:

```
$ conda activate custodian
$ which cstdn
/home/user/envs/custodian/bin/cstdn
```

If the the full path to the `cstdn` executable is printed to the screen the installation was successful. Note that the printed path should point to the folder in which the new Anaconda environment was installed previously. After installing the required executable, the next step is now to setup the code on your local computer (i.e. the computer you're actually running AiiDA on!) To setup the code simply run the `verdi code setup` command and provide all the necessary information you're asked for. (Please refer to the offical AiiDA documentation for detailed instructions on how to setup a new code)

**Note:** When asked for the input plugin, choose the `cusp.vasp` entry-point in order to connect the code to the plugin's calculator. At the very end of the setup the code's prepend / append sections are requested: Please make sure the Custodian installation is made available at the code's loading time by adding the appropriate commands to the requested prepend section (See the example below)

After successful code setup you may run the *verdi code show* command on your newly configured Custodian code which should give an output similar to the following:

```
$ verdi code show Custodian@RemoteComputer
--------------------  ------------------------------------------------------
PK                    14166
UUID                  ec3d6056-4d9c-452b-8453-410b28e7a126
Label                 Custodian
Description           Custodian code on remote Computer
Default plugin        cusp.vasp
Type                  remote
Remote machine        RemoteComputer
Remote absolute path  /home/user/envs/custodian/bin/cstdn
```

(continues on next page)

(continued from previous page)

```
Prepend text
                      module load anaconda                                    # load␣
↪anaconda module and conda command
                      source "$(conda info --base)/etc/profile.d/conda.sh"  # make
↪'conda activate' command available
                      conda activate custodian                                # load␣
↪the actual environment and add cstdn to PATH
Append text           No append text
--------------------  ------------------------------------------------------
```

## 2.2 Populating the Database with VASP Pseudo-Potentials

With the code now being setup we're almost set to run the first calculation. However, before doing so we first need to populate the AiiDA database with appropriate pseudo-potentials. To this end the plugin extends the `verdi data` command with the additional `potcar` sub-command. This new sub-command allows to interact with VASP pseudo-potential files and offers two different ways of adding potentials:

- adding only single potentials using `verdi data potcar add single`
- or adding a batch of potentials at once using `verdi data potcar add family`

---

**Note:** Type `verdi data potcar --help` on the command line to get more information on the provided commands and the expected syntax. The command is also documented *here*.

---

In the following, only a single pseudo-potential for silicon, required to run the calculation example presented in the next section, is added to the database. As stated above a single pseudo-potential may be added to the database using the `verdi data potcar add single` command, thus:

```
$ verdi potcar add single /home/andreas/plugin_dev/testing/potcar/potpaw_PBE/Si/
↪POTCAR --name Si --functional pbe

New pseudo-potential(s) to be stored:

name    element    functional      version  path
------  ---------  ------------  ---------  ----------------------------------------
↪--------------------------------------
Si      Si         pbe            19990402  /home/andreas//plugin_dev/testing/potcar/
↪potpaw_PBE/Si/POTCAR

File location: /home/andreas/plugin_dev/testing/potcar/potpaw_PBE/Si/POTCAR

Discovered a total of 1 POTCAR file(s) of which
        1       will be stored to the database,
        0       are already available in the database and
        0       will be skipped due to errors

Before continuing, please check the displayed list for possible errors! Continue and␣
↪store? [y/N]: y
Created new VaspPotcarFile node with UUID c6dd3acc-7ffe-44de-b638-4dff4ff8bab8 at ID␣
↪918
```

Check the printed summary to check if the potential was recognized correctly and press `Y` to continue and save the potential with the shown attributes to the database.

**Note:** In later calculations you can choose from the different stored potentials by referencing to the `name`, `functional` and `version` printed to the screen when adding the potential. Fixing all of the three attributes uniquely defines a pseudo-potential which is the reason why these attributes are used as potential identifiers throughout this plugin.

If you want know which potentials are already stored, use the `vasp data potcar list` command to get an overview of the available potentials, i.e.

```
$ verdi data potcar list --element Si

Showing available pseudo-potentials for
        name:       all
        element:    Si
        functional: all

  id  uuid                                  name          element    functional
----  ------------------------------------  ------------  ---------  ------------
 209  d31eea80-f1fc-432c-b68d-1553f44f73a8  Si_d_GW_nr    Si         pbe
 210  bee20ab8-8b38-4255-9885-ab7e53605678  Si_d_GW       Si         pbe
 211  47787525-9dc1-4c8b-a327-72dd6223df96  Si_h_old      Si         pbe
 212  1991a70b-440a-4626-ac27-330b4b546b7e  Si_h          Si         pbe
 213  6730058f-e2d9-4a51-baa1-cee8887f9a70  Si_nopc       Si         pbe
 214  b8d542b6-dd56-49b3-8e57-6281b4971ff7  Si            Si         pbe
 215  d832b49d-6c36-469e-afef-0fc8b8533fb3  Si_pv_GW      Si         pbe
 216  c19da65f-c696-4d02-bdbe-c5211e1c896f  Si_sv_GW_nr   Si         pbe
 217  537a85fa-34b8-4267-bbc0-aed06346a03f  Si_sv_GW      Si         pbe
```

## 2.3 Calculation Example

As an example the following code snippet describes the relaxation for a simple silicon diamond structure using both Custodian and the VASP code. (Note that this is only for demonstration purposes and simply adding a custodian code will **not** enable any error correction for that calculation! Please refer to the calculator section on how to run a calculation with error corrections) For the sake of simplicity, here, all calculation input parameters are taken as defined by pymatgen's `MPRelaxSet`.

```python
#!/usr/bin/env python

from aiida.plugins import CalculationFactory, DataFactory
from aiida.engine import submit
from aiida.orm import Code

from pymatgen.io.vasp.sets import MPRelaxSet

# load the plugin's datatypes
VaspIncarData = DataFactory('cusp.incar')
VaspKpointData = DataFactory('cusp.kpoints')
VaspPoscarData = DataFactory('cusp.poscar')
VaspPotcarData = DataFactory('cusp.potcar')

def si_diamond_structure():
    """
    Setup a cubic unitcell containing the Si diamond structure
    """
```

(continues on next page)

(continued from previous page)

```
    from pymatgen import Lattice, Structure
    lattice = Lattice.cubic(5.4309)
    species = ['Si']
    coords = [[.0, .0, .0]]
    # setup the structure
    structure = Structure.from_spacegroup('Fd-3m', lattice, species, coords)
    return structure

# define the vasp and custodian codes to be used for the calculation
code_vasp = 'vasp_5.4.1_openmpi_4.0.3_scalapack_2.1.0@CompMPI'
code_custodian = 'custodian_2020427@CompMPI'

# get the builder for the VASP calculation object and setup the codes
# and job resources
VaspSiRelax = CalculationFactory('cusp.vasp').get_builder()
VaspSiRelax.code = Code.get_from_string(code_vasp)
VaspSiRelax.custodian.code = Code.get_from_string(code_custodian)
VaspSiRelax.metadata.options.resources = {
    'tot_num_mpiprocs': 4,
    'num_machines': 1
}
# simplest case: simply use the calculation inputs as defined by
# pymatgen's MPRelaxSet
mprelaxset = MPRelaxSet(si_diamond_structure())
# set the calculation parameters
VaspSiRelax.incar = VaspIncarData(incar=mprelaxset.incar)
VaspSiRelax.kpoints = VaspKpointData(kpoints=mprelaxset.kpoints)
VaspSiRelax.poscar = VaspPoscarData(structure=mprelaxset.poscar)
VaspSiRelax.potcar = VaspPotcarData.from_structure(
                            mprelaxset.poscar, mprelaxset.potcar_functional,
                            potcar_params=mprelaxset.potcar_symbols)
# submit the code to the daemon
calc_node = submit(VaspSiRelax)
```

Saving the above contents to a new python file, i.e. `test_calc.py`, we are now ready to actually run the calculation. One the command line simply execute the following command to start the calculation:

```
$ verdi run test_calc.py
```

After the calculation has been successfully deployed to the daemon it should now appear in the list of active processes. You may check this using AiiDA's `verdi process list` which will output all active processes:

```
$ verdi process list
  PK  Created    Process label         Process State    Process status
----  ---------  --------------------  ---------------  ------------------------------
↪---------
1332  5s ago     VaspCalculation        Waiting        Monitoring scheduler: job␣
↪state RUNNING
```

**Note:** You should be able to run this example by simply copy and pasting the code to a local file on your computer. Of course, the code names used in the snippet have to be adapted accordingly before submission.

CHAPTER 3

# Next Steps

Following the previous two steps discussing the *installation* and *basic setup* the plugin should now be ready to use. Which steps to take from here depends on your personal preference:

## 3.1 I would like to directly jump into the calculations and learn by doing

Checkout the *Tutorials section* for a collection if example calculations and workflows.

## 3.2 Nah, uh, I'd rather get to known the plugin's intestines first

Jump to the *User-Guide section* to learn everything about the details of calculation plugins and the data types used by this plugin.

## 3.3 I found a bug and / or would like to contribute to the plugin

Refer to the Development section to learn about the possible ways you can contribute to the plugin or how to file a bug report.

CHAPTER 4

# Calculation Tutorials

This section contains tutorials using the plugin's calculation objects to illustrate their setup and usage for VASP calculations using this plugin and AiiDA.

**Note:** Note that a copy and paste-able example of the calculation's code discussed in the tutorial is provided at the end of each tutorial included in this section.

## 4.1 Relaxation of a Si Diamond Structure

Topics covered in this tutorial:

- Setting up calculation inputs
- Setting up an (error corrected) VASP calculation
- Access generated outputs

### 4.1.1 Introduction

In this (simple) tutorial the `VaspCalculation` calculator of this plugin is used to run an error corrected VASP calculation. In particular, a silicon diamond structure is relaxed and finally the ground state energy per Si atom is calculated for the relaxed structure.

### 4.1.2 Setting up the Inputs

Before we can start the calculation we need to setup all the required inputs. As every normal VASP calculation the plugin also requires the basic VASP inputs to be provided, i.e.

- *POSCAR*
- *INCAR*

- *KPOINTS*
- *POTCAR*

We will start in the following with setting up the structure inputs for the silicon diamond relaxation. To this end we first need to setup the silicon diamond structure. Since the plugin is tightly connected to pymatgen's data types the structure is setup using pymatgen:

```
from pymatgen import Lattice, Structure
lattice = Lattice.cubic(5.431)
species = ["Si"]
coords = [[0.0, 0.0, 0.0]]
si_diamond_structure = Structure.from_spacegroup(227, lattice, species, coords)
```

Using this structure we can now directly setup the required *POSCAR* data for the calculation expecting a *VaspPoscarData* as input:

```
from aiida.plugins import DataFactory
VaspPoscarData = DataFactory('cusp.poscar')
poscar = VaspPoscarData(structure=si_diamond_structure)
```

**Note:** You're not restricted to the pymatgen structure type here but you may also initialize the *VaspPotcarData* from several other types, i.e.: Poscar or StructureData are also accepted input types.

With the *POSCAR* data now being set we are already done with the structure setup for our calculation. Next we define the *INCAR* parameters for the calculation allowing for a full cell relaxation of the passed structure (i.e. *ISIF=3*). As this is only a tutorial and not a serious calculation we, of course, want the calculation to finish quickly. Thus, a not very sophisticated force-convergence threshold of *EDIFFG=-0.1* is chosen for the tutorial. The expected input for the *INCAR* parameters is of type *VaspIncarData* and we can set it up by simply passing a dictionary containing the *INCAR* settings we want to apply for the calculation:

```
VaspIncarData = DataFactory('cusp.incar')
incar_params = {'ISIF': 3, 'EDIFFG': -0.1}
incar = VaspIncarData(incar=incar_params)
```

For the *KPOINTS* parameters we also use a rather sparse grid. In the following the grid is setup using the automatic method:

```
VaspKpointData = DataFactory('cusp.kpoints')
kpoint_params = {'mode': 'auto', 'kpoints': 100}
kpoints = VaspKpointData(kpoints=kpoint_params)
```

Finally, we setup the last missing input: the pseudo-potential. For the pseudo-potential the *VaspPotcarData* type is expected by the calculator. Here, the pseudo-potentials have to be passed as dictionary with key-value pairs where each key defines an element present in the *POSCAR* data with the pseudo-potential that should be used for the element as value. However, we do not need to construct this dictionary by hand: We simply use the *from_structure()* method and let the function do the job. For the calculation we use the default PBE potential for silicon (i.e. the pseudo-potential with name *'Si'*):

```
VaspPotcarData = DataFactory('cusp.potcar')
potcar = VaspPotcarDara.from_structure(poscar, 'pbe')
```

which initializes a dictionary containing a single entry of the following form

```
{'Si': <VaspPotcarData: uuid: 1f6ea785-876f-4942-9f30-51a8eac39573 (unstored)>}
```

**Note:** Before you can initialize the pseudo-potential data using the aforementioned *from_structure()* you have to add the required potentials to the database. This can be easily done using the implemented `verdi data potcar add` command. For the required silicon pseudo-potential this command could look like the following:

```
$ verdi data potcar add single --name "Si" --functional "pbe" /vasp_pseudos/potpaw_
↪PBE/Si/POTCAR
```

Please refer to the *potcar command documentation* for a detailed introduction to the command and the expected parameters and their meanings.

### 4.1.3 Preparing and Running the Calculation

Since we now have defined all required inputs we are ready to setup and finally also run the calculation. To setup the calculation we need to define a code that should be used to run the VASP calculation. We can check for available codes using the `verdi code list` command which will list all codes available in the database:

```
$ verdi code list
# List of configured codes:
# (use 'verdi code show CODEID' to see the details)
* pk 1228 - vasp_5.4.1_openmpi_4.0.3_scalapack_2.1.0@CompMPI
* pk 1271 - custodian_2020427@CompMPI
* pk 1366 - vasp_5.4.1_openmpi_4.0.3_scalapack_2.1.0_vtst@CompMPI
```

Here, three different codes are available from the database, two VASP codes

- *vasp_5.4.1_openmpi_4.0.3_scalapack_2.1.0@CompMPI*
- *vasp_5.4.1_openmpi_4.0.3_scalapack_2.1.0_vtst@CompMPI*

and one custodian code that can be used for the error correction

- *custodian_2020427@CompMPI*

In the following the *vasp_5.4.1_openmpi_4.0.3_scalapack_2.1.0@CompMPI* code is used to run the calculation. To connect the code and setup the calculation we first load the corresponding builder for the VASP calculator *aiida_cusp.calculators.VaspCalculation* implemented by this plugin.

```
from aiida.plugins import CalculationFactory
VaspSiRelax = CalculationFactory('cusp.vasp').get_builder()
```

Using the returned builder we can now simply add our inputs to the calculation. For the VASP code and the required calculation inputs, setup in the previous step, this could look like the following

```
from aiida.orm import Code
# setup the VASP code
VaspSiRelax.code = Code.get_from_string('vasp_5.4.1_openmpi_4.0.3_scalapack_2.1.
↪0@CompMPI')
resources = {'tot_num_mpiprocs': 4, 'num_machines': 1}
VaspSiRelax.metadata.options.resources = resources
# setup the VASP calculation inputs
VaspSiRelax.incar = incar
VaspSiRelax.kpoints = kpoints
VaspSiRelax.poscar = poscar
VaspSiRelax.potcar = potcar
```

**Note:** Note the added resources for the job defined via the *metadata.options.resources* option. These define the calculation jobs resources the scheduler acquires upon submission, i.e. the number of cores and machines to be used on the computer to run the job. As the settings defined here usually depend on the type of scheduler you are using, please refer to the AiiDA scheduler documentation for the options available for your scheduler.

Finally, we want to run the VASP calculation defined by the above inputs with automated error correction using Custodian. To do so we need to add the Custodian executable, defined by the *custodian_2020427@CompMPI* code object, and the error handlers we want to use to the calculation as additional inputs:

```
# enable error correction by adding an  **additional** custodian code ...
VaspSiRelax.custodian.code = Code.get_from_string('custodian_2020427@CompMPI')
# ... and the corresponding custodian error handlers
VaspSiRelax.custodian.handlers = ['VaspErrorHandler']
```

In the above example only a single error handlers, i.e. the *'VaspErrorHandler'*, is set in the calculation and the default settings as defined by the plugin are used for the connected Custodian code. For a complete overview of the available error handlers and the available Custodian settings that may be set for the code, please refer to the *Custodian section* of this documentation.

**Note:** You can also run this example without error corrections by simply leaving the *VaspSiRelax.custodian.code* and *VaspSiRelax.custodian.handler* inputs empty (those inputs are optional!) In that case the calculator will call the VASP executable defined by the code given in the *VaspSiRelax.code* input directly instead of wrapping VASP with Custodian.

With all required inputs defined, we are now ready to run the code. The following code shows how the calculation can be submitted to the AiiDA daemon via the `submit()` function provided by the `aiida.engine` module:

```
from aiida.engine import submit
node = submit(VaspSiRelax)
```

**Note:** If you want to run the calculation in your interpreter replace the used `submit()` function with the `run()` function.

We can check that the calculation was indeed submitted to the daemon by checking the output of the `verdi process list` command which should now list our submitted calculation as running process:

```
$ verdi process list
  PK  Created    Process label         Process State    Process status
----  ---------  --------------------  ---------------  ------------------------------
↪---------
1377  43s ago    VaspCalculation        Waiting        Monitoring scheduler: job␣
↪state RUNNING
```

### 4.1.4 Inspecting the Outputs

After the job has finished the automatically connected default `VaspFileParser` will add the generated *vasprun.xml*, *OUTCAR* and *CONTCAR* files as outputs to the stored calculation node. As the stored files are available from the node using the *outputs.parsed_results* namespace we can easily determine the energy per Si atom in the relaxed structure using the parsed *vasprun.xml* file by loading the calculation node and inspecting the stored file contents. Using the *PK* of the stored calculation node printed, next to the running calculation in the output of *verdi process list* (see above),

the node can be loaded from the database using AiiDA's `load_node()` function. In the following a verdi shell is used to load the node and calculated the energy per Si atom by inspecting the loaded node's outputs:

```
>>> from aiida.orm import load_node
>>> si_relax_node = load_node(1377)
>>> print(si_relax_node)
uuid: f97a5909-6a3d-4cec-b4ea-39a69a3a125e (pk: 1337)
>>> print(si_relax_node.outputs.parsed_results__vasprun_xml)
<VaspVasprunData: uuid: 136e55a1-b1d4-4aeb-9661-8830808552f5 (pk: 1339)>
```

Since the plugin tightly integrates AiiDA with the Pymatgen framework we can easily get to the total energy of the system (and actually many more quantities) using the *get_vasprun()* method implemented by the *VaspVasprunData* class:

```
>>> pymatgen_vasprun = si_relax_node.outputs.parsed_results.vasprun_xml.get_vasprun()
>>> print(type(pymatgen_vasprun))
<class 'pymatgen.io.vasp.outputs.Vasprun'>
>>> total_energy = float(pymatgen_vasprun.final_energy)
>>> num_atoms = float(len(pymatgen_vasprun.final_structure))
>>> energy_per_atom = total_energy / num_atoms
>>> print("Energy per atom: {} (eV/atom)".format(energy_per_atom))
Energy per atom: -5.41045657375 (eV/atom)
```

### 4.1.5 Copy-and-Paste

```
from pymatgen import Lattice, Structure
from aiida.orm import Code
from aiida.plugins import CalculationFactory, DataFactory
from aiida.engine import submit

# setup the code-labels defining the codes to be used
vasp_code_label = 'place_your_vasp_code_label_here'
custodian_code_label = 'place_your_custodian_code_label_here'

# define all input datatypes
VaspIncarData = DataFactory('cusp.incar')
VaspKpointData = DataFactory('cusp.kpoints')
VaspPoscarData = DataFactory('cusp.poscar')
VaspPotcarData = DataFactory('cusp.potcar')

# setup the silicon diamond structure
lattice = Lattice.cubic(5.431)
species = ["Si"]
coords = [[0.0, 0.0, 0.0]]
si_diamond_structure = Structure.from_spacegroup(227, lattice, species, coords)

# define calculation inputs
incar = VaspIncarData(incar={'ISIF': 3, 'EDIFFG': -0.1})
poscar = VaspPoscarData(structure=si_diamond_structure)
potcar = VaspPotcarData.from_structure(poscar, 'pbe')
kpoints = VaspKpointData(kpoints={'mode': 'auto', 'kpoints': 25})

# fetch codes from the AiiDA database
vasp_code = Code.get_from_string(vasp_code_label)
custodian_code = Code.get_from_string(custodian_code_label)
```

(continues on next page)

(continued from previous page)

```
# setup the calculation object
VaspSiRelax = CalculationFactory('cusp.vasp').get_builder()
resources = {'tot_num_mpiprocs': 4, 'num_machines': 1}
VaspSiRelax.metadata.options.resources = resources
VaspSiRelax.code = vasp_code
VaspSiRelax.incar = incar
VaspSiRelax.poscar = poscar
VaspSiRelax.potcar = potcar
VaspSiRelax.kpoints = kpoints

# optional inputs for the custodian error correction (skip this if you
# do not want to enable error correction)
VaspSiRelax.custodian.code = custodian_code
VaspSiRelax.custodian.handlers = ['VaspErrorHandler']

# submit calculation the daemon
node = submit(VaspSiRelax)

# print out the PK of the submitted job
print("Submitted VaspSiRelax with PK: {}".format(node.pk))
```

## 4.2 NEB Calculation for the Interstitial Migration of $Li^+$ in $Li_4P_2S_6$

Topics covered in this tutorial:

- Setting up and running a (error corrected) VASP NEB calculation
- Access generated outputs to plot the NEB path

---

**Note:** In this tutorial the VASP code with the additional VTST tools package is used

---

### 4.2.1 Introduction

In this tutorial the *VaspCalculation* calculator of this plugin is used to run an error corrected, complex NEB calculation using the AiiDA framework. In the following example the NEB calculation dealing with migration of interstitial $Li^+$ ions in $Li_4P_2S_6$ is shown. Here, three different steps are required:

- First the initial, defect free $Li_4P_2S_6$ structure is relaxed allowing a full cell relaxation
- In a second step the structures defining the NEB path endpoints are relaxed
- Finally, the NEB calculation is carried out based on the relaxed endpoint structures

---

**Note:** This tutorial assumes that you have already a basic knowledge about how to setup a calculation using the *VaspCalculation* class and what kind of inputs are expected. If not, checkout the *previous tutorial* dealing the relaxation of a simple Si diamond structure which contains more details about the expected inputs and how to setup a calculation.

---

### 4.2.2 Relaxing the initial Structure

In order to calculate the migration energy of a $Li^+$ ion in the $Li_4P_2S_6$ crystal structure using NEB we first need to calculated the ground state structure of the defect free $Li_4P_2S_6$ structure. We can easily setup the initial structure using pymatgen:

```
from pymatgen import Structure, Lattice
# lattice vector lengths
a = 6.10622  # a = b
c = 6.61513
lattice = Lattice.hexagonal(a=a, c=c)
# Structure setup with P-31m spacegroup
positions, species = list(zip(*[
    ([0.3333, 0.6667, 0.0000], "Li"),  # Li1
    ([0.6667, 0.3333, 0.5000], "Li"),  # Li2
    ([0.0000, 0.0000, 0.1715], "P"),
    ([0.3237, 0.0000, 0.2500], "S"),
]))
spacegroup = 162  # P-31m (planar Li4P2S6 structure)
li4p2s6_structure = Structure.from_spacegroup(spacegroup, lattice, species,
                                              positions, tol=1.0E-3)
```

Calculating the ground state structure the ionic positions as well as the cell is allowed to relax (*ISIF=3*). However, in the following a loose convergence criterion for the remaining ionic forces of *0.1 eV / atom* is applied to speed up the calculations. In particular, the following *INCAR* parameters are used as calculation inputs:

```
incar_params = {
    'ALGO': 'Fast',
    'IBRION': 1,
    'ISIF': 3,
    'EDIFF': 1.0e-6,
    'EDIFFG': -0.1,
    'ENCUT': 420.0,
    'POTIM': 0.55,
    'SIGMA': 0.1,
    # override VASP's NSW=0 default which would abort after the first ionic step
    'NSW': 99,
}
```

For the required k-points a gamma centered grid is employed:

```
kpoint_params = {
   'mode': 'gamma',
   'kpoints': [3, 3, 2],
}
```

With all *INCAR*, *KPOINTS* and structure (i.e. *POSCAR*) inputs being defined we can now initialize the required inputs for the calculation. Here, AiiDA's `DataFactory()` function is used to load the corresponding data types provided by the plugin using their defined entry points:

```
from aiida.plugins import DataFactory

# import the required input datatypes
VaspIncarData = DataFactory('cusp.incar')
VaspKpointData = DataFactory('cusp.kpoints')
VaspPoscarData = DataFactory('cusp.poscar')
VaspPotcarData = DataFactory('cusp.potcar')
```

(continues on next page)

(continued from previous page)

```
# setup calculation inputs
incar = VaspIncarData(incar=incar_params)
kpoints = VaspKpointData(kpoints=kpoint_params)
poscar = VaspPoscarData(structure=li4p2s6_structure)
potcar = VaspPotcarData.from_structure(poscar, 'pbe')
```

Finally, the relaxation of the created $Li_4P_2S_6$ structure can be started by loading the calculation plugin and connecting the defined inputs to the calculation object,

```
from aiida.orm import Code
from aiida.plugins import CalculationFactory
from aiida.engine import submit

# load the VASP and Custodian codes
vasp_code = Code.get_from_string("vasp_5.4.1_openmpi_4.0.3_scalapack_2.1.0_
↪vtst@CompMPI")
custodian_code = Code.get_from_string("custodian_2020427@CompMPI")

# Setup and run the relaxation
VaspRelax = CalculationFactory('cusp.vasp').get_builder()
resources = {'tot_num_mpiprocs': 4, 'num_machines': 1}
VaspRelax.metadata.options.resources = resources
VaspRelax.code = vasp_code
VaspRelax.custodian.code = custodian_code
VaspRelax.custodian.handlers = ["VaspErrorHandler"]
VaspRelax.incar = incar
VaspRelax.poscar = poscar
VaspRelax.kpoints = kpoints
VaspRelax.potcar = potcar
# submit the calculation
node = submit(VaspRelax)
```

To check if the calculation was submitted to the daemon, simply run `verdi process list` and check that the submitted calculation is listed as process in the running state:

```
$ verdi process list
  PK  Created    Process label         Process State    Process status
----  ---------  --------------------  ---------------  ------------------------------
↪---------
1692  6s ago     VaspCalculation        Waiting        Monitoring scheduler: job␣
↪state RUNNING
```

**Note:** In the example output shown above you can see that the process was submitted and is currently running as process with associated *PK=1692*. Remember the shown process ID associated with the calculation since it is needed for the next step.

### 4.2.3 Relaxing the NEB Path Endpoints

In the previous step the ground state structure of $Li_4P_2S_6$ was calculated featuring a full cell relaxation. Using the relaxed structure of this calculation we can now setup the endpoints defining the NEB path of interest. In this example the added, interstitial $Li^+$ is assumed to migrate in between two neighboring Li lattice sites. Thus, the positions defining the start and the end of the migration path are given by

```
positions = [
    [0.3333, 0.3333, 0.4500],  # initial Li+ position (start of NEB path)
    [0.6667, 0.6667, 0.5500],  # final Li+ position (end of NEB path)
]
```

Since the migration of an interstitial Li ion is calculated, the Li ion has to be included as additional atom. Using the relaxed structure of the previous calculation two new structures are generated giving the complete structure with the additional Li ion before and after the migration took place:

```
from aiida.orm import load_node
# load the relaxed Li4P2S6 structure without additional Li-ion
relaxed_structure = load_node(1692).outputs.parsed_results.contcar.get_structure()
# create two new structures with an additional Li-ion sitting at the
# initial NEB-path and final NEB-path position
endpoints = []
for position in positions:
   structure = relaxed_structure.copy()
   structure.append("Li", position)
   endpoints.append(structure)
```

In order to relax the created structure containing the additional Li-ions we use the same *INCAR*, *KPOINTS* and *POTCAR* parameters used for the relaxation of the initial, defect free unit cell. However, now no cell relaxation is allowed and only ionic positions are considered as variable degree of freedom. In addition, a constant background charge is added to the calculation to account for the *+1* charge of the added Li-ion, thus:

```
incar_params.update({'ISIF': 2})
incar_params.update({'NELECT': 50})
incar = VaspIncarData(incar=incar_params)
```

With the updated incar we can again submit the relaxation of both endpoint structures to the daemon, equivalent to the procedure shown for the unit cell structure before.

```
# shared settings
VaspRelaxEndpoint = CalculationFactory('cusp.vasp').get_builder()
resources = {'tot_num_mpiprocs': 4, 'num_machines': 1}
VaspRelaxEndpoint.metadata.options.resources = resources
VaspRelaxEndpoint.code = vasp_code
VaspRelaxEndpoint.custodian.code = custodian_code
VaspRelaxEndpoint.custodian.handlers = ["VaspErrorHandler"]
VaspRelaxEndpoint.incar = incar
VaspRelaxEndpoint.kpoints = kpoints
# individual settings per endpoint, i.e. the structure
for endpoint in endpoints:
    VaspRelaxEndpoint.poscar = VaspPoscarData(structure=endpoint)
    VaspRelaxEndpoint.potcar = VaspPotcarData.from_structure(endpoint, 'pbe')
    node = submit(VaspRelaxEndpoint)
```

Again, using the `verdi process list` command to show the active processes should now output two calculations corresponding to the submitted relaxations for the two endpoints.

```
$ verdi process list
  PK  Created    Process label         Process State    Process status
----  ---------  --------------------  ---------------  ------------------------------
↪---------
1722  15s ago    VaspCalculation        Waiting        Monitoring scheduler: job␣
↪state RUNNING
1727  15s ago    VaspCalculation        Waiting        Monitoring scheduler: job␣
↪state RUNNING
```

(continues on next page)

(continued from previous page)

**Note:** The relaxed endpoint structures generated by those calculations are used to interpolate the NEB path fed into the final NEB calculation. So, once again, remember the process IDs displayed in this output, i.e. *1722* and *1727*.

### 4.2.4 Running the NEB Calculation

To setup the final NEB calculation we first need to find intermediate images by interpolating the path between the defined NEB path endpoints. In the following this interpolation step is shown based on the interpolation method naturally implemented in the `Structure` class. Using the relaxed endpoints of the previous step a NEB path containing a single intermediate image is generated running the following code.

```
# get the start and final node of the NEB path (previously relaxed)
struct_start = load_node(1722).outputs.parsed_results.contcar.get_structure()
struct_final = load_node(1727).outputs.parsed_results.contcar.get_structure()
# interpolate NEB path between struct_start and struct_final featuring a
# single intermediate image
neb_path_struct = struct_start.interpolate(struct_final, nimages=2)
# transform list of pymatgen structures to list of Poscar input
# structures
neb_path_poscar = [VaspPoscarData(structure=s) for s in neb_path_struct]
```

The list `neb_path_poscar` now contains three *aiida_cusp.data.VaspPoscarData* structures defining the NEB path of the interstitially migrating Li-ion as shown in the following image:

In order to combine the single structures to a connected NEB path in the calculation we need to tell VASP that it has to run a NEB calculation. To this end additional NEB parameters have to be added to the *INCAR* parameters of the previous endpoint relaxation:

```
incar_params.update({'IMAGES': 1})  # a single intermediate image
incar_params.update({'LCLIMB': True})  # use the climbing image algorithm
incar_params.update({'SPRING': -5.0})  # spring force of the nudged elastic band
incar = VaspIncarData(incar=incar_params)
```

Setting up the NEB calculation using the *VaspCalculation* is straightforward and does not differ from the setup of a regular calculation except for one point. Instead of a single structure passed to the calculation via the *inputs.poscar* option, NEB calculations expect a dictionary of node labels and corresponding structures defining the NEB path passed to the calculation via the *inputs.neb_path* option. For the defined NEB path in this example the path is passed to the calculation as follows.

```
VaspNeb = CalculationFactory('cusp.vasp').get_builder()
VaspNeb.neb_path = {
   'node_00': neb_path_poscar[0],  # endpoint1: the intial position
   'node_01': neb_path_poscar[1],  # intermediate: the interpolated intermediate image
   'node_02': neb_path_poscar[2],  # endpoint2: the final position
}
```

All other expected parameters are defined in the same way as for regular VASP calculations, i.e.

```
resources = {'tot_num_mpiprocs': 4, 'num_machines': 1}
VaspNeb.metadata.options.resources = resources
```

(continues on next page)

(continued from previous page)

```
VaspNeb.code = vasp_code
VaspNeb.custodian.code = custodian_code
VaspNeb.custodian.handlers = ["VaspErrorHandler"]
VaspNeb.incar = incar
VaspNeb.kpoints = kpoints
VaspNeb.potcar = potcar
# after setup: submit the calculation
node = submit(VaspNeb)
```

which sets up the remaining inputs and submits the NEB calculation to the daemon:

```
$ verdi process list
  PK  Created    Process label         Process State    Process status
----  ---------  --------------------  ---------------  --------------------------------
 ↪---------
1747  5s ago     VaspCalculation        Waiting        Monitoring scheduler: job␣
 ↪state RUNNING
```

### 4.2.5 Plotting the Calculated NEB Path

Usually the NEB calculations are analyzed based on the generated OUTCAR files. However, only a single OUTCAR file is generated by the NEB calculation started here, i.e. for the intermediate image, while the start and final structure are held fixed. Thus, to analyze the NEB calculation the NEB calculation outputs for the intermediate image as well as the outputs of the relaxation runs for the endpoints (containing the missing OUTCAR files of the fixed endpoints) are required. Since all calculations were done using the AiiDA framework and each calculation is identified by an associated id this is not a very complicated task and both, relaxed structures as well as the outcar files can be retrieved very easily:

```
from aiida.orm import load_node
# OUTCARs for the initial, intermediate and final NEB position
# Here the initial and final OUTCARs are taken from the endpoint
# relaxations while the intermediate OUTCAR is taken from the performed
# NEB calculation!
outcars = [
    load_node(1722).outputs.parsed_results.outcar.get_outcar(),
    load_node(1747).outputs.parsed_results.node_01.outcar.get_outcar(),
    load_node(1727).outputs.parsed_results.outcar.get_outcar(),
]

# CONTCARs for the initial, intermediate and final NEB position
# Here the initial and final CONTCARs are taken from the endpoint
# relaxations while the intermediate CONTCAR is taken from the performed
# NEB calculation!
structures = [
    load_node(1722).outputs.parsed_results.contcar.get_structure(),
    load_node(1747).outputs.parsed_results.node_01.contcar.get_structure(),
    load_node(1727).outputs.parsed_results.contcar.get_structure(),
]
```

With all *OUTCAR* and *CONTCAR* outputs being loaded for all NEB path nodes we can now simply analyze and plot the minimum energy path associated with the chosen NEB migration path. Using the `NEBAnalysis` module:

```
from pymatgen.analysis.transition_state import NEBAnalysis
```

(continues on next page)

(continued from previous page)

```
neb_path_analyzer = NEBAnalysis.from_outcars(outcars, contcars)
neb_path_plot = neb_path_analyzer.get_plot()
neb_path_plot.show()
```

which should output the minimum energy path similar to the plot shown below.

### 4.2.6 Copy-and-Paste

**Note:** In the following copy and paste-able code snippets are given for the calculation steps discussed in this tutorial. These calculation steps contain:

1. *Initial relaxation of the structure*
2. *Setup and relaxation of the NEB path endpoints*
3. *The actual NEB calculation*
4. *Final analysis of the generated minimum energy path*

In order to use the snippets given below, copy the code snippets to four different files and subsequently execute the files in the discussed order using `verdi run`. After each submitted calculation is finished do not forget to update the code-labels and the calculation IDs before running the next calculation!

#### 1. Relax Initial Structure

```
from pymatgen import Structure, Lattice, PeriodicSite
from aiida.orm import Code
from aiida.plugins import CalculationFactory, DataFactory
from aiida.engine import submit

# the VASP and Custodian codes to be used for the calculation
vasp_code_label = "your_vasp_vtst_code_label"
custodian_code_label = "your_custodian_code_label"

# define all input datatypes
VaspIncarData = DataFactory('cusp.incar')
VaspKpointData = DataFactory('cusp.kpoints')
VaspPoscarData = DataFactory('cusp.poscar')
VaspPotcarData = DataFactory('cusp.potcar')

#
# Setup the Li4P2S6 structure
#
# lattice vector lengths
a = 6.10622  # a = b
c = 6.61513
lattice = Lattice.hexagonal(a=a, c=c)
# Structure setup with P-31m spacegroup
positions, species = list(zip(*[
    ([0.3333, 0.6667, 0.0000], "Li"),  # Li1
    ([0.6667, 0.3333, 0.5000], "Li"),  # Li2
```

(continues on next page)

(continued from previous page)

```
    ([0.0000, 0.0000, 0.1715], "P"),
    ([0.3237, 0.0000, 0.2500], "S"),
]))
spacegroup = 162  # P-31m (planar Li4P2S6 structure)
structure = Structure.from_spacegroup(spacegroup, lattice, species, positions,
                                      tol=1.0E-3)
# Setup the input parameters
incar_params = {
    'ALGO': 'Fast',
    'IBRION': 1,
    'ISIF': 3,
    'EDIFF': 1.0e-6,
    'EDIFFG': -1.0e-1,
    'ENCUT': 420.0,
    'POTIM': 0.55,
    'SIGMA': 0.1,
    'NSW': 99,
}
incar = VaspIncarData(incar=incar_params)
poscar = VaspPoscarData(structure=structure)
kpoints = VaspKpointData(kpoints={'mode': 'gamma', 'kpoints': [3, 3, 2]})
# Setup the caclulation inputs
VaspRelax = CalculationFactory('cusp.vasp').get_builder()
resources = {'tot_num_mpiprocs': 4, 'num_machines': 1}
VaspRelax.metadata.options.resources = resources
VaspRelax.code = Code.get_from_string(vasp_code_label)
VaspRelax.incar = incar
VaspRelax.poscar = poscar
VaspRelax.kpoints = kpoints
VaspRelax.potcar = VaspPotcarData.from_structure(poscar, 'pbe')
# custodian error correction
VaspRelax.custodian.code = Code.get_from_string(custodian_code_label)
VaspRelax.custodian.handlers = ["VaspErrorHandler"]
# submit the calculation
node = submit(VaspRelax)
# print out the PK of the submitted job
print("Submitted Relaxtion of Li4P2S6 with PK: {}".format(node.pk))
```

### 2. Relax the NEB Path Endpoints

```
from aiida.orm import Code, load_node
from aiida.plugins import CalculationFactory, DataFactory
from aiida.engine import submit

# Add the node-ID for the previously performed unit cell relaxation
previous_calc_id =

# code labels
vasp_code_label = "your_vasp_vtst_code"
custodian_code_label = "your_custodian_code"

# define all input datatypes
VaspIncarData = DataFactory('cusp.incar')
VaspKpointData = DataFactory('cusp.kpoints')
VaspPoscarData = DataFactory('cusp.poscar')
```

(continues on next page)

(continued from previous page)

```
VaspPotcarData = DataFactory('cusp.potcar')

# load relaxed structure from calculation
outputs = load_node(previous_calc_id).outputs
relaxed_structure = outputs.parsed_results.contcar.get_structure()
# NEB path positions start / stop
positions = [
    [0.3333, 0.3333, 0.4500],  # NEB path start position
    [0.6667, 0.6667, 0.5500],  # NEB path end position
]
# the structures with the initial and final interstitial Li position
endpoints = []
for position in positions:
    structure = relaxed_structure.copy()
    structure.append("Li", position)
    endpoints.append(structure)
# Setup the calculation Inpts
incar_params = {
    'ALGO': 'Fast',
    'IBRION': 1,
    'ISIF': 2,
    'EDIFF': 1.0e-6,
    'EDIFFG': -1.0e-1,
    'ENCUT': 420.0,
    'POTIM': 0.55,
    'SIGMA': 0.1,
    'NELECT': 50,  # Background charge for added Li+ interstitial
    'NSW': 99,
}
incar = VaspIncarData(incar=incar_params)
kpoints = VaspKpointData(kpoints={'mode': 'gamma', 'kpoints': [3, 3, 2]})
# Setup the calculation
VaspRelaxEndpoint = CalculationFactory('cusp.vasp').get_builder()
resources = {'tot_num_mpiprocs': 4, 'num_machines': 1}
VaspRelaxEndpoint.metadata.options.resources = resources
VaspRelaxEndpoint.code = Code.get_from_string(vasp_code_label)
VaspRelaxEndpoint.incar = incar
VaspRelaxEndpoint.kpoints = kpoints
# custodian error correction
VaspRelaxEndpoint.custodian.code = Code.get_from_string(custodian_code_label)
VaspRelaxEndpoint.custodian.handlers = ["VaspErrorHandler"]
# Run the NEB path endpoint relaxations in parallel, i.e. submit both
# calculations at once
for endpoint in endpoints:
    VaspRelaxEndpoint.poscar = VaspPoscarData(structure=endpoint)
    VaspRelaxEndpoint.potcar = VaspPotcarData.from_structure(endpoint, 'pbe')
    node = submit(VaspRelaxEndpoint)
    ## print out the PK of the submitted job
    print("Submitted VaspSiRelax with PK: {}".format(node.pk))
```

### Running the NEB Calculation

```
from aiida.orm import Code
from aiida.plugins import CalculationFactory, DataFactory
from aiida.engine import submit
```

(continues on next page)

(continued from previous page)

```
# Add the node-ID for the previously performed endpoint relaxations for the
# initial and final endpoint
initial_endpoint_calc_id =
final_endpoint_calc_id =

# code labels
vasp_code_label = "your_vasp_vtst_code"
custodian_code_label = "your_custodian_code"

# define all input datatypes
VaspIncarData = DataFactory('cusp.incar')
VaspKpointData = DataFactory('cusp.kpoints')
VaspPoscarData = DataFactory('cusp.poscar')
VaspPotcarData = DataFactory('cusp.potcar')

# get the start and final node of the NEB path (previoulsy relaxed)
outputs_start = load_node(initial_endpoint_calc_id).outputs
struct_start = outputs.parsed_results__contcar.get_structure()
outputs_final = load_node(final_endpoint_calc_id).outputs
struct_final = outputs.parsed_results__contcar.get_structure()

# build the neb path using pymatgen featuring a single intermediate image
neb_path_struct = struct_start.interpolate(struct_final, nimages=2)
neb_path_poscar = [VaspPoscarData(structure=s) for s in neb_path_struct]

# Setup the calculation Inpts
incar_params = {
    'ALGO': 'Fast',
    'IBRION': 1,
    'ISIF': 2,
    'EDIFF': 1.0e-6,
    'EDIFFG': -1.0e-1,
    'ENCUT': 420.0,
    'POTIM': 0.55,
    'SIGMA': 0.1,
    'NELECT': 50,  # Background charge for added Li+ interstitial
    'NSW': 99,
    # NEB parameters
    'IMAGES': 1,  # a single intermediate image
    'LCLIMB': True,
    'SPRING': -5.0,
}
neb_incar = VaspIncarData(incar=incar_params)
neb_path = {
    'node_00': neb_path_poscar[0],
    'node_01': neb_path_poscar[1],
    'node_02': neb_path_poscar[2],
}
kpoints = VaspKpointData(kpoints={'mode': 'gamma', 'kpoints': [3, 3, 2]})
# simply use the first structure to initialize the potcars
potcar = VaspPotcarData.from_structure(neb_path_poscar[0], 'pbe')
# Setup the caclulation inputs
VaspNeb = CalculationFactory('cusp.vasp').get_builder()
resources = {'tot_num_mpiprocs': 4, 'num_machines': 1}
VaspNeb.metadata.options.resources = resources
VaspNeb.code = Code.get_from_string(vasp_code_label)
```

(continues on next page)

(continued from previous page)

```
VaspNeb.incar = neb_incar
VaspNeb.neb_path = neb_path
VaspNeb.kpoints = kpoints
VaspNeb.potcar = potcar
# custodian error correction
VaspNeb.custodian.code = Code.get_from_string(custodian_code_label)
VaspNeb.custodian.handlers = ["VaspErrorHandler"]
# submit the NEB calculation
node = submit(VaspNeb)
# print out the PK of the submitted job
print("Submitted VaspSiRelax with PK: {}".format(node.pk))
```

### NEB Path Analysis

```
from aiida.orm import load_node
from pymatgen.analysis.transition_state import NEBAnalysis

# Node-IDs for the previous endpoint calculations
endpoint_start_calc_id =
endpoint_final_calc_id =
# Node-ID for the actual NEB calculation
neb_calc_id =
# initial, intermediate and final NEB node OUTCARs
outcars = [
    load_node(endpoint_start_calc_id).outputs.parsed_results__outcar.get_outcar(),
    load_node(neb_calc_id).outputs.parsed_results__node_01__outcar.get_outcar(),
    load_node(endpoint_final_calc_id).outputs.parsed_results__outcar.get_outcar(),
]
# initial, intermediate and final NEB node CONTCARs
structures = [
    load_node(endpoint_start_calc_id).outputs.parsed_results.contcar.get_structure(),
    load_node(neb_calc_id).outputs.parsed_results.node_01.contcar.get_structure(),
    load_node(endpoint_final_calc_id).outputs.parsed_results.contcar.get_structure(),
]
# populate pymatgen NEB analyzer and plot the calculated minimumenergy path
neb_analyzer = NEBAnalysis.from_outcars(outcars, structures)
neb_plot = neb_analyzer.get_plot()
neb_plot.show()
```

CHAPTER 5

# Workflow Tutorials

This section contains tutorials using the plugin's implemented workflows to illustrate their setup and usage for VASP calculations using this plugin and AiiDA.

**Note:** Currently there are no workflow tutorials available!

CHAPTER 6

# Calculator Classes

This section contains the documentation for the available calculator classes. In the following, detailed information on all calculator classes for running VASP calculations, that are implemented by the plugin, is given.

**Note:** Although this may change in the future, currently, only a single calculation class is implemented in this plugin. However, the VASP calculations run by this class is solely controlled by the input parameters you provide for the calculation (similar to running VASP manually) Thus, although providing a single calculator only, the plugin is capable of running **all** kind of VASP calculations! This, of course, also includes AIMD and NEB calculations (See the calculator class documentation for detailed information)

## 6.1 Vasp Calculator (`cusp.vasp`)

The *VaspCalculation* is available using the `cusp.vasp` entry point allows you to perform all kind of VASP calculation ranging from simple calculations to AIMD and even NEB calculations. Similar to VASP itself the behavior of the calculation is entirely controllable by the passed input parameters defined for *KPOINTS*, *INCAR* and the *POTCAR* pseudo-potential. (Please refer to the *potcar command documentation* how to add pseudo-potentials for VASP to the database) Whether a simple calculation or complex NEB calculation is run is decided by the calculation object based on the given structural inputs (i.e. `poscar` or `neb_path`). Note that by default the output files containing calculation results are parsed using the *VaspFileParser class*. Of course the default parser may be changed to a different parser class using the calculation's *metadata.options.parser_class* option with corresponding parser options passed to the parser through the *metadata.options.parser_settings* option. For an overview of the available parsers and the accepted settings please refer to the *Parser section*. Despite the already mentioned, optional parser options the calculator accepts several other (non-)optional inputs that are used to setup the actual VASP calculation. In the following these calculation inputs are discussed and, for clarity, have been clustered into three main input groups that can be set with the calculation class:

- *Vasp Calculation Inputs:* All inputs found in this group are the direct VASP inputs that are used to setup the VASP calculation, i.e. *KPOINTS*, *INCAR*, etc.
- *Custodian Inputs:* Set of input parameters for the Custodian executable needed to run an error corrected VASP calculation through the Custodian framework. If a Custodian code and corresponding error handlers are defined

error correction is enabled by wrapping the previously defined VASP calculation in Custodian. However, note that all inputs listed in this group are completely optional!

- *Restart Options:* Available options when a calculation is restarted from a previous run using the remote folder of the parent calculation.

### 6.1.1 Calculator Inputs

#### VASP Calculation Inputs:

---

**Note:** Note that for the `VaspCalculation` class the input options *inputs.poscar* and *inputs.neb_path* may not be set simultaneously. If both are set at the same time an error will be raised and the calculation will fail!

---

- **code** (`aiida.orm.Code`) – VASP code used to run the calculation.
- **incar** (`aiida_cusp.data.VaspIncarData`) – i INCAR data input defining the calculation parameters for the VASP calculation (see *INCAR*)
- **kpoints** (`aiida_cusp.data.VaspKpointData`) – KPOINTS data input defining the k-point grid to be used for the calculation (see *KPOINTS*)
- **poscar** (`aiida_cusp.data.VaspPoscarData`) – Structure data input required for regular VASP or AIMD simulations (see *POSCAR*)
- **neb_path** (`dict`) – Structure data input required for VASP NEB calculations.

---

**Note:** For NEB calculations a dictionary of multiple structures defining the NEB path is expected as input to the *neb_path* option. Here, every structure has to passed under the corresponding key *'node_XX'* where *'XX'* represents the name of the NEB sub-folder the image data is written to. As an example consider the following input:

```
inputs.neb_path = {'node_00': poscar_1, 'node_01': poscar_2, 'node_02': poscar_3}
```

Then, upon submission of the calculation the contents of *poscar_1* are written to the calculation's *'00'* sub-folder, the contents of *poscar_2* to the *'01'* sub-folder and so on.

---

- **potcar** (`dict`) – The VASP pseudo-potentials to be used for the calculation. Potentials are expected to be defined as dictionary containing the structure's elements as keys and the `aiida_cusp.data.VaspPotcarData` of the potential to be used for that element

---

**Note:** There is no need to build this dictionary manually and it is highly recommended to setup the *options.potcar* inputs using the `aiida_cusp.data.VaspPotcarData.from_structure()` method. Please refer to the *VaspPotcarData documentation* for more details in how this method is used to generate the appropriate inputs.

---

#### Custodian Settings:

Options passed to the Custodian executable if a custodian code is set for the *custodian.code* option. (Also refer to the *Custodian section* for more details on the available settings)

**Note:** If no settings are defined for Custodian the VASP code is not wrapped by Custodian (i.e. the *vasp* executable defined by the VASP code set for the *code* input is called directly)

- **custodian.code** (`aiida.orm.Code`) –
- **custodian.handlers** (`list` or `dict`) – Optional input option defining the error handlers connected to the calculation. For a complete list of available error handlers that may be set here please refer to the *handler section* in the Custodian documentation of this plugin. (optional, default: `{}`)

  **Warning:** Be advised that setting no error handlers for Custodian is perfectly fine, however, defining a Custodian code without setting any handlers will disable the error correction.

- **custodian.settings** (`dict`) – Optional dictionary containing the settings that should be set to customize the behavior of the Custodian executable. If no settings are passed (default) then the plugin's default settings for Custodian will be used. For a complete list of available settings that may be set here and their corresponding default values, please refer to the *settings section* in the Custodian documentation of this plugin. (default: `{}`)

#### Restart Options:

- **restart.folder** (`aiida.orm.RemoteData`) – Remote folder of the parent calculation from which the calculation is restarted. All files in the remote folder will be copied to the restarted calculation's folder and are used as input to the new calculation.

  **Note:** For restarted calculations the previous used *INCAR* and *KPOINTS* data can be ignored by setting new parameters through the *inputs.incar* and *inputs.kpoints* options. Note, however, that setting an alternative structure or using different pseudo-potentials is not allowed for a restarted calculation which will raise an error.

- **restart.contcar_to_poscar** (`bool`) – If this option is set to *True* the *POSCAR* file of the restarted calculations will be replaced with the parent calculation's *CONTAR* contents. (optional, default: *True*)

### 6.1.2 Default Calculator Outputs

After the calculation has finished, parsed outputs are available via the calculation nodes *outputs.parsed_results* key. Note that the contents that are stored to this output key of course depend the parser plugin used for the calculation (see the *Parsers section*). By default the `VaspCalculation` class uses the *VaspFileParser* to parse the generated results. Note that if no additional parser options are passed to this parser class only the *CONTCAR*, *vasprun.xml* and *OUTCAR* files will be avilable in the calculation's outputs.

**Note:** Files not generated as a result of the calculation, i.e. the logged scheduler and stdout / stderr outputs as well as the used submit script and custodian inputs are not stored under the *outputs.parsed_results* key. You can find these files in the calculation's retrieved folder located under the *output.retrieved* key.

CHAPTER 7

# Custodian

One of the most distinct characteristics of this plugin is its ability to wrap VASP calculations in Custodian to enable error correction on the runtime level. To this end, calculators implemented by this plugin, in addition to the VASP code, accept a second, optional custodian code object under the `inputs.custodian.code` input. Similarly to the actual VASP code input, defined by the `inputs.code` option, additional settings for the Custodian code may be passed through the `inputs.custodian.settings` option to customize Custodian's behavior. Possible settings that can be set for calculations employing the Custodian executable are documented in the *Custodian settings* section. Error handlers defining which errors are monitored and corrected can be defined for each calculation via the `inputs.custodian.handlers` input. The available set of error handlers that can be used with this plugin is documented in the *handler section*.

**Warning:** Note that all custodian inputs, i.e. `custodian.code`, `custodian.settings` and `custodian.handlers` are entirely optional and do not need to be specified. However, be advised that setting a custodian code via the `custodian.code` input without specifying any handler does not enable any error correction for the calculation! In that case custodian will simply run the VASP calculation with its default settings, ignoring any emerging errors.

## 7.1 Custodian Settings

Settings changing the default behavior of the custodian executable can be passed for every VASP calculation via the `inputs.custodian.settings` input. Inputs passed to the calculation by this option are expected to be of type `dict` containing the parameters and the corresponding values to be set as key-value pairs. For instance, if you are using a builder to setup the calculation the settings may be given as

```
builder.inputs.custodian.settings = {
  'max_errors': 10,
  'monitor_freq': 5,
  ...
}
```

In the following all available options that may be defined and the corresponding default options that are used in case they are undefined are shown in the following.

**Available Options:**

- **max_errors** (`int`) – This sets the maximum number of errors that may occur for a single calculation before terminating the calculation (default: *10*)
- **polling_time_step** (`int`) – Seconds between two consecutive checks for the calculation being completed (default: *10*)
- **monitor_freq** (`int`) – Number of performed polling steps before the calculation is checked for possibly encountered errors (default: *30*)
- **skip_over_errors** (`bool`) – Set this option to *True* to skip over any failed error handler (default: *False*)

## 7.2 Custodian Error Handlers

Custodian uses different error handlers to check and correct for different failures of VASP calculations. Thus, one can individually decide which error should be corrected and which error should fail the calculation and connect the corresponding handlers. Error handlers for the calculations run with the calculation classes implemented in this plugin can be passed via the `inputs.custodian.handlers` input. Here, the input is expected to be a dictionary of the following form:

```
handler_inputs = {
    'HandlerName1': {
        'option': 'value',
        ...
        },
    'HandlerName2': {
        'option', 'value',
        ...
        }
    ...
}
```

In that case every dictionary entry corresponds to an individual handler specified by its name and the corresponding handler options to be used for it. Note if an empty dictionary is passed as option to any handler, i.e. *'HandlerName': {}*, the plugin's default options are used for the handler. Since one often wants to use the defined default for **all** handlers in the first place handlers may also be passed as a simple list of handler names. In that case the default values are used for every defined handler and the example above would simplify to

```
handler_inputs = ['HandlerName1', 'HandlerName2', ...]
```

In the following all handlers available for VASP calculations are described in more detail and the available settings and their used defaults are shown. (For more complete overview please refer to the original Custodian documentation)

### 7.2.1 *'VaspErrorHandler'*

Most basic error handler for VASP calculations checking for the most common VASP errors logged to the stdout output (i.e. Call to ZHEGV failed, TOO FEW BANDS, etc.) (See also VaspErrorHandler)

**Handler Settings:**

- **natoms_large_cell** (`int`) – Number of atoms above which a structure is considered as large. This has implications on the strategy of resolving some kind of the errors (for instance whether LREAL should be set to True or False, etc.) (default: *100*)
- **errors_subset_to_catch** (`list`) – List of errors that will be caught (other errors not in the list are ignored!). If set to *None* all errors known to the handler will be detected. (default: `None`)

---

**Note:** A complete list of errors known to the handler that may be passed in this list can be directly found from the VaspErrorHandler:

```
>>> from custodian.vasp.handlers import VaspErrorHandler
>>> known_errors = list(VaspErrorHandler.error_msgs.keys())
>>> for error in known_errors:
...     print(error)
...
tet
inv_rot_mat
brmix
subspacematrix
tetirr
incorrect_shift
real_optlay
rspher
dentet
too_few_bands
triple_product
rot_matrix
brions
pricel
zpotrf
amin
zbrent
pssyevx
eddrmm
edddav
grad_not_orth
nicht_konv
zheev
elf_kpar
elf_ncl
rhosyg
posmap
point_group
```

---

### 7.2.2 *'FrozenJobErrorHandler'*

Considers a calculation as frozen if the output to stdout is not updated for a defined amount of time and restarts the job if frozen. (See also FrozenJobErrorHandler)

**Warning:** If using this handler do not set the timeout too low for demanding calculations with many atoms and / or a dense kpoint grid!

**Handler Settings:**

- **timeout** (`int`) – Seconds without activity on the stdout output after which the job is considered as frozen (default: *21600*)

### 7.2.3 *'PotimErrorHandler'*

Check for positive energy changes in electronic steps (dE) larger than the defined threshold and reduce POTIM parameter accordingly.

**Handler Settings:**

- **dE_threshold** (`float`) – Maximum threshold (in eV) for energy changes between consecutive electronic steps. For energy changes larger than the defined value the handler will restart the calculation with reduced POTIM parameter (default: *1.0*)

### 7.2.4 *'NonConvergingErrorHandler'*

Check if NELM is reach for `nionic_steps` consecutive ionic steps and correct by first switching to more stable algorithms (ALGO) and secondly adjusting the mixing parameters (AMIX, BMIX, BMIN). (See also NonConvergingErrorHandler)

**Handler Settings:**

- **nionic_steps** (`int`) – Number of consecutive ionic steps with the maximum number of electronic steps being reached before considered an error. (default: *10*)

### 7.2.5 *'DriftErrorHandler'*

Checks if the final drift exceeds the force convergence criterion defined by *EDIFFG* tag and restarts if true. (See also DriftErrorHandler)

**Handler Settings:**

- **max_drift** (`float`) – Defines the maximal acceptable drift. If set to *None* the value is set to the supplied value defined by *EDIFFG* in the *INCAR* parameters. (default: *None*)
- **to_average** (`int`) – Demand at least that many steps to calculate the average drift (default: *3*)
- **enaug_multiply** (`int`) – Value used to multiply the value of *ENAUG* found from the *INCAR* parameters when restarting the calculation (default: *2*)

### 7.2.6 *'WalltimeHandler'*

Write a STOPCAR to the calculation folder if the calculation's runtime approaches the defined wall time. (See also WalltimeHandler)

**Handler Settings:**

- **wall_time** (`int`) – Total wall time of the job in seconds. (If running using the PBS scheduler this value is retrieved from the PBS_WALLTIME environment variable if not set here) (default: *None*)
- **buffer_time** (`int`) – Buffer time in seconds between writing the STOPCAR and the total wall time is reached. Note that if the average time required to complete 3 ionic steps is larger the set buffer_time this value will be used as buffer_time. (default: *300*)
- **electronic_step_stop** (`bool`) – If set to *True* compare the defined buffer_time to the time required to complete electronic steps rather than ionic steps. (default: *False*)

### 7.2.7 *'StdErrHandler'*

Handler checking for common errors only written to stderr. (See also StdErrHandler)

**Handler Settings:**

- This handler does not provide any custom settings

### 7.2.8 *'AliasingErrorHandler'*

Corrects for aliasing (small wrap around) errors encountered for insufficient FFT grids. (See also AliasingErrorHandler)

**Handler Settings:**

- This handler does not provide any custom settings

### 7.2.9 *'UnconvergedErrorHandler'*

Check for both ionic and electronic convergence and restart the job with different strategies if convergence was not reached. (See also UnconvergedErrorHandler)

**Handler Settings:**

- This handler does not provide any custom settings

### 7.2.10 *'PositiveEnergyErrorHandler'*

Check for a positive absolute energy at the end of a calculation and restart with ALGO=Normal (Stops calculation is ALGO is already set to Normal). (See also PositiveEnergyErrorHandler)

**Handler Settings:**

- This handler does not provide any custom settings

### 7.2.11 *'MeshSymmetryErrorHandler'*

Check for symmetry errors and switch off symmetry (i.e. set ISYM=0) if reciprocal lattices and kpoint lattices belong to different classes of lattices. (See also MeshSymmetryErrorHandler)

**Handler Settings:**

- This handler does not provide any custom settings

### 7.2.12 *'LrfCommutatorErrorHandler'*

Checks for *LRF_COMMUTATOR* errors corrects the error by switching to finite-differences when calculating the cell-periodic derivative or orbitals (i.e. sets LPEAD=True). (See also LrfCommutatorErrorHandler)

**Handler Settings:**

- This handler does not provide any custom settings

# CHAPTER 8

# Parser Classes

In the following the parser classes implemented by this plugin are documented. In general, the implemented parser plugins are used to parse the calculation outputs generated by VASP calculations that have been performed using the plugins' calculation classes. Parsers can be individually set for each calculation through the `metadata.options.parser_name` option by passing the entry point name of the parser you intent to use.

**Note:** You can use the `verdi plugin list aiida.parsers` command to get a list of all currently installed parsers:

```
$ verdi plugin list aiida.parsers
Registered entry points for aiida.parsers:
* arithmetic.add
* cusp.default
* templatereplacer.doubler
```

Similarly to setting the parser name, additional parser settings that alter the parser behavior may be passed through the `metadata.options.parser_settings` option. Of course, the available `parser_settings` and generated calculation outputs depend on the parser class you chose to use for each calculation. For an overview of the generated outputs and available settings for each of the plugin's parser classes, please refer to the documentation of the individual parsers given below.

## 8.1 Vasp File Parser (`cusp.default`)

The *VaspFileParser* class is the default class used by each calculation if no alternative parser is set through the `metadata.options.parser_name` option. This parser does not apply any parser magic but simply checks the output files and adds them as gzip-compressed archives to the repository. Since no parsing of individual values is applied this parser class is suitable for all VASP calculations (i.e. normal calculations, AIMD as well as NEB). Of course, the set of files added for each calculation can be changes based on the available parser settings discussed in the following.,

### 8.1.1 Parser Settings

Parser settings are expected to be passed as dictionary to the calculations `metadata.options.parser_settings` option. The dictionary thereby accepts the settings show in the following.

**Warning:** Note that if the dictionary passed through the `metadata.options.parser_settings` contains any setting unknown to the parser the parsing will fail!

**Settings:**

- **parse_files** (`list`, optional) – List of filenames (i.e. *'OUTCAR'*, *'vasprun.xml'*, etc.) or wildcards (i.e. *'*CAR'*, *'*'*, etc.). Every file in the list of retrieved files matching a defined filename or a given wildcard is added to the calculation.

  **Note:** Mixing filenames and wildcards is allowed, i.e. *['*CAR', 'vasprun.xml', ...]* is perfectly fine.

  By default (i.e. if unset) only the files *'CONTCAR'*, *'OUTCAR'* and *'vasprun.xml'* are added to the outputs. (default: *['CONTCAR', 'OUTCAR', 'vasprun.xml']*)

- **fail_on_missing_files** (`bool`, optional) – Set this flag to *True* if the parsing should fail if a file defined in the `parse_files` list cannot be found in the retrieved outputs. (default: *False*)

### 8.1.2 Parser Outputs

All files defined by the `parse_files` options are added by the parser to the calculation as outputs available under the *outputs.parsed_results* key.

**Note:** Note that only the files *vasprun.xml*, *OUTCAR*, *WAVECAR*, *CHGCAR* and *CONTCAR* are added using their corresponding *output datatype*. All other files will be added using the `VaspGenericData` class (see also *generic datatype*).

Parsed files are registered in the output namespace using the corresponding lower case filename neglecting any suffixes. Thus, after parsing a stored *CONTCAR* file can be accessed from the stored calculation node via

```
calulation_node.outputs.parsed_results.contcar
```

while a parsed *vasprun.xml* file would be accessible (with the corresponding *.xml* suffix being replaced by *_xml*) via

```
calculation_node.outputs.parsed_results.vasprun_xml
```

In case of NEB calculation featuring files stored at different NEB sub-folders all files found inside the top-folder are added corresponding to the scheme above. However, files located in NEB sub-folders will be added to an individual namespace corresponding to the sub-folder it was found in prefixed with `node_`. As an example consider parsing the *CONTCAR* outputs of a NEB calculation located in the NEB calculation's three sub-folders *'00'*, *'01'* and *'02'*. Then each of the individual output files is accessible via the output links

```
calculation_node.outputs.parsed_results.node_00.contcar  # output 00/CONTCAR
calculation_node.outputs.parsed_results.node_01.contcar  # output 01/CONTCAR
calculation_node.outputs.parsed_results.node_02.contcar  # output 02/CONTCAR
```

**Note:** This scheme applies to all calculation output files found in NEB sub-folders which will also be added to the corresponding sub-namespace.

# CHAPTER 9

# Available Data Types

In this chapter the calculation input and output data types for VASP calculations as implemented by this plugin are documented. Here, input data types implement a representation of input parameters passed to a VASP calculation (i.e. corresponding to *INCAR*, *KPOINTS*, *POSCAR* and *POTCAR* files) that is compatible with and storable to AiiDA databases. Contrary, the implemented output data types are used to store the generated outputs of VASP calculations to the AiiDA database. In order to see how the input data types can be initialized and how you can interact with both calculation inputs and outputs, please refer to the individual documentation of each datatype given below.

**Note:** Note that VASP output files that are not listed here explicitly does not mean that the plugin does not know about them or does not know how to store or handle them. For such files simply no individual output datatype was implemented and those will be stored to the database using the implemented general *VaspGenericData* class.

## 9.1 INCAR (`cusp.incar`)

The *VaspIncarData* is the central object used to define and pass INCAR parameters to the VASP calculation object. This data class is published by the plugin using the `cusp.incar` entry point and can be loaded via AiiDA's `DataFactory()` function. Note that in order to use the class as input for VASP calculations it has to be first initialized with the desired input parameters. As discussed in the following, the parameters may be passed to the constructor either directly (i.e. as dictionary) or indirect via a pymatgen `Incar` instance.

### 9.1.1 Initializing the class

In general the constructor of the *VaspIncarData* class is of the following form:

```
VaspIncarData(incar=None)
```

**Arguments:**

- **incar** (`dict` or `pymatgen.io.vasp.inputs.Incar`) – The INCAR parameters used to run a VASP calculation. May be given as `Incar` object which is useful if you want to use the parameters directly from a

pymatgen set. Alternatively, if you do not use a pymatgen set the desired parameters may also be passed directly as `dict` of the form

```
incar_params = {
    'PARAM_1': VALUE_1,
    'PARAM_2': VALUE_2,
    ...
    'PARAM_N': VALUE_N,
}
```

In that case, the dictionary keys (`PARAM`) define the INCAR parameter to be set and the values (`VALUE`) the corresponding value.

### 9.1.2 Implemented Methods and Attributes

`VaspIncarData.`**`get_incar`**`()`

Create and return a `Incar` instance initialized from the node's stored incar data contents.

**Returns** a pymatgen Incar data instance

**Return type** `pymatgen.io.vasp.inputs.Incar`

### 9.1.3 Example

The following example illustrates how to initialize the *`aiida_cusp.data.VaspIncarData`* object from an INCAR parameter `dict`

```
>>> from aiida.plugins import DataFactory
>>> IncarData = DataFactory('cusp.incar')
>>> incar_params = {
...     'ALGO': 'Fast',
...     'EDIFF': 1.0E-6,
...     'EDIFFG': -0.01,
...     'LWAVE': False,
...     'MAGMOM': 56*[0.6],
... }
>>> incar = IncarData(incar=incar_params)
>>> print(incar)
uuid: dc37533e-9d70-4493-8c6f-f51a503cd3e5 (unstored)
>>> print(incar.get_incar())
ALGO = Fast
EDIFF = 1e-06
EDIFFG = -0.01
LWAVE = False
MAGMOM = 56*0.6
```

## 9.2 KPOINTS (`cusp.kpoints`)

Using the *`VaspKpointData`* the KPOINT parameters of a calculation can be passed to the VASP calculation object. This data class is published by the plugin using the `cusp.kpoints` entry point and can be loaded via AiiDA's `DataFactory()` function. To simplify the setup of different k-point grids (i.e. *Monkhorst*, *Gamma*, *Automatic*, etc.), the class is closely connected to Pymatgen's `Kpoints` class supporting multiple initialization modes. Which of the implemented initialization mode is used to generate the k-point data is decided by the plugin based on the type and set of parameters passed to the constructor (see the *following section*)

### 9.2.1 Initializing the class

In general the constructor of the *VaspKpointData* class accepts k-point settings (defining the k-point grid) and an optional structure input required for density based on k-point grid definition:

```
VaspKpointData(kpoints=None, structure=None)
```

**Arguments:**

- **kpoints** (dict or pymatgen.io.vasp.inputs.Kpoints) – The KPOINT parameters used to run a VASP calculation. May be given as pymatgen.io.vasp.inputs.Kpoints object which is useful if k-points parameters defined by a pymatgen set should be used. Alternatively (i.e. no pre-defined pymatgen set should be used) the desired k-point grid may also be set using a parameter dict. In that case the grid is generated using the class-internal methods depending on the type of parameters passed by the dict (See the *following section* for more details on the possible parameters and corresponding generation modes) kpoint parameters
- **structure** (optional, pymatgen.core.structure.Structure, pymatgen.io.vasp.inputs.Poscar or aiida.orm.StructureData) – Optional structure input for initialization modes that are based on a k-point density. This is only required if the KPOINT grid is initialized using the internal methods based on a passed kpoint density. (See the initialization modes discussed in the *following section* for more details)

### 9.2.2 Available initialization modes and their corresponding parameters

Passing a parameters dictionary to the *VaspKpointData* the passed dictionary may only contain the following keys:

- **mode** (str) – The initialization mode to be used for the KPOINT generation. In general the *VaspKpointData* class supports the four distinct initialization modes: auto, monkhorst, gamma and line
- **kpoints** (int, float or list) – Defining the actual KPOINT grid

  **Note:** The expected type passed for the kpoints depends on the initialization mode defined by the mode key.

- **shift** (list) – Shift the kpoint grid by the defined amount
- **sympath** (HighSymmKpath) – Path along high symmetry lines used in band structure calculations (i.e. only required if mode is set to line)

In the following the different initialization modes and expected parameters are discussed in more detail.

#### Mode: auto

Setting the mode to auto the KPOINT grid initialized automatically using a single integer. This corresponds to setting *Auto* in the KPOINT file. In this mode the expected input parameters passed in the input dictionary are:

- **mode** (str) – auto
- **kpoints** (int) – Integer used to automatically determine the grid's subdivisions
- **shift** (None) – Unused by this mode
- **sympath** (None) – Unused by this mode

**Example:**

```
>>> auto_mode_params = {
...     'mode': 'auto',
...     'kpoints': 100,
... }
>>> kpoints = VaspKpointData(kpoints=auto_mode_params)
>>> print(kpoints.get_kpoints())
Fully automatic kpoint scheme
0
Automatic
100
```

### Mode: monkhorst

Setting the mode to `monkhorst` calls the internal KPOINT grid initialization for Monkhorst grids. In this mode the expected input parameters passed in the input dictionary are:

- **mode** (`str`) – `monkhorst`
- **kpoints** (`list` or `float`) – Explicit 3x1 list of `int` defining the grid's subdivisions or a kpoint density of type:class:*float*

  **Note:** In case the kpoint grid is initialized from density the structure has to be passed to the constructor as well. However, the structure is not required for the initialization using an explicit kpoint grid.

- **shift** (`list`) – 3x1 list of `float` defining the kpoint grid shift applied to the grid

  **Note:** If the grid is initialized from a density (i.e. kpoints is of type `float`) any defined shift is ignored.

- **sympath** (`None`) – Unused by this mode

**Example for explicit kpoint grid:**

```
>>> monkhorst_mode_params = {
...     'mode': 'monkhorst',
...     'kpoints': [2, 2, 2],
... }
>>> kpoints = VaspKpointData(kpoints=monkhorst_mode_params)
>>> print(kpoints.get_kpoints())
Automatic kpoint scheme
0
Monkhorst
2 2 2
```

**Example for kpoint density**

```
>>> from pymatgen import Structure, Lattice
>>> lattice = Lattice.cubic(1.0)
>>> structure = Structure(lattice, ['H'], [[.0, .0, .0]])
>>> monkhorst_mode_params = {
...     'mode': 'monkhorst',
...     'kpoints': 10.0,
... }
>>> kpoints = VaspKpointData(kpoints=monkhorst_mode_params, structure=structure)
>>> print(kpoints.get_kpoints())
```

(continues on next page)

(continued from previous page)

```
pymatgen v2020.4.29 with grid density = 10 / atom
0
Monkhorst
2 2 2
```

### Mode: gamma

Setting the mode to `gamma` calls the internal KPOINT grid initialization for Gamma grids. This initialization is basically equivalent to the previously discussed Monkhorst initialization mode but generates a Gamma grid In this mode the expected input parameters passed in the input dictionary are:

- **mode** (`str`) – `gamma`
- **kpoints** (`list` or `float`) – Explicit 3x1 list of `int` defining the grid's subdivisions or a kpoint density of type:class:*float*

  **Note:** In case the kpoint grid is initialized from density the structure has to be passed to the constructor as well. However, the structure is not required for the initialization using an explicit kpoint grid.

- **shift** (`list`) – 3x1 list of `float` defining the kpoint grid shift applied to the grid

  **Note:** If the grid is initialized from a density (i.e. kpoints is of type `float`) any defined shift is ignored.

- **sympath** (`None`) – Unused by this mode

**Example for explicit kpoint grid:**

```
>>> gamma_mode_params = {
...     'mode': 'gamma',
...     'kpoints': [2, 2, 2],
... }
>>> kpoints = VaspKpointData(kpoints=gamma_mode_params)
>>> print(kpoints.get_kpoints())
Automatic kpoint scheme
0
Gamma
2 2 2
```

**Example for kpoint density**

```
>>> from pymatgen import Structure, Lattice
>>> lattice = Lattice.cubic(1.0)
>>> structure = Structure(lattice, ['H'], [[.0, .0, .0]])
>>> gamma_mode_params = {
...     'mode': 'gamma',
...     'kpoints': 10.0,
... }
>>> kpoints = VaspKpointData(kpoints=gamma_mode_params, structure=structure)
>>> print(kpoints.get_kpoints())
pymatgen v2020.4.29 with grid density = 10 / atom
0
Gamma
2 2 2
```

### Mode: line

Kpoint grids for bandstructure calculations can be generated by setting the mode to `line` Using line mode the expected input parameters passed in the input dictionary are:

- **mode** (`str`) – `line`
- **kpoints** (`int`) – Integer value defining the number of kpoints for each path segment
- **shift** (`None`) – Unused by this mode
- **sympath** (`HighSymmKpath`) – `HighSymmKpath` object defining a path along high symmetry lines in the Brillouin zone

**Example:**

```
>>> from pymatgen import Structure, Lattice
>>> from pymatgen.symmetry.bandstructure import HighSymmKpath
>>> lattice = Lattice.cubic(1.0)
>>> structure = Structure(lattice, ['H'], [[.0, .0, .0]])
>>> sympath = HighSymmKpath(structure, path_type='sc')

>>> line_mode_params = {
...     'mode': 'line',
...     'kpoints': 50,
...     'sympath': symmetry_path,
... }

>>> kpoints = VaspKpointData(kpoints=line_mode_params)
>>> print(kpoints.get_kpoints())
Line_mode KPOINTS file
50
Line_mode
Reciprocal
0.0 0.0 0.0 ! \Gamma
0.0 0.5 0.0 ! X

0.0 0.5 0.0 ! X
0.5 0.5 0.0 ! M

0.5 0.5 0.0 ! M
0.0 0.0 0.0 ! \Gamma

0.0 0.0 0.0 ! \Gamma
0.5 0.5 0.5 ! R

0.5 0.5 0.5 ! R
0.0 0.5 0.0 ! X

0.5 0.5 0.0 ! M
0.5 0.5 0.5 ! R
```

## 9.3 POSCAR (`cusp.poscar`)

Input data type for structure inputs passed to VASP calculations. Contrary to a plain structure data type the *VaspPoscarData* class offers multiple additional attributes like constraints or atomic velocities

### 9.3.1 Initializing the class

In general the constructor of the *VaspPoscarData* class is of the following form:

```
VaspPoscarData(structure=None, constraints=None, velocities=None, temperature=None)
```

**Arguments:**

- **structure** (`StructureData`, `Structure` or `Poscar`) – The structure data used to generate the POSCAR data from.
- **constraints** (`list`, optional) – Nx3 list (N the number of atoms) of `bool` introducing selective dynamics by adding constraints on the coordinates for every atom contained in the structure. (i.e. *False*: coordinate is not allowed to change, *True* coordinate is allowed to change)
- **velocities** (`list`, optional) – Nx3 list (N the number of atoms) of `float` setting the velocities for every atom on the structure
- **temperature** (`float`, optional) – Instead of passing an explicit list of velocities initialize the atomic velocities from a Maxwell-Boltzmann distribution at the given temperature

**Example plain structure:**

```
>>> from aiida.plugins import DataFactory
>>> from pymatgen import Lattice, Structure
>>> lattice = Lattice.cubic(3.359)
>>> structure = Structure(lattice, ["Po"], [.0, .0, .0])
>>> VaspPoscarData = DataFactory('cusp.poscar')
>>> cusp_poscar = VaspPoscarData(structure=structure)
>>> print(cusp_poscar.get_poscar())
Po1
1.0
3.359000 0.000000 0.000000
0.000000 3.359000 0.000000
0.000000 0.000000 3.359000
Po
1
direct
0.000000 0.000000 0.000000 Po
```

**Example with constraints:**

```
>>> from aiida.plugins import DataFactory
>>> from pymatgen import Lattice, Structure
>>> lattice = Lattice.cubic(3.359)
>>> structure = Structure(lattice, ["Po"], [.0, .0, .0])
>>> VaspPoscarData = DataFactory('cusp.poscar')
>>> cusp_poscar = VaspPoscarData(structure=structure, constraints=[[True, False,␣
↪True]])
>>> print(cusp_poscar.get_poscar())
Po1
1.0
3.359000 0.000000 0.000000
0.000000 3.359000 0.000000
0.000000 0.000000 3.359000
Po
1
Selective dynamics
```

(continues on next page)

(continued from previous page)

```
direct
0.000000 0.000000 0.000000 T F T Po
```

### 9.3.2 Recovering the stored structure data

In order to further analyze or re-use structures the *VaspPoscarData* class offers several methods to retrieve the stored structure. In particular, four different methods are available to recover the stored structure data in different formats:

VaspPoscarData.**get_poscar**()
Create and return a pymatgen.io.vasp.inputs.Poscar instance initialized from the node's stored structure data contents.

**Returns** a pymatgen Poscar instance

**Return type** pymatgen.io.vasp.inputs.Poscar

VaspPoscarData.**get_structure**()
Create and return a pymatgen.core.structure.Structure instance from the node's stored structure data contents.

**Returns** a pymatgen Structure instance

**Return type** pymatgen.core.structure.Structure

VaspPoscarData.**get_atoms**()
Create and return a ase.Atoms instance from the node's stored structure data contents

**Returns** an ASE-Atoms instance

**Return type** ase.Atoms

VaspPoscarData.**get_aiida_structure**()
Create and return a aiida.orm.StructureData instance from the node's stored structure data contents

**Returns** an AiiDA StructureData instance

**Return type** aiida.orm.StructureData

## 9.4 POTCAR (cusp.potcar)

The class *VaspPotcarData* is used to represent the VASP pseudo-potentials used as the inputs for a VASP calculation. To this end the class implements the simple *from_structure()* method that creates the pseudo-potential inputs for VASP calculations. Based on the input structure and the optionally passed potcar names and versions this function queries the database for the requested potentials and automatically builds the list of potentials that can be passed via the calculation's *input.potcar* option (see also the *following section* and the *example* given at the end of this section)

---

**Note:** Note that the *VaspPotcarData* class was introduced to respect the copyright enforced on the VASP pseudo-potentials. Because of that no actual contents of the original pseudo-potential are stored to this node to avoid any copyright infringement when exporting and sharing the calculation. Rather than storing the actual contents only a link to the corresponding pseudo-potential data stored in the database is added such that the node is independent of the underlying data.

---

### 9.4.1 Generating Pseudo-Potential Inputs for VASP Calculations

**classmethod** VaspPotcarData.**from_structure**(*structure*, *functional*, *potcar_params={}*)

Create potcar input data for a given structure and functional.

Reads elements present in the passed structure and builds the input list of VaspPotcarData instances defining the potentials to be used for each element. By default the potential names are assumed to equal the corresponding element names and potentials of the latest version are used. This default behavior can be changed on a per element basis by using the *potcar_params* dictionary to explicitly define the potential name and / or version to be used for a given element

Assume a structure containig elements 'A' and 'B'. For both elements the potentials with name 'A_pv' and 'B_pv' should be used and additionally element 'B' requires the use of a specific potential version, i.e. version 10000101 (Note that the version number is simply an integer representation of the creation date found in the potcar files title in the format YYYYMMDD) The above can be achived by passing the following *potcar_params* to the method:

```
potcar_params = {
    'A': {'name': 'A_pv'},
    'B': {'name': 'B_pv', 'version': 10000101},
}
```

**Note:** Alternatively it is also possible to use the *potcar_params* to pass a simple list defining the potential names (i.e. A_pv, B_pv, ...) to be used for the calculation (for all elements in the structure not defined by the passed list the default potentials will be used!) In that case potentials of the latest version are used and the method tries to figure out the elements corresponding to the potential name automatically. Because of that: make sure that each potential name starts with the corresponding element name (**case sensitive!**) followed by any non-letter character, for instance: **Ge**_d_GW, **Li**_sv, **Cu**, **H**1.75, etc...

**Parameters**

- **structure** (Structure, Poscar, StructureData or *VaspPoscarData*) – input structure for which the potential list is generated
- **functional** (*str*) – functional type of the used potentials, accepted functionals inputs are: *'lda_us'*, *'lda'*, *'lda_52'*, *'pbe'*, *'pbe_52'*, *'pbe_54'*, *'pw91_us'* and *'pw91'*.
- **potcar_params** (*dict* or *list*) – optional dictionary overriding the default potential name and version used for the element given as key or a list of potential names to be used for the calculation

### 9.4.2 Example

In the following some examples are given to show how the *aiida_cusp.data.VaspPotcarData.from_structure()* method can be used to initialize the pseudo-potential inputs for a VASP calculation. To test this functionality a test structure is required. In the following example a simple fcc structure is used to demonstrate the process of the pseudo-potential setup:

```
>>> from pymatgen import Structure, Lattice
>>> from aiida.plugins import DataFactory
>>> VaspPotcarData = DataFactory('cusp.potcar')
>>> lattice = Lattice.cubic(3.524)
>>> structure = Structure.from_spacegroup(225, lattice, ["Cu"], [[0.0, 0.0, 0.0]])
```

Simply passing the structure and the desired functional to the method without specifying any *potcar_params* initializes the pseudo-potential list with the default settings. This means: For each element present in the structure the database is queried for a pseudo-potential with the defined functional and a potential name matching the element name. From the list of potentials matching these requirements the potential with the highest version number will be return as input potential:

```
>>> potcar = VaspPotcarData.from_structure(structure, 'pbe')
>>> potcar
{'Cu': <VaspPotcarData: uuid: 0ea424b7-9b9e-446a-8d44-7690cb0006d7 (unstored)>}
>>> print("{} {} {}".format(potcar['Cu'].name, potcar['Cu'].version, potcar['Cu'].
↪functional))
Cu 20010105 pbe
```

If your calculation requires a different potential, for instance the *Cu_pv* potential, the above shown default behavior can be overridden by passing the desired potential parameters to the function using the *potcar_params* dictionary. To use the *Cu_pv* potential instead of the *Cu* potential, chosen by default, the following *potcar_params* need to passed:

```
>>> potcar = VaspPotcarData.from_structure(structure, 'pbe', potcar_params={'Cu': {
↪'name': 'Cu_pv'}})
>>> potcar
{'Cu': <VaspPotcarData: uuid: c55928e9-3f3a-4d03-87f7-5b2c4be5fd9a (unstored)>}
>>> print("{} {} {}".format(potcar['Cu'].name, potcar['Cu'].version, potcar['Cu'].
↪functional))
Cu_pv 20000906 pbe
```

Note that the *potcar_params* also allows a *'version'* key for each element to not only define the potential's name to be used but also potentially fix the potential's version. However, since in the above example only the potential name is changed and the version remains unchanged (i.e. whatever highest version found) the above given is equivalent to passing the pseudo-potential name only:

```
>>> potcar = VaspPotcarData.from_structure(structure, 'pbe', potcar_params=['Cu_pv'])
>>> potcar
{'Cu': <VaspPotcarData: uuid: 1f6ea785-876f-4942-9f30-51a8eac39573 (unstored)>}
>>> print("{} {} {}".format(potcar['Cu'].name, potcar['Cu'].version, potcar['Cu'].
↪functional))
Cu_pv 20000906 pbe
```

## 9.5 CONTCAR (`cusp.contcar`)

Output data type for structure data written to the *CONTCAR* generated by a VASP calculation. This data type is identical to the *VaspPoscarData* and only implemented to differentiate between the input and output structure of a VASP calculation. In order to simplify the access to the file's contents the class implements the *get_contcar()* (*see below*) For other methods also available from the class please refer to the *VaspPoscarData documentation*.

### 9.5.1 Implemented Methods and Attributes

VaspContcarData.**get_contcar**()

Create and return a pymatgen.io.vasp.inputs.Poscar instance initialized from the node's stored output structure data content.

**Returns** a pymatgen Poscar instance

**Return type** pymatgen.io.vasp.inputs.Poscar

## 9.6 vasprun (cusp.vasprun)

Data types of class *VaspVasprunData* are used to store the data contained in the VASP calculation output files of type *vasprun.xml*. As those files may grow large a gzip-compressed copy of the file is stored to the AiiDA repository instead of the plain file. In order to simplify the access to the file's contents several methods are implemented by the class to access the stored file contents are implemented.

### 9.6.1 Implemented Methods and Attributes

VaspVasprunData.**get_vasprun**(***kwargs*)

Return a pymatgen.io.vasp.outputs.Vasprun instance initialized from *vasprun.xml* data stored by the node

Given arguments are directly passed on to pymatgen's Vasprun constructor. By default, a minimal setup is used to parse the *vasprun.xml* contents:

```
kwargs_default = {
    ionic_step_skip: None,
    ionic_step_offset: 0,
    parse_dos: False,
    parse_eigen: False,
    parse_projected_eigen: False,
    occu_tol: 1.0E-8,
    exception_on_bad_xml: True,
}
```

However, note that the default parameters may be overridden at any time by passing the desired values when calling this function.

**Parameters**

- **ionic_step_skip** (*int*) – read structures and energies only for every 'ionic_step_skip'th ionic step
- **ionic_step_offset** (*int*) – ignore energies and structures at ionic steps smaller than 'ionic_step_offset'
- **parse_dos** (*bool*) – parse density of states from the vasprun.xml
- **parse_eigen** (*bool*) – parse eigenvalues from the vasprun.xml
- **parse_projected_eigen** (*bool*) – parse projected eigenvalues from the vasprun.xml
- **occu_tol** (*float*) – minimum occupation tolerances for determining the valence and conduction band minima
- **except_on_bad_xml** (*bool*) – throw an exception if the parsed vasprun.xml is malformed

**Returns** Vasprun instance initialized from the node's stored vasprun.xml data

**Return type** Vasprun

VaspVasprunData.**get_content**(*decompress=True*)

Load the node and return either the archive (i.e. compressed) or the file (i.e. the decompressed) contents stored in the node.

**Parameters** **decompress** (*bool*) – Indicate wether comressed or uncompressed contents are returned

VaspVasprunData.**write_file**(*filepath*, *decompress=True*)
Write the node's contents to a file

**Params filepath** path to the output file to which the stored contents will be written

**Params decompress** write the decompressed or the archive contents to the file

VaspVasprunData.**filepath**
Return the path of the object in the repository.

Basically replicates the procedure in the internal open method without the final call to the open()

## 9.7 OUTCAR (cusp.outcar)

Data types of class *VaspOutcarData* are used to store the data contained in the VASP calculation output files of type *OUTCAR*. As those files may grow large a gzip-compressed copy of the file is stored to the AiiDA repository instead of the plain file. In order to simplify the access to the file's contents several methods are implemented by the class to access the stored file contents are implemented.

### 9.7.1 Implemented Methods and Attributes

VaspOutcarData.**get_outcar**()
Return a pymatgen.io.vasp.outputs.Outcar instance initialized from the OUTCAR data stored by the node

**Returns** Outcar instance initialized from the node's stored OUTCAR data

**Return type** Outcar

VaspOutcarData.**get_content**(*decompress=True*)
Load the node and return either the archive (i.e. compressed) or the file (i.e. the decompressed) contents stored in the node.

**Parameters** **decompress** (*bool*) – Indicate wether comressed or uncompressed contents are returned

VaspOutcarData.**write_file**(*filepath*, *decompress=True*)
Write the node's contents to a file

**Params filepath** path to the output file to which the stored contents will be written

**Params decompress** write the decompressed or the archive contents to the file

VaspOutcarData.**filepath**
Return the path of the object in the repository.

Basically replicates the procedure in the internal open method without the final call to the open()

## 9.8 CHGCAR (cusp.chgcar)

Data types of class *VaspChgcarData* are used to store the data contained in the VASP calculation output files of type *CHGCAR*. As those files may grow large a gzip-compressed copy of the file is stored to the AiiDA repository instead of the plain file. Note that this data types is only implemented for convenience to simplify the sharing of *CHGCAR* contents between calculations. However, you may use the class as you wish but be advised that only the basic methods for accessing the file's contents are implemented as it can be seen in the following.

### 9.8.1 Implemented Methods and Attributes

VaspChgcarData.**get_content**(*decompress=True*)
Load the node and return either the archive (i.e. compressed) or the file (i.e. the decompressed) contents stored in the node.

**Parameters** **decompress** (*bool*) – Indicate wether comressed or uncompressed contents are returned

VaspChgcarData.**write_file**(*filepath*, *decompress=True*)
Write the node's contents to a file

**Params filepath** path to the output file to which the stored contents will be written

**Params decompress** write the decompressed or the archive contents to the file

VaspChgcarData.**filepath**
Return the path of the object in the repository.

Basically replicates the procedure in the internal open method without the final call to the open()

## 9.9 WAVECAR (cusp.wavecar)

Data types of class *VaspWavecarData* are used to store the data contained in the VASP calculation output files of type *WAVECAR*. As those files may grow large a gzip-compressed copy of the file is stored to the AiiDA repository instead of the plain file. Note that this data types is only implemented for convenience to simplify the sharing of *WAVECAR* contents between calculations. However, you may use the class as you wish but be advised that only the basic methods for accessing the file's contents are implemented as it can be seen in the following.

### 9.9.1 Implemented Methods and Attributes

VaspWavecarData.**get_content**(*decompress=True*)
Load the node and return either the archive (i.e. compressed) or the file (i.e. the decompressed) contents stored in the node.

**Parameters** **decompress** (*bool*) – Indicate wether comressed or uncompressed contents are returned

VaspWavecarData.**write_file**(*filepath*, *decompress=True*)
Write the node's contents to a file

**Params filepath** path to the output file to which the stored contents will be written

**Params decompress** write the decompressed or the archive contents to the file

VaspWavecarData.**filepath**
Return the path of the object in the repository.

Basically replicates the procedure in the internal open method without the final call to the open()

## 9.10 Generic (cusp.generic)

Data types of class *VaspGenericData* are used to store the data contained in the VASP calculation output files for which no individual data type is available. However, the node is inherited from the same base class as all other output nodes. This means a gzip-compressed copy of every file marked as *VaspGenericData* is stored to the repository rather than the plain file contents. Similar to the data types available for *WAVECAR* and *CHGCAR* this type is not

optimized for user interaction. However, you still may use the type as you like and the stored contents can be easily accessed via the implemented basic methods for file access.

### 9.10.1 Implemented Methods and Attributes

VaspGenericData.**get_content**(*decompress=True*)

Load the node and return either the archive (i.e. compressed) or the file (i.e. the decompressed) contents stored in the node.

**Parameters** **decompress** (*bool*) – Indicate wether comressed or uncompressed contents are returned

VaspGenericData.**write_file**(*filepath*, *decompress=True*)

Write the node's contents to a file

**Params filepath** path to the output file to which the stored contents will be written

**Params decompress** write the decompressed or the archive contents to the file

VaspGenericData.**filepath**

Return the path of the object in the repository.

Basically replicates the procedure in the internal open method without the final call to the open()

# CHAPTER 10

# Commands

The AiiDA package provides a useful command line interface accessible through the `verdi` command. To simplify certain tasks the plugin introduces additional commands to extend the default `verdi` command line interface. In the following sections an overview of the set of commands added by the plugin and their usage is given.

## 10.1 verdi data potcar

```
Usage: verdi data potcar [OPTIONS] COMMAND [ARGS]...

  Manage VASP POTCAR files.

Options:
  -h, --help  Show this message and exit.

Commands:
  add   Add one or multiple VASP pseudo-potential files to the database.
  list  List VASP pseudo-potentials available in the AiiDA database.
  show  Display a pseudo-potential's header contents.
```

CHAPTER 11

# aiida_cusp package

Custodian based VASP Plugin for AiiDA enabling automated error correction for AiiDA managed VASP calculations.

## 11.1 Subpackages

### 11.1.1 aiida_cusp.calculators package

**class** aiida_cusp.calculators.**VaspCalculation**(**args*, ***kwargs*)

Bases: *aiida_cusp.calculators.vasp_basic_calculation.VaspBasicCalculation*, *aiida_cusp.calculators.vasp_neb_calculation.VaspNebCalculation*

**create_calculation_inputs**(*folder*, *calcinfo*)

Write the calculation inputs and setup the retrieve lists based on the type of calculation (i.e. Basic or NEB calculation)

**classmethod define**(*spec*)

Defined the required inputs for the calculation process.

**is_neb**()

**restart_files_exclude**()

Extend the default list of excluded files defined by the parent CalculationBase class if neccessary.

As the sandbox folder gets uploaded in an early submission stage adding files given as inputs to a restarted calculation assures that those files are not overwritten by the possibly available corresponding remote file

**verify_structure_inputs**()

#### Submodules

#### aiida_cusp.calculators.calculation_base module

Base class serving as parent for other VASP calculator implementations

**class** aiida_cusp.calculators.calculation_base.**CalculationBase**(**args*, ***kwargs*)
Bases: aiida.engine.processes.calcjobs.calcjob.CalcJob

Base class implementing the basic inputs and features commond to all kind of VASP calculations. Includes a list of available inputs common to both basic VASP calculations as well as NEB calculations.

**create_calculation_inputs**(*folder*, *calcinfo*)
Write the calculation inputs and set the rerieve lists (This method has to be implemented by the inherited subclass)

**classmethod define**(*spec*)
Defined the required inputs for the calculation process.

**prepare_for_submission**(*folder*)
The baseclass will only setup the basic calcinfo arguments but will not write **any** files which has to be implemented in the subclassed prepare_for_submission() method

**remote_filelist**(*remote_data*, *relpath='.'*)
Recurse remote folder contents and return a list of all files found on the remote with the list containing the files names, relative paths and the absolute file path on the remote.

**Returns** list of tuples of type (filename, absolut_path_on_remote, relative_path (without the filename)

**restart_copy_remote**(*folder*, *calcinfo*)
Copy and write remote input files for a restarted VASP calcualtion

**restart_files_exclude**()
Create a list of files that will not be copied from the remote restart folder to the current calculation folder.

**retrieve_permanent_list**()
Define the list of files marked for permanent retrieval.

Here only calculation independent files are marked for retrieval, i.e. files like the submit-script, scheduler outputs, etc. Retrieval of calculation files is the responsibility of the connected parser.

**retrieve_temporary_list**()
Defines the list of files marked for temporary retrieval.

By default each calculaltion will retrieve **all** available files created by the calculation and store them to a temporary folder. Which of the files are actually kept is then decided by the subsequently running parser.

**setup_custodian_settings**(*is_neb=False*)
Create custodian settings instance from the given handlers and settings.

**vasp_calc_mpi_args**()
Obtain the mpirun commands and the provided extra mpi parameters

This function is basically a copy of the procedure internally used by AiiDA in it's CalcJob.presubmit() method to build the list of MPI and extra MPI parameters.

**vasp_run_line**()
Get the VASP run line used by the custodian script to start the VASP calculation

Populates the CalcInfo object with all required parameters such that the generated CalcInfo instance can be passed to the schedulers _get_run_line() method to obtain the runline.

aiida_cusp.calculators.calculation_base.**lidi_serializer**(*value*)

### aiida_cusp.calculators.vasp_basic_calculation module

Calculator class performing regular VASP calculations

**class** aiida_cusp.calculators.vasp_basic_calculation.**VaspBasicCalculation**(**args*, ***kwargs*)

Bases: *aiida_cusp.calculators.calculation_base.CalculationBase*

Calculator class for basic VASP calculations.

This calculator implements the required features for setting up and running basic VASP calculations (i.e. all calculations that are **not** of the kind NEB calculation, refer to the VaspNebCalculation class for this kind of calculation) through the AiiDA framework.

**create_calculation_inputs**(*folder*, *calcinfo*)
Write the calcultion inputs and setup the retrieve lists for a regular VASP calculation that is not of type NEB

**classmethod define**(*spec*)
Defined the required inputs for the calculation process.

### aiida_cusp.calculators.vasp_calculation module

Calculator class performing VASP calculations

**class** aiida_cusp.calculators.vasp_calculation.**VaspCalculation**(**args*, ***kwargs*)

Bases: *aiida_cusp.calculators.vasp_basic_calculation.VaspBasicCalculation*, *aiida_cusp.calculators.vasp_neb_calculation.VaspNebCalculation*

**create_calculation_inputs**(*folder*, *calcinfo*)
Write the calculation inputs and setup the retrieve lists based on the type of calculation (i.e. Basic or NEB calculation)

**classmethod define**(*spec*)
Defined the required inputs for the calculation process.

**is_neb**()

**restart_files_exclude**()
Extend the default list of excluded files defined by the parent CalculationBase class if neccessary.

As the sandbox folder gets uploaded in an early submission stage adding files given as inputs to a restarted calculation assures that those files are not overwritten by the possibly available corresponding remote file

**verify_structure_inputs**()

### aiida_cusp.calculators.vasp_neb_calculation module

Calculator class performing VASP NEB calculations

**class** aiida_cusp.calculators.vasp_neb_calculation.**VaspNebCalculation**(**args*, ***kwargs*)

Bases: *aiida_cusp.calculators.calculation_base.CalculationBase*

Calculator class for VASP NEB calculations.

This calculator implements the required features for setting up and running a VASP NEB calculation through the AiiDA framewirk. Contrary to the calculator for regular runs (refer to the VaspCalculation class) instead of a single structure this calculator class supports the input of multiple POSCAR instances defining the NEB path.

**create_calculation_inputs**(*folder*, *calcinfo*)
Write the calcultion inputs and setup the retrieve lists for a VASP calculation of type NEB

**classmethod define**(*spec*)
Defined the required inputs for the calculation process.

### 11.1.2 aiida_cusp.cli package

#### Submodules

#### aiida_cusp.cli.potcar_cmd module

AiiDA command line extension for handling VASP pseudo-potential files

### 11.1.3 aiida_cusp.data package

**class** aiida_cusp.data.**VaspKpointData**(*kpoints=None*, *structure=None*)
Bases: aiida.orm.nodes.data.dict.Dict

AiiDA compatible node representing a VASP k-point grid based on the Kpoints datatype.

**Parameters**

- **kpoints** (pymatgen.io.vasp.inputs.Kpoints or dict) – k-point grid definitions either given as a dictionary to call the internal initialization modes or directly as a pymatgen pymatgen.io.vasp.inputs.Kpoints object.
- **structure** (Structure, Poscar or StructureData) – calculation input structure (optional, only required if k-points are initialized from a density)

**get_description**()
Return a descriptive string of the stored k-point contents.

**get_kpoints**()
Create and return a pymatgen.io.vasp.inputs.Kpoints instance initialized from the node's stored k-point data contents.

**Returns** a pymatgen Kpoints instance

**Return type** pymatgen.io.vasp.inputs.Kpoints

**write_file**(*filename*)
Write the stored k-point grid data to VASP input file.

File creation is redirected to the pymatgen.io.vasp.inputs.Kpoints.write_file() method and the created output file will be formatted as VASP input file (KPOINTS)

**Parameters** **filename** (*str*) – destination for the output file including the desired output filename

**Returns** None

**class** aiida_cusp.data.**VaspPoscarData**(**args*, ***kwargs*)
Bases: aiida.orm.nodes.data.dict.Dict

**get_aiida_structure**()
Create and return a aiida.orm.StructureData instance from the node's stored structure data contents

**Returns** an AiiDA StructureData instance

**Return type** aiida.orm.StructureData

**get_atoms**()
Create and return a ase.Atoms instance from the node's stored structure data contents

**Returns** an ASE-Atoms instance

**Return type** ase.Atoms

**get_description**()
Return a short descriptive string for the stored structure data contents containing the structural composition. :return: structural composition of the stored structure :rtype: *str*

**get_poscar**()
Create and return a pymatgen.io.vasp.inputs.Poscar instance initialized from the node's stored structure data contents.

**Returns** a pymatgen Poscar instance

**Return type** pymatgen.io.vasp.inputs.Poscar

**get_structure**()
Create and return a pymatgen.core.structure.Structure instance from the node's stored structure data contents.

**Returns** a pymatgen Structure instance

**Return type** pymatgen.core.structure.Structure

**write_file**(*filename*)
Write the stored Poscar contents to VASP input file.

File creation is redirected to the pymatgen.io.vasp.inputs.Poscar.write_file() method and the created output file will be formatted as VASP input file (POSCAR)

**Parameters** **filename** (*str*) – destination for the output file including the desired output filename

**Returns** None

**class** aiida_cusp.data.**VaspIncarData**(*incar=None*)
Bases: aiida.orm.nodes.data.dict.Dict

AiiDA compatible node representing a VASP incar data object based on the Incar datatype.

**Parameters** **incar** (dict or Incar) – input parameters used to construct the Incar object or a incar object itself (Note: may also be set to *None* to initialize an empty incar object and use the VASP default parameters)

**get_incar**()
Create and return a Incar instance initialized from the node's stored incar data contents.

**Returns** a pymatgen Incar data instance

**Return type** pymatgen.io.vasp.inputs.Incar

**write_file**(*filename*)
Write the stored incar data to VASP input file.

Output of the contents to the file is redirected to the pymatgen.io.vasp.inputs.Incar.write_file() method and the created output file will be formatted as VASP input file (INCAR)

**Parameters** **filename** (*str*) – destination for writing the output file

**Returns** None

**class** aiida_cusp.data.**VaspPotcarData**(**args*, ***kwargs*)

Bases: aiida.orm.nodes.data.dict.Dict

Pseudopotential input datatype for VASP calculations.

Contrary to the related VaspPotcarFile class this object does **not** contain any proprietary potential information and only stores the potential's uuid value and the unique potential identifiers (name, functional, version and the content hash)

**element**

Make associated potential element a property

**filenode_uuid**

Make associated potential UUID a property

**classmethod from_structure**(*structure*, *functional*, *potcar_params={}*)

Create potcar input data for a given structure and functional.

Reads elements present in the passed structure and builds the input list of VaspPotcarData instances defining the potentials to be used for each element. By default the potential names are assumed to equal the corresponding element names and potentials of the latest version are used. This default behavior can be changed on a per element basis by using the *potcar_params* dictionary to explicitly define the potential name and / or version to be used for a given element

Assume a structure containig elements 'A' and 'B'. For both elements the potentials with name 'A_pv' and 'B_pv' should be used and additionally element 'B' requires the use of a specific potential version, i.e. version 10000101 (Note that the version number is simply an integer representation of the creation date found in the potcar files title in the format YYYYMMDD) The above can be achived by passing the following *potcar_params* to the method:

```
potcar_params = {
    'A': {'name': 'A_pv'},
    'B': {'name': 'B_pv', 'version': 10000101},
}
```

**Note:** Alternatively it is also possible to use the *potcar_params* to pass a simple list defining the potential names (i.e. A_pv, B_pv, ...) to be used for the calculation (for all elements in the structure not defined by the passed list the default potentials will be used!) In that case potentials of the latest version are used and the method tries to figure out the elements corresponding to the potential name automatically. Because of that: make sure that each potential name starts with the corresponding element name (**case sensitive!**) followed by any non-letter character, for instance: **Ge**_d_GW, **Li**_sv, **Cu**, **H**1.75, etc...

**Parameters**

- **structure** (Structure, Poscar, StructureData or *VaspPoscarData*) – input structure for which the potential list is generated
- **functional** (*str*) – functional type of the used potentials, accepted functionals inputs are: *'lda_us'*, *'lda'*, *'lda_52'*, *'pbe'*, *'pbe_52'*, *'pbe_54'*, *'pw91_us'* and *'pw91'*.
- **potcar_params** (*dict* or *list*) – optional dictionary overriding the default potential name and version used for the element given as key or a list of potential names to be used for the calculation

**functional**

Make associated potential functional a property

**hash**
Make associated potential hash a property

**init_from_tags**(*name*, *version*, *functional*)
Initialized VaspPotcarData node from given potential tags

**Parameters**

- **name** (*str*) – name of the potential
- **version** (*int*) – version of the potential
- **functional** (*str*) – functional of the potential

**Raises** ***VaspPotcarDataError*** – if one of the non-optional parameters name, version or functional is missing

**load_potential_file_node**()
Load the actual potential node associated with the potential data node

First tries to load the potential file node from the given UUID. If this fails (for instance because the VaspPotcarData node was imported from some other database) a second attempt will try to load a potential file node based on the stored content hash.

**Returns** the associated VaspPotcarFile node

**Return type** *VaspPotcarFile*

**Raises** ***VaspPotcarDataError*** – if no corresponding potential file node is found in the database

**name**
Make associated potential name a property

**classmethod potcar_from_linklist**(*poscar_data*, *linklist*)
Assemble pymatgen Potcar object from a list of VaspPotcarData instances

Reads pseudo-potential from the passed list connecting each element with it's potential and creates the complete Potcar file according to the element ordering fodun in the passed poscar data object.

**Parameters**

- **poscar_data** – input structure for VASP calculations
- **linklist** (*dict*) – dictionary mapping element names to VaspPotcarData instances

**Returns** pymatgen Potcar data instance with containing the concatenated pseudo-potential information for all elements defined in the linklist

**Return type** Potcar

**classmethod potcar_props_from_name_list**(*potcar_name_list*)
Create a valid potcar properties dictionary from a list of given potential names

**version**
Make associated potential version a property

**class** aiida_cusp.data.**VaspVasprunData**(*file*, *filename=None*, ***kwargs*)
Bases: *aiida_cusp.utils.single_archive_data.SingleArchiveData*

AiiDA compatible node representing a VASP vasprun.xml output data object as gzip-compressed node in the repository.

**get_vasprun**(***kwargs*)
Return a pymatgen.io.vasp.outputs.Vasprun instance initialized from *vasprun.xml* data stored by the node

Given arguments are directly passed on to pymatgen's Vasprun constructor. By default, a minimal setup is used to parse the *vasprun.xml* contents:

```
kwargs_default = {
    ionic_step_skip: None,
    ionic_step_offset: 0,
    parse_dos: False,
    parse_eigen: False,
    parse_projected_eigen: False,
    occu_tol: 1.0E-8,
    exception_on_bad_xml: True,
}
```

However, note that the default parameters may be overridden at any time by passing the desired values when calling this function.

**Parameters**

- **ionic_step_skip** (*int*) – read structures and energies only for every 'ionic_step_skip'th ionic step
- **ionic_step_offset** (*int*) – ignore energies and structures at ionic steps smaller than 'ionic_step_offset'
- **parse_dos** (*bool*) – parse density of states from the vasprun.xml
- **parse_eigen** (*bool*) – parse eigenvalues from the vasprun.xml
- **parse_projected_eigen** (*bool*) – parse projected eigenvalues from the vasprun.xml
- **occu_tol** (*float*) – minimum occupation tolerances for determining the valence and conduction band minima
- **except_on_bad_xml** (*bool*) – throw an exception if the parsed vasprun.xml is malformed

**Returns** Vasprun instance initialized from the node's stored vasprun.xml data

**Return type** Vasprun

**parser_settings**(**kwargs*)
Update default parser settings with user defined inputs

**class** aiida_cusp.data.**VaspContcarData**(**args*, ***kwargs*)
Bases: *aiida_cusp.data.inputs.vasp_poscar.VaspPoscarData*

AiiDA compatible node representing a VASP CONTCAR output data object

Added as separate output node for easier access when used as calculation input for restarted VASP calculations.

**get_contcar**()
Create and return a pymatgen.io.vasp.inputs.Poscar instance initialized from the node's stored output structure data content.

**Returns** a pymatgen Poscar instance

**Return type** pymatgen.io.vasp.inputs.Poscar

**class** aiida_cusp.data.**VaspOutcarData**(*file*, *filename=None*, ***kwargs*)
Bases: *aiida_cusp.utils.single_archive_data.SingleArchiveData*

AiiDA compatible node representing a VASP OUTCAR output data object as gzip-compressed node in the repository.

**get_outcar**()
Return a pymatgen.io.vasp.outputs.Outcar instance initialized from the OUTCAR data stored by the node

**Returns** Outcar instance initialized from the node's stored OUTCAR data

**Return type** Outcar

**class** aiida_cusp.data.**VaspChgcarData**(*file*, *filename=None*, ***kwargs*)
Bases: *aiida_cusp.utils.single_archive_data.SingleArchiveData*

AiiDA compatible node representing a VASP CHGCAR output data object as gzip-compressed node in the repository.

Added as separate output node for easier access when used as calculation input for restarted VASP calculations.

**class** aiida_cusp.data.**VaspWavecarData**(*file*, *filename=None*, ***kwargs*)
Bases: *aiida_cusp.utils.single_archive_data.SingleArchiveData*

AiiDA compatible node representing a VASP WAVECAR output data object as gzip-compressed node in the repository.

Added as separate output node for easier access when used as calculation input for restarted VASP calculations.

**class** aiida_cusp.data.**VaspGenericData**(*file*, *filename=None*, ***kwargs*)
Bases: *aiida_cusp.utils.single_archive_data.SingleArchiveData*

AiiDA compatible node representing an arbitrary VASP output data object for which no own output datatype was implemented.

### Subpackages

### aiida_cusp.data.inputs package

### Submodules

### aiida_cusp.data.inputs.vasp_incar module

Datatype and methods to initialize and interact with VASP specific INCAR input data

**class** aiida_cusp.data.inputs.vasp_incar.**IncarWrapper**
Bases: object

Utility class for initializing pymatgen.io.vasp.inputs.Incar data objects

Accepts either a Incar instance or a dictionary containing valid VASP Incar parameters which will be passed through to the Incar constructor. Note: If *incar* is set to *None* and empty incar file will be initialized.

**Parameters** **incar** (dict or Incar) – input parameters used to construct the Incar object or a incar object itself.

**classmethod keys_to_upper_case**(*rawdict*)
Transform all incar parameter keys to upper case

**Parameters** **rawdict** (*dict*) – input dictionary containing incar parameter key value pairs with keys possibly of mixed case

**class** aiida_cusp.data.inputs.vasp_incar.**VaspIncarData**(*incar=None*)
Bases: aiida.orm.nodes.data.dict.Dict

AiiDA compatible node representing a VASP incar data object based on the Incar datatype.

**Parameters** **incar** (dict or Incar) – input parameters used to construct the Incar object or a incar object itself (Note: may also be set to *None* to initialize an empty incar object and use the VASP default parameters)

**get_incar**()
Create and return a Incar instance initialized from the node's stored incar data contents.

**Returns** a pymatgen Incar data instance

**Return type** pymatgen.io.vasp.inputs.Incar

**write_file**(*filename*)
Write the stored incar data to VASP input file.

Output of the contents to the file is redirected to the pymatgen.io.vasp.inputs.Incar.write_file() method and the created output file will be formatted as VASP input file (INCAR)

**Parameters** **filename** (*str*) – destination for writing the output file

**Returns** None

### aiida_cusp.data.inputs.vasp_kpoint module

Datatype and methods to initialize and interact with VASP specific KPOINTS input data

**class** aiida_cusp.data.inputs.vasp_kpoint.**KpointWrapper**(*kpoints=None*, *structure=None*)
Bases: object

Utility class for initializing pymatgen.io.vasp.inputs.Kpoints data objects.

Provides a wrapper for the different initialization modes provided by the pymatgen Kpoints object. Different modes can be access by setting the 'mode' keyword in the kpoints dictionary and by defining the datatypes passed by the 'kpoints' keyword (i.e. a list of kpoints will initialize an explicit kpoint grid while passing a float to the constructor will call a density based initialization mode)

Example:

```
>>> kpoint_param = {'mode': 'auto', 'kpoints': 100}
>>> kpts = KpointWrapper(kpoints=kpoint_params)
>>> print(kpts)
Automatic Mesh
0
Auto
  40
```

**Parameters**

- **kpoints** (dict or Kpoints) – input parameters that are used to construct the Kpoints object or a kpoints object itself
- **structure** (Structure, Poscar or StructureData) – optional input structure for the calculation required for density based kpoint initialization

**classmethod auto_int**()
Initialize automatic kpoint grid following the VASP automatic mode

Example:

```
Automatic Mesh
0
Auto
  40
```

**classmethod build_init_mode**()
Figure out the initialization mode using the defined mode and the specified type of kpoints

**classmethod complete_parameter_list**()
Complete the parameter list and perform minimal initial checking

**classmethod gamma_float**()
Initialize gamma grid using a given kpoint density. Enforces the usage of gamma grids.

Example:

```
Automatic Kpoint Scheme
0
Gamma
5 5 5
```

**classmethod gamma_list**()
Initialize gamma grid from a list explicitly defining the number of kpoint subdivisions along the crystal axis

Example:

```
Automatic Kpoint Scheme
0
Gamma
5 5 5
```

**classmethod line_int**()
Initialize list of kpoints along a high-symmetry path through the Brillouin zone. Uses a path defined by the pymatgen HighSymmPath class with a defined number of kpoint subdivisions between the path nodes.

Example:

```
Line_mode KPOINTS file
100
Line_mode
Reciprocal
0.0 0.0 0.0 ! Gamma
0.0 0.5 0.0 ! X

0.0 0.5 0.0 ! X

...

0.0 0.5 0.0 ! X

0.5 0.5 0.0 ! M
0.5 0.5 0.5 ! R
```

**classmethod monkhorst_float**()
Initialize kpoint grid using a given kpoint density. If the number of subdivisions is even a Monkhorst grid is constructured whereas gamma grids are used for odd kpoint subdivisions

Example:

```
Automatic Kpoint Scheme
0
Monkhorst
4 4 4
```

**classmethod monkhorst_list**()
Initialize Monkhorst grid from a list explicitly defining the number of kpoint subdivisions along the crystal axis

Example:

```
Automatic Kpoint Scheme
0
Monkhorst
4 4 4
```

**classmethod structure_from_input**()
Obtain `Structure` object from the given input structure

**classmethod validate_and_initialize**()
Initialize a new Kpoint object from user inputs

**class** aiida_cusp.data.inputs.vasp_kpoint.**VaspKpointData**(*kpoints=None*, *structure=None*)

Bases: `aiida.orm.nodes.data.dict.Dict`

AiiDA compatible node representing a VASP k-point grid based on the `Kpoints` datatype.

**Parameters**

- **kpoints** (`pymatgen.io.vasp.inputs.Kpoints` or dict) – k-point grid definitions either given as a dictionary to call the internal initialization modes or directly as a pymatgen `pymatgen.io.vasp.inputs.Kpoints` object.
- **structure** (`Structure`, `Poscar` or `StructureData`) – calculation input structure (optional, only required if k-points are initialized from a density)

**get_description**()
Return a descriptive string of the stored k-point contents.

**get_kpoints**()
Create and return a `pymatgen.io.vasp.inputs.Kpoints` instance initialized from the node's stored k-point data contents.

**Returns** a pymatgen Kpoints instance

**Return type** `pymatgen.io.vasp.inputs.Kpoints`

**write_file**(*filename*)
Write the stored k-point grid data to VASP input file.

File creation is redirected to the `pymatgen.io.vasp.inputs.Kpoints.write_file()` method and the created output file will be formatted as VASP input file (KPOINTS)

**Parameters** **filename** (*str*) – destination for the output file including the desired output filename

**Returns** None

### aiida_cusp.data.inputs.vasp_poscar module

Datatype and methods to initialize and interact with VASP specific POSCAR input data

**class** aiida_cusp.data.inputs.vasp_poscar.**PoscarWrapper**
Bases: object

Utility class for initializing pymatgen.io.vasp.inputs.Poscar data objects.

Requires at least a structure object to initialize the VASP Poscar input file and allows to set additional parameters like initial velocities or predictor-corrector coordinates. If the input structure is of type Poscar all other input parameters will be ignored.

**Parameters**

- **structure** (StructureData, Structure or Poscar) – Structure data used to initialize the VASP Poscar input file.
- **constraints** (*list*) – Nx3 list containing boolean values defining constraints on the coordinates of all atoms present in the structure with values set to *False* indicating fixed coordinates.
- **velocities** (*list*) – Nx3 list of floats defining the velocities of the atoms contained in the structure
- **temperature** (*float*) – initialize atom velocities from a Maxwell-Boltzmann distribution at the given temperature

**Returns** a pymatgen Poscar instance

**Return type** Poscar

**classmethod structure_from_input**()
Obtain a Structure instance from the given input structure

**classmethod validate_and_initialize**()
Initialize new Poscar instance from given user inputs

**class** aiida_cusp.data.inputs.vasp_poscar.**VaspPoscarData**(**args*, ***kwargs*)
Bases: aiida.orm.nodes.data.dict.Dict

**get_aiida_structure**()
Create and return a aiida.orm.StructureData instance from the node's stored structure data contents

**Returns** an AiiDA StructureData instance

**Return type** aiida.orm.StructureData

**get_atoms**()
Create and return a ase.Atoms instance from the node's stored structure data contents

**Returns** an ASE-Atoms instance

**Return type** ase.Atoms

**get_description**()
Return a short descriptive string for the stored structure data contents containing the structural composition. :return: structural composition of the stored structure :rtype: *str*

**get_poscar**()
Create and return a pymatgen.io.vasp.inputs.Poscar instance initialized from the node's stored structure data contents.

**Returns** a pymatgen Poscar instance

**Return type** pymatgen.io.vasp.inputs.Poscar

**get_structure**()
Create and return a pymatgen.core.structure.Structure instance from the node's stored structure data contents.

**Returns** a pymatgen Structure instance

**Return type** pymatgen.core.structure.Structure

**write_file**(*filename*)
Write the stored Poscar contents to VASP input file.

File creation is redirected to the pymatgen.io.vasp.inputs.Poscar.write_file() method and the created output file will be formatted as VASP input file (POSCAR)

**Parameters** **filename** (*str*) – destination for the output file including the desired output filename

**Returns** None

### aiida_cusp.data.inputs.vasp_potcar module

Datatypes and methods to initialize and interact with VASP specific pseudo-potential files

**class** aiida_cusp.data.inputs.vasp_potcar.**VaspPotcarData**(**args*, ***kwargs*)
Bases: aiida.orm.nodes.data.dict.Dict

Pseudopotential input datatype for VASP calculations.

Contrary to the related VaspPotcarFile class this object does **not** contain any proprietary potential information and only stores the potential's uuid value and the unique potential identifiers (name, functional, version and the content hash)

**element**
Make associated potential element a property

**filenode_uuid**
Make associated potential UUID a property

**classmethod from_structure**(*structure*, *functional*, *potcar_params={}*)
Create potcar input data for a given structure and functional.

Reads elements present in the passed structure and builds the input list of VaspPotcarData instances defining the potentials to be used for each element. By default the potential names are assumed to equal the corresponding element names and potentials of the latest version are used. This default behavior can be changed on a per element basis by using the *potcar_params* dictionary to explicitly define the potential name and / or version to be used for a given element

Assume a structure containig elements 'A' and 'B'. For both elements the potentials with name 'A_pv' and 'B_pv' should be used and additionally element 'B' requires the use of a specific potential version, i.e. version 10000101 (Note that the version number is simply an integer representation of the creation date found in the potcar files title in the format YYYYMMDD) The above can be achived by passing the following *potcar_params* to the method:

```
potcar_params = {
    'A': {'name': 'A_pv'},
    'B': {'name': 'B_pv', 'version': 10000101},
}
```

**Note:** Alternatively it is also possible to use the *potcar_params* to pass a simple list defining the potential names (i.e. A_pv, B_pv, ...) to be used for the calculation (for all elements in the structure not defined by the passed list the default potentials will be used!) In that case potentials of the latest version are used and the method tries to figure out the elements corresponding to the potential name automatically. Because of that: make sure that each potential name starts with the corresponding element name (**case sensitive!**) followed by any non-letter character, for instance: **Ge**_d_GW, **Li**_sv, **Cu**, **H**1.75, etc...

**Parameters**

- **structure** (Structure, Poscar, StructureData or *VaspPoscarData*) – input structure for which the potential list is generated
- **functional** (*str*) – functional type of the used potentials, accepted functionals inputs are: *'lda_us'*, *'lda'*, *'lda_52'*, *'pbe'*, *'pbe_52'*, *'pbe_54'*, *'pw91_us'* and *'pw91'*.
- **potcar_params** (*dict* or *list*) – optional dictionary overriding the default potential name and version used for the element given as key or a list of potential names to be used for the calculation

**functional**
Make associated potential functional a property

**hash**
Make associated potential hash a property

**init_from_tags**(*name*, *version*, *functional*)
Initialized VaspPotcarData node from given potential tags

**Parameters**

- **name** (*str*) – name of the potential
- **version** (*int*) – version of the potential
- **functional** (*str*) – functional of the potential

**Raises** ***VaspPotcarDataError*** – if one of the non-optional parameters name, version or functional is missing

**load_potential_file_node**()
Load the actual potential node associated with the potential data node

First tries to load the potential file node from the given UUID. If this fails (for instance because the VaspPotcarData node was imported from some other database) a second attempt will try to load a potential file node based on the stored content hash.

**Returns** the associated VaspPotcarFile node

**Return type** *VaspPotcarFile*

**Raises** ***VaspPotcarDataError*** – if no corresponding potential file node is found in the database

**name**
Make associated potential name a property

**classmethod potcar_from_linklist**(*poscar_data*, *linklist*)
Assemble pymatgen Potcar object from a list of VaspPotcarData instances

Reads pseudo-potential from the passed list connecting each element with it's potential and creates the complete Potcar file according to the element ordering fodun in the passed poscar data object.

**Parameters**

- **poscar_data** – input structure for VASP calculations
- **linklist** (*dict*) – dictionary mapping element names to VaspPotcarData instances

**Returns** pymatgen Potcar data instance with containing the concatenated pseudo-potential information for all elements defined in the linklist

**Return type** Potcar

**classmethod potcar_props_from_name_list**(*potcar_name_list*)
Create a valid potcar properties dictionary from a list of given potential names

**version**
Make associated potential version a property

**class** aiida_cusp.data.inputs.vasp_potcar.**VaspPotcarFile**(*path_to_potcar*, *name*, *element*, *version*, *functional*, *checksum*)
Bases: aiida.orm.nodes.data.singlefile.SinglefileData

Datatype representing an actual POTCAR file object stored to the AiiDA database.

Note that this class should never be called directly to add any pseudo-potentials to the database but always using the provided add_potential()-classmethod.

**Parameters**

- **path_to_potcar** (*str*) – Absolute path pointing to the potcar file
- **name** (*str*) – qualified name of the potential at location *path_to_potcar* (i.e. Li, Li_sv, Ge_pv_GW, ...)
- **element** (*str*) – element associated with the pseudo-potential
- **version** (*int*) – version of the pseudo-potential, i.e. the creation date as 8-digit integer in numerical YYYYMMDD representation
- **functional** (*str*) – functional group of the potential located at *path_to_potcar*. Valid choices are: lda_us, lda, lda_52, lda_54, pbe, pbe_52, pbe_54, pw91 and pw91_us
- **checksum** (*str*) – SHA-256 hash of the potential contents

**classmethod add_potential**(*path_to_potcar*, *name=None*, *functional=None*)
Create and return unstored potential node from POTCAR file.

Loads the POTCAR file at the given location and parses the basic identifiers from the file's contents.

**Parameters**

- **path_to_potcar** (*pathlib.Path*) – absolute path pointing to a potcar file
- **name** (*str*) – qualified name of the potential at location *path_to_potcar* (i.e. Li, Li_sv, Ge_pv_GW, ...)
- **functional** (*str*) – functional group of the potential located at *path_to_potcar*. Valid choices are: lda_us, lda, lda_52, lda_54, pbe, pbe_52, pbe_54, pw91 and pw91_us

**element**

**classmethod from_tags**(*name=None*, *element=None*, *version=None*, *functional=None*, *checksum=None*)

Query database for potentials containing a set of given tags.

To query the database at least one of the available tags has to be given. If multiple tags are defined only potentials matching **all** of the defined tags will be returned.

**Parameters**

- **name** (*str*) – fully qualified name of the potential (i.e. Li_sv, Li, Ge_sv_GW, P, ...)
- **element** (*str*) – name of the element associated with a given potential (i.e. Cl, Li, S, ...)
- **version** (*int*) – version (i.e. the creation date) of the potential in numerical 8-digit integer YYYYMMDD representation
- **functional** (*str*) – functional filter to query only for potentials associated with a specific functional. Allowed values are: lda_us, lda, lda_52, lda_54, pbe, pbe_52, pbe_54, pw91 and pw91_us
- **checksum** – the SHA-256 hash value associated with the contents of a potcar file

**Returns** a list of *VaspPotcarFile* nodes in the database matching the given tags

**Return type** list(*VaspPotcarFile*)

**functional**

**hash**

**classmethod is_unique**(*potcar_attrs*)

Check if a potential is unqiue (i.e. not already stored).

Uniqueness is checked based on the unique potential identifiers, i.e. the potential name, it's version and the associated functional.

**Parameters** **potcar_attrs** (*aiida_cusp.utils.PotcarParser*) – a PotcarParser instance of the potential to be checked for uniqueness

**Returns** *True* if no other potential with identical identifiers exists in the database

**Return type** *bool*

**Raises** ***MultiplePotcarError*** – if the a potential with identical identifiers is already stored in the database

**name**

**validate_element**(*element*)

Assert the given element is in the list of elements.

**validate_functional**(*functional*)

Assert the functional is in the list of valid functionals.

**validate_name**(*name*)

Assert potential name starts with valid element name.

**validate_version**(*version*)

Assert the parsed potential version is valid.

**version**

### aiida_cusp.data.outputs package

#### Submodules

#### aiida_cusp.data.outputs.vasp_chgcar module

Datatype and methods to initialize and interact with VASP specific CHGCAR output data

**class** aiida_cusp.data.outputs.vasp_chgcar.**VaspChgcarData**(*file*, *filename=None*, ***kwargs*)

Bases: *aiida_cusp.utils.single_archive_data.SingleArchiveData*

AiiDA compatible node representing a VASP CHGCAR output data object as gzip-compressed node in the repository.

Added as separate output node for easier access when used as calculation input for restarted VASP calculations.

#### aiida_cusp.data.outputs.vasp_contcar module

Datatype and methods to initialize and interact with VASP specific CONTCAR output data

**class** aiida_cusp.data.outputs.vasp_contcar.**VaspContcarData**(**args*, ***kwargs*)

Bases: *aiida_cusp.data.inputs.vasp_poscar.VaspPoscarData*

AiiDA compatible node representing a VASP CONTCAR output data object

Added as separate output node for easier access when used as calculation input for restarted VASP calculations.

**get_contcar**()

Create and return a pymatgen.io.vasp.inputs.Poscar instance initialized from the node's stored output structure data content.

**Returns** a pymatgen Poscar instance

**Return type** pymatgen.io.vasp.inputs.Poscar

#### aiida_cusp.data.outputs.vasp_generic module

Generic datatype used for all VASP output files for which no specific output type is implemented.

**class** aiida_cusp.data.outputs.vasp_generic.**VaspGenericData**(*file*, *filename=None*, ***kwargs*)

Bases: *aiida_cusp.utils.single_archive_data.SingleArchiveData*

AiiDA compatible node representing an arbitrary VASP output data object for which no own output datatype was implemented.

#### aiida_cusp.data.outputs.vasp_outcar module

Datatype and methods to initialize and interact with VASP specific OUTCAR output data

**class** aiida_cusp.data.outputs.vasp_outcar.**VaspOutcarData**(*file*, *filename=None*, ***kwargs*)

Bases: *aiida_cusp.utils.single_archive_data.SingleArchiveData*

AiiDA compatible node representing a VASP OUTCAR output data object as gzip-compressed node in the repository.

**get_outcar**()
Return a pymatgen.io.vasp.outputs.Outcar instance initialized from the OUTCAR data stored by the node

**Returns** Outcar instance initialized from the node's stored OUTCAR data

**Return type** Outcar

### aiida_cusp.data.outputs.vasp_vasprun module

Datatype and methods to initialize and interact with VASP specific vasprun.xml output data

**class** aiida_cusp.data.outputs.vasp_vasprun.**VaspVasprunData**(*file*, *filename=None*, ***kwargs*)
Bases: *aiida_cusp.utils.single_archive_data.SingleArchiveData*

AiiDA compatible node representing a VASP vasprun.xml output data object as gzip-compressed node in the repository.

**get_vasprun**(***kwargs*)
Return a pymatgen.io.vasp.outputs.Vasprun instance initialized from *vasprun.xml* data stored by the node

Given arguments are directly passed on to pymatgen's Vasprun constructor. By default, a minimal setup is used to parse the *vasprun.xml* contents:

```
kwargs_default = {
    ionic_step_skip: None,
    ionic_step_offset: 0,
    parse_dos: False,
    parse_eigen: False,
    parse_projected_eigen: False,
    occu_tol: 1.0E-8,
    exception_on_bad_xml: True,
}
```

However, note that the default parameters may be overridden at any time by passing the desired values when calling this function.

**Parameters**

- **ionic_step_skip** (*int*) – read structures and energies only for every 'ionic_step_skip'th ionic step
- **ionic_step_offset** (*int*) – ignore energies and structures at ionic steps smaller than 'ionic_step_offset'
- **parse_dos** (*bool*) – parse density of states from the vasprun.xml
- **parse_eigen** (*bool*) – parse eigenvalues from the vasprun.xml
- **parse_projected_eigen** (*bool*) – parse projected eigenvalues from the vasprun.xml
- **occu_tol** (*float*) – minimum occupation tolerances for determining the valence and conduction band minima
- **except_on_bad_xml** (*bool*) – throw an exception if the parsed vasprun.xml is malformed

**Returns** Vasprun instance initialized from the node's stored vasprun.xml data

**Return type** Vasprun

**parser_settings**(***kwargs*)
Update default parser settings with user defined inputs

#### aiida_cusp.data.outputs.vasp_wavecar module

Datatype and methods to initialize and interact with VASP specific WAVECAR output data

**class** aiida_cusp.data.outputs.vasp_wavecar.**VaspWavecarData**(*file*, *filename=None*, ***kwargs*)
Bases: *aiida_cusp.utils.single_archive_data.SingleArchiveData*

AiiDA compatible node representing a VASP WAVECAR output data object as gzip-compressed node in the repository.

Added as separate output node for easier access when used as calculation input for restarted VASP calculations.

### 11.1.4 aiida_cusp.parsers package

#### Submodules

#### aiida_cusp.parsers.parser_base module

The parser base class

**class** aiida_cusp.parsers.parser_base.**ParserBase**(*node*)
Bases: aiida.parsers.parser.Parser

The ParseBase class provides the general methods like the check for missing folders and the setup of the retrieved file lists that may be used by the derived parser classes

**check_retrieved_temp_folder**(***kwargs*)
As all calculation output files are expected to be retrieved in the retrieved_temporary_folder only checks for the presence of this folder.

**complete_linkname**(*partial_linkname*)
Create the complete linkname by adding the default outputs namespace prefix used to register results generated by the connected parser to the given partial linkname.

**parse**(***kwargs*)
Parse the contents of the output files retrieved in the *FolderData*.

This method should be implemented in the sub class. Outputs can be registered through the *out* method. After the *parse* call finishes, the runner will automatically link them up to the underlying *CalcJobNode*.

**Parameters** **kwargs** – output nodes attached to the *CalcJobNode* of the parser instance.

**Returns** an instance of ExitCode or None

**parser_settings**(*node*)
Return the possibly available optional parser settings defined by the calling calculation plugin.

**register_output_nodes**(*output_nodes*)
Add the parsed results stored in the output_nodes list as output nodes to the calculation

### aiida_cusp.parsers.vasp_file_parser module

Default parser class used with the cusp pluing for parsing the results of AiiDA managed VASP calculations

**class** aiida_cusp.parsers.vasp_file_parser.**VaspFileParser**(*node*)
Bases: *aiida_cusp.parsers.parser_base.ParserBase*

The VaspFileParser parsing class represents a basic parser class. In order to make it as flexible as possible this default parser only checks for the given files an adds them as SingleArchiveNodes to the calling calculation, giving you the full VASP experience (i.e. files-in –> file-out. In particular, no special parsing actions parsing special values or attributes of the calculation are employed at the parsing level to ensure the default parser class is compatible with all types of VASP calculations.

All discovered files present in the retrieve list defined by the parser settings will be added to the calculation with the filename as linkname with .suffix replaced by _suffix, i.e. the vasprun.xml would be stored under the vasprun_xml linkname) If files are found to be located in a subfolder (this is only the case for instance in NEB calculations) an additional namespace with linkname 'node_{foldername}' under which the files will be added.

Accepted options set by the metadata.options.parser_settings:

**parse_files (list) – A list of files that will be parsed from the** retrieved folder and added as nodes to the calculation output. Note that wilcards like 'W*.tmp' or even '*' are allowed! If not list is defined, by default, only the OUTCAR, CONTCAR and vasprun.xml files are added.

**build_parsing_list**()
Setup a list of files to be parsed based on the parsing list supplied by the parser's parse_file option

**linkname**(*filepath*)
Generate the output linkname under which the file at the given location will be available

**normalized_filename**(*filepath*)
Return a normalized version of the filename, i.e. lower case and .suffix replace with _suffix

**parse**(***kwargs*)
Parse the contents of the output files retrieved in the *FolderData*.

This method should be implemented in the sub class. Outputs can be registered through the *out* method. After the *parse* call finishes, the runner will automatically link them up to the underlying *CalcJobNode*.

**Parameters** **kwargs** – output nodes attached to the *CalcJobNode* of the parser instance.

**Returns** an instance of ExitCode or None

**parse_chgcar**(*filepath*)
Parsing hook triggered for files of type CHGCAR

**parse_contcar**(*filepath*)
Parsing hook triggered for files of type CONTCAR

**parse_generic**(*filepath*)
Generic parsing hook trigger for all files not featuring a specific parsing hook

**parse_outcar**(*filepath*)
Parsing hook triggered files of type OUTCAR

**parse_vasprun_xml**(*filepath*)
Parsing hook triggered files of type vasprun.xml

**parse_wavecar**(*filepath*)
Parsing hook triggered for files of type WAVECAR

**parsing_hook**(*filepath*)

**verify_and_set_parser_settings**()
Fail the parsing process if the parsin list if empty meaning no files will be parsed from the retrieved folder

### 11.1.5 aiida_cusp.utils package

**class** aiida_cusp.utils.**PotcarParser**(*path_to_potcar_file*, *name=None*, *functional=None*)
Bases: object

Minimalistic parsing library for VASP POTCAR files.

Only tries to parse the basic potential identifiers from the file, i.e. the potential version and the associated element. Does not perform any checks on the validity of the parsed results but only raises if no match was retrieved at all.

**Parameters**

- **path_to_potcar_file** (*pathlib.Path*) – path to a VASP POTCAR file
- **name** (*str*) – name of the potential (i.e. Li, Li_sv, . . . )
- **functional** (*str*) – functional of the potential, i.e. one of lda_us, lda, lda_52, lda_54, pbe, pbe_52, pbe_54, pw91 or pw91_us

**apply_quirks**()
Update invalid and erroneous parameters running the stored quirks

**hash_contents**()
Hash potential contents using secure hashing.

Calculates the hash based on the reduced instead of the full potential contents to avoid potentials being assumed as different if whitespaces differ.

**Returns** return sha256 hash for potential contents

**Return type** str

**load_potential_contents**()
Load POTCAR file contents.

**Returns** the contents of the potcar file

**Return type** str

**load_reduced_contents**()
Load POTCAR file contents and return a reduced version without consecutive whitespaces and occasionally occuring '^' chars

**Returns** returns the reduced contents of the potcar file

**Return type** str

**potential_element**()
Extract the element stored with the potential

**Returns** string containing the element name

**Return type** str

**Raises** ***PotcarParserError*** – if regex returns with no match

**potential_header**()
Extract header section from potential contents

**Returns** string containing the potential header

**Return type** str

**Raises** ***PotcarParserError*** – if regex returns with no match

**potential_title**()
Extract the potential title (i.e. the first line of the file)

**potential_version**()
Extract creation date from the potential header

**Returns** integer representation of the potential creation date

**Return type** int

**verify_parsed**()
Check if the parsed contents do make sense.

## Submodules

### aiida_cusp.utils.custodian module

Utility for translating plugin input parameters for Custodian and setting up the input scripts for the Custodian executable.

**class** aiida_cusp.utils.custodian.**CustodianSettings**(*vasp_cmd*, *stdout_fname*, *stderr_fname*, *settings={}*, *handlers={}*, *is_neb=False*)

Bases: object

Class to store Custodian settings and generate the required input files

Any error handlers may be passed as either a *list* containing the handler names or a *dict* with handler names set as keys and the corresponding item defining the handler-settings as *dict*. In case of the handlers being passed as a *list* the default settings for the defined handlers will be applied.

**Parameters**

- **vasp_cmd** (*list*) – list of commands requird to run the VASP calculation
- **stdout_fname** (*str*) – name of the file used to log messages send to stdout
- **stderr_fname** (*str*) – name of the file used to log messages send to stderr
- **settings** (*dict*) – settings for the Custodian job
- **handlers** (*list* or *dict*) – a set of error-correction handlers used to correct any errors occuring during the VASP calculation

**setup_custodian_handlers**(*handlers*)
Define error handlers used by custodian.

Accepts a list of handler names which will initialize the handlers with their corresponding default values. If *handlers* is given as dictionary keys are assumed to be of type handler name and the corresponding item to be of type *dict* containing the non-default parameters for the corresponding handler.

**Parameters** **handlers** (*dict* or *list*) – list of handler names or a dictionary of handlers and their corresponding non-default parameters

**Returns** dictionary of handler module paths and the corresponding handler setings for all handlers defined in the input *handlers*

**Return type** *dict*

**Raises**

- **CustodianSettingsError** – if input parameter handlers is not of the correct type (i.e. list, tuple or dict)
- **CustodianSettingsError** – if invalid parameter settings are found for a handler
- **CustodianSettingsError** – if an invalid handler name is found

**setup_custodian_settings** (*settings*)
Define settings for the custodian program.

Removes all Custodian program specific input parameters from the passed *settings* and complements missing parameters with given defaults.

**Parameters** **settings** (*dict*) – dictionary containing the settings for the Custodian program

**Returns** input settings for the Custodian program

**Return type** *dict*

**setup_vaspjob_settings** (*settings*)
Define settings for the VASP program.

Removes all VASP program specific input parameters from the passed *settings* and complements missing parameters with given defaults

**Parameters**

- **settings** (*dict*) – dictionary containing the settings for the VASP job
- **is_neb** (*bool*) – set to *True* if the VASP job is of type NEB

**validate_handlers** (*handlers*)
Check if *handlers* is empty, i.e. verify all handlers have been consumed.

**Parameters** **handlers** (*dict*) – dictionary of handlers and parameters

**Raises** **CustodianSettingsError** – if any remaining handler is found in the passed dictionary

**validate_settings** (*settings*)
Check if *settings* is empty, i.e. verify that all custodian and vasp job settinhs have been consumed.

**Parameters** **settings** (*dict*) – dictionary containing custodian and VASP job settings

**Raises** **CustodianSettingsError** – if any remaining setting is found in the passed dictionary

**write_custodian_spec** (*path_to_file*)
Generate custodian specification yaml-file.

Before writing the file all settings and handler contents are properly re-arranged such that the generated yaml-file is understood by the custodian command-line executable.

**Parameters** **path_to_file** –

**Raises** **CustodianSettingsError** – if the file defined by the passed *path_to_file* variable does not contain the .yaml suffix

**Returns** None

### aiida_cusp.utils.decorators module

Simple read-only implementation of a classproperty decorator as suggested on StackOverflow: https://stackoverflow.com/a/13624858

**class** aiida_cusp.utils.decorators.**classproperty**(*fget*)
Bases: object

### aiida_cusp.utils.defaults module

**class** aiida_cusp.utils.defaults.**CustodianDefaults**
Bases: object

Collection of default values for the custodian calculator comprising default job options, handlers and corresponding handler options.

**CUSTODIAN_SETTINGS = {'checkpoint': False, 'gzipped_output': False, 'max_errors': 1**

**ERROR_HANDLER_SETTINGS = {'AliasingErrorHandler': {'output_filename': 'aiida.out'},**

**HANDLER_IMPORT_PATH = 'custodian.vasp.handlers'**

**MODIFIABLE_SETTINGS = ['max_errors', 'polling_time_step', 'monitor_freq', 'skip_over_e**

**RUN_LOG_FNAME = 'run.log'**

**VASP_JOB_IMPORT_PATH = 'custodian.vasp.jobs.VaspJob'**

**VASP_JOB_SETTINGS = {'auto_continue': False, 'auto_gamma': False, 'auto_npar': Fals**

**VASP_NEB_JOB_IMPORT_PATH = 'custodian.vasp.jobs.VaspNEBJob'**

**VASP_NEB_JOB_SETTINGS = {'auto_continue': False, 'auto_gamma': False, 'auto_npar': 1**

**class** aiida_cusp.utils.defaults.**PluginDefaults**
Bases: object

**CSTDN_SPEC_FNAME = 'cstdn_spec.yaml'**

**NEB_NODE_PREFIX = 'node_'**

**NEB_NODE_REGEX = re.compile('^node_[0-9]{2}$')**

**PARSER_OUTPUT_NAMESPACE = 'parsed_results'**

**STDERR_FNAME = 'aiida.err'**

**STDOUT_FNAME = 'aiida.out'**

**class** aiida_cusp.utils.defaults.**VaspDefaults**
Bases: object

Collection of default values for VASP

**FNAMES = {'bsefatband': 'BSEFATBAND', 'chg': 'CHG', 'chgcar': 'CHGCAR', 'contcar':**

**FUNCTIONAL_MAP = {'potpaw_gga': 'pw91', 'potpaw_lda': 'lda', 'potpaw_lda.52': 'lda_**

**class** aiida_cusp.utils.defaults.**VasprunParsingDefaults**
Bases: object

Default settings used to parse vasprun.xml files

**PARSER_ARGS = {'exception_on_bad_xml': False, 'ionic_step_offset': 0, 'ionic_step_sk**

### aiida_cusp.utils.exceptions module

Custom exceptions

**exception** aiida_cusp.utils.exceptions.**CommandLineError**
Bases: Exception

Exception raised by CLI commands.

**exception** aiida_cusp.utils.exceptions.**CustodianSettingsError**
Bases: Exception

Exception raised by the CustodianSettings class.

**exception** aiida_cusp.utils.exceptions.**IncarWrapperError**
Bases: Exception

Exception raised by the Incar wrapper class.

**exception** aiida_cusp.utils.exceptions.**KpointWrapperError**
Bases: Exception

Exception raised by the KpointWrapper class.

**exception** aiida_cusp.utils.exceptions.**MultiplePotcarError**
Bases: Exception

Exception raised if POTCAR file is already stored in the database.

**exception** aiida_cusp.utils.exceptions.**PoscarWrapperError**
Bases: Exception

Exception raised by the PoscarWrapper class.

**exception** aiida_cusp.utils.exceptions.**PotcarParserError**
Bases: Exception

Exception raised by the PotcarParser class.

**exception** aiida_cusp.utils.exceptions.**PotcarPathError**
Bases: Exception

Exception raised by the PotcarPathParser class.

**exception** aiida_cusp.utils.exceptions.**VaspPotcarDataError**
Bases: Exception

Exception raised by the VaspPotcarData class.

**exception** aiida_cusp.utils.exceptions.**VaspPotcarFileError**
Bases: Exception

Exception raised by the VaspPotcarFile class.

### aiida_cusp.utils.potcar module

Utility module implementing several methods for parsing and interacting with VASP potential files.

**class** aiida_cusp.utils.potcar.**PotcarParser**(*path_to_potcar_file*, *name=None*, *functional=None*)
Bases: object

Minimalistic parsing library for VASP POTCAR files.

Only tries to parse the basic potential identifiers from the file, i.e. the potential version and the associated element. Does not perform any checks on the validity of the parsed results but only raises if no match was retrieved at all.

**Parameters**

- **path_to_potcar_file** (*pathlib.Path*) – path to a VASP POTCAR file
- **name** (*str*) – name of the potential (i.e. Li, Li_sv, ...)
- **functional** (*str*) – functional of the potential, i.e. one of lda_us, lda, lda_52, lda_54, pbe, pbe_52, pbe_54, pw91 or pw91_us

**apply_quirks**()
Update invalid and erroneous parameters running the stored quirks

**hash_contents**()
Hash potential contents using secure hashing.

Calculates the hash based on the reduced instead of the full potential contents to avoid potentials being assumed as different if whitespaces differ.

**Returns** return sha256 hash for potential contents

**Return type** str

**load_potential_contents**()
Load POTCAR file contents.

**Returns** the contents of the potcar file

**Return type** str

**load_reduced_contents**()
Load POTCAR file contents and return a reduced version without consecutive whitespaces and occasionally occuring '^' chars

**Returns** returns the reduced contents of the potcar file

**Return type** str

**potential_element**()
Extract the element stored with the potential

**Returns** string containing the element name

**Return type** str

**Raises** ***PotcarParserError*** – if regex returns with no match

**potential_header**()
Extract header section from potential contents

**Returns** string containing the potential header

**Return type** str

**Raises** ***PotcarParserError*** – if regex returns with no match

**potential_title**()
Extract the potential title (i.e. the first line of the file)

**potential_version**()
Extract creation date from the potential header

**Returns** integer representation of the potential creation date

**Return type** int

**verify_parsed**()
Check if the parsed contents do make sense.

**class** aiida_cusp.utils.potcar.**PotcarPathParser**(*potcar_file_path*)
Bases: object

Utility class for parsing POTCAR file paths.

Assumes a POTCAR file path of the form /path/[FUNCTIONAL]/[NAME]/POTCAR with [FUNCTIONAL] containing the archive name of the potential libraries as shipped by VASP, i.e. one of potuspp_lda, potpaw_lda, potpaw_lda.52, potpaw_lda.54, potpaw_pbe, potpaw_pbe.52, potpaw_pbe.54, potuspp_gga or potpaw_gga. Tries to determine the potential's functional and name from the given path and raises an exception if this fails.

**Parameters** **potcar_file_path** (*pathlib.Path*) – file path pointing to a VASP pseudo-potential

**Raises** ***PotcarPathError*** – if the path does not point to a POTCAR file or if the path does not contain a valid archive name to identify the corresponding functional

**parse_functional**(*path*)
Parse the potential's functional type from the given path.

Search the given path for functional identifiers corresponding to the archive names chosen by the pseudo-potential libraries shipped with VASP.

**Parameters** **path** (*pathlib.Path*) – file path pointing to a VASP pseudo-potential

**Returns** plugin internal functional identifier

**Return type** str

**Raises** ***PotcarPathError*** – if no valid functional identifier can be found in the given path.

**parse_name**(*path*)
Parse the unique functional name from the path.

Simply returns the name of the parent folder containing the POTCAR file.

**Parameters** **path** (*pathlib.Path*) – file path pointing to a VASP pseudo-potential

**Returns** the pseudo-potential's unique name

**Return type** str

**validate_path_is_potcar_file**(*path*)
Check if the given path points to a file of type POTCAR .

**Parameters** **path** (*pathlib.Path*) – file path pointing to a VASP pseudo-potential

**Raises** ***PotcarPathError*** – if the parser cannot extract the functional from the path or if the path does not point to a potential

### aiida_cusp.utils.single_archive_data module

Extension to AiiDA's SinglefileData storing the given file as gzip-compressed archive instead of simply storing the file as-is

**class** aiida_cusp.utils.single_archive_data.**SingleArchiveData**(*file*, *filename=None*, ***kwargs*)
Bases: aiida.orm.nodes.data.singlefile.SinglefileData

Data class used to store the contents of a file as gzip compressed archive in its repository

**ARCHIVE_SUFFIX = '.gz'**

**filepath**
Return the path of the object in the repository.

Basically replicates the procedure in the internal open method without the final call to the open()

**get_compressed_file_contents** (*filepath*)
Load the file and return a gzip compressed version of the contents

**Parameters** **filepath** (*pathlib.Path*) – path to the file whose contents will be loaded and compressed using gzip

**Returns** a tuple containing the compressed file contents and the file's name with '.gz' suffix appended

**Return type** tuple

**get_content** (*decompress=True*)
Load the node and return either the archive (i.e. compressed) or the file (i.e. the decompressed) contents stored in the node.

**Parameters** **decompress** (*bool*) – Indicate wether comressed or uncompressed contents are returned

**set_file** (*file*, *filename=None*)
Compress given file and store it to the node's repository.

This method differs from the original SinglefileData.set_file() method in that it only accepts a filepath. The additional filename argument is only here to make AiiDA happy and is ignored since we only allow file path inputs from which the filename is deduced.

**Parameters** **file** (*pathlib.Path*) – path to the file whose contents will be compressed and stored to the repository

**Raises** **ValueError** – if the given path does not point to a file

**write_file** (*filepath*, *decompress=True*)
Write the node's contents to a file

**Params filepath** path to the output file to which the stored contents will be written

**Params decompress** write the decompressed or the archive contents to the file

CHAPTER 12

# Indices and tables

- genindex
- modindex
- search

# Python Module Index

## a

# Index

## D



## E



## F



## G

## H



## I



## K



## L

## M



## N



## P

## R



## S



## V

## W