AiiDA VASP Plugin Documentation

Release 0.1

Mario Žic

Aug 31, 2017

Contents

1 Introduction **3**
- 1.1 AiiDA VASP Plugin Introduction . . . 3

2 Installation **5**
- 2.1 List of dependencies . . . 5
- 2.2 Installing the Plugin . . . 5

3 Users **7**
- 3.1 Basic Usage . . . 7
- 3.2 Preparing Pymatgen Input . . . 7
- 3.3 Preparing AiiDA calculation . . . 8

4 Default Behaviour **11**
- 4.1 Plugin's Default Behaviour . . . 11
- 4.2 Mandatory & Optional Input . . . 11
- 4.3 Parser Instructions . . . 11

5 Advanced Users **15**
- 5.1 Introduction to Parser Instructions . . . 15

6 Developers **19**
- 6.1 Introduction . . . 19
- 6.2 Input Plugin . . . 19
- 6.3 Output Plugin . . . 19

7 Indices and tables **21**

AiiDA VASP Plugin is an AiiDA extension, writen in Python, which provides a user friendly and flexible way to run highly automated density functional theory (DFT) calculations using the Viena Ab initio Simulation Package, VASP . The code consists of two main parts, the Input and the Output plugin, which facilitate VASP calculation setup and output postprocessing within the AiiDA framework.

Warning: This plugin is still in its early testing phase !!!

CHAPTER 1

Introduction

Covers a brief overview of the main functionality provided by the plugin.

AiiDA VASP Plugin Introduction

The AiiDA VASP Plugin aims to provide a full support for running VASP calculations using the AiiDA package.

The User Interface is based heavily on the VASP support already provided by the Materials Project through pymatgen. Main reasons for this are:

- to support a **standard VASP interface** across different high-throughput frameworks.
- to avoid code duplication and utilize the mature and well supported pymatgen code base.

The *philosophy* behind this plugin can be summarised as follows: *Take a well defined VASP input and return parsed data in a form the User has specified - no more, no less.*

In order to facilitate the latter, we have implemented **Parser Instructions**, a simple way to customize and extend the parsing capabilities of the output parser.

CHAPTER 2

Installation

A quick guide to get you running.

List of dependencies

We assume you are running alinux distribution and have a working installation of AiiDA (version u‘5.0’, and above).

In addition to AiiDA, this plugin requires pymatgen (version u‘3.2.10’, and above), to be installed.

Installing the Plugin

The plugin consists of two parts: the input and the output plugin. These can be found in correspondingly named directories at the top-level of the AiiDA VASP Plugin distribution directory.

There are two ways to add the VASP Plugin to AiiDA. You can either *symlink* or *copy* the plugin directories to the appropriate AiiDA distribution directory. The **directories must be renamed** as explained below. The package imports inside the modules will be broken if this is not done correctly!

Note: In the following we will assume that the AiiDA VASP Plugin is located inside the *~/aiida_vasp_plugin/* directory and the AiiDA distribution is located inside the *~/aiida_dir/*. **Please modify these paths correspondingly!**

To proceed with the installation follow method A or method B, according to your preferences.

Method A - Symlink

You need to do the following symlink:

```
ln -s ~/aiida_vasp_plugin/input ~/aiida_dir/aiida/orm/calculation/job/vasp
```

for the input plugin, and:

```
ln -s ~/aiida_vasp_plugin/output ~/aiida_dir/aiida/parsers/plugins/vasp
```

for the output plugin.

If everything went ok, now you whould be able to import these modules from your python console.

Method B - Copy

First you need to copy the input plugin:

```
cp -r ~/aiida_vasp_plugin/input ~/aiida_dir/aiida/orm/calculation/job/vasp
```

and then the output plugin:

```
cp -r ~/aiida_vasp_plugin/output ~/aiida_dir/aiida/parsers/plugins/vasp
```

to the corresponding AiiDA directories.

If everything went ok you should be able to import these modules from your python console.

Check installation

Open your python console and try to import the packages:

```
import aiida.orm.calculation.job.vasp
import aiida.parsers.plugins.vasp
```

Note: If you get an error here please check that you have provided the correct paths to the copy/symlink commands above.

CHAPTER 3

Users

An example based introduction to the code usage. Covers the plugin logic and the basics needed to run the VASP code.

Basic Usage

In this section we cover the basics of setting up a VASP calculation using the plugin. The procedure can be logically split into two steps. The first step is to set up VASP input using the VASP interface provided by the pymatgen package. In the second step these objects, together with a set of user specified *output parser instructions*, are passed as an input to the AiiDA calculation.

Note: **The pymatgen syntax will not be covered here in great detail!** - just a short use-case example will be provided. For more details on pymatgen we refer you to pymatgen documentation.

Preparing Pymatgen Input

A short example of setting up pymatgen VASP input is given below. The goal is to create: *POSCAR*, *INPUTCAR*, *KPOINTS*, and *POTCAR* files, which represent a minimal input for any VASP calculation.

An excerpt from the full code is shown below to illustrate the input setup procedure:

```
import numpy as np
#
# Pymatgen imports
import pymatgen as mg
from pymatgen.io import vasp as vaspio
#
# POSCAR
lattice_constant = 5.97
lattice = lattice_constant * np.array([
```

```
    [0.0, 0.5, 0.5],
    [0.5, 0.0, 0.5],
    [0.5, 0.5, 0.0]
])
lattice = mg.Lattice(lattice)

struct = mg.Structure(
    lattice,
    [Mn, Mn, Ga],
    # site coords
    [[0.00, 0.00, 0.00], [0.25, 0.25, 0.25], [0.50, 0.50, 0.50]]
)
poscar = vaspio.Poscar(struct, comment='cubic Mn2Ga')

# KPOINTS
kpoints = vaspio.Kpoints.monkhorst_automatic(
    kpts=(10, 10, 10), shift=(0.0, 0.0, 0.0)
)
```

Therefore, for each VASP input file we have a pymatgen object representing it, e.g. *KPOINTS* is represented by the *pymatgen.io.vasp.Kpoints* object. Our task here is just to provide basic information needed to construct the VASP input files.

Full code used for this example can be found here .

Preparing AiiDA calculation

The aim of this section is to set up a working AiiDA calculation. We will assume that all pymatgen objects representing the VASP input have already been created. Our task then is to create a VASP calculation object and pass it the content of the pymatgen input files.

Before we pass the input files to the AiiDA calculation we need to **split** the *POSCAR* file, since it may contain both dictionary and array data. This is achieved by the *disassemble_poscar* function which returns a dictonary of *POSCAR* parts. It its important to note that each of these parts is already an instance of AiiDA's *Data* class and can be directly stored in the AiiDA database. The split is done like this:

```
# AiiDA imports
from aiida.orm import Code, DataFactory
from aiida.orm.calculation.job.vasp import vasp as vplugin
from aiida import load_dbenv
load_dbenv()

# split the poscar for AiiDA serialization
poscar_parts = vplugin.disassemble_poscar(poscar)
```

Note: This intermediate step represents only a transitional solution which will be improved in future versions!

The next step is to create an instance of the AiiDA VASP calculation and pass it the input files. The code to do this is shown below:

```
# split the poscar for AiiDA serialization
poscar_parts = vplugin.disassemble_poscar(poscar)

# === Prepare Calculation
```

```
ParameterData = DataFactory('parameter')
StructureData = DataFactory('structure')

codename = 'Vasp'  # this may be differ from user-to-user
code = Code.get(codename)  # executable to call, module imports etc

calc = code.new_calc()
calc.label = "VASP plugin development"
calc.description = "Test input plugin"
calc.set_max_wallclock_seconds(5*60)  # 5 min
calc.set_resources({
    "num_machines": 1,
    "num_mpiprocs_per_machine": 1,
    'num_cores_per_machine': 24  # this will differ from machine-to-machine
})
calc.set_withmpi(True)

calc.use_poscar(poscar_parts['poscar'])
calc.use_structure(poscar_parts['structure'])
calc.use_incar(
    ParameterData(dict=incar.as_dict())
)
calc.use_kpoints(
    ParameterData(dict=kpoints.as_dict())
)
calc.use_potcar(
    ParameterData(dict=potcar.as_dict())
)

# settings
settings = {'PARSER_INSTRUCTIONS': []}
pinstr = settings['PARSER_INSTRUCTIONS']
pinstr.append({
    'instr': 'dummy_data',
    'type': 'data',
    'params': {}
})

# additional files to return
settings.setdefault(
    'ADDITIONAL_RETRIEVE_LIST', [
        'OSZICAR',
        'CONTCAR',
        'OUTCAR',
        'vasprun.xml'
        ]
)
calc.use_settings(ParameterData(dict=settings))
```

The calculation can now be submitted.

What is **important to notice** are the *calc.use_method*'s which are specific to the VASP plugin. These can be logically divided into four groups:

- *use_incar, use_potcar, use_kpoints* - passed as a *ParameterData* object, which store the *dict* representation of the pymatgen object
- *use_poscar, use_structure, use_structure_extras* - passed as correspondingly named objects in the *poscar_parts* dict, which was obtained by splitting up the *POTCAR* object. **Note:** the *structure_extras* in the example is not

shown because this data is optional, i.e. it may contain array data that can be found in the *CONTCAR* file, e.g. the final velicities of ions, etc.

- *use_settings* - pased as *ParameterData*. Used to specify additional files to retreive and output parser instructions.
- *use_chgcar*, *use_wavecar* - passed as a *SinglefileData* object. See the next section for more details on using these inputs.

Full code used for this example can be found here .

CHGCAR and WAVECAR Files

The *CHGCAR* and *WAVECAR* files are usually used for continuation runs. The plugin treats them as an *optional input*. The *SinglefileData* object can be created like this:

```
from aiida.orm.data.singlefile import SinglefileData

input_file = SinglefileData()
input_file.set_file('path/to/the/file/CHGCAR')
```

The *input_file* now points to the actual file on the disc and will be copied to the AiiDA database when the calculation's *store_all* method is called. It is important to note here that we **must** have an input *CHACAR/WAVECAR* file written at some location on the disc before we can create a *SinglefileData* object.

Once we have created a *SinglefileData* representation of the *CHACAR/WAVECAR* file we can pass it to AiiDA as an input like this:

```
chgcar = SinglefileData()
chgcar.set_file('path/to/the/file/CHGCAR')
...
calc.use_chgcar(chgcar)
```

and similarly for the *WAVECAR* file.

CHAPTER 4

Default Behaviour

An overview of the features that may confuse you.

Plugin's Default Behaviour

Plugin's default behaviour includes every bit of plugin's behaviour which is not obvious to the user. These are mostly intended as commodity features, but for a new user it may lead to confusion.

Note: Here we try to document all "hidden" features of the plugin which may result with undesired outcome for the user. **If a feature is not well documented, then it should be treated as a bug in the documentation!**

Mandatory & Optional Input

Preparation of the input for an AiiDA calculation was described *here*. The main thing to recall is that we use the *calc.use_method*'s to pass the input to the AiiDA calculation object. It was noted that we did not use the *calc.use_structure_extras* method in that example. Therefore, the *structure_extras* is an *optional* input parameter. Currently this is the only optional input !

Note: All inputs are mandatory, except the structure_extras !

Parser Instructions

A significant fraction of implemented default behaviours is related to parser instructions. The parser instructions are used to specify how the calculation output is to be parsed. It is reasonable to assume that some basic quantities like energy and magnetization, will be of interest to all users. Hence, it is convenient to define *default parsers*. Since we

divide the output into three parts: *errors, data, and structure*, we have an assigned default parser for each of these data types. More on these output types can be found *here*.

Instruction Specification

What may be unusual is that **we require at least one parser to be specified for each of the data types above**. This involves a number of default behaviours:

1. If nothing is specified, the output plugin will load a default parser for each of the data types.
2. If only some of the parsers are not provided, the output plugin will load a default parser for the missing data type.
3. *However*, if a provided instruction can not be loaded during the calculation submission, e.g. if the calculation does not conform to the specifications or there is a typo in the input, the input plugin will *raise an exception*.

Note: The plugin will forgive you if a required instruction is not specified, and load defaults for you, *but it will crash* if you specify an instruction that can not be loaded.

Dummy Instructions

There are perfectly legitimate reasons not to parse the whole output, we only require the user to be explicit about it !

The dummy instructins are introduced to allow the user to express that he does not want to parse something, e.g. the user decides not to parse the output structure because he is doing static calculations and he wants to save memory space.

The output nodes are still created, but they are empty! *(This may change in future versions!)*

Note: Dummy instructions allow the user to skip the parsing !

Static Instructions

Static parser instructions require the developer to specify the files which are required for parsing. This is used to automatically fetch the required files for the output parser. *The user does not need to provide any additional input !*

Note: Mandatory input files are automatically appended to the retrieve list !

Instruction Execution

The output parsing is concieved as a loop that iterates over the list of parser instructions and executes them one-by-one, in an arbitrary order. What happens if an instruction crashes during the execution ?! There are two possible scenarios:

1. the instruction handled the crash on it's own and returned without raising an exception. In this case you should probably expect to see an error message in the error log for that instruction, e.g. in the *errors@TheCrashedInstruction* node, but this depends on the implementation of that particular instruction.
2. the instruction didn't handle the crash. In this case the error log inside the *errors@TheCrashedInstruction* node will probably not be complete an we will have an additional error inside the *parser_warnings* node, created by the output parser itself, giving extra information about the crash.

Warning: **Always check for output parsing errors !** (If an instruction crashes the output parser will just move to the next instruction.)

CHAPTER 5

Advanced Users

Extends the topic of Parser Instructions and gives an example of a new parser instruction implementation.

Note: Knowledge of Python programing is assumed.

Introduction to Parser Instructions

Parser instructions are the central concept of the output plugin. They provide the output plugin with an easily configurable, and extensible, parsing functionality. The role of the output plugin can thus be understood, in simple terms, as a boiler-plate needed to load, execute, and store the results returned by the parser instructions.

Note: *Parsing of the output is achieved by executing a sequence of parser instructions!*

Specifying the Parser Instruction Input

In order to customize the output parsing process we need to specify which instructions should be used as a part of the input. The instructions are specified using a special key *PARSER_INSTRUCTIONS*, within the *settings* input node, as shown below:

```
settings = {'PARSER_INSTRUCTIONS': []}
instr = settings['PARSER_INSTRUCTIONS']  # for easier access
instr.append({
    'instr': 'dummy_data_parser',
    'type': 'data',
    'params': {}
})

...
```

```
calc.use_settings(ParameterData(dict=settings))
```

Where the *calc* is an instance of the *VaspCalculation* class.

In the example above we are appending a single *data* parser instruction called *dummy_data_parser*. **The parser instructions are supposed to be specified as a dictionary with three keys:** *instr*, *type*, and *params*.

Currently there are three parser instruction types implemented: *data*, *error*, and *structure*. The distinction between these types comes into play during the instruction loading, where the output parser appends different auxiliary parameters to the instruction based on its type. For example, to every *error* type instruction a *SCHED_ERROR_FILE* parameter is appended. More information about the plugin's default behaviour can be found *here*.

Defining New Parser Instructions

All parser instructions inherit from the base class, *BaseInstruction*, which provides the interface towards the output plugin. Therefore, the *BaseInstruction* is *a template for implementing custom parser instructions*.

In order to implement a new parser instruction one must inherit from the base class and overrride the two following things:

1. list of input files, given by the class property *_input_file_list_*, or by setting the *_dynamic_file_list_ = True*, when the names of the input files are not known in advance.
2. override the *BaseInstruction._parser_function(self)*, which is a method that is called when the instruction is executed - it implements the actual parsing of the output.

Below we give examples on how to implement these two different instruction types.

Note: In future versions we may implement *BaseInstruction* subclasses for each instruction type, i.e. *StaticInstruction* and *DynamicInstruction*, in order to be explicit about our intents.

Static Parser Instruction

Static parser instruction is just an ordinary parser instruction for which we can specify the list of input file names in advance, i.e. the *input file names are static*.

The use of this method is advantageous since the input plugin will automatically update the list of files to be retreived, and the instruction itself will automatically check if the required files are present before the parsing starts.

Since the **static parser instructions** offer both the user commodity and additional safeties against an invalid user input, they should be **prefered** over the dynamic parser instructions!

Example:

The *Default_vasprun_parserInstruction* is an example of the static parsing instruction. It operates only on the statically named *vasprun.xml* file.

First thing in defining a static parsing instruction is to override the *_input_file_list_*:

In the case above the only input file is the *vasprun.xml*.

Next follows the implementation of the *_parser_function*. The *_parser_function* implements the output parsing logic. This part depends only on the user preferences and does not depend on the internal working of the AiiDA.

Finally, **the output** must be returned as a tuple:

The *nodes_list* is just an arbitrary *list of tuples*, e.g. *[('velocities', ArrayData_type), ('energies', ParameterData_type), ...]*, where the second tuple value needs to be an instance of the AiiDA's *Data* type.

The second item in the return tuple is the *parameter_warnings* object, which is just a dictionary in which we can log useful information, e.g. non-critical errors, during the instruction execution. For example:

```
parser_warnings.setdefault('error name', 'details about the error')
```

After the instruction returns, the parser warnings are converted to a node, *('errors@Default_vasprun_parserInstruction', ParameterData(parser_warnings))*, which is stored in the AiiDA database as a part of the output.

Note: In summary, static parser instruction is implemented by overriding the *_input_file_list_* and the *_parser_function*. The parsed output must be returned in a format described above.

Dynamic Parser Instruction

Dynamic parser instruction differ from the static parser instructions in that the *input file names must be provided by the user* as an instruction parameter during the instruction specification, i.e. during the VASP calculation setup. This represents and overhead and allows for a typo to cause an instruction execution crash during the output parsing. For this reason the static parser methods should be used whenever that is possible.

Example:

The *Default_error_parserInstruction* is an example of the dynamic parser instruction.

The whole code is given below:

First the *_dynamic_files_list_* is set to *True*, followed by the *_parser_function* implementation:

1. get the input file name, *self._params['SCHED_ERROR_FILE']*, to open for parsing. (See the note below.)
2. read the whole standard error file. We could be looking for a particular kind of error here instead!
3. set up the output node list and return. In this example only one node, *runtime_errors*, is created. The *parser_warnings* is just an empty dictionary.

Note: The *SCHED_ERROR_FILE* parameter is appended automatically by the output parser to every *error* instruction type. This is an example of the *default behaviour*.

CHAPTER 6

Developers

An in-depth cover of the plugin implementation.

Note: Advanced Python programing skills and familiarity with the AiiDA internals are assumed.

TODO !

Introduction

This section is a stub !!!

TODO!

Input Plugin

Few words about the input plugin.

Input Plugin Introduction

Cover the basic concepts here.

Output Plugin

Few words about the output plugin.

Output Plugin Introduction

Cover the basic concepts here.

CHAPTER 7

Indices and tables

- genindex
- modindex
- search