AiiDa-FLEUR Documentation

Release 1.1.3

The AiiDA-FLEUR team.

Jan 19, 2021

Contents

The AiiDA-FLEUR python package enables the use of the all-electron Density Functional Theory (DFT) code FLEUR (http://www.flapw.de) with the AiiDA framework (http://www.aiida.net).

It is open source under the MIT license and is available under (https://github.com/JuDFTteam/aiida-fleur). The package is developed within the MaX EU Center of Excellence (www.max-center.eu) at Forschungszentrum Jülich GmbH (http://www.fz-juelich.de/pgi/pgi-1/DE/Home/home_node.html), (IAS-1/PGI-1), Germany. Check out the AiiDA registry to find out more about what other packages for AiiDA exists, that might be helpful for you.

If you use this package please cite:

- The plugin and workflows:

  J. Broeder, D. Wortmann, and S. Blügel, Using the AiiDA-FLEUR package for all-electron ab initio electronic structure data generation and processing in materials science, In Extreme Data Workshop 2018 Proceedings, 2019, vol 40, p 43-48

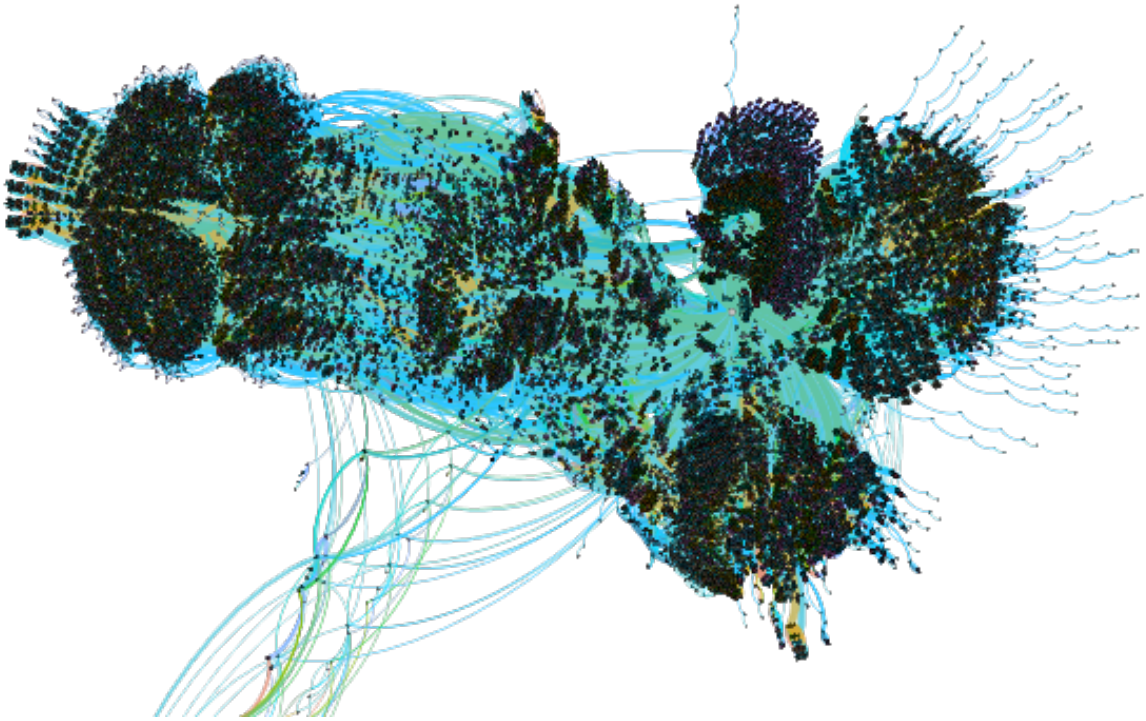
- The FLEUR code: http:/www.flapw.de

CHAPTER 1

Features, Illustrations, Usage examples:

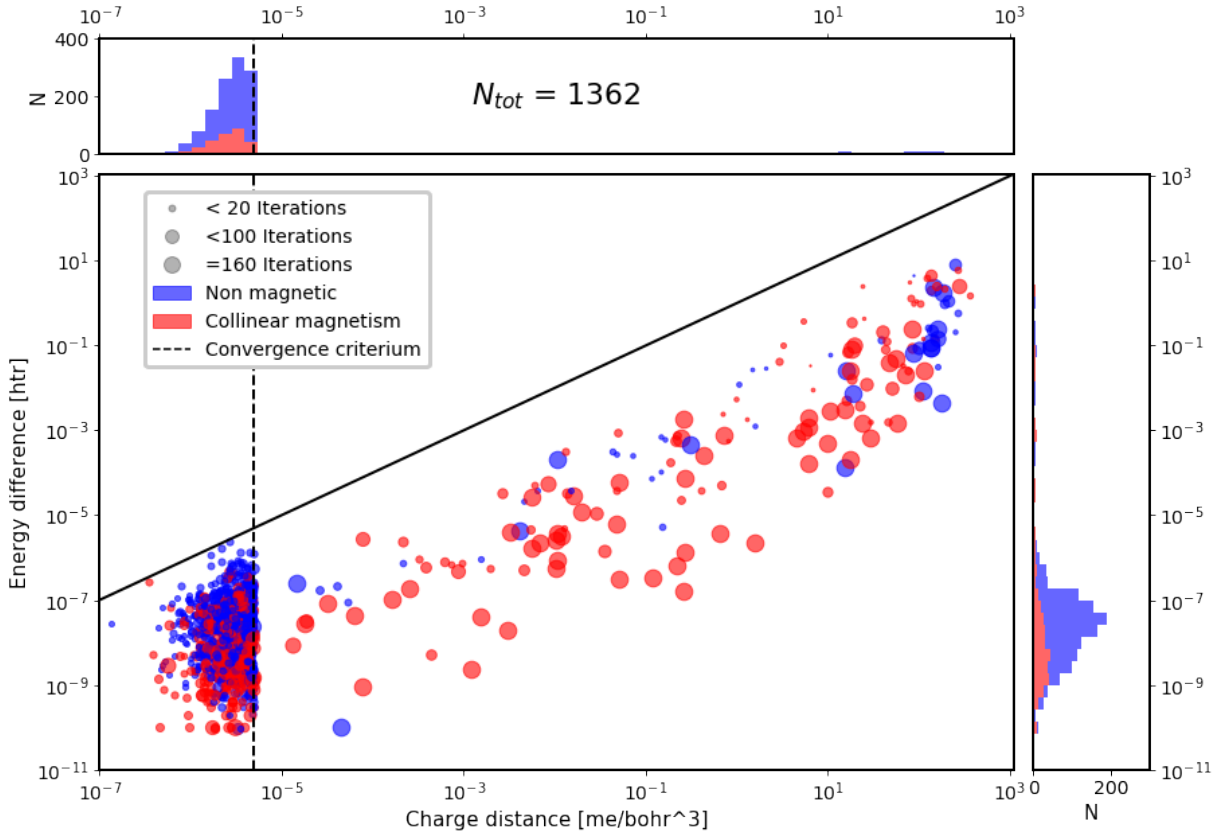
Example 1, Full Provenance tracking trough AiiDA:

AiiDA graph visualization of a small database containing about 130 000 nodes from Fleur calculations. (Visualized with Gephi)



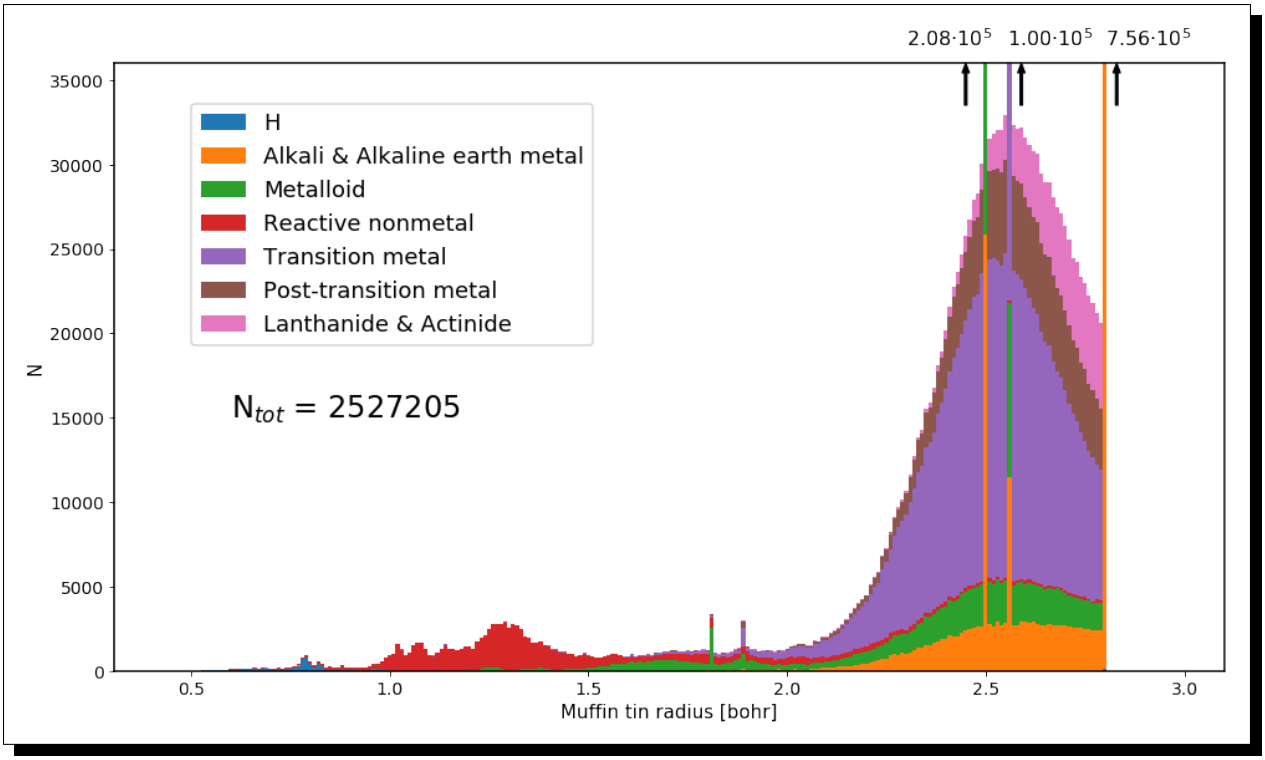
Example 2, Material screening:

Fleur SCF convergence of 1362 different screened Binary systems managed by the scf workchain



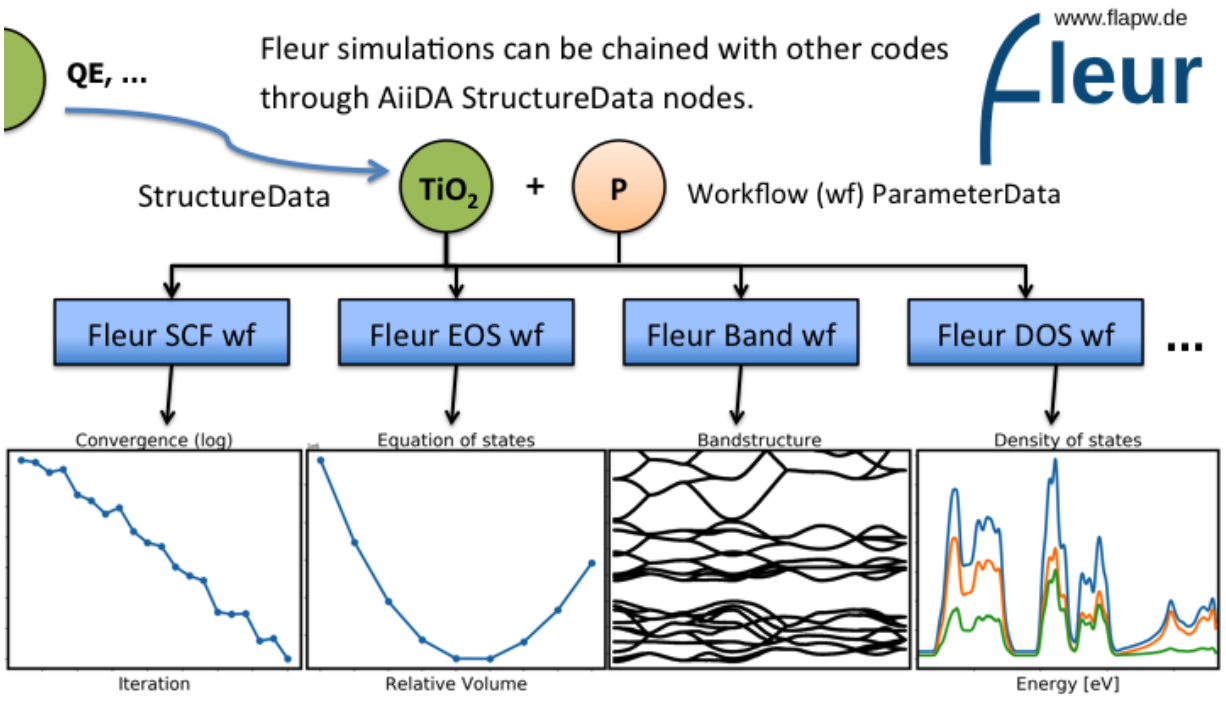
Example 3 Method robustness, tuning:

FLAPW muffin tin radii for all materials (>820000) in the OQMD .



Example 4, DFT Code Interoperability:

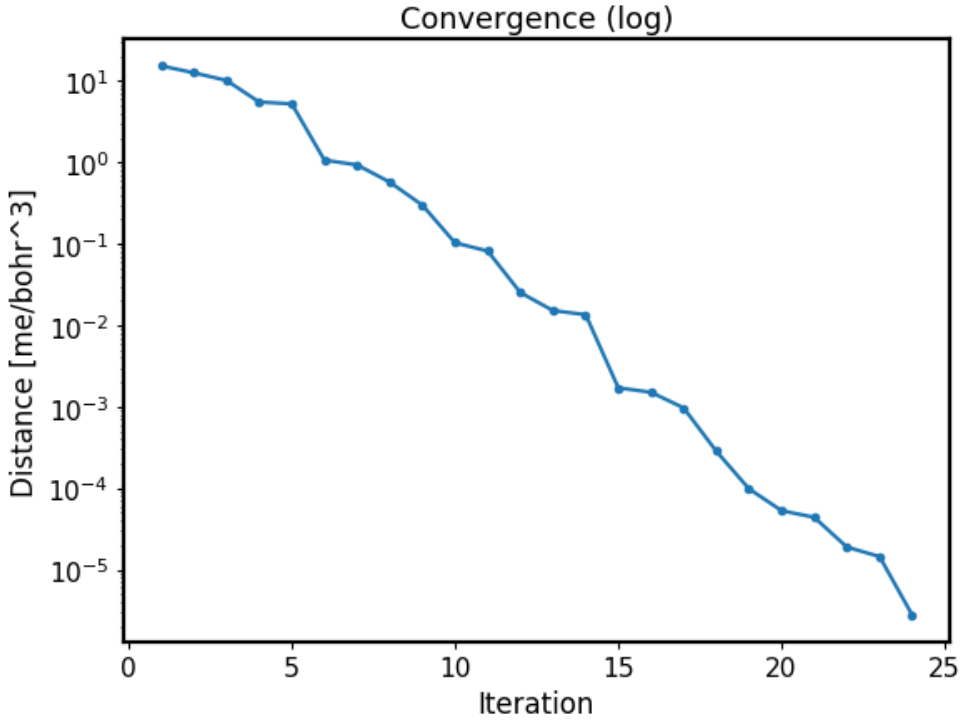
If an DFT code has an AiiDA plugin, one can run successive calculations using different codes. For example, it is possible to perform a structure relaxation with VASP or Quantum Espresso and run an all-electron FLEUR workflow for the output structure.



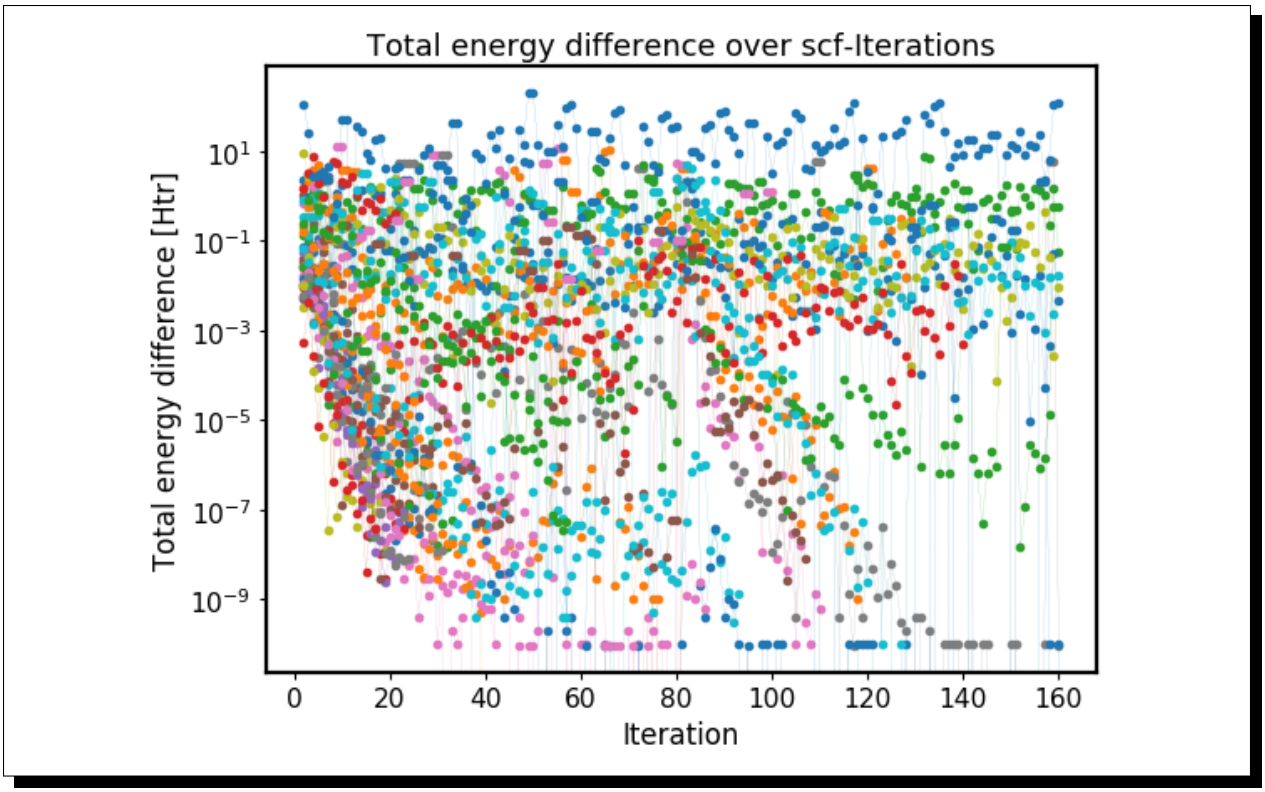
Example 5, Quick Visualizations:

AiiDA-FLEUR contains a function ('plot_fleur') to get a quick visualization of some database node(s). For example, to make a convergence plot of one or several SCF runs in your scripts, or notebook.:

```
plot_fleur(scf_node)
```




```
plot_fleur(scf_node_list)
```


Total energy difference over scf-Iterations
Total energy difference [Htr]
10^{1}
10^{-1}
10^{-3}
10^{-5}
10^{-7}
10^{-9}
0
20
40
60
80
100
120
140
160
Iteration

CHAPTER 2

Basic overview

2.1 Requirements to use this code:

- A running AiiDA version (and postgresql database)
- Executables of the Fleur code

Other packages (in addition to all requirements of AiiDA):

- lxml
- ase
- masci-tools

2.2 AiiDA-package Layout:

1. *Fleur input generator*
2. *FleurinpData structure*
3. *Fleur code*

The overall plugin for Fleur consists out of three AiiDA plugins. One for the Fleur input generator (inpgen), one data structure (fleurinpData) representing the inp.xml file and a plugin for the Fleur code (fleur, fleur_MPI). Other codes from the Fleur family (GFleur) or which build on top (Spex) are not supported.

The package also contains workflows

1. *Fleur base workchain*
2. *Self-Consistent Field* (Scf)
3. *Density Of States* (DOS)
4. *Structure optimization* (relax)
5. *Band structure*

6. *Equation of States* (EOS)
7. *Initial corelevel shifts*
8. *Corehole*
9. Force-theorem *Magnetic Anisotropy Energy*
10. Force-theorem *Spin Spiral Dispersion*
11. Force-theorem *Dzjaloshinskii-Moriya Interaction energy dispersion*
12. Scf *Magnetic Anisotropy Energy*
13. Scf *Spin Spiral Dispersion*

The package also contains AiiDA dependent tools around the workflows and plugins. All tools independent on aiida-core are moved to the masci-tools repository, to be available to other non AiiDA related projects and tools.

2.3 Acknowledgments:

We acknowledge partial support from the EU Centre of Excellence "MaX – Materials Design at the Exascale" (http://www.max-centre.eu). (Horizon 2020 EINFRA-5, Grant No. 676598). We thank the AiiDA team for their help and work. Also the vial exchange with developers of AiiDA packages for other codes was inspiring.

CHAPTER 3

User's Guide

Everything you need for using AiiDA-FLEUR

3.1 User's guide

This is the AiiDA-FLEUR user's guide.

3.1.1 Getting started

3.1.1.1 Installation of AiiDA-FLEUR

To use AiiDA, it has to be installed on your local machine and configured properly. The detailed description of all required steps can be found in the AiiDA documentation. However, a small guide presented below shows an example of installation of AiiDA-FLEUR.

Installation of python packages

First of all, make sure that you have all required libraries that are needed for AiiDA.

Note: If you use a cooperative machine, you might need to contact to your IT department to help you with setting up some libraries such as postgres and RabbitMQ.

In order to safely install AiiDA, you need to set up a virtual environment to protect your local settings and packages. To set up a python3 environment, run:

```
python3 -m venv ~/.virtualenvs/aiidapy
```

This will create a directory in your home directory named `.virtualenvs/aiidapy` where all the required packages will be installed. Next, the virtual environment has to be activated:

```
source ~/.virtualenvs/aiidapy/bin/activate
```

After activation, your prompt should have `(aiidapy)` in front of it, indicating that you are working inside the virtual environment.

To install the latest official releases of AiiDA and AiiDA-FLEUR, run:

```
(aiidapy)$ pip install aiida-fleur>=1.0
```

The command above will automatically install AiiDA itself as well since AiiDA-FLEUR has a corresponding requirement.

If you want to work with the development version of AiiDA-FLEUR, you should consider installing AiiDA and AiiDA-FLEUR from corresponding GitHub repositories. To do this, run:

```
(aiidapy)$ mkdir <your_directory_AiiDA>
(aiidapy)$ git clone https://github.com/aiidateam/aiida-core.git
(aiidapy)$ cd aiida_core
(aiidapy)$ pip install -e .
```

Which will install aiida_core. Note `-e` option in the last line: it allows one to fetch updates from GitHub without package reinstallation. AiiDA-FLEUR can be installed the same way:

```
(aiidapy)$ mkdir <your_directory_FLEUR>
(aiidapy)$ git clone https://github.com/JuDFTteam/aiida-fleur.git
(aiidapy)$ cd aiida-fleur
(aiidapy)$ git checkout develop
(aiidapy)$ pip install -e .
```

3.1.1.2 AiiDA setup

Once AiiDA-FLEUR is installed, it it necessary to setup a profile, computers and codes.

Profile setup

First, to set up a profile with a database, use:

```
(aiidapy)$ verdi quicksetup
```

You will be asked to specify information required to identify data generated by you. If this command does not work for you, please set up a profile manually via *verdi setup* following instructions from the AiiDA tutorial.

Before setting up a computer, run:

```
(aiidapy)$ verdi daemon start
(aiidapy)$ verdi status
```

The first line launches a daemon which is needed for AiiDA to work. The second one makes an automated check if all necessary components are working. If all of your checks passed and you see something like

```
✓ profile:     On profile quicksetup
✓ repository:  /Users/tsep/.aiida/repository/quicksetup
✓ postgres:    Connected to aiida_qs_tsep_060f34d14612eee921b9ec5433b36abf@None:None
✓ rabbitmq:    Connected to amqp://127.0.0.1?heartbeat=600
✓ daemon:      Daemon is running as PID 8369 since 2019-07-12 09:56:31
```

your AiiDA is set up properly and you can continue with next section.

Computers setup

AiiDA needs to know how to access the computer that you want to use for FLEUR calculations. Therefore you need to set up a computer - this procedure will create a representation (node) of computational computer in the database which will be used later. It can be done by:

```
(aiidapy)$ verdi computer setup
```

An example of the input:

```
Computer label: my_laptop
Hostname: localhost
Description []: This is my laptop.
Transport plugin: local
Scheduler plugin: direct
Shebang line (first line of each script, starting with #!) [#!/bin/bash]:
Work directory on the computer [/scratch/{username}/aiida/]: /Users/I/home/workaiida
Mpirun command [mpirun -np {tot_num_mpiprocs}]:
Default number of CPUs per machine: 1
```

after that, a vim editor pops out, where you need to specify prepend and append text where you can specify required imports for you system. You can skip add nothing there if you need no additional imports.

If you want to use a remote machine via ssh, you need to specify this machine in `~/.ssh/config/`:

```
Host super_machine
  HostName super_machine.institute.de
  User user_1
  IdentityFile ~/.ssh/id_rsa
  Port 22
  ServerAliveInterval 60
```

and then use:

```
Computer label: remote_cluster
Hostname: super_machine
Description []: This is a super_machine cluster.
Transport plugin: ssh
Scheduler plugin: slurm
Shebang line (first line of each script, starting with #!) [#!/bin/bash]:
Work directory on the computer [/scratch/{username}/aiida/]: /scratch/user_1/workaiida
Mpirun command [mpirun -np {tot_num_mpiprocs}]: srun
Default number of CPUs per machine: 24
```

Note: *Work directory on the computer* is the place where all computational files will be stored. Thus, if you have a faster partition on your machine, I recommend you to use this one.

The last step is to configure the computer via:

```
verdi computer configure ssh remote_cluster
```

for ssh connections and

```
verdi computer configure local remote_cluster
```

for local machines.

If you are using aiida-fleur inside FZ Jülich, you can find additional helpful instructions on setting up the connection to JURECA (or other machine) on iffwiki.

FLEUR and inpgen setup

AiiDA-FLEUR uses two codes: FLEUR itself and an input generator called inpgen. Thus, two codes have to be set up independently.

input generator

I recommend running input generator on your local machine because it runs fast and one usually spends more time waiting for the input to be uploaded to the remote machine. You need to install inpgen code to your laptop first which can be done following the official guide.

After inpgen is successfully installed, it has to be configured by AiiDA. Run:

```
(aiidapy)$ verdi code setup
```

and fill all the required forms. An example:

```
Label: inpgen
Description []: This is an input generator code for FLEUR
Default calculation input plugin: fleur.inpgen
Installed on target computer? [True]: True
Computer: my_laptop
Remote absolute path: /Users/User/Codes/inpgen
```

after that, a vim editor pops out and you need to specify prepend and append text where you can add required imports and commands for you system. Particularly in my case, I need to set proper library paths. Hence my prepend text looks like:

```
export LD_LIBRARY_PATH=$LD_LIBRARY_PATH:/usr/local/intel/mkl/lib:/usr/local/intel/
↪compilers_and_libraries_2019.3.199/mac/compiler/lib/
```

Now inpgen code is ready to be used.

FLEUR code

FLEUR code can be set up the same way as the input generator. However, there is an important note that has to be mentioned.

Note: If you use an HDF version of the FLEUR code then AiiDA-FLEUR plugin should know this. The main reason for this is that names of output files vary between HDF and standard FLEUR versions. To properly set up an HDF version of the code, you *must* mention HDF5 (or hdf5) in the code description and not change it in the future. An example of setting up an HDF version:

```
Label: fleur
Description []: This is the FLEUR code compiled with HDF5.
Default calculation input plugin: fleur.fleur
Installed on target computer? [True]: True
Computer: remote_cluster
Remote absolute path: /scratch/user/codes/fleur_MPI
```

Installation test

To test if the aiida-fleur installation was successful use:

```
(aiidapy)$ verdi plugin list aiida.calculations
```

Example output containing FLEUR calculations:

```
* arithmetic.add
* fleur.fleur
* fleur.inpgen
* templatereplacer
```

You can pass as a further parameter one (or more) plugin names to get more details on a given plugin.

After you have installed AiiDA-FLEUR it is always a good idea to run the automated standard test set once to check on the installation (make sure that postgres can be called via 'pg_ctl' command)

```
cd aiida_fleur/tests/
./run_all_cov.sh
```

the output should look something like this

```
(env_aiida)% ./run_all_cov.sh
================================== test session starts␣
↪==================================
platform darwin -- Python 3.7.6, pytest-5.3.1, py-1.8.0, pluggy-0.12.0
Matplotlib: 3.1.1
Freetype: 2.6.1
rootdir: /Users/tsep/Documents/aiida/aiida-fleur, inifile: pytest.ini
plugins: mpl-0.10, cov-2.7.1
collected 555 items

test_entrypoints.py ...................                                  [ ␣
↪3%]
data/test_fleurinp.py ..................................................... [␣
↪14%]
.....................................                                    [␣
↪21%]
data/test_fleurinpmodifier.py ..                                         [␣
↪21%]
parsers/test_fleur_parser.py ........                                    [␣
↪23%]
tools/test_StructureData_util.py ...................                     [␣
↪26%]
tools/test_common_aiida.py .....                                         [␣
↪27%]
tools/test_common_fleur_wf.py ...s..s.s.                                 [␣
↪29%]
```

(continues on next page)

(continued from previous page)

```
tools/test_common_fleur_wf_util.py .....s.s....s.....s                          [␣
↪32%]
tools/test_data_handling.py .                                                   [␣
↪32%]
tools/test_dict_util.py ......                                                  [␣
↪33%]
tools/test_element_econfig_list.py .......                                      [␣
↪35%]
tools/test_extract_corelevels.py ...                                            [␣
↪35%]
tools/test_io_routines.py ..                                                    [␣
↪36%]
tools/test_read_cif_folder.py .                                                 [␣
↪36%]
tools/test_xml_util.py ..........s.............................................. [␣
↪46%]
....sss..ssss..............s.............ss.....................sssssss..s.. [␣
↪61%]
.................sssssssssssssssssss.........sss.......................s......... [␣
↪76%]
...................s.....................sss.......................s............ [␣
↪91%]
...............s...............                                                 [␣
↪96%]
workflows/test_workflows_builder_init.py .................                      ␣
↪[100%]

+ coverage report

====================== 500 passed, 55 skipped, 21 warnings in 51.09s␣
↪=======================
```

No worries about skipped tests - they appear due to technical implementation of tests and contain some information for developers. For a user it is important to make sure that the others do not fail: if anything (especially a lot of tests) fails it is very likely that your installation is messed up. Some packages might be missing (reinstall them by hand and report to development team). The other problem could be that the AiiDA-FLEUR version you have installed is not compatible with the AiiDA version you are running, since not all AiiDA versions are back-compatible.

3.1.2 AiiDA-FLEUR Data Plugins

AiiDA-FLEUR data plugins include:

1. `FleurinpData` structure (*FleurinpData*)
2. `FleurinpModifier` structure (*FleurinpModifier*)

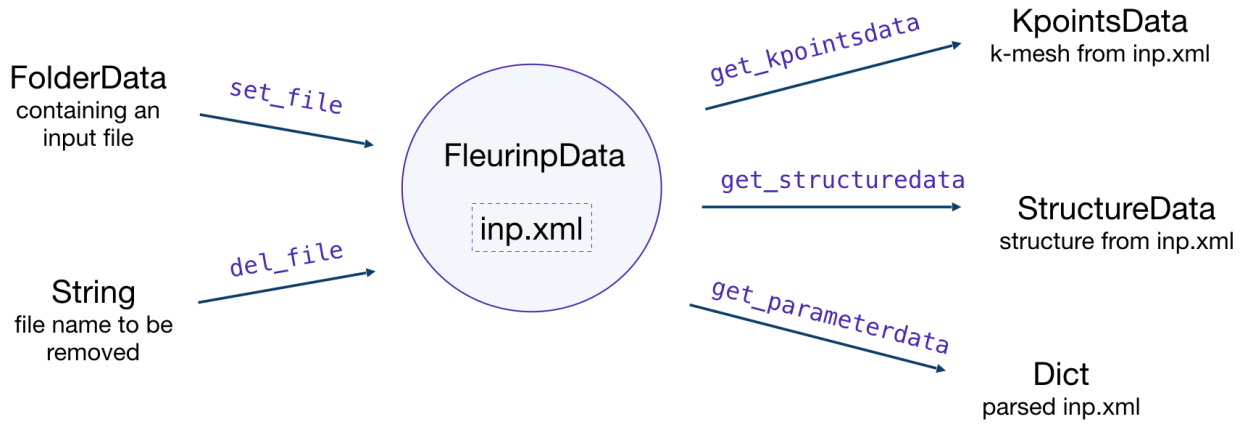
`FleurinpData` is a `Data` type that represents input files needed for the FLEUR code and methods to work with them. They include inp.xml and some other situational files. Finally, `FleurinpModifier` consists of methods to change existing `FleurinpData` in a way to preserve data provenance.

3.1.2.1 FleurinpData

- **Class**: `FleurinpData`
- **String to pass to the** `DataFactory()`: `fleur.fleurinp`

- **Aim**: store input files for the FLEUR code and provide user-friendly editing.
- **What is stored in the database**: the filenames, a parsed inp.xml files as nested dictionary
- **What is stored in the file repository**: inp.xml file and other optional files.
- **Additional functionality**: Provide user-friendly methods. Connected to structure and Kpoints AiiDA data structures

Description/Features



FleurinpData is an additional AiiDA data structure which represents everything a *FleurCalculation* needs, which is mainly a complete `inp.xml` file.

Note: Currently, *FleurinpData* methods support *ONLY* `inp.xml` files, which have everything in them (kpoints, energy parameters, ...), i.e which were created with the `-explicit` inpgen command line switch. In general it was designed to account for several separate files too, but this is no the default way Fleur should be used with AiiDA.

FleurinpData was implemented to make the plugin more user-friendly, hide complexity and ensure the connection to AiiDA data structures (`StructureData`, `KpointsData`). More detailed information about the methods can be found below and in the module code documentation.

Note: If you want to change the input file use the *FleurinpModifier* (*FleurinpModifier*) class, because a *FleurinpData* object has to be stored in the database and usually sealed.

Initialization:

```
from aiida_fleur.data.fleurinp import FleurinpData
# or FleurinpData = DataFactory('fleur.fleurinp')

F = FleurinpData(files=['path_to_inp.xml_file', <other files>])
# or
F = FleurinpData(files=['inp.xml', <other files>], node=<folder_data_pk>)
```

If the `node` attribute is specified, AiiDA will try to get files from the `FolderData` corresponding to the node. If not, it tries to find an `inp.xml` file using an absolute path `path_to_inp.xml_file`.

Be aware that the `inp.xml` file name has to be named 'inp.xml', i.e. no file names are changed because the filenames will not be changed before submitting a Fleur Calculation. If you add another inp.xml file the first one will be overwritten.

Properties

- *inp_dict*: Returns the inp_dict (the representation of the inp.xml file) as it will or is stored in the database.
- *files*: Returns a list of files, which were added to FleurinpData. Note that all of these files will be copied to the folder where FLEUR will be run.

Note: *FleurinpData* will first look in the `aiida_fleur/fleur_schema/input/` for matching Fleur xml schema files to the `inp.xml` files. If it does not find a match there, it will recursively search in your PYTHONPATH and the current directory. If you installed the package with pip there should be no problem, as long the package versions is new enough for the version of the Fleur code you are deploying.

User Methods

- *del_file()* - Deletes a file from *FleurinpData* instance.
- *set_file()* - Adds a file from a folder node to *FleurinpData* instance.
- *set_files()* - Adds several files from a folder node to *FleurinpData* instance.
- *get_fleur_modes()* - Analyses inp.xml and get a corresponding calculation mode.
- *get_structuredata()* - A CalcFunction which returns an AiiDA `StructureData` type extracted from the inp.xml file.
- *get_kpointsdata()* - A CalcFunction which returns an AiiDA `KpointsData` type produced from the inp.xml file. This only works if the kpoints are listed in the in inp.xml.
- *get_parameterdata()* - A CalcFunction that extracts a `Dict` node containing FLAPW parameters. This node can be used as an input for inpgen.
- *set_kpointsdata_f()* - A Function of fleurmodifier used to writes kpoints of a `KpointsData` node to the inp.xml file. It replaces old kpoints.

Setting up atom labels

Label is a string that marks a certain atom in the `inp.xml` file. Labels are created automatically by the inpgen, however, in some cases it is helpful to control atom labeling. This can be done by setting up the kind name while initialising the structure:

```
structure = StructureData(cell=cell)
structure.append_atom(position=(0.0, 0.0, -z), symbols='Fe', name='Fe123')
structure.append_atom(position=(x, y, 0.0), symbols='Pt')
structure.append_atom(position=(0., 0., z), symbols='Pt')
```

in this case both of the Pr atoms will get default labels, but 'Fe' atom will the label '123' (actually ' 123', but all of the methods in AiiDA-Fleur are implemented in a way that user should know only last digit characters).

Note: Kind name, which is used for labeling, must begin from the element name and end up with a number. It is **very important** that the first digit of the number is smaller than 4: `152`, `3`, `21` can be good choices, when `492`, `66`, `91` are forbidden.

Warning: Except setting up the label, providing a kind name also creates a new specie. This is because the `123` will not only appear as a label, but also in the atom number. In our case, the line in the inpgen input corresponding to Fe atom will look like `26.123 0 0 -z 123`. Hence, if we would have another Fe atom with the default kind name, both of the Fe atom would belong to different atom group, generally resulting in lower symmetry.

Given labels can be used for simplified xml methods. For example, when one performs SOC calculations it might be needed to vary `socscale` parameter for a certain atom. Knowing the correct label of the atom, it is straightforward to make such a change in *`FleurinpData`* object (*using* the *`FleurinpModifier`*) or pass a corresponding line to `inpxml_changes` of workchain parameters:

```
# an example of inpxml_changes list, that sets socscale of the iron atom
# from the above structure to zero
inpxml_changes = [('set_species_label', {'at_label': '123',
                                         'attributedict': {
                                             'special': {'socscale': 0.0}
                                             },
                                         'create': True
                                        })]

# in this example the atomgroup, to which the atom with label '222' belongs,
# will be modified
fm = FleurinpModifier(SomeFleurinp)
fm.set_atomgr_att_label(attributedict={'force': [('relaxXYZ', 'FFF')]}, atom_label=' ␣
↪               222')
```

3.1.2.2 FleurinpModifier

Description

The *`FleurinpModifier`* class has to be used if you want to change anything in a stored *`FleurinpData`*. It will store and validate all the changes you wish to do and produce a new *`FleurinpData`* node after you are done making changes and apply them.

`FleurinpModifier` provides a user with methods to change the Fleur input. In principle a user can do everything, since he could prepare a FLEUR input himself and create a *`FleurinpData`* object from that input.

Note: In the open provenance model nodes stored in the database cannot be changed (except extras and comments). Therefore, to modify something in a stored *inp.xml* file one has to create a new *`FleurinpData`* which is not stored, modify it and store it again. However, this node would pop into existence unlinked in the database and this would mean we loose the origin from what data it comes from and what was done to it. This is the task of *`FleurinpModifier`*.

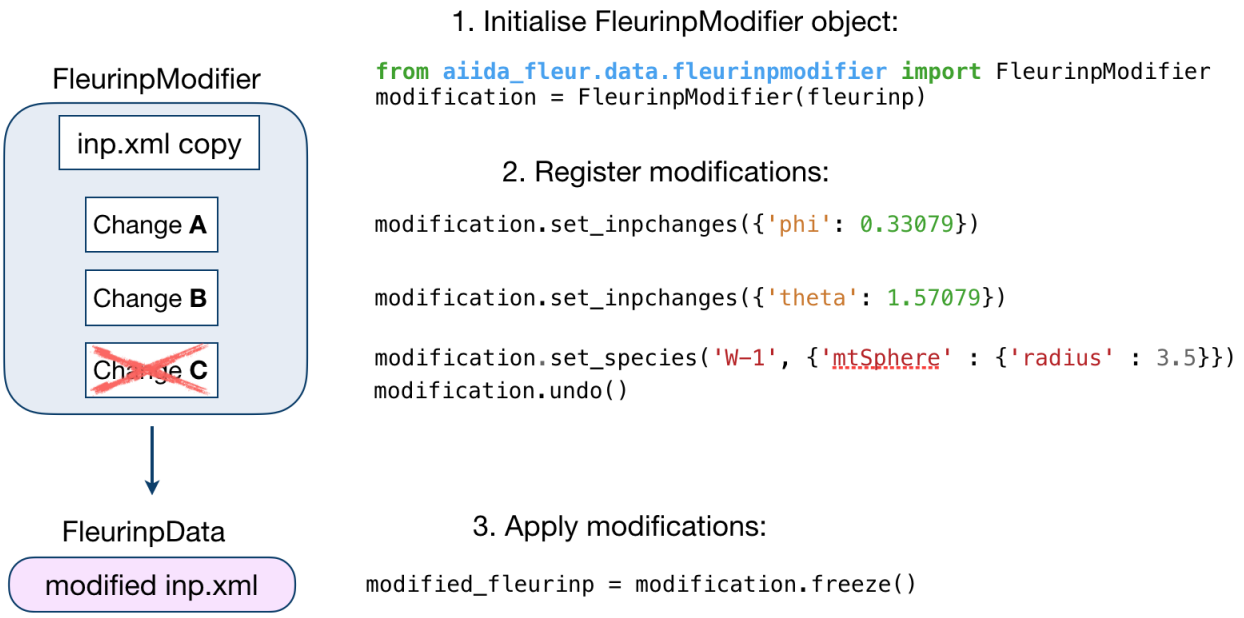
Usage

To modify an existing *`FleurinpData`*, a *`FleurinpModifier`* instance has to be initialised staring from the *`FleurinpData`* instance. After that, a user should register certain modifications which will be cached and can be

previewed. They will be applied on a new *FleurinpData* object when the freeze method is executed. A code example:

```
from aiida_fleur.data.fleurinpmodifier import  FleurinpModifier

F = FleurinpData(files=['inp.xml'])
fm = FleurinpModifier(F)                                    # Initialise FleurinpModifier␣
↪class
fm.set_inpchanges({'dos' : True, 'Kmax': 3.9 })             # Add changes
fm.show()                                                   # Preview
new_fleurinpdata = fm.freeze()                              # Apply
```

The figure below illustrates the work of the *FleurinpModifier* class.



User Methods

General methods

- *validate()*: Tests if the changes in the given list are validated.
- *freeze()*: Applies all the changes in the list, calls *modify_fleurinpdata()* and returns a new *FleurinpData* object.
- *changes()*: Displays the current list of changes.
- *show()*: Applies the modifications and displays/prints the resulting `inp.xml` file. Does not generate a new *FleurinpData* object.

Modification registration methods

The registration methods can be separated into two groups. First of all, there are XML methods that require deeper knowledge about the structure of an `inp.xml` file. All of them require an xpath input:

- *xml_set_attribv_occ()*: Set an attribute of a specific occurrence of xml elements

- *xml_set_first_attribv()*: Set an attribute of first occurrence of xml element
- *xml_set_all_attribv()*: Set attributes of all occurrences of the xml element
- *xml_set_text()*: Set the text of first occurrence of xml element
- *xml_set_text_occ()*: Set an attribute of a specific occurrence of xml elements
- *xml_set_all_text()*: Set the text of all occurrences of the xml element
- *create_tag()*: Insert an xml element in the xml tree.
- *delete_att()*: Delete an attribute for xml elements from the xpath evaluation.
- *delete_tag()*: Delete an xml element.
- *replace_tag()*: Replace an xml element.
- *add_num_to_att()*: Adds a value or multiplies on it given attribute.

On the other hand, there are shortcut methods that already know some paths:

- *set_species()*: Specific user-friendly method to change species parameters.
- *set_atomgr_att()*: Specific method to change atom group parameters.
- *set_species_label()*: Specific user-friendly method to change a specie of an atom with a certain label.
- *set_atomgr_att_label()*: Specific method to change atom group parameters of an atom with a certain label.
- *set_inpchanges()*: Specific user-friendly method for easy changes of attribute key value type.
- *shift_value()*: Specific user-friendly method to shift value of an attribute.
- *shift_value_species_label()*: Specific user-friendly method to shift value of an attribute of an atom with a certain label.
- *set_nkpts()*: Specific method to set the number of kpoints.
- *set_nmmpmat()*: Specific method for initializing or modifying the density matrix file for a LDA+U calculation (details see below)

The figure below shows a comparison between the use of XML and shortcut methods.

XML methods	Shortcuts
Total number of iterations	
`xml_set_first_attribv('/fleurInput/calculationSetup/scfLoop', 'itmax', 29)`	`set_inpchanges({'itmax': 29})`
Muffin tin radius	
`xml_set_first_attribv('/fleurInput/atomSpecies/' 'species[@name = "W-1"]/mtSphere', 'radius', 2.2)`	`set_specie('W-1', {'mtSphere' : {'radius' : 2.2}})`
Beta noco parameter	
`xml_set_first_attribv('/fleurInput/atomGroups/atomGroup/' 'atomGroup[@species = "W-1"]/nocoParams', 'beta', 1.57)`	`set_atomgr_att({'nocoParams': [('beta', 1.57)]}, species='W-1')`

Modifying the density matrix for LDA+U calculations

The above mentioned *set_nmmpmat()* takes a special role in the modification registration methods, as the modifications are not done on the inp.xml file but the density matrix file n_mmp_mat used by Fleur for LDA+U calculations. The resulting density matrix file is stored next to the inp.xml in the new *FleurinpData* instance produced by calling the *freeze()* method and will be used as the initial density matrix if a calculation is started from this *FleurinpData* instance.

The code example below shows how to use this method to add a LDA+U procedure to an atom species and provide an initial guess for the density matrix.

```
from aiida_fleur.data.fleurinpmodifier import  FleurinpModifier

F = FleurinpData(files=['inp.xml'])
fm = FleurinpModifier(F)                                                    # Initialise␣
↪FleurinpModifier class
fm.set_species('Nd-1', {'ldaU':                                             # Add LDA+U␣
↪procedure
                        {'l': 3, 'U': 6.76, 'J': 0.76, 'l_amf': 'F'}})
fm.set_nmmpmat('Nd-1', orbital=3, spin=1, occStates=[1,1,1,1,0,0,0]) # Initialize n_
↪mmp_mat file with the states
                                                                            # m = -3 to m =␣
↪0 occupied for spin up
                                                                            # spin down is␣
↪initialized with 0 by default
new_fleurinpdata = fm.freeze()                                              # Apply
```

Note: The n_mmp_mat file is a simple text file with no knowledge of which density matrix block corresponds to which LDA+U procedure. They are read in the same order as they appear in the inp.xml. For this reason the n_mmp_mat file can become invalid if one adds/removes a LDA+U procedure to the inp.xml after the n_mmp_mat file was initialized. To circumvent these problems always remove any existing n_mmp_mat file from the *FleurinpData* instance, before adding/removing or modifying the LDA+U configuration. Furthermore the *set_nmmpmat()* should always be called after any modifications to the LDA+U configuration.

3.1.3 AiiDA-FLEUR Calculations

AiiDA-FLEUR plugin consists of three main parts:

1. FLEUR input generator (*Fleur input generator plugin*)
2. FLEUR code (*FLEUR code plugin*)

Fleur input generator represents inpgen code, FLEUR code represents fleur and fleur_MPI codes.

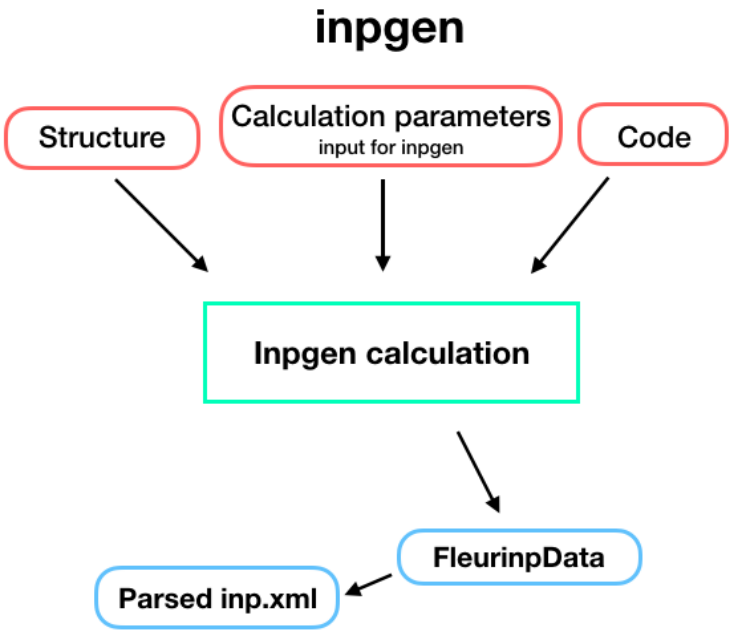
Other codes from the Fleur family (GFleur) or which are built on top of FLEUR (Spex) are not supported yet.

3.1.3.1 Fleur input generator plugin

Description

The input generator plugin is capable of running the Fleur input generator (inpgen). Similarly to inpgen code, *FleurinputgenCalculation* accepts a StructureData and a list of other parameters via **calc_parameters** (Dict type) containing all other parameters that inpgen accepts as an input. As a result, an *FleurinpData* node will be created which is a database representation of inp.xml and all other input files for FLEUR.

To set up an input dictionary, consider using *get_inputs_inpgen()* which assembles input nodes in a ready-to-use single dictionary.



Inputs

The table below shows all the input nodes that can be passed into additional *FleurinputgenCalculation*:

name	type	description	required
code	Code	Inpgen code	yes
structure	StructureData	Structure data node	yes
parameters	Dict	FLAPW parameters	no
metadata.options	Dict	computational resources	yes
metadata.label	string	computational resources	yes
metadata.description	string	computational resources	yes

- **code**: `Code` - the Code node of an inpgen executable
- **structure**: `StructureData` - a crystal structure that will be written into simplified input file. The plugin will run inpgen always with relative coordinates (crystal coordinates) in the 3D case. In the 2D case in rel, rel, abs. Currently for films no crystal rotations are be performed, therefore the coordinates need to be given as Fleur needs them. (x, y in plane, z out of plane)
- **calc_parameters**: `Dict`, optional - Input parameters of inpgen as a nested dictionary. An example:

```
# -*- coding: utf-8 -*-
Cd = Dict(dict={
    'atom':{'element' : 'Cd', 'rmt' : 2.5, 'jri' : 981, 'lmax' : 12,
            'lnonsph' : 6, 'lo' : '4d',
            'econfig' : '[Ar] 4s2 3d10 4p6 | 4d10 5s2'},
    'comp': {'kmax': 4.7, 'gmaxxc' : 12.0, 'gmax' : 14.0},
    'kpt': {'div1' : 17, 'div2': 17, 'div3' : 17, 'tkb' : 0.0005}})

# Magnetism and spin orbit coupling
```

(continues on next page)

(continued from previous page)

```
Cr = Dict(dict={
    'atom1':{'element' : 'Cr', 'id': '24.0', 'rmt' : 2.1, 'jri' : 981,
             'lmax' : 12, 'lnonsph' : 6, 'lo' : '3s 3p', 'bmu':1.5},
    'atom2':{'element' : 'Cr', 'id': '24.1', 'rmt' : 2.1, 'jri' : 981,
             'lmax' : 12, 'lnonsph' : 6, 'lo' : '3s 3p', 'bmu':1.4},
    'comp': {'kmax': 5.2, 'gmaxxc' : 12.5, 'gmax' : 15.0},
    'kpt': {'div1' : 24, 'div2': 24, 'div3' : 24, 'tkb' : 0.0005},
    'soc' : {'theta' : 0.0, 'phi' : 0.0}})
```

The list of all possible keys:

```
'input': ['film', 'cartesian', 'cal_symm', 'checkinp', 'symor', 'oldfleur']

'atom': ['id', 'z', 'rmt', 'dx', 'jri', 'lmax', 'lnonsph', 'ncst', 'econfig',
         'bmu', 'lo', 'element', 'name']

'comp': ['jspins', 'frcor', 'ctail', 'kcrel', 'gmax', 'gmaxxc', 'kmax']

'exco': ['xctyp', 'relxc'],

'film': ['dvac', 'dtild'],

'soc': ['theta', 'phi'],

'qss': ['x', 'y', 'z'],

'kpt': ['nkpt', 'kpts', 'div1', 'div2', 'div3','tkb', 'tria'],

'title': {}
```

See the Fleur documentation for the meaning of each key.

The *atom* namelist can occur several times in the parameter dictionary representing different atom species. However, python does not accept the same key twice and one must use *atomN* keys where *N* is an integer which will be ignored during the simplified input generation. Note that there is no need to set *&input film* because it is set automatically according to the given **structure** input node. That is also the reason why *&lattice* input parameter is ignored, we only support setting structure via **structure** input node.

- **settings**: class `Dict`, optional - An optional dictionary that allows the user to specify if additional files shall be received and other advanced non default stuff for inpgen.

To set up an input dictionary, consider using *`get_inputs_inpgen()`* which assembles input nodes in a ready-to-use single dictionary.

Outputs

The table below shows all the output nodes generated by *`FleurinputgenCalculation`*:

name	type	comment
fleurinpData	FleurinpData	represents *inp.xml*
remote_folder	FolderData	represents calculation folder
retrieved	FolderData	represents retrieved folder

All output nodes can be accessed via `calculation.outputs`.

- **fleurinpData**: *FleurinpData* - Data structure which represents the inp.xml file and provides useful methods. For more information see *FleurinpData*
- **remote_folder**: RemoteData - RemoteData which represents the calculation folder on the remote machine.
- **retrieved**: FolderData - FolderData which represents the retrieved folder on the remote machine.

Errors

When a certain error appears, the calculation finishes with a non-zero *exit code*.

Exit code	Reason
251	Input parameters for inpgen contain unknown keys
253	Fleur lattice needs atom positions as input
254	Excessive input parameters were specified
300	No retrieved folder found
301	One of the output files can not be opened
306	XML input file was not found
307	Some required files were not retrieved

Additional advanced features

While the input link with name calc_parameters is used for the content of the namelists and parameters of the inpgen input file, additional parameters for changing the plugin behavior can be specified in the 'settings': class Dict input.

Below we summarise some of the options that you can specify and their effect. In each case, after having defined the content of settings_dict, you can use it as input of a calculation calc by doing:

```
calc.use_settings(Dict(dict=settings_dict))
```

Retrieving more files

The inpgen plugin retrieves per default the files : inp.xml, out, struct.xsf.

If you know that your inpgen calculation is producing additional files that you want to retrieve (and preserve in the AiiDA repository in the long term), you can add those files as a list as follows (here in the case of a file named testfile.txt):

```
settings_dict = {
  'additional_retrieve_list': ['testfile.txt'],
}
```

Retrieving less files

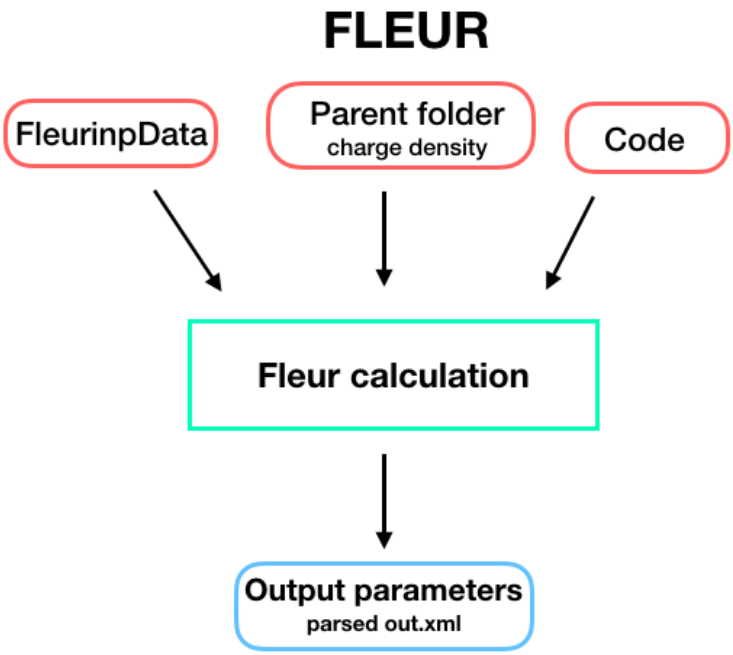
If you know that you do not want to retrieve certain files (and preserve in the AiiDA repository in the long term) you can add those files as a list as follows (here in the case of a file named testfile.txt):

```
settings_dict = {
  'remove_from_retrieve_list': ['testfile.txt'],
}
```

3.1.3.2 FLEUR code plugin

Description

The *FleurCalculation* runs Fleur executable e.g. `fleur` or `fleur_MPI`.



Inputs

To set up an input dictionary, consider using *get_inputs_fleur()* which assembles input nodes in a ready-to-use single dictionary.

The table below shows all possible inputs for the FleurCalculation:

name	type	description	required
code	Code	Fleur code	yes
fleurinpdata	FleurinpData	Object representing inp.xml	no
parent_folder	RemoteData	Remote folder of another calculation	no
settings	Dict	special settings	no
metadata.options	Dict	computational resources	yes

- **fleurinp**: *FleurinpData*, optional - Data structure which represents the inp.xml file and everything a Fleur calculation needs. For more information see *FleurinpData*.
- **parent_folder**: `RemoteData`, optional - If specified, certain files in the previous Fleur calculation folder are copied in the new calculation folder.

Note: **fleurinp** and **parent_folder** are both optional. Depending on the setup of the inputs, one of five scenarios will happen:

1. **fleurinp**: files belonging to **fleurinp** will be used as input for FLEUR calculation.
2. **fleurinp** + **parent_folder** (FLEUR): files, given in **fleurinp** will be used as input for FLEUR calculation. Moreover, initial charge density will be copied from the folder of the parent calculation.

3. **parent_folder** (FLEUR): Copies inp.xml file and initial charge density from the folder of the parent FLEUR calculation.
4. **parent_folder** (input generator): Copies inp.xml file from the folder of the parent inpgen calculation.
5. **parent_folder** (input generator) + **fleurinp**: files belonging to **fleurinp** will be used as input for FLEUR calculation. Remote folder is ignored.

Outputs

The table below shows all the output nodes generated by *FleurCalculation*:

name	type	comment
output_parameters	Dict	contains parsed *out.xml*
remote_folder	FolderData	represents calculation folder
retrieved	FolderData	represents retrieved folder

All the outputs can be found in `calculation.outputs`.

- **remote_folder**: `RemoteData` - RemoteData which represents the calculation folder on the remote machine.
- **retrieved**: `FolderData` - FolderData which represents the retrieved folder on the remote machine.
- **output_parameters**: `Dict` - Contains all kinds of information of the calculation and some physical quantities of the last iteration.

  An example output node:

```
# -*- coding: utf-8 -*-
(aiidapy)% verdi data dict show 425
{
    'CalcJob_uuid': 'a6511a00-7759-484a-839d-c100dafd6118',
    'bandgap': 0.0029975592,
    'bandgap_units': 'eV',
    'charge_den_xc_den_integral': -3105.2785777045,
    'charge_density1': 3.55653e-05,
    'charge_density2': 6.70788e-05,
    'creator_name': 'fleur 27',
    'creator_target_architecture': 'GEN',
    'creator_target_structure': ' ',
    'density_convergence_units': 'me/bohr^3',
    'end_date': {
        'date': '2019/07/17',
        'time': '12:50:27'
    },
    'energy': -4405621.1469633,
    'energy_core_electrons': -99592.985569309,
    'energy_hartree': -161903.59225823,
    'energy_hartree_units': 'Htr',
    'energy_units': 'eV',
    'energy_valence_electrons': -158.7015525598,
    'fermi_energy': -0.2017877885,
    'fermi_energy_units': 'Htr',
    'force_largest': 0.0,
    'magnetic_moment_units': 'muBohr',
    'magnetic_moments': [
```

(continues on next page)

(continued from previous page)


```
        2.7677822875,
        2.47601e-05,
        2.22588e-05,
        6.05518e-05,
        0.0001608849,
        0.0001504687,
        0.0001321699,
        -3.35528e-05,
        1.87169e-05,
        -0.0002957294
    ],
    'magnetic_spin_down_charges': [
        5.8532354421,
        6.7738647125,
        6.8081938915,
        6.8073232631,
        6.8162583243,
        6.8156475799,
        6.8188399492,
        6.813423175,
        6.7733972589,
        6.6797683064
    ],
    'magnetic_spin_up_charges': [
        8.6210177296,
        6.7738894726,
        6.8082161503,
        6.8073838149,
        6.8164192092,
        6.8157980486,
        6.8189721191,
        6.8133896222,
        6.7734159758,
        6.679472577
    ],
    'number_of_atom_types': 10,
    'number_of_atoms': 10,
    'number_of_iterations': 49,
    'number_of_iterations_total': 49,
    'number_of_kpoints': 240,
    'number_of_species': 1,
    'number_of_spin_components': 2,
    'number_of_symmetries': 2,
    'orbital_magnetic_moment_units': 'muBohr',
    'orbital_magnetic_moments': [],
    'orbital_magnetic_spin_down_charges': [],
    'orbital_magnetic_spin_up_charges': [],
    'output_file_version': '0.27',
    'overall_charge_density': 7.25099e-05,
    'parser_info': 'AiiDA Fleur Parser v0.2beta',
    'parser_warnings': [],
    'spin_density': 7.91911e-05,
    'start_date': {
        'date': '2019/07/17',
        'time': '10:38:24'
    },
    'sum_of_eigenvalues': -99751.687121869,
```


(continues on next page)

(continued from previous page)

```
    'title': 'A Fleur input generator calulation with aiida',
    'unparsed': [],
    'walltime': 7923,
    'walltime_units': 'seconds',
    'warnings': {
        'debug': {},
        'error': {},
        'info': {},
        'warning': {}
    }
}
```

Errors

Errors of the parsing are reported in the log of the calculation (accessible with the `verdi process report` command). Everything that Fleur writes into stderr is also shown here, i.e all JuDFT error messages. Example:

```
(aiidapy)% verdi process report 513
*** 513 [scf: fleur run 1]: None
*** (empty scheduler output file)
*** (empty scheduler errors file)
*** 3 LOG MESSAGES:
+-> ERROR at 2019-07-17 14:57:01.108964+00:00
| parser returned exit code<302>: FLEUR calculation failed.
+-> ERROR at 2019-07-17 14:57:01.097337+00:00
| FLEUR calculation did not finishsuccessfully.
+-> WARNING at 2019-07-17 14:57:01.056220+00:00
| The following was written into std error and piped to out.error :
|  I/O warning : failed to load external entity "relax.xml"
| rm: cannot remove 'cdn_last.hdf': No such file or directory
| **************juDFT-Error*****************
| Error message:e>vz0
| Error occurred in subroutine:vacuz
| Hint:Vacuum energy parameter too high
| Error from PE:0/24
```

Moreover, all warnings and errors written by Fleur in the out.xml file are stored in the ParameterData under the key `warnings`, and are accessible with `Calculation.res.warnings`.

More serious FLEUR calculation failures generate a non-zero *exit code*. Each exit code has it's own reason:

Exit code	Reason
300	One of output files can not be opened
301	No retrieved folder found
302	FLEUR calculation failed for unknown reason
303	XML output file was not found
304	Parsing of XML output file failed
305	Parsing of relax XML output file failed
310	FLEUR calculation failed due to memory issue
311	FLEUR calculation failed because atoms spilled to the vacuum
312	FLEUR calculation failed due to MT overlap
313	FLEUR calculation failed due to MT overlap during relaxation
314	Problem with cdn is suspected
315	Invalid Elements found in the LDA+U density matrix.
316	Calculation failed due to time limits.

Additional advanced features

In general see the FLEUR documentation.

While the input link with name **fleurinpdata** is used for the content of the inp.xml, additional parameters for changing the plugin behavior, can be specified in the **settings** input, also of type `Dict`.

Below we summarise some of the options that you can specify, and their effect. In each case, after having defined the content of `settings_dict`, you can use it as input of a calculation `calc` by doing:

```
calc.use_settings(Dict(dict=settings_dict))
```

Adding command-line options

If you want to add command-line options to the executable (particularly relevant e.g. '-hdf' use hdf, or '-magma' use different libraries, magma in this case), you can pass each option as a string in a list, as follows:

```
settings_dict = {
    'cmdline': ['-hdf', '-magma'],
}
```

The default command-line of a fleur execution of the plugin looks like this for the torque scheduler:

```
'mpirun' '-np' 'XX' 'path_to_fleur_executable' '-wtime' 'XXX' < 'inp.xml' > 'shell.out
↪' 2> 'out.error'
```

If the code node description contains 'hdf5' in some form, the plugin will use per default hdf5, it will only copy the last hdf5 density back, not the full cdn.hdf file. The Fleur execution line becomes in this case:

```
'mpirun' '-np' 'XX' 'path_to_fleur_executable' '-last_extra' '-wtime' 'XXX' < 'inp.xml
↪' > 'shell.out' 2> 'out.error'
```

Retrieving more files

AiiDA-FLEUR does not copy all output files generated by a FLEUR calculation. By default, the plugin copies only `out.xml`, `cdn1` and `inp.xml` and other technical files. Depending on certain switches in used `inp.xml`, the

plugin is capable of automatically adding additional files to the copy list:

- if `band=T` : `bands.1`, `bands.2`
- if `dos=T` : `DOS.1`, `DOS.2`
- if `pot8=T` : `pot*`
- if `l_f=T` : `relax.xml`

If you know that your calculation is producing additional files that you want to retrieve (and preserve in the AiiDA repository in the long term), you can add those files as a list as follows (here in the case of a file named `testfile.txt`):

```
settings_dict = {
  'additional_retrieve_list': ['testfile.txt'],
}
```

Retrieving less files

If you know that you do not want to retrieve certain files(and preserve in the AiiDA repository in the long term). i.e. the `cdn1` file is to large and it is stored somewhere else anyway, you can add those files as a list as follows (here in the case of a file named `testfile.txt`):

```
settings_dict = {
  'remove_from_retrieve_list': ['testfile.txt'],
}
```

Copy more files remotely

The plugin copies by default the `mixing_history*` files if a parent_folder is given in the input.

If you know that for your calculation you need some other files on the remote machine, you can add those files as a list as follows (here in the case of a file named `testfile.txt`):

```
settings_dict = {
  'additional_remotecopy_list': ['testfile.txt'],
}
```

Copy less files remotely

If you know that for your calculation do not need some files which are copied per default by the plugin you can add those files as a list as follows (here in the case of a file named `testfile.txt`):

```
settings_dict = {
  'remove_from_remotecopy_list': ['testfile.txt'],
}
```

3.1.4 AiiDA-FLEUR WorkChains

3.1.4.1 General design

All of the WorkChains have a similar interface and they share several common input nodes.

Inputs

There is always a `wf_parameters`: `Dict` node for controlling the workflow behavior. It contains all the parameters related to physical aspects of the workchain and its content vary between different workchains.

Note: `inpxml_changes` key of `wf_parameters` exists for most of the workchains. This list can be used to apply any supported change to inp.xml that will be used in calculations. To add a required change, simply append a two-element tuple where the first element is the name of the registration method and the second is a dictionary of inputs for the registration method. For more information about possible registration methods and their inputs see *FleurinpModifier*. An example:

```
inpxml_changes = [('set_inpchanges', {'change_dict': {'l_noco': False}})]
```

The other common input is an `options`: `Dict` node where the technical parameters (AiiDA options) are specified i.e resources, queue name and so on.

Regarding an input crystal structure, it can be set in two ways in the most of the workflows:

1. Provide a `structure`: `StructureData` node and an optional `calc_parameters`: `Dict`. It this case an inpgen code node is required. The workflow will call inpgen calculation and create a new FleurinpData that will be used in the workchain.
2. Provide a `fleurinp`: *`FleurinpData`* node which contains a complete input for a FLEUR calculation.
3. Provide a `remote_data`: `RemoteData` and the last charge density and `inp.xml` from the previous calculation will be used.

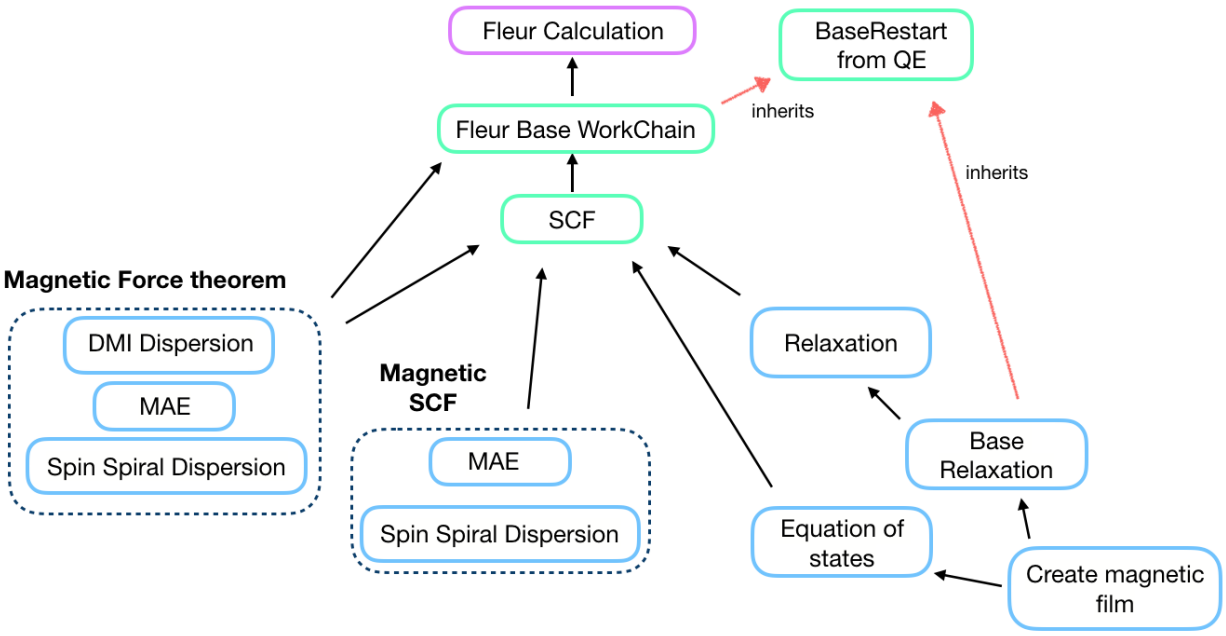
Input for the nested workchains has to be specified via a corresponding namespace. Please, refer to the documentation of a particular workchain to see the details.

Outputs

Most of the workchains return a workflow specific *ParameterData* (`Dict`) node named `output_name_wc_para` or simple `out` which contains the main results and some information about the workchain.

There are additional workflow specific input and output nodes, please read the documentation of a particular workchain that you are interested in.

3.1.4.2 Workchain classification



All of the workchains are divided into the groups. First, we separate *technical* and *scientific* workflows. This separation is purely subjective: *technical* workchains tend to be less complex and represent basic routine tasks that people usually encounter. *Scientific* workflows are less general and aimed at certain tasks.

There are the sub-group of the force theorem calculations and their self-consistent analogs in the scientific workchains group.

Note: The `plot_fleur` function provides a quick visualization for every workflow node or node list. Inputs are *uuid*, *pk*, *workchain* nodes or *ParameterData* (workchain output) nodes.

Basic (Technical) Workchains

Fleur base restart workchain

- **Current version**: 0.1.1
- **Class**: *FleurBaseWorkChain*
- **String to pass to the** `WorkflowFactory()`: `fleur.base`
- **Workflow type**: Technical
- **Aim**: Automatically resubmits a FleurCalculation in case of failure
- **Computational demand**: Corresponding to a single `FleurCalculation`
- **Database footprint**: Links to the FleurCalculation output nodes and full provenance
- **File repository footprint**: no addition to the `CalcJob` run

Contents

Import Example:

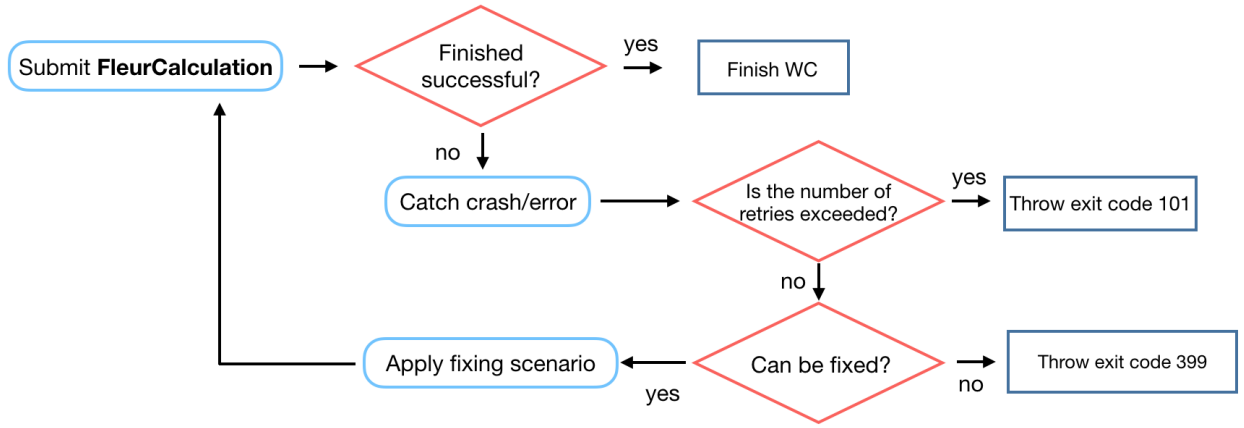
```
from aiida_fleur.workflows.base_fleur import FleurBaseWorkChain
#or
WorkflowFactory('fleur.base')
```

Description/Purpose

This workchain wraps *`FleurCalculation`* into BaseRestartWorkChain workchain, which is a plain copy of a BaseRestartWorkChain originally implemented in AiiDA-QE. The workchain automatically tracks and fixes crashes of the *`FleurCalculation`*.

Note: This workchain accepts all of the inputs that is needed for the FleurCalculation. It also contains all the links of the outputs generated by the FleurCalculation. It most of the cases, a user does not feel the difference in the front-end behaviour between FleurCalculation and FleurBaseWorkChain.

The workchain works as follows:



For now only problems with memory can be fixed in *`FleurBaseWorkChain`* :if a FleurCalculation finishes with exit status 310 the FleurBaseWorkChain will resubmit it setting twice larger number of computational nodes.

Warning: The exit status 310 can be thrown only in a few tested cases. Different machines and different compilers can report the memory issue in various ways. Now only a few kinds of reports are supported:

1. 'Out Of Memory' or 'cgroup out-of-memory handler' string found in *out.error* file.
2. If memory consumption, which is printed out in *out.error* as 'used' or in *usage.json* as 'VmPeak', is larger than 93% of memory available (printed out into *out.xml* as *memoryPerNode*).

All other possible memory issue reports are not implemented - please contact to the AiiDA-Fleur developers to add new report message.

`check_kpts()` method

Fixing failed calculations is not the only task of *FleurBaseWorkChain*. It also implements automatic parallelisation routine called *check_kpts()*. The task of this method is to ensure the perfect k-point parallelisation of the FLEUR code. It tries to set up the number of nodes and mpi tasks in a way that the total number of used MPI tasks is a factor of the total number of k-points. Therefore a user actually specifies not the actual resources to be used in a calculation but their maximum possible values.

The *optimize_calc_options()*, which is used by *check_kpts()*, has five main inputs: maximal number of nodes, first guess for a number of MPI tasks per node, first guess for a number of OMP threads per MPI task, required MPI_per_node / OMP_per_MPI ratio and finally, a switch that sets up if OMP parallelisation is needed. A user does not have to use the *optimize_calc_options()* explicitly, it will be run automatically taking `options['resources']` specified by user. `nodes` input (maximal number of nodes that can be used) is taken from `"num_machines"`. `mpi_per_node` is copied from `"num_mpiprocs_per_machine"` and `omp_per_mpi` is taken from `"num_cores_per_mpiproc"` if the latter is given. In this case `use_omp` is set to **True** (calculation will use OMP threading), `mpi_omp_ratio` will be set to `"num_mpiprocs_per_machine"` / `"num_cores_per_mpiproc"` and number of available CPUs per node is calculated as `"num_mpiprocs_per_machine"` * `"num_cores_per_mpiproc"`. In other case, when `"num_cores_per_mpiproc"` is not given, `use_omp` is set to **False** and the number of available CPUs per node is assumed to be equal to `"num_mpiprocs_per_machine"` and `mpi_omp_ratio` will be ignored.

Note: The error handler, which is responsible for dealing with memory issues, tries to decrease the MPI_per_node / OMP_per_MPI ratio and additionally decreases the value passed to `mpi_omp_ratio` by the factor of 0.5.

Note: Before setting the actual resources to the calculation, *check_kpts()* can throw an exit code if the suggested load of each node is smaller than 60% of what specified by user. For example, if user specifies:

```
options = {'resources' : {"num_machines": 4, "num_mpiprocs_per_machine" : 24}
```

and *check_kpts()* suggested to use 4 `num_machines` and 13 `num_mpiprocs_per_machine` the exit code will be thrown and calculation will not be submitted.

Warning: This method works with PBS-like schedulers only and if `num_machines` and `num_mpiprocs_per_machine` are specified. Thus it method can be updated/deprecated for other schedulers and situations. Please feel free to write an issue on this arguable function.

Errors

See *Exit codes*.

Exit code	Reason
101	Maximum number of fixing an error is exceeded
102	The calculation failed for an unknown reason, twice in a row This should probably never happen since there is a 399 exit code
360	*check_kpts()* suggests less than 60% of node load
389	FLEUR calculation failed due to memory issue and it can not be solved for this scheduler

Exit codes duplicating FleurCalculation exit codes:

Exit code	Reason
311	FleurCalculation failed because an atom spilled to the vacuum during relaxation.
312	FleurCalculation failed due to MT overlap.
399	FleurCalculation failed and FleurBaseWorkChain has no strategy to resolve this

Fleur self-consistency field workflow

- **Current version**: 0.4.0
- **Class**: *FleurScfWorkChain*
- **String to pass to the** `WorkflowFactory()`: `fleur.scf`
- **Workflow type**: Technical
- **Aim**: Manage FLEUR SCF convergence
- **Computational demand**: Corresponding to several `FleurCalculation`
- **Database footprint**: Output node with information, full provenance, `~ 10+10*FLEUR Jobs` nodes
- **File repository footprint**: no addition to the `CalcJob` run

Contents

Import Example:

```
from aiida_fleur.workflows.scf import FleurScfWorkChain
#or
WorkflowFactory('fleur.scf')
```

Description/Purpose

Converges the charge *density*, the *total energy* or the *largest force* of a given structure, or stops because the maximum allowed retries are reached.

The workchain is designed to converge only one parameter independently on other parameters (*largest force* is an exception because FLEUR code first checks if density was converged). Simultaneous convergence of two or three parameters is not implemented to simplify the code logic and because one almost always interested in a particular parameter. Moreover, it was shown that the total energy tend to converge faster than the charge density.

This workflow manages an inpgen calculation (if needed) and several Fleur calculations. It is one of the most core workchains and often deployed as a sub-workflow.

Input nodes

The table below shows all the possible input nodes of the SCF workchain.

name	type	description	required
fleur	`Code`	Fleur code	yes
inpgen	`Code`	Inpgen code	no
wf_parameters	`Dict`	Settings of the workchain	no
structure	`StructureData`	Structure data node	no
calc_parameters	`Dict`	inpgen *parameters*	no
fleurinp	*`FleurinpData`*	*FLEUR input*	no
remote_data	`RemoteData`	Remote folder of another calculation	no
options	`Dict`	AiiDA options (computational resources)	no
settings	`Dict`	Special *settings* for Fleur calculation	no

Only `fleur` input is required. However, it does not mean that it is enough to specify `fleur` only. One *must* keep one of the supported input configurations described in the *Layout* section.

Workchain parameters and its defaults

- `wf_parameters`: `Dict` - Settings of the workflow behavior. All possible keys and their defaults are listed below:

```
# -*- coding: utf-8 -*-
'fleur_runmax': 4,                  # Maximum number of fleur jobs/starts
'density_converged': 0.00002,       # Charge density convergence criterion
'energy_converged': 0.002,          # Total energy convergence criterion
'force_converged': 0.002,           # Largest force convergence criterion
'mode': 'density',                  # Parameter to converge: 'density', 'force'␣
↪or 'energy'
'serial': False,                    # Execute fleur with mpi or without
'only_even_MPI': False,             # True if suppress parallelisation having␣
↪odd number of MPI
'itmax_per_run': 30,                # Maximum iterations run for one␣
↪FleurCalculation
```

(continues on next page)

(continued from previous page)

```
'force_dict': {'qfix': 2,            # parameters required for the 'force' mode
               'forcealpha': 0.5,
               'forcemix': 'BFGS'},
'inpxml_changes': [],                # Modifications to inp.xml
```

'force_dict' contains parameters that will be inserted into the `inp.xml` in case of force convergence mode. Usually this sub-dictionary does not affect the convergence, it affects only the generation of `relax.xml` file. Read more in FLEUR relaxation documentation.

Note: Only one of `density_converged`, `energy_converged` or `force_converged` is used by the workchain that corresponds to the **'mode'**. The other two are ignored. Exception: force mode uses both `density_converged` and `force_converged` because FLEUR code always converges density before forces.

- `options`: `Dict` - AiiDA options (computational resources). Example:

```
'resources': {"num_machines": 1, "num_mpiprocs_per_machine": 1},
'max_wallclock_seconds': 6*60*60,
'queue_name': '',
'custom_scheduler_commands': '',
'import_sys_environment': False,
'environment_variables': {}
```

Returns nodes

The table below shows all the possible output nodes of the SCF workchain.

name	type	comment
output_scf_wc_para	`Dict`	results of the workchain
fleurinp	*`FleurinpData`*	FleurinpData that was used (after all modifications)
last_fleur_calc_output	`Dict`	Link to last FleurCalculation output dict

More details:

- `fleurinp`: *`FleurinpData`* - A *`FleurinpData`* that was actually used for last *`FleurScfWorkChain`*. It usually differs from the input *`FleurinpData`* because there are some hard-coded modifications in the SCF workchain.
- `last_fleur_calc_output`: `Dict` - A link to the output node of the last Fleur calculation.
- `output_scf_wc_para`: `Dict` - Main results of the workchain. Contains errors, warnings, convergence history and other information. An example:

```
# -*- coding: utf-8 -*-
{
    'conv_mode': 'density',
    'distance_charge': 0.1406279038,
    'distance_charge_all': [
        61.1110641131,
        43.7556515683,
        ...
    ],
```

(continues on next page)

(continued from previous page)

```
    'distance_charge_units': 'me/bohr^3',
    'errors': [],
    'force_diff_last': 'can not be determined',
    'force_largest': 0.0,
    'info': [],
    'iterations_total': 23,
    'last_calc_uuid': 'b20b5b94-5d80-41a8-82bf-b4d8eee9bddc',
    'loop_count': 1,
    'material': 'FePt2',
    'total_energy': -38166.176928494,
    'total_energy_all': [
        -38166.542950054,
        -38166.345602746,
        ...
    ],
    'total_energy_units': 'Htr',
    'total_wall_time': 245,
    'total_wall_time_units': 's',
    'warnings': [],
    'workflow_name': 'FleurScfWorkChain',
    'workflow_version': '0.4.0'
}
```

Layout

Similarly to *FleurCalculation*, SCF workchain has several input combinations that implicitly define the behaviour of the workchain during inputs processing. Depending on the setup of the inputs, one of the four supported scenarios will happen:

1. **fleurinp** + **remote_data** (FLEUR):

   Files, belonging to the **fleurinp**, will be used as input for the first FLEUR calculation. Moreover, initial charge density will be copied from the folder of the remote folder.

2. **fleurinp**:

   Files, belonging to the **fleurinp**, will be used as input for the first FLEUR calculation.

3. **structure** + **inpgen** + *calc_parameters*:

   inpgen code and optional *calc_parameters* will be used to generate a new *FleurinpData* using a given **structure**. Generated *FleurinpData* will be used as an input for the first FLEUR calculation.

3. **structure** + **inpgen** + *calc_parameters* + **remote_data** (FLEUR):

   inpgen code and optional *calc_parameters* will be used to generate a new *FleurinpData* using a given **structure**. Generated *FleurinpData* will be used as an input for the first FLEUR calculation. Initial charge density will be taken from given **remote_data** (FLEUR). **Note**: make sure that **remote_data** (FLEUR) corresponds to the same structure.

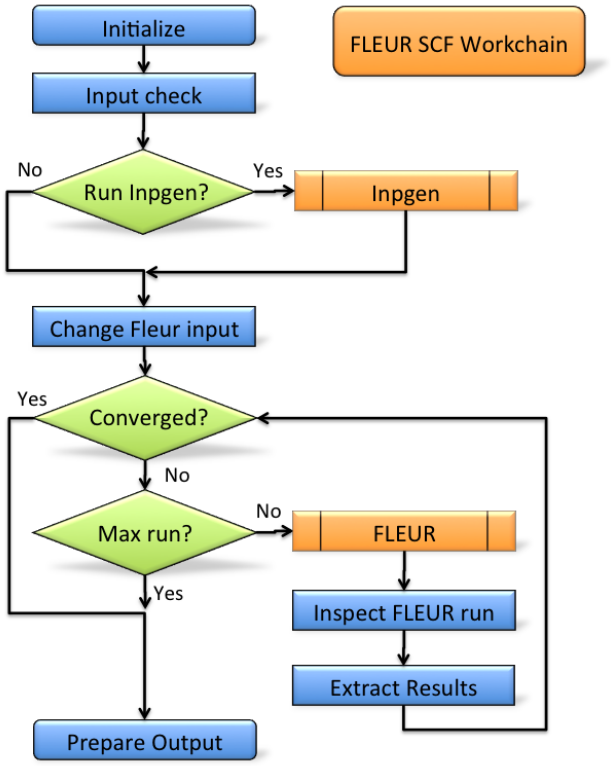
5. **remote_data** (FLEUR):

   inp.xml file and initial charge density will be copied from the remote folder.

For example, if you want to continue converging charge density, use the option 3. If you want to change something in the inp.xml and use old charge density you should use option 2. To do this, you can retrieve a FleurinpData produced by the parent calculation and change it via FleurinpModifier, use it as an input together with the RemoteFolder.

Warning: One *must* keep one of the supported input configurations. In other case the workchain will stop throwing exit code 230.

The general layout does not depend on the scenario, SCF workchain sequentially submits several FLEUR calculation to achieve a convergence criterion.



Error handling

In case of failure the SCF WorkChain should throw one of the *exit codes*:

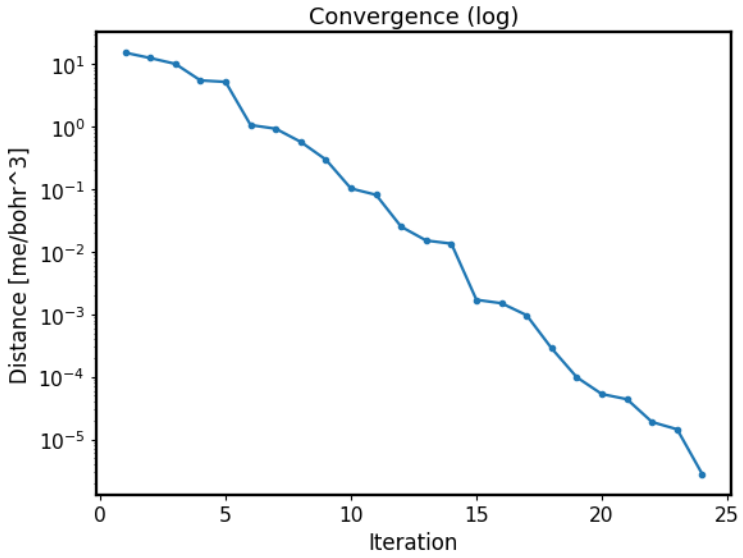
Exit code	Reason
230	Invalid input, please check input configuration
231	Invalid code node specified, check inpgen and fleur code nodes
232	Input file modification failed
233	Input file was corrupted after modifications
360	Inpgen calculation failed
361	Fleur calculation failed
362	SCF cycle did not lead to convergence, maximum number of iterations exceeded

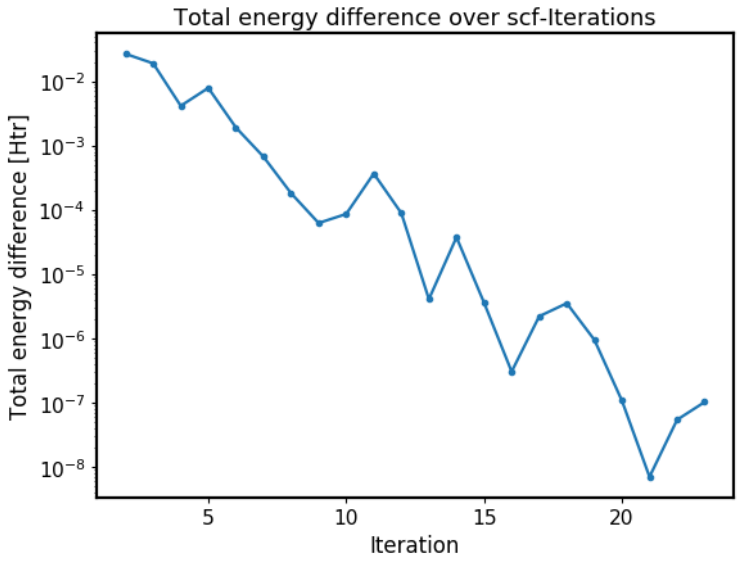
If your workchain crashes and stops in *Excepted* state, please open a new issue on the Github page and describe the details of the failure.

Plot_fleur visualization

Single node

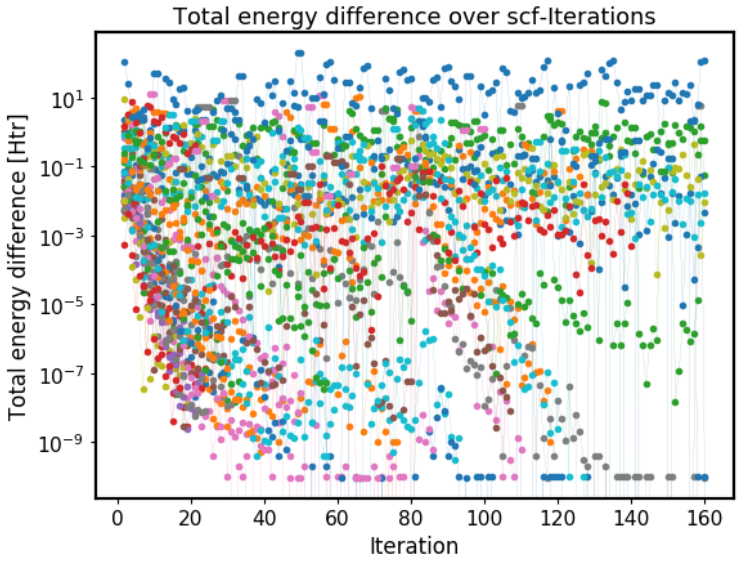
```
from aiida_fleur.tools.plot import plot_fleur

plot_fleur(50816)
```

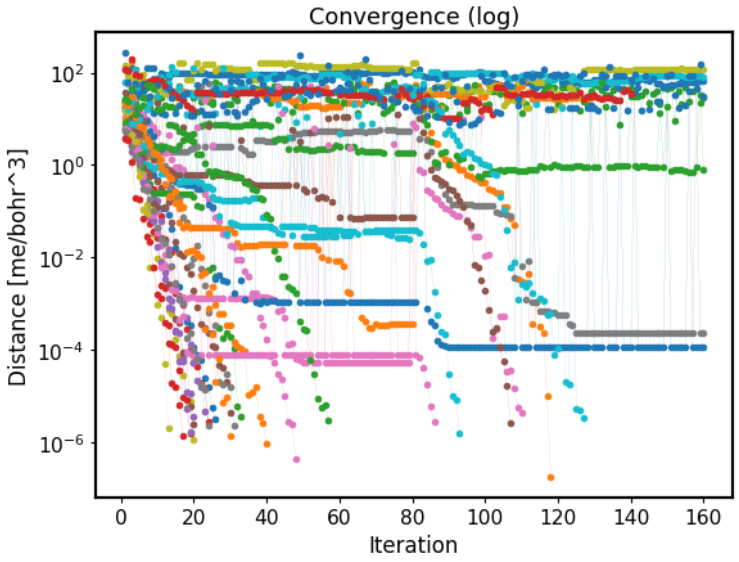




Multi node

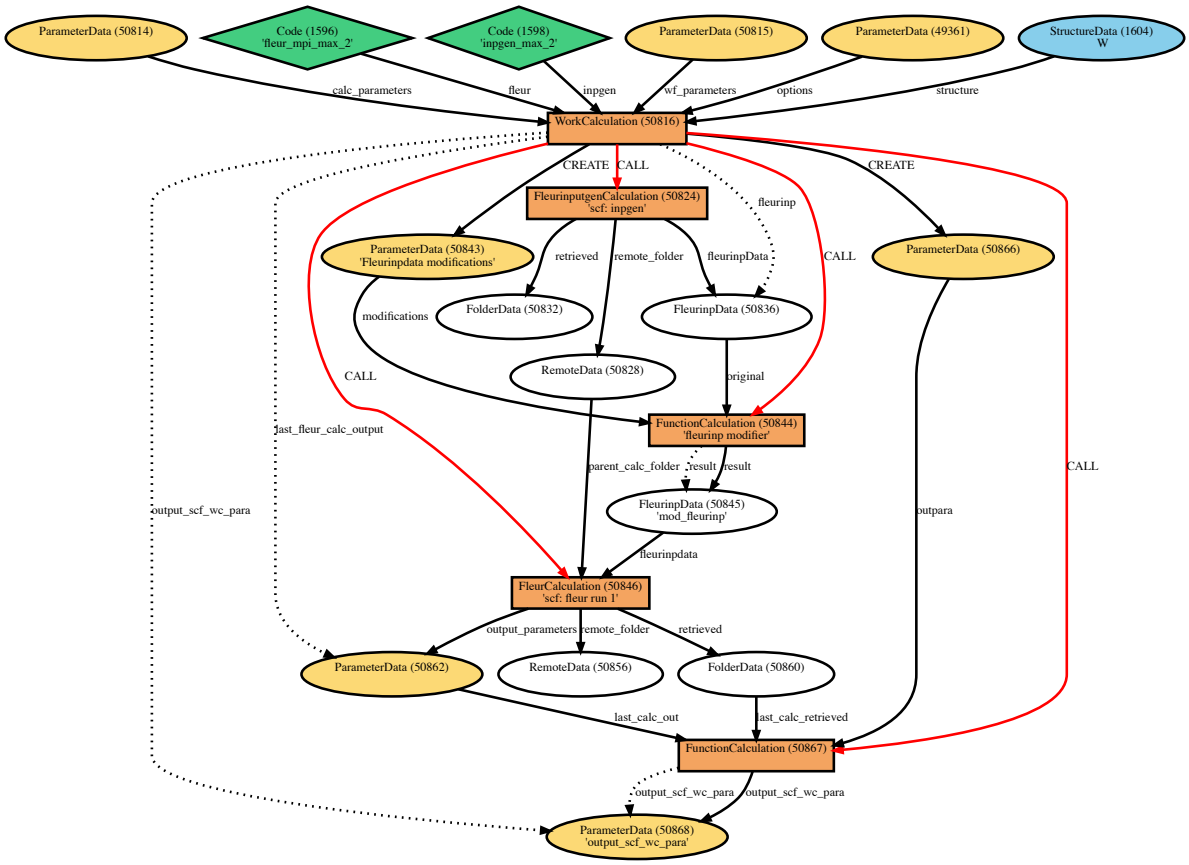
```
from aiida_fleur.tools.plot import plot_fleur

plot_fleur(scf_pk_list)
```


Total energy difference over scf-Iterations
Total energy difference [Htr]
10^{1}
10^{-1}
10^{-3}
10^{-5}
10^{-7}
10^{-9}
0
20
40
60
80
100
120
140
160
Iteration


Convergence (log)
Distance [me/bohr^3]
10^{2}
10^{0}
10^{-2}
10^{-4}
10^{-6}
0
20
40
60
80
100
120
140
160
Iteration

Database Node graph

```
from aiida_fleur.tools.graph_fleur import draw_graph

draw_graph(50816)
```



Example usage

```
# -*- coding: utf-8 -*-
from aiida_fleur.workflows.scf import FleurScfWorkChain
from aiida.orm import Dict, load_node

fleur_code = load_node(FLEUR_PK)
inpgen_code = load_node(INPGEN_PK)
structure = load_node(STRUCTURE_PK)

wf_para = Dict(dict={'fleur_runmax': 3,
                     'density_converged': 0.001,
                     'mode': 'density',
                     'itmax_per_run': 30,
                     'serial': False,
                     'only_even_MPI': False})
```

(continues on next page)

(continued from previous page)

```
options = Dict(dict={'resources': {'num_machines': 1, 'num_mpiprocs_per_
↪machine': 2},
                     'withmpi': True,
                     'max_wallclock_seconds': 600})

calc_parameters = Dict(dict={'kpt': {'div1': 2,
                                     'div2': 2,
                                     'div3': 2
                                     }})

SCF_workchain = submit(FleurScfWorkChain,
                       fleur=fleur_code,
                       inpgen=inpgen_code,
                       calc_parameters=calc_parameters,
                       structure=structure,
                       wf_parameters=wf_para,
                       options=options)
```

Fleur equation of states (eos) workflow

- **Current version**: 0.3.5
- **Class**: *FleurEosWorkChain*
- **String to pass to the** `WorkflowFactory()`: `fleur.eos`
- **Workflow type**: Technical
- **Aim**: Vary the cell volume, to fit an equation of states, (Bulk modulus, . . .)

Contents

Import Example:

```
from aiida_fleur.workflows.eos import fleur_eos_wc
#or
WorkflowFactory('fleur.eos')
```

Description/Purpose

Calculates equation of states for a given crystal structure.

First, an input structure is scaled and a list of scaled structures is constructed. Then, total energies of all the scaled structures are calculated via *FleurScfWorkChain* (*SCF*). Finally, resulting total energies are fitted via the Birch–Murnaghan equation of state and the cell volume corresponding to the lowest energy is evaluated. Other fit options are also available.

Input nodes

The *FleurEosWorkChain* employs exposed feature of the AiiDA-core, thus inputs for the *SCF* sub-workchain should be passed in the namespace called `scf` (see *example of usage*). Please note that the *structure* input node is excluded from the *scf* namespace since the EOS workchain should process input structure before performing energy calculations.

name	type	description	required
scf	namespace	inputs for nested SCF WorkChain. structure input is excluded	no
wf_parameters	Dict	Settings of the workchain	no
structure	StructureData	input structure	no

Returns nodes

name	type	comment
output_eos_wc_para	Dict	results of the workchain
output_eos_wc_structure	StructureData	Crystal structure with the volume of the lowest total energy

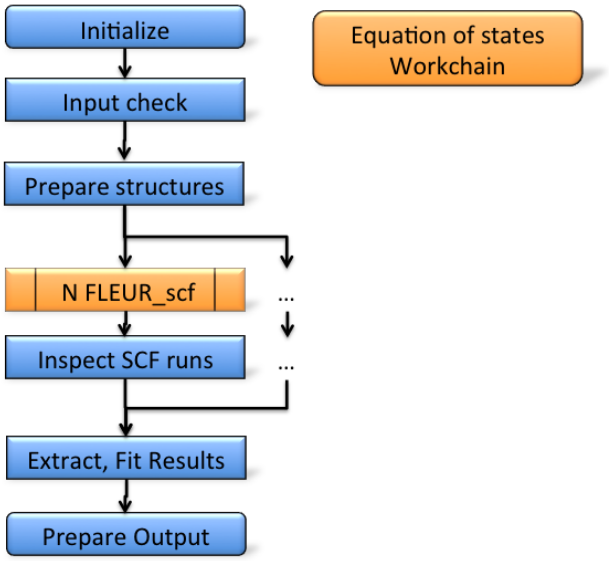
Layout

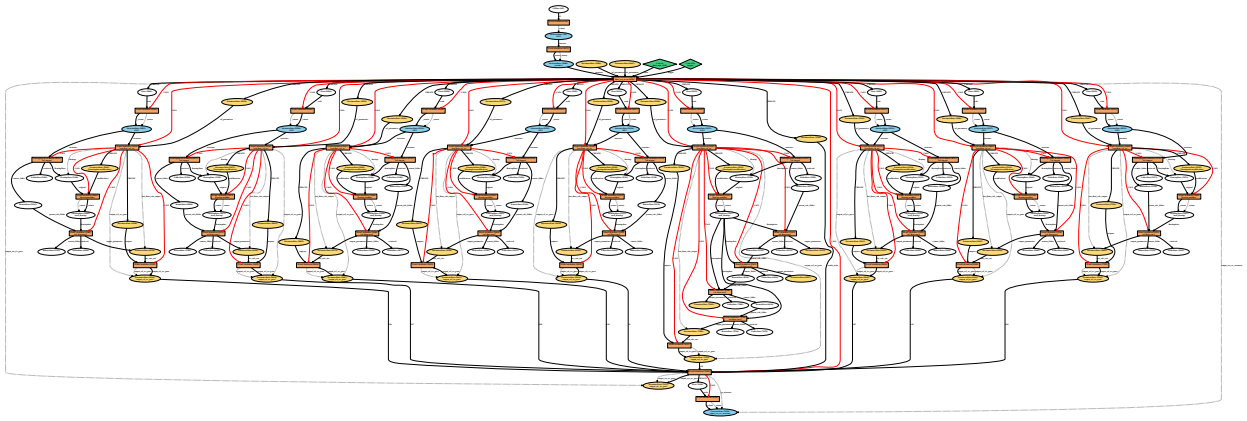
Database Node graph

```
from aiida_fleur.tools.graph_fleur import draw_graph

draw_graph(49670)
```

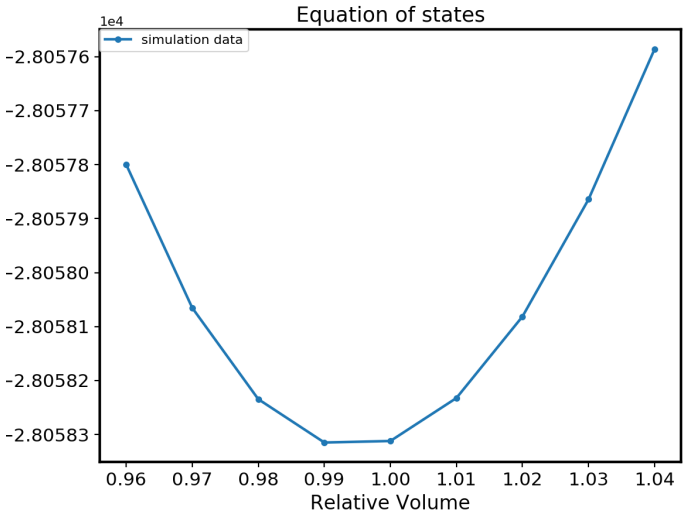
Plot_fleur visualization

Single node


Initialize
Input check
Prepare structures
N FLEUR_scf
Inspect SCF runs
Extract, Fit Results
Prepare Output
Equation of states Workchain
...
...

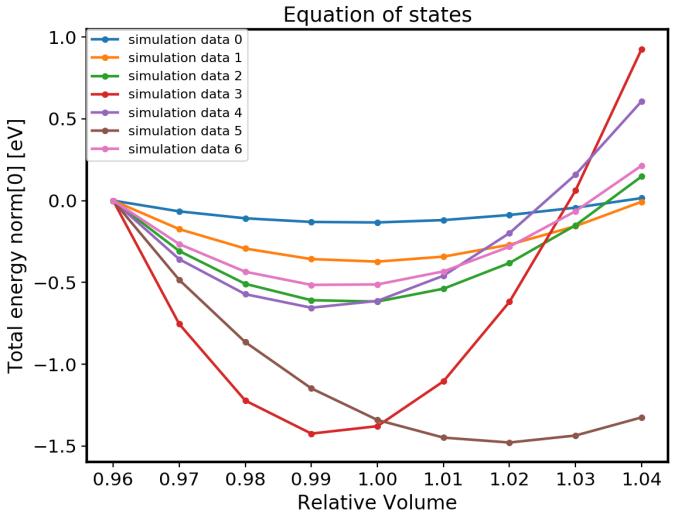


```
from aiida_fleur.tools.plot import plot_fleur

plot_fleur(49670)
```



Multi node

```
from aiida_fleur.tools.plot import plot_fleur

plot_fleur(eos_pk_list)
```



Example usage

```
# -*- coding: utf-8 -*-
from aiida_fleur.workflows.ssdisp import FleurSSDispWorkChain
```

(continues on next page)

(continued from previous page)

```
from aiida.orm import Dict, load_node

fleur_code = load_node(FLEUR_PK)
inpgen_code = load_node(INPGEN_PK)
structure = load_node(STRUCTURE_PK)

wf_para = Dict(dict={'points': 9,
                     'step': 0.002,
                     'guess': 1.00
                     })


wf_para_scf = Dict(dict={'fleur_runmax': 2,
                         'itmax_per_run': 120,
                         'density_converged': 0.2,
                         'serial': False,
                         'mode': 'density'
                })


options_scf = Dict(dict={'resources': {'num_machines': 1, 'num_mpiprocs_per_
↪machine': 8},
                         'queue_name': 'devel',
                         'custom_scheduler_commands': '',
                         'max_wallclock_seconds':  60*60})

inputs = {'scf': {
                  'wf_parameters': wf_para_scf,
                  'calc_parameters': parameters,
                  'options': options_scf,
                  'inpgen': inpgen_code,
                  'fleur': fleur_code
                 },
          'wf_parameters': wf_para,
          'structure': structure
}

SCF_workchain = submit(FleurSSDispWorkChain,
                       fleur=fleur_code,
                       inpgen=inpgen_code,
                       calc_parameters=calc_parameters,
                       structure=structure,
                       wf_parameters=wf_para,
                       options=options)
```

Output node example

```
# -*- coding: utf-8 -*-
{
  'bulk_deriv': -612.513884563477,
  'bulk_modulus': 29201.4098068761,
  'bulk_modulus_units': 'GPa',
  'calculations': [],
  'distance_charge': [
```

(continues on next page)

(continued from previous page)


```
    4.4141e-06,
    4.8132e-06,
    1.02898e-05,
    1.85615e-05
  ],
  'distance_charge_units': 'me/bohr^3',
  'errors': [],
  'guess': 1.0,
  'info': [
    'Consider refining your basis set.'
  ],
  'initial_structure': 'd6985712-7eca-4730-991f-1d924cbd1062',
  'natoms': 1,
  'nsteps': 4,
  'residuals': [],
  'scaling': [
    0.998,
    1.0,
    1.002,
    1.004
  ],
  'scaling_gs': 1.00286268683922,
  'scf_wfs': [],
  'stepsize': 0.002,
  'structures': [
    'f7fddbb5-51af-4dac-a4ba-021d1bf5795b',
    '28e9ed28-837c-447e-aae7-371b70454dc4',
    'fc340850-1a54-4be4-abed-576621b3015f',
    '77fd128b-e88c-4d7d-9aea-d909166926cb'
  ],
  'successful': true,
  'total_energy': [
    -439902.565469453,
    -439902.560450163,
    -439902.564547518,
    -439902.563105211
  ],
  'total_energy_units': 'Htr',
  'volume_gs': 16.2724654374658,
  'volume_units': 'A^3',
  'volumes': [
    16.1935634057491,
    16.2260154366224,
    16.2584674674955,
    16.290919498369
  ],
  'warnings': [
    'Abnormality in Total energy list detected. Check entr(ies) [1].'
  ],
  'workflow_name': 'fleur_eos_wc',
  'workflow_version': '0.3.3'
}
```


Error handling

Total energy check:

The workflow quickly checks the behavior of the total energy for outliers. Which might occur, because the chosen FLAPW parameters might not be good for all volumes. Also local Orbital setup and so on might matter.

- Not enough points for fit
- Some calculations did not converge
- Volume ground state does not lie in the calculated interval, interval refinement

Exit codes

A list of implemented *exit codes*:

Code	Meaning
230	Invalid workchain parameters

Fleur structure optimization Base workchain

- **Current version**: 0.1.0
- **Class**: *~aiida_fleur.workflows.base_relax.FleurBaseRelaxWorkChain*
- **String to pass to the** `WorkflowFactory()`: `fleur.base_relax`
- **Workflow type**: Technical
- **Aim**: Stable execution of *`FleurRelaxWorkChain`*

Contents

Import Example:

```
from aiida_fleur.workflows.base_relax import FleurBaseRelaxWorkChain
#or
WorkflowFactory('fleur.base_relax')
```

Description/Purpose

Optimizes the structure in a way the largest force is lower than a given threshold.

Wraps :ref:'relax_wc' and thus has the same input/output nodes.

Error handling

A list of implemented error handlers:

To be documented.

Example usage

To be documented.

Fleur structure optimization workchain

- **Current version**: 0.2.1
- **Class**: *FleurRelaxWorkChain*
- **String to pass to the** `WorkflowFactory()`: `fleur.relax`
- **Workflow type**: Technical
- **Aim**: Structure optimization of a given structure
- **Computational demand**: Several *FleurScfWorkChain*
- **Database footprint**: Output node with information, full provenance, ~ `10+10*FLEUR Jobs` nodes

Contents

Import Example:

```
from aiida_fleur.workflows.relax import FleurRelaxWorkChain
#or
WorkflowFactory('fleur.relax')
```

Description/Purpose

Optimizes the structure in a way the largest force is lower than a given threshold.

Uses *FleurScfWorkChain* to converge forces first, checks if the largest force is smaller than the threshold. If the largest force is bigger, submits a new *FleurScfWorkChain* for next step structure proposed by FLEUR.

All structure optimization routines implemented in the FLEUR code, the workchain only wraps it.

Input nodes

The FleurSSDispWorkChain employs exposed feature of the AiiDA, thus inputs for the nested *SCF* workchain should be passed in the namespace `scf`.

name	type	description	required
scf	namespace	inputs for nested SCF WorkChain	yes
final_scf	namespace	inputs for a final SCF WorkChain	no
wf_parameters	`Dict`	Settings of the workchain	no

Workchain parameters and its defaults

- `wf_parameters`: `Dict` - Settings of the workflow behavior. All possible keys and their defaults are listed below:

```
# -*- coding: utf-8 -*-
'film_distance_relaxation': False,    # if True, sets relaxXYZ="FFT" for all atoms
'force_criterion': 0.049,             # Sets the threshold of the largest force
'relax_iter': 5                       # Maximum number of optimization iterations
```

Output nodes

- `output_relax_wc_para`: `Dict` - Information of workflow results
- `optimized_structure`: `StructureData` - Optimized structure

Layout

Geometry optimization workchain always submits SCF WC using inputs given in the `scf` namespace. Thus one can start with a structure, FleurinpData or converged/not-fully-converged charge density.

Output nodes

name	type	comment
output_relax_wc_para	`Dict`	results of the workchain
optimized_structure	*FleurinpData*	FleurinpData that was used (after all modifications)

For now output node contains the minimal amount of information. The content can be easily extended on demand, please contact to developers for request.

```
# this is a content of out output node
{
    "errors": [],
    "force": [
        0.03636428
    ],
    "force_iter_done": 1,
    "info": [],
    "initial_structure": "181c1e8d-3c56-4009-b0bb-e8b76cb417e2",
```

(continues on next page)

(continued from previous page)

```
    "warnings": [],
    "workflow_name": "FleurRelaxWorkChain",
    "workflow_version": "0.1.0"
}
```

Error handling

A list of implemented exit codes:

+———+———————————————- | Code | Meaning | +———+———————————————+ | 230 | Input: Invalid workchain parameters given. | +———+———————————————+ | 231 | Input: Inpgen missing in input for final scf. | +———+———————————————- | 350 | The workchain execution did not lead to | | | relaxation criterion. Thrown in the very | | | end of the workchain. | +———+———————————————+ | 351 | SCF Workchains failed for some reason. | +———+———————————————+ | 352 | Found no relaxed structure info in the output of SCF | +———+———————————————+ | 353 | Found no SCF output | +———+———————————————+ | 354 | Force is small, switch to BFGS | +———+———————————————+

Exit codes duplicating FleurCalculation exit codes:

Exit code	Reason
311	FLEUR calculation failed because atoms spilled to the vacuum
313	Overlapping MT-spheres during relaxation

Example usage

```
# -*- coding: utf-8 -*-
from aiida.orm import load_node, Dict
from aiida.engine import submit

from aiida_fleur.workflows.relax import FleurRelaxWorkChain

fleur_code = load_node(FLEUR_PK)
inpgen_code = load_node(INPGEN_PK)


wf_relax = {'film_distance_relaxation': False,
            'force_criterion': 0.049,
            'relax_iter': 5
            }

wf_relax_scf = {'fleur_runmax': 5,
                'serial': False,
                'itmax_per_run': 50,
                'alpha_mix': 0.015,
                'relax_iter': 25,
                'force_converged': 0.001,
                'force_dict': {'qfix': 2,
                               'forcealpha': 0.75,
                               'forcemix': 'straight'},
```

(continues on next page)

(continued from previous page)

```
                'inpxml_changes': []
                }

wf_relax = Dict(dict=wf_relax)
wf_relax_scf = Dict(dict=wf_relax_scf)

calc_relax = {'comp': {'kmax': 4.0,
                       },
              'kpt': {'div1': 24,
                      'div2': 20,
                      'div3': 1
                      },
              'atom': {'element': 'Pt',
                       'rmt': 2.2,
                       'lmax': 10,
                       'lnonsph': 6,
                       'econfig': '[Kr] 5s2 4d10 4f14 5p6| 5d9 6s1',
                       },
              'atom2': {'element': 'Fe',
                        'rmt': 2.1,
                        'lmax': 10,
                        'lnonsph': 6,
                        'econfig': '[Ne] 3s2 3p6| 3d6 4s2',
                        },
              }

calc_relax = Dict(dict=calc_relax)

options_relax = {'resources': {'num_machines': 1, 'num_mpiprocs_per_machine': 4, 'num_
↪cores_per_mpiproc': 6},
                 'queue_name': 'devel',
                 'custom_scheduler_commands': '',
                 'max_wallclock_seconds':  1*60*60}

inputs = {
    'scf': {
        'wf_parameters': wf_relax_scf,
        'calc_parameters': calc_relax,
        'options': options_relax,
        'inpgen': inpgen_code,
        'fleur': fleur_code
    },
    'wf_parameters': wf_relax
}

res = submit(FleurRelaxWorkChain, **inputs)
```

Fleur dos/band workflows

Warning: These workchains do not work with AiiDA >1.0 version yet. They need to be updated.

These are two seperate workflows which are pretty similar so we treat them here together

- **Class**: *fleur_dos_wc* and *FleurBandDosWorkChain*

- **String to pass to the** `WorkflowFactory()`: `fleur.dos`, `fleur.banddos`
- **Workflow type**: Workflow (lv 1)
- **Aim**: Calculate a density of states. Calculate a Band structure.
- **Compuational demand**: 1 `Fleur Job calculation`
- **Database footprint**: Outputnode with information, full provenance, `~ 10` nodes
- **File repository footprint**: The `JobCalculation` run, plus the DOS or Bandstructure files
- **Additional Info**: Use alone.

Contents

Import Example:

```
from aiida_fleur.workflows.dos import fleur_dos_wc
#or
WorkflowFactory('fleur.dos')

from aiida_fleur.workflows.banddos import FleurBandDosWorkChain
#or
WorkflowFactory('fleur.banddos')
```

Description/Purpose

DOS:

Calculates an Density of states (DOS) ontop of a given Fleur calculation (converged or not).

BandDos:

Calculates an electronic band structure ontop of a given Fleur calculation (converged or not).

In the future we plan to add the possibility to converge a calculation before, and choose the kpaths automatic. This version should be able start simply from a crystal structure.

Each of these workflows prepares/chances the Fleur input and manages one Fleur calculation.

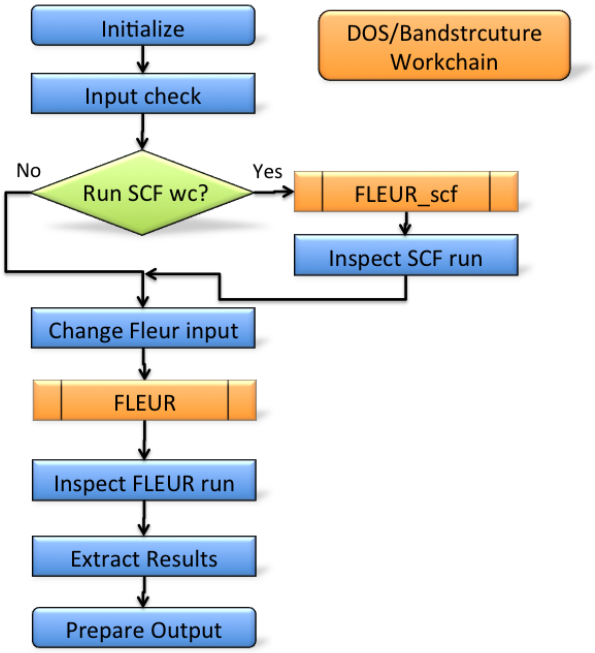
Input nodes:

- `fleur` (`Code`): Fleur code using the `fleur.fleur` plugin
- `wf_parameters` (`Dict`, optional): Some settings of the workflow behavior (e.g. number of kpoints, path, energy sampling and smearing, ...)
- `fleurinp` (*`FleurinpData`*, path 2): Fleur input data object representing the fleur input files.
- `remote_data` (`RemoteData`, optional): The remote folder of the (converged) calculation whose output density is used as input for the DOS, or band structure run.
- `options` (`Dict`, optional): All options available in AiiDA, i.e resource specification, queue name, extras scheduler commands, ...
- `settings` (`Dict`, optional): special settings for Fleur calculations, will be given like it is through to calculationss.

Returns nodes

- `output_dos_wc_para` (`Dict`): Information of the dos workflow results like success, last result node, list with convergence behavior
- `output_band_wc_para` (`Dict`): Information node from the band workflow
- `last_fleur_calc_output` (`Dict`) Output node of last Fleur calculation is returned.

Layout



Database Node graph

```
from aiida_fleur.tools.graph_fleur import draw_graph

draw_graph(76867)
```

Plot_fleur visualization

Single node

```
from aiida_fleur.tools.plot import plot_fleur

# DOS calc
plot_fleur(76867)
```

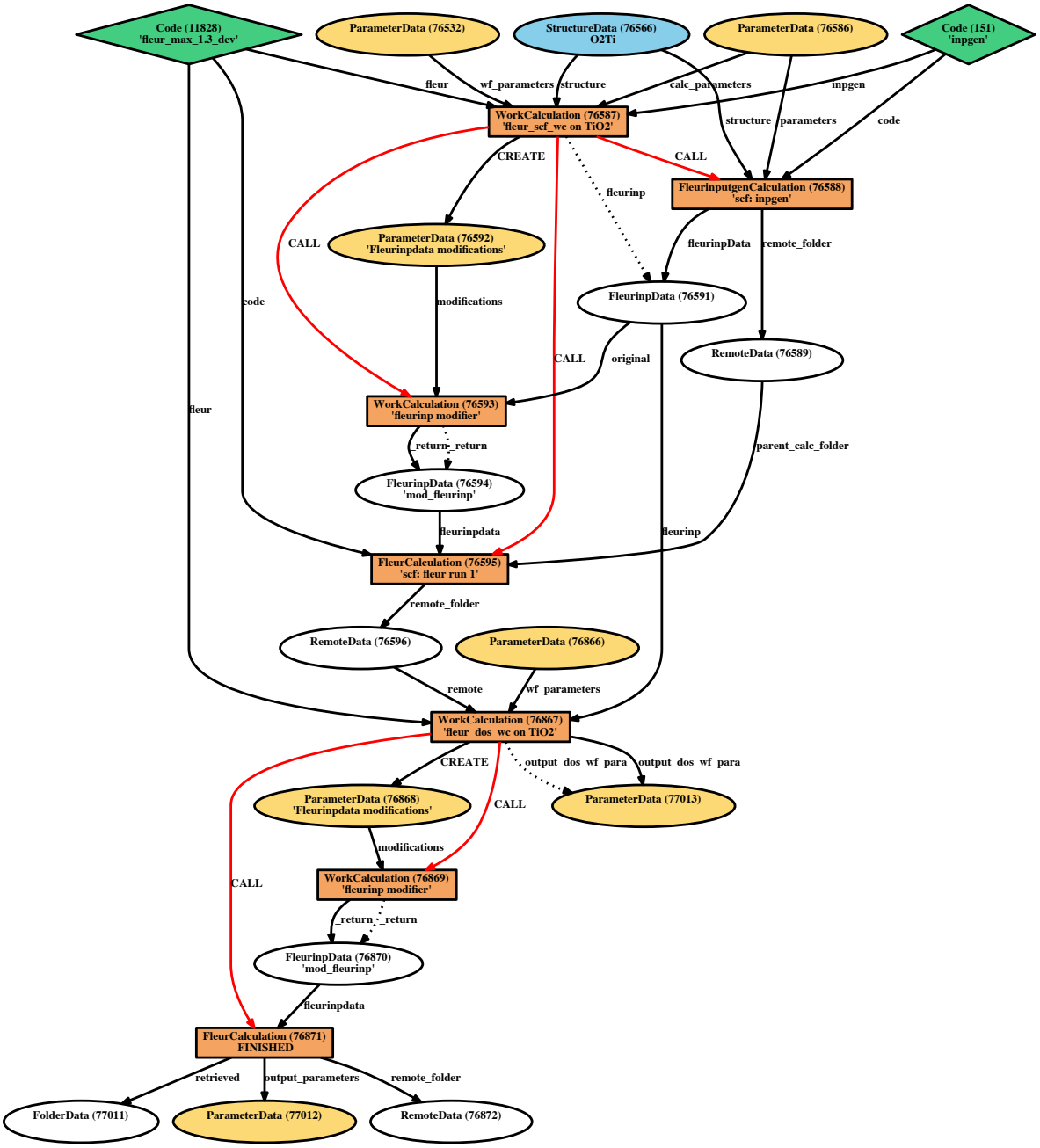
Multi node just does a bunch of single plots for now.

```
from aiida_fleur.tools.plot import plot_fleur

plot_fleur(dos_pk_list)
```

Example usage

```
# -*- coding: utf-8 -*-
###############################################################################
↪##
# Copyright (c), Forschungszentrum Jülich GmbH, IAS-1/PGI-1, Germany.
↪ #
#                All rights reserved.
↪ #
# This file is part of the AiiDA-FLEUR package.
↪ #
#
↪ #
# The code is hosted on GitHub at https://github.com/broeder-j/aiida-fleur
↪ #
# For further information on the license, see the LICENSE.txt file
↪ #
# For further information please visit http://www.flapw.de or
↪ #
# http://aiida-fleur.readthedocs.io/en/develop/
↪ #
###############################################################################
↪##
"""
Here we run the fleur_dos_wc for a Fleur calculation which has been
↪converged before
Layout:

1. Database env load, Import, create base classes
2. Creation of  input nodes
3. Lauch workchain
"""
```

(continues on next page)


Code (11828)
'fleur_max_1.3_dev'
ParameterData (76532)
StructureData (76566)
O2Ti
ParameterData (76586)
Code (151)
'inpgen'
fleur
wf_parameters
structure
calc_parameters
inpgen
WorkCalculation (76587)
'fleur_scf_wc on TiO2'
structure
parameters
code
CREATE
CALL
FleurinputgenCalculation (76588)
'scf: inpgen'
fleurinp
CALL
ParameterData (76592)
'Fleurinpdata modifications'
fleurinpData
remote_folder
code
modifications
FleurinpData (76591)
RemoteData (76589)
CALL
original
WorkCalculation (76593)
'fleurinp modifier'
fleur
_return
_return
parent_calc_folder
FleurinpData (76594)
'mod_fleurinp'
fleurinpdata
fleurinp
FleurCalculation (76595)
'scf: fleur run 1'
remote_folder
RemoteData (76596)
ParameterData (76866)
remote
wf_parameters
WorkCalculation (76867)
'fleur_dos_wc on TiO2'
CREATE
output_dos_wf_para
output_dos_wf_para
ParameterData (76868)
'Fleurinpdata modifications'
CALL
ParameterData (77013)
modifications
CALL
WorkCalculation (76869)
'fleurinp modifier'
_return
_return
FleurinpData (76870)
'mod_fleurinp'
fleurinpdata
FleurCalculation (76871)
FINISHED
retrieved
output_parameters
remote_folder
FolderData (77011)
ParameterData (77012)
RemoteData (76872)

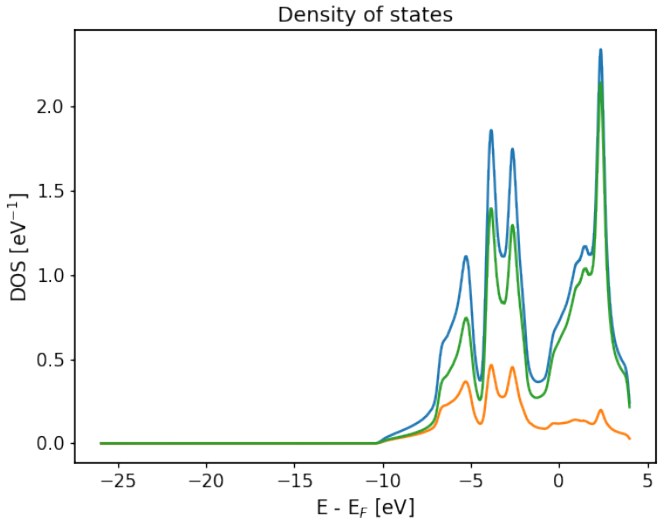


Fig. 1: For the bandstructure visualization it depends on the File produced. Old bandstructure file:

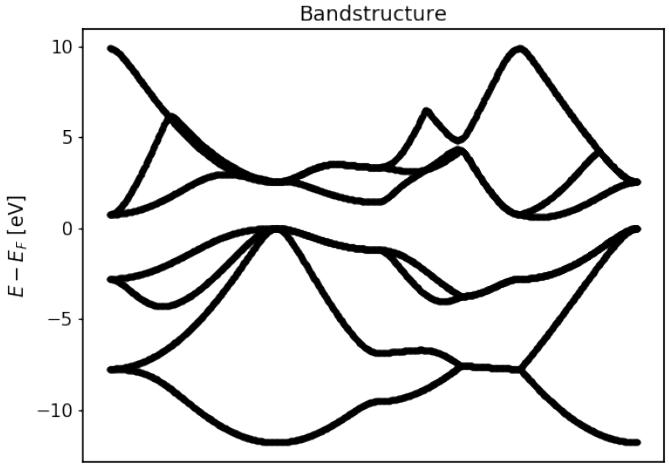
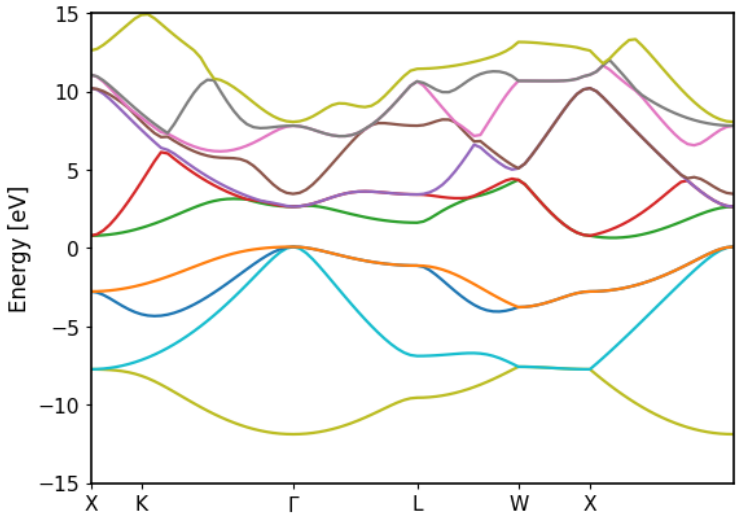
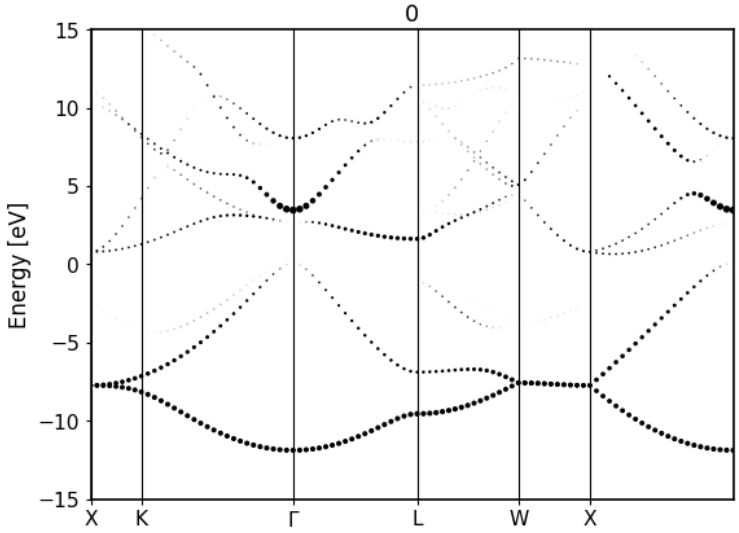
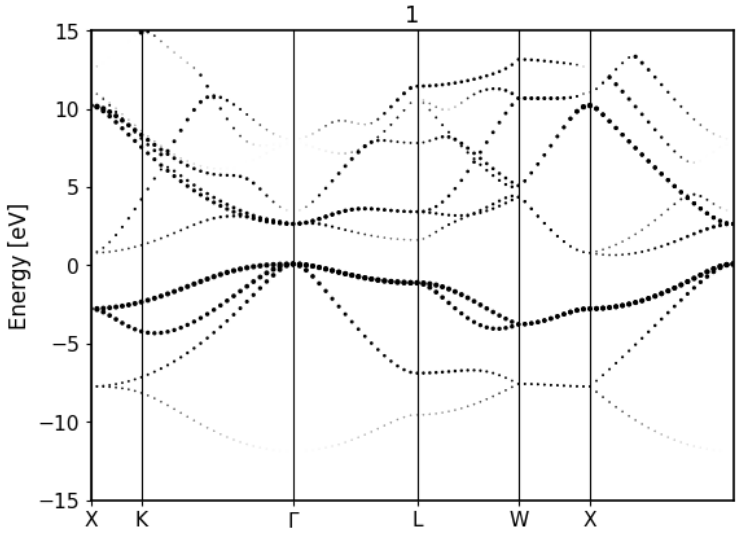
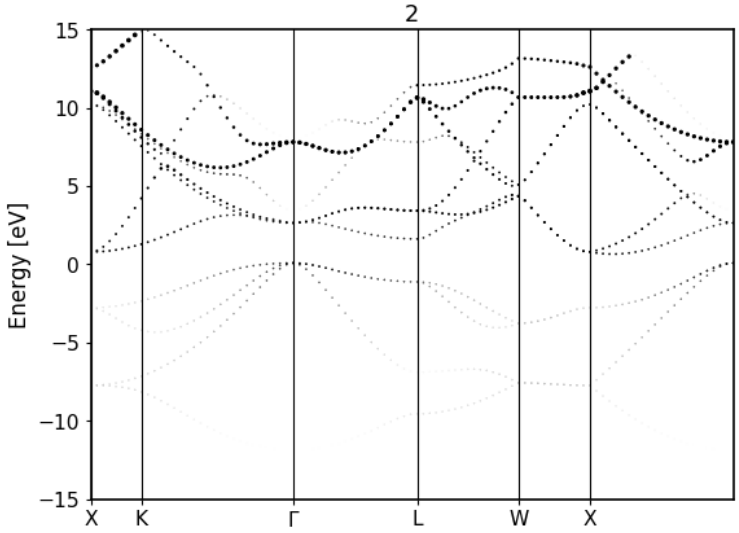


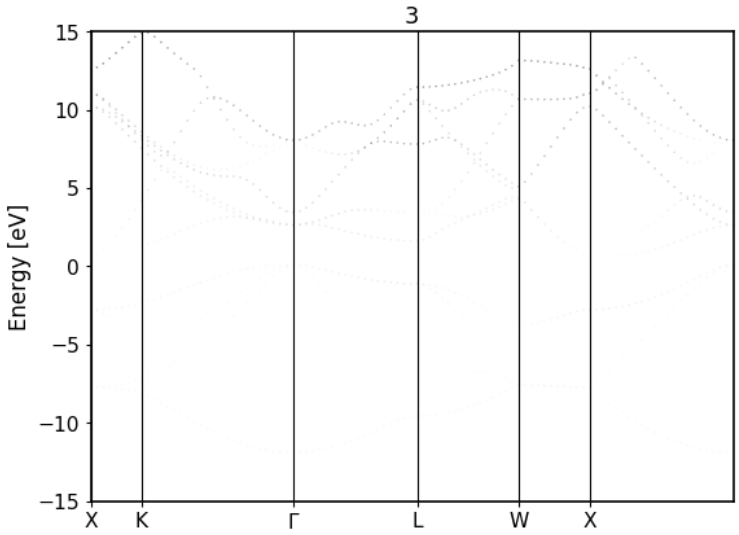
Fig. 2: Bandstructure `band_dos.hdf` file with l-like charge information: Band resolved bandstructure and fat-bands for the different channels. Spin and combinded DOS plus band structure visualizations are in progress...


15
10
5
0
−5
−10
−15
Energy [eV]
X
K
Γ
L
W
X


0
15
10
5
0
−5
−10
−15
Energy [eV]
X
K
Γ
L
W
X


1
15
10
5
0
−5
−10
−15
Energy [eV]
X
K
Γ
L
W
X


2
Energy [eV]
15
10
5
0
−5
−10
−15
X
K
Γ
L
W
X


3
Energy [eV]
15
10
5
0
−5
−10
−15
X
K
Γ
L
W
X

(continued from previous page)

```
######################
# 1. Load the database environment. Imports and base class creation

from __future__ import absolute_import
from aiida import load_dbenv, is_dbenv_loaded
if not is_dbenv_loaded():
    load_dbenv()

from aiida.plugins import DataFactory
from aiida.orm import Code, load_node
from aiida.engine.launch import submit, run
from aiida_fleur.workflows.dos import fleur_dos_wc

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

######################
# 2. Creation/loding of input nodes

# Load the codes, thwy have to be setup in your database.
fleur_label = 'fleur@localhost'
fleur_code = Code.get_from_string(fleur_label)

### Create wf_parameters (optional) and options
wf_para = Dict(dict={'fleur_runmax': 4, 'density_criterion': 0.000001,
↪'serial': False})

options = Dict(dict={'resources': {'num_machines': 1}, 'queue_name': '',
↪'max_wallclock_seconds': 60 * 60})

# load a fleurino data object from a scf_wc before
##############################
# 3. submit the workchain with its inputs.

inputs = {}
inputs['wf_parameters'] = wf_para
inputs['fleurinp'] = fleurinp
inputs['fleur'] = fleur_code
inputs['description'] = 'test fleur_dos_wc run on W'
inputs['label'] = 'dos test '
inputs['options'] = options

# submit workchain to the daemon
# Noice that the nodes we created before are not yet stored in the database,
# but AiiDA will do so automaticly when we launch the workchain.
# To reuse nodes it might be a good idea, to save them before by hand and␣
↪then load them
res = submit(fleur_dos_wc, **inputs)

# You can also run the workflow in the python interpreter as blocking
#res = run(fleur_dos_wc, **inputs)
```

Output node example

Error handling

Still has to be documented

Warning if parent calculation was not converged.

More advanced (Scientific) Workchains

- **Current version**: 0.4.0

Fleur initial core-level shifts workflow

Class name, import from:

```
from aiida_fleur.workflows.initial_cls import fleur_initial_cls_wc
#or
WorkflowFactory('fleur.init_cls')
```

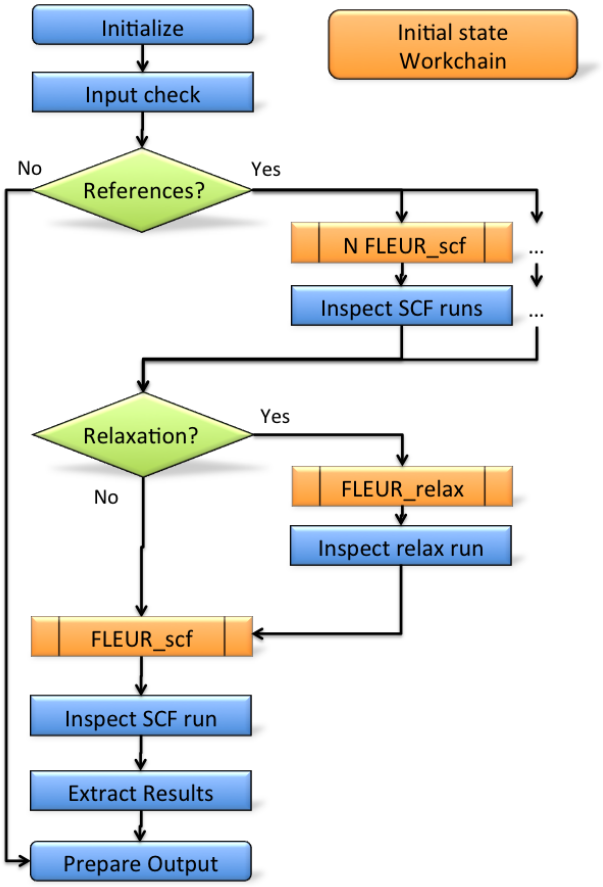
Description/Purpose

The initial-state workflow *fleur_initial_cls_wc* calculates core-level shifts of a system with respect to the elemental references via normal SCF calculations. If required, the SCF calculations of the corresponding elemental references are also managed by the workflow. Furthermore, the workflow extracts the enthalpy of formation for the investigated compound from these SCF runs. The workflow calculates core-level shifts (CLS) as the difference of Kohn-Sham core-level energies with respect to the respected Fermi level.

This workflow manages none to one inpgen calculation and one to several Fleur calculations. It is one of the most core workflows and often deployed as sub-workflow.

Note: To minimize uncertainties on CLS it is important that the compound as well as the reference systems are calculated with the same atomic parameters (muffin-tin radius, radial grid points and spacing, radial basis cutoff). The workflow tests for this equality and tries to assure it, though it does not know what is a good parameter set nor if the present set works well for both systems. It is currently best practice to enforce the FLAPW parameters used within the workflow, i.e., provide them as input for the system as for the references. For low high-throughput failure rates and smallest data footprint we advice to calculate the references first alone and parse a converged calculation as a reference, that way references are not rerun and produce less overhead. Otherwise one can also turn on *caching* in AiiDA which will save the recalculation of the references, but won't decrease their data footprint.

Layout


Initialize
Initial state Workchain
Input check
No
References?
Yes
N FLEUR_scf
...
Inspect SCF runs
...
Relaxation?
Yes
No
FLEUR_relax
Inspect relax run
FLEUR_scf
Inspect SCF run
Extract Results
Prepare Output

Input nodes

name	type	description	required
inpgen	`Code`	Inpgen code	yes
fleur	`Code`	Fleur code	yes
wf_parameters	`Dict`	Settings of the workchain	no
fleurinp	*`FleurinpData`*	*FLEUR input*	no
structure	`StructureData`	Crystal structure	no
calc_parameters	`Dict`	FLAPW parameters for given structure	no
options	`Dict`	AiiDA options (computational resources)	no

More details:

- `fleur` (*aiida.orm.Code*): Fleur code using the `fleur.fleur` plugin
- `inpgen` (*aiida.orm.Code*): Inpgen code using the `fleur.inpgen` plugin
- `wf_parameters` (*Dict*, optional): Some settings of the workflow behavior
- `structure` (*StructureData*, path 1): Crystal structure data node.
- `calc_parameters` (*Dict*, optional): Longer description of the workflow
- `fleurinp` (*FleurinpData*, path 2): Label of the workflow

Workchain parameters and its defaults

wf_parameters

`wf_parameters`: `Dict` - Settings of the workflow behavior. All possible keys and their defaults are listed below:

```
# -*- coding: utf-8 -*-
'relax': True,                # Not implemented, relax the structure
'relax_mode': 'Fleur',        # Not implemented, how to relax the structure
'relax_para': 'default',      # Not implemented, parameter for the relaxation
'scf_para': 'default',        # Use these parameters for the SCFs
'same_para': True,            # enforce the same parameters
'serial': False,              # Run everthing in serial
'references': {}              # Dict to provide the elemental references
                              # i.e { 'W': calc, outputnode of SCF workflow or␣
↪fleurinp,
                              # or structure data or (structure data + Parameter),
                              # 'Be' : ...}
```

options

`options`: `Dict` - AiiDA options (computational resources). Example:

```
'resources': {"num_machines": 1, "num_mpiprocs_per_machine": 1},
'max_wallclock_seconds': 6*60*60,
'queue_name': '',
'custom_scheduler_commands': '',
'import_sys_environment': False,
'environment_variables': {}
```

Returns nodes

The table below shows all the possible output nodes of the fleur_initial_cls_wc workchain.

name	type	comment
output_initial_cls_wc_para	Dict	Link to last FleurCalculation output dict

More details:

- `output_initial_cls_wc_para`: `Dict` - Main results of the workchain. Contains core-level shifts, band gaps, core-levels, atom-type information, errors, warnings, other information. An example:


```
# -*- coding: utf-8 -*-
{
    'atomtypes': {
        'W': [
            {
                'atomic_number': 74,
                'coreconfig': '[Kr] (5s1/2) (4d3/2) (4d5/2) (4f5/2) (4f7/2)',
                'element': 'W',
                'natoms': 1,
                'species': 'W-1',
                'stateOccupation': [
                    {
                        '(5d3/2)': [
                            '2.00000000',
                            '.00000000'
                        ]
                    },
                    {
                        '(5d5/2)': [
                            '2.00000000',
                            '.00000000'
                        ]
                    }
                ],
                'valenceconfig': '(5p1/2) (5p3/2) (6s1/2) (5d3/2) (5d5/2)'
            }
        ]
    },
    'bandgap': 0.0074571775,
    'bandgap_units': 'htr',
    'binding_energy_convention': 'negativ',
    'corelevel_energies': {
        'W': [
            [
                -2550.2512204246,
                -439.7260486989,
                -420.4442892264,
                -370.7259449483,
                -101.1391871143,
                -92.5627547497,
                -81.8114542005,
                -20.7351164096,
                -67.3928879745,
                -65.0551729884,
                -17.1796863155,
```


(continues on next page)

(continued from previous page)


```
                -14.6884205438,
                -2.7326665018,
                -8.8548575156,
                -8.3959093745,
                -1.0995859461,
                -1.0173114662
            ]
        ]
    },
    'corelevel_energies_units': 'htr',
    'corelevelshifts': {
        'W': [
            [
                0.0,
                0.0,
                0.0,
                ...
            ]
        ]
    },
    'corelevelshifts_units': 'htr',
    'fermi_energy': 0.6026436555,
    'fermi_energy_units': 'htr',
    'formation_energy': [
        0.0
    ],
    'formation_energy_units': 'eV/atom',
    'material': 'W',
    'reference_bandgaps': [
        0.0074571775
    ],
    'reference_bandgaps_des': [
        'W'
    ],
    'reference_corelevel_energies': {
        'W': [
            [
                -2550.2512204246,
                -439.7260486989,
                -420.4442892264,
                -370.7259449483,
                -101.1391871143,
                -92.5627547497,
                -81.8114542005,
                -20.7351164096,
                -67.3928879745,
                -65.0551729884,
                -17.1796863155,
                -14.6884205438,
                -2.7326665018,
                -8.8548575156,
                -8.3959093745,
                -1.0995859461,
                -1.0173114662
            ]
        ]
    },
```


(continues on next page)

(continued from previous page)


```
    'reference_corelevel_energies_units': 'htr',
    'reference_fermi_energy': [
        0.6026436555
    ],
    'reference_fermi_energy_des': [
        'W'
    ],
    'successful': true,
    'total_energy': -439902.56049548,
    'total_energy_ref': [
        -439902.56049548
    ],
    'total_energy_ref_des': [
        'W'
    ],
    'total_energy_units': 'eV',
    'warnings': [],
    'workflow_name': 'fleur_initial_cls_wc',
    'workflow_version': '0.4.0'
}
```


Plot_fleur visualization

Single node

```
from aiida_fleur.tools.plot import plot_fleur

plot_fleur(50816)
```

Example usage

```
# -*- coding: utf-8 -*-
from aiida.orm import load_node, Dict
from aiida.engine import submit
from aiida.plugins import WorkflowFactory


fleur_init_cls_wc = WorkflowFactory('fleur.initial_cls')
struc = load_node(STRUCTURE_PK)
flapw_para = load_node(PARAMETERS_PK)
fleur_code = load_node(FLEUR_PK)
inpgen_code = load_node(INPGEN_PK)

options = Dict(dict={'resources': {'num_machines': 2, 'num_mpiprocs_per_
↪machine': 24},
                     'queue_name': '',
                     'custom_scheduler_commands': '',
                     'max_wallclock_seconds':  60*60})

wf_para_initial = Dict(dict={
  'references': {'Be': '257d8ae8-32b3-4c95-8891-d5f527b80008',
                 'W': 'c12c999c-9a00-4866-b6ef-9bb5d28e7797'},
  'scf_para': {'density_criterion': 5e-06, 'fleur_runmax': 3, 'itmax_per_run
↪': 80}})
```

(continues on next page)

(continued from previous page)

```
# launch workflow
initial_res = submit(fleur_init_cls_wc, wf_parameters=wf_para_initial,␣
↪structure=struc,
            calc_parameters=flapw_para, options=options, fleur=fleur,␣
↪inpgen=inpgen,
            label='test initial cls', description='fleur_initial_cls test')
```

Error handling

Still has to be documented.

So far only the input is checked. All other errors are currently not handled. The SCF sub-workchain comes with its own error handling of course.

Fleur core-hole workflow

Class name, import from:

```
from aiida_fleur.workflows.corehole import fleur_corehole_wc
#or
WorkflowFactory('fleur.corehole')
```

Description/Purpose

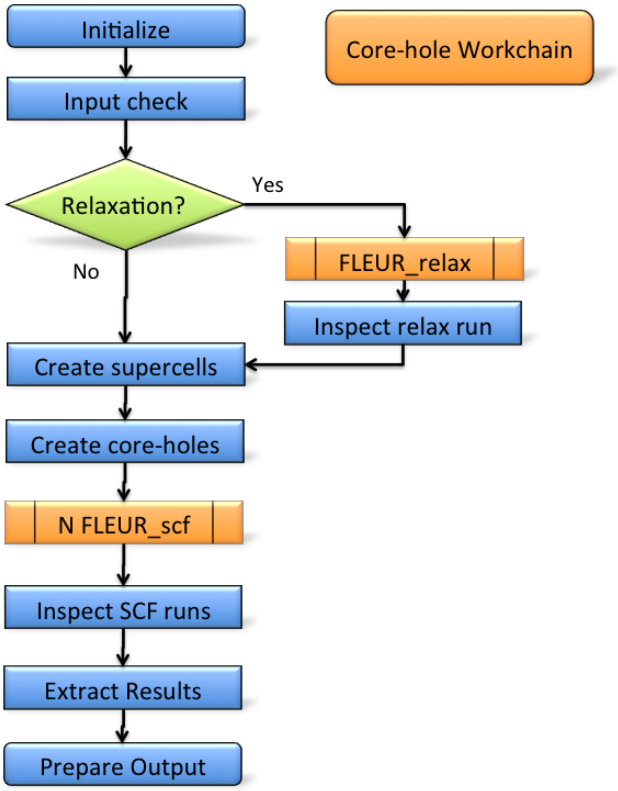
The core-hole workflow can be deployed to calculate absolute core-level binding energies.

Such core-hole calculations are performed through a super-cell setup. The workflow allows for arbitrary corehole charges and of valence and charged type core-holes. From a computational cost perspective it may be cheaper to calculate all relative initial-state shifts of a structure and then launch one core-hole calculation on the structure to get an absolute reference energy instead of performing expensive core-hole calculations for all atom-types in the structure. The core-hole workflow implements the usual FLEUR workflow interface with a workflow control parameter node.

Layout

Input nodes

name	type	description	required
inpgen	`Code`	Inpgen code	yes
fleur	`Code`	Fleur code	yes
wf_parameters	`Dict`	Settings of the workchain	no
fleurinp	*`FleurinpData`*	*FLEUR input*	no
structure	`StructureData`	Crystal structure	no
calc_parameters	`Dict`	FLAPW parameters for given structure	no
options	`Dict`	AiiDA options (computational resources)	no



More details:

- `fleur` (*aiida.orm.Code*): Fleur code using the `fleur.fleur` plugin
- `inpgen` (*aiida.orm.Code*): Inpgen code using the `fleur.inpgen` plugin
- `wf_parameters` (*Dict*, optional): Some settings of the workflow behavior
- `structure` (*StructureData*, path 1): Crystal structure data node.
- `calc_parameters` (*Dict*, optional): Longer description of the workflow
- `fleurinp` (*FleurinpData*, path 2): Label of the workflow

Workchain parameters and its defaults

wf_parameters

`wf_parameters`: `Dict` - Settings of the workflow behavior. All possible keys and their defaults are listed below:

```
# -*- coding: utf-8 -*-
'method' : 'valence',       # what method to use, default for valence to highest open␣
↪shell
'hole_charge' : 1.0,        # what is the charge of the corehole? 0<1.0
'atoms' : ['all'],          # coreholes on what atoms, positions or index for list,
                            # or element ['Be', (0.0, 0.5, 0.334), 3]
'corelevel': ['all'],       # coreholes on which corelevels [ 'Be1s', 'W4f', 'Oall'...]
'supercell_size' : [2,1,1],# size of the supercell [nx,ny,nz]
```

(continues on next page)

(continued from previous page)

```
'para_group' : None,        # use parameter nodes from a parameter group
'relax' : False,            # relax the unit cell first?
'relax_mode': 'Fleur',      # what releaxation do you want
'relax_para' : 'default',   # parameter dict for the relaxation
'scf_para' : 'default',     # wf parameter dict for the scfs
'same_para' : True,         # enforce the same atom parameter/cutoffs on the corehole␣
↪calc and ref
'serial' : True,            # run fleur in serial, or parallel?
'magnetic' : True           # jspins=2, makes a difference for coreholes
```

options

options: Dict - AiiDA options (computational resources). Example:

```
'resources': {"num_machines": 1, "num_mpiprocs_per_machine": 1},
'max_wallclock_seconds': 6*60*60,
'queue_name': '',
'custom_scheduler_commands': '',
'import_sys_environment': False,
'environment_variables': {}
```

Returns nodes

- output_corehole_wc_para (*Dict*): Information of workchain results

More details:

- output_corehole_wc_para: Dict - Main results of the workchain. Contains Binding energies, band gaps, core-levels, atom-type information, errors, warnings, other information. An example:

```
# -*- coding: utf-8 -*-
{'atomtypes': [[
  {'atomic_number': 4, 'coreconfig': '(1s1/2)', 'element': 'Be', 'natoms': 1,
   'species': 'Be_corehole1', 'stateOccupation': [
      {'(1s1/2)': ['1.00000000', '.50000000']},
      {'(2p1/2)': ['.50000000', '.00000000']}], 'valenceconfig': '(2s1/2) (2p1/2)
↪'},
  {'atomic_number': 4, 'coreconfig': '[He]', 'element': 'Be', 'natoms': 1,
   'species': 'Be-2', 'stateOccupation': [{'(2p1/2)': ['.00000000', '.00000000']}
↪],
   'valenceconfig': '(2s1/2) (2p1/2)'},
  {'atomic_number': 4, 'coreconfig': '[He]', 'element': 'Be', 'natoms': 1,
   'species': 'Be-2', 'stateOccupation': [{'(2p1/2)': ['.00000000', '.00000000']}
↪],
   'valenceconfig': '(2s1/2) (2p1/2)'},
  {'atomic_number': 4, 'coreconfig': '[He]', 'element': 'Be', 'natoms': 1,
   'species': 'Be-2', 'stateOccupation': [{'(2p1/2)': ['.00000000', '.00000000']}
↪],
   'valenceconfig': '(2s1/2) (2p1/2)'}]], 'bandgap': [0.0004425914],
  'bandgap_units':'eV', 'binding_energy': [53.57027767044], 'corehole_type':
↪'valence',
  'binding_energy_units': 'eV', 'binding_energy_convention': 'negativ',
  'coreholes_calculated': 'Be1s', 'coreholes_calculated_details': '', 'coresetup
↪': [],
```

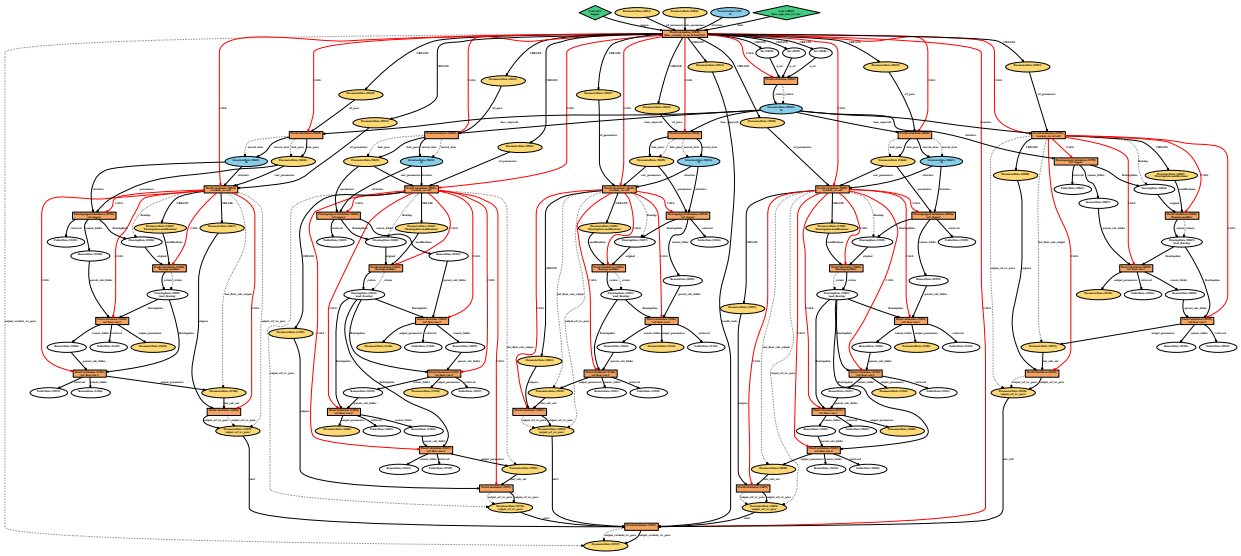
(continues on next page)

(continued from previous page)

```
  'errors': [], 'fermi_energy': [0.3138075709], 'fermi_energy_unit': 'eV',
  'reference_bandgaps': [0.0225936434], 'reference_coresetup': [],
  'successful': true, 'total_energy_all': [-1554.08485250996],
  'total_energy_all_units': 'eV', 'total_energy_ref': [-1607.6551301804],
  'total_energy_ref_units': 'eV', 'warnings': [], 'hints': [],
  'weighted_binding_energy': [107.14055534088], 'weighted_binding_energy_units':
↪'eV',
  'workflow_name': 'fleur_corehole_wc', 'workflow_version': '0.3.2'
}
```

Database Node graph

```
from aiida_fleur.tools.graph_fleur import draw_graph

draw_graph(30528)
```



Plot_fleur visualization

Currently there is no visualization directly implemented for plot fleur. Through there in masci-tools there are methods to visualize spectra and binding energies

Example usage

```
# -*- coding: utf-8 -*-
from aiida.orm import load_node, Dict
from aiida.engine import submit
from aiida.plugins import WorkflowFactory


fleur_corehole_wc = WorkflowFactory('fleur.corehole')
```

(continues on next page)

(continued from previous page)

```
struc = load_node(STRUCTURE_PK)
flapw_para = load_node(PARAMETERS_PK)
fleur_code = load_node(FLEUR_PK)
inpgen_code = load_node(INPGEN_PK)

options = Dict(dict={'resources': {'num_machines': 2, 'num_mpiprocs_per_
↪machine': 24},
                         'queue_name': '',
                         'custom_scheduler_commands': '',
                         'max_wallclock_seconds':  60*60})

wf_para_corehole = Dict(dict={u'atoms': [u'Be'], #[u'all'],
  u'supercell_size': [2, 2, 2], u'corelevel': ['1s'], #[u'all'],
  u'hole_charge': 1.0, u'magnetic': True, u'method': u'valence', u'serial':␣
↪False})

# launch workflow
dos = submit(fleur_corehole_wc, wf_parameters=wf_para_corehole,␣
↪structure=struc,
             calc_parameters=flapw_para, options=options,
             fleur=fleur, inpgen=inpgen, label='test core hole wc',
             description='fleur_corehole test')
```

Error handling

Still has to be documented

Fleur Create Magnetic Film workchain

- **Current version**: 0.1.1
- **Class**: FleurCreateMagneticWorkChain
- **String to pass to the** `WorkflowFactory()`: `fleur.create_magnetic`
- **Workflow type**: Scientific workchain

Contents

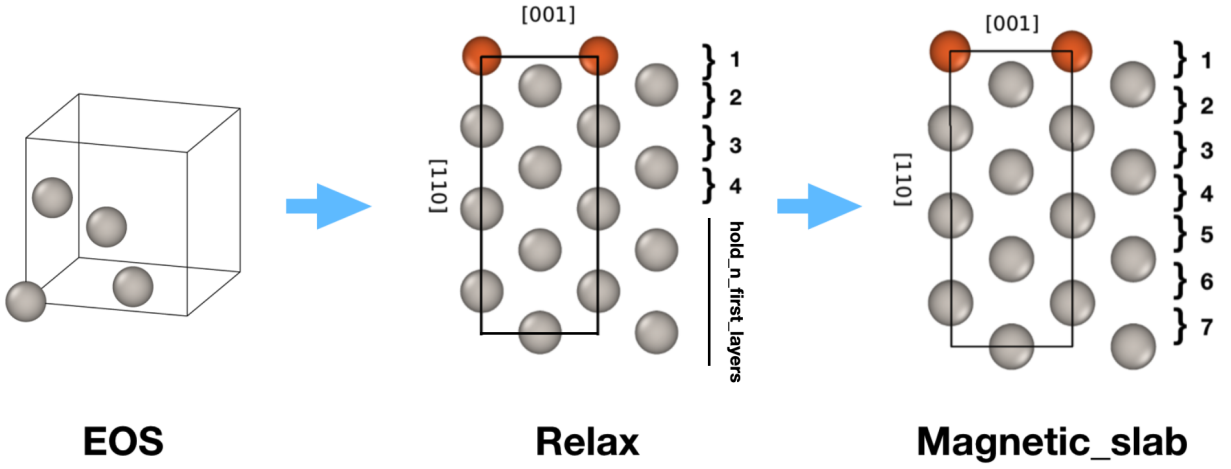
Import Example:

```
from aiida_fleur.workflows.create_magnetic_film import FleurCreateMagneticWorkChain
#or
WorkflowFactory('fleur.create_magnetic')
```

Description/Purpose

The workchain constructs a relaxed film structure which is ready-to-use in the subsequent magnetic workchains, such as *DMI*, *MAE* or *SSDisp* workchains.

The main inputs include information about the substrate (structure type, miller indices, element) and deposited material. The main logic of the workchain is depicted on the figure below:



First, the workchain uses *EOS workchain* to find the equilibrium lattice parameters for the substrate. For now only bcc and fcc substrate lattices are supported. Note, the algorithm always uses conventional unit cells e.g. one gets 4 atoms in the unit cell for fcc lattice (see the figure above).

After that, the workchain constructs a film which will be used for interlayer distance relaxation via the *relaxation workchain*. The algorithm creates a film using given miller indices and the ground state lattice constant and replaces some layers with another elements given in the input. For now only single-element layer replacements are possible i.e. each resulting layer can be made of a single element. It is not possible to create e.g. B-N monolayer using this workchain. If we refer to the figure above, in ideal case one constructs a structure with an inversion or z-reflection symmetry to calculate interlayer distances 1-4. However, the workchain does not ensure an inversion or z-reflection symmetry, that is user responsibility to make it. For instance, if you want to achieve one of these symmetries you should pass positive and negative numbers of layer in the replacements dictionary of the wf parameters, see an example in *defaults*.

Note: z-reflection or inversion symmetries are not ensured by the workchain even if you specify symmetric replacements. Sometimes you need to remove a few layers before replacements. For example, consider the case of fcc (110) film: if `size` is equal to (1, 1, 4) there are will be 8 layers in the template before the replacements since there are 2 layers in the unit cell. That means the x,y positions of the first atom are equal to (0.0, 0.0) when the 8th atom coordinates are equal to (0.5, 0.5). Thus, to achieve z-reflection symmetry one needs to remove the 8th layer by specifying `'pop_last_layers' : 1` in the wf parameters.

Finally, using the result of the relaxation workchain, a magnetic structure having no z-reflection symmetry is created. For this the workchain takes first N layers from the relaxed structure and attaches M substrate layers to the bottom. The final structure is z-centralised.

Input nodes

The FleurCreateMagneticWorkChain employs exposed feature of the AiiDA-core, thus inputs for the *EOS* and *relaxation* workchains should be passed in the namespaces `eos` and `relax` correspondingly (see *example of usage*). Please note that the *structure* input node is excluded from the EOS namespace and from the Relax SCF namespace since corresponding input structures are created within the CreateMagnetic workchain.

name	type	description	required
eos	namespace	inputs for nested EOS WC. structure input is excluded.	no
relax	namespace	inputs for nested Relax WC. structure input of SCF sub-namespace is excluded	no
wf_parameters	`Dict`	Settings of the workchain	no
eos_output	`Dict`	*EOS* output dictionary	no
optimized_structure	`StructureData`	relaxed film structure	no
distance_suggestion	`Dict`	interatomic distance suggestion, output of py:func:*~aiida_fleur.tools.StructureData_util.request_average_bond_length_store()*	no

Similarly to other workchains, FleurCreateMagneticWorkChain behaves differently depending on the input nodes setup. The list of supported input configurations is given in the section *Supported input configurations*.

Workchain parameters and its defaults

`wf_parameters`

`wf_parameters`: `Dict` - Settings of the workflow behavior. All possible keys and their defaults are listed below:

```
# -*- coding: utf-8 -*-
'lattice': 'fcc',                  # type of the substrate lattice: 'bcc' or 'fcc'
'miller': [[-1, 1, 0],             # miller indices to orient the lattice
           [0, 0, 1],
           [1, 1, 0]],
'host_symbol': 'Pt',               # chemical element of the substrate
'latticeconstant': 4.0,            # initial guess for the substrate lattice constant
'size': (1, 1, 5),                 # sets the size of the film unit cell for relax␣
↪step
'replacements': {-2: 'Fe',         # sets the layer number to be replaced by another␣
↪element
                 -1: 'Fe'},        # NOTE: negative number means that replacement␣
↪will take place
                                   #       on layers on the other side from the held␣
↪layers

'decimals': 10,                    # set the accuracy of writing atom positions
'pop_last_layers': 1,              # number of bottom layers to be removed before␣
↪relaxation
'hold_n_first_layers': 5,          # number of bottom layers to be held during the␣
↪relaxation
                                   # (relaxXYZ = 'FFF')

'total_number_layers': 4,          # use this total number of layers
'num_relaxed_layers': 2,           # use this number of relaxed interlayer distances
```

Output nodes

- `magnetic_structure`: `StructureData`- the relaxed film structure.

Supported input configurations

CreateMagnetic workchain has several input combinations that implicitly define the workchain layout. **eos**, **relax**, **optimized_structure** and **eos_output** are analysed. Depending on the given setup of the inputs, one of four supported scenarios will happen:

1. **eos + relax + distance_suggestion**:

   The EOS will be used to calculate the equilibrium structure of the substrate, then Relax WC will be used to relax the interlayer distances. Finally, the non-symmetrical magnetic structure will be created. A good choice if there is nothing to begin with. **distance_suggestion** will be used to guess a better starting interlayer distances before submitting Relax WC.

2. **eos_output + relax + distance_suggestion**:

   The equilibrium substrate structure will be extracted from the **eos_output**, then Relax WC will be used to relax the interlayer distances. Finally, the non-symmetrical magnetic structure will be created. A good choice if EOS was previously done for the substrate. **distance_suggestion** will be used to guess a better starting interlayer distances before submitting Relax WC.

3. **optimized_structure**:

   optimized_structure will be treated as a result of Relax WC and directly used to construct the final non-symmetrical magnetic structure. A good choice if everything was done except the very last step.

4. **relax**:

   Relax WC will be submitted using inputs of the namespace, which means one can for instance continue a relaxation procedure. After Relax WC is finished, the non-symmetrical magnetic structure will be created. A good choice if something wrong happened in one of the relaxation steps of another CreateMagnetic workchain submission.

All the other input configuration will end up with an exit code 231, protecting user from misunderstanding.

Error handling

A list of implemented *exit codes*:

Code	Meaning
230	Invalid workchain parameters
231	Invalid input configuration
380	Specified substrate is not bcc or fcc, only them are supported
382	Relaxation calculation failed.
383	EOS WorkChain failed.

Example usage

```
# -*- coding: utf-8 -*-
from aiida.orm import load_node, Dict
from aiida.engine import submit

from aiida_fleur.workflows.create_magnetic_film import␣
↪FleurCreateMagneticWorkChain

fleur_code = load_node(FLEUR_PK)
inpgen_code = load_node(INPGEN_PK)

wf_para = {
    'lattice': 'fcc',
    'miller': [[-1, 1, 0],
               [0, 0, 1],
               [1, 1, 0]],
    'host_symbol': 'Pt',
    'latticeconstant': 4.0,
    'size': (1, 1, 5),
    'replacements': {0: 'Fe', -1: 'Fe'},
    'decimals': 10,
    'pop_last_layers': 1,

    'total_number_layers': 8,
    'num_relaxed_layers': 3,
}

wf_para = Dict(dict=wf_para)

wf_eos = {'points': 15,
          'step': 0.015,
          'guess': 1.00
          }

wf_eos_scf = {'fleur_runmax': 4,
              'density_converged': 0.0002,
              'serial': False,
              'itmax_per_run': 50,
              'inpxml_changes': []
              }

wf_eos_scf = Dict(dict=wf_eos_scf)

wf_eos = Dict(dict=wf_eos)

calc_eos = {'comp': {'kmax': 3.8,
                     },
            'kpt': {'div1': 4,
                    'div2': 4,
                    'div3': 4
                    }
            }

calc_eos = Dict(dict=calc_eos)

options_eos = {'resources': {'num_machines': 1, 'num_mpiprocs_per_machine':␣
↪4, 'num_cores_per_mpiproc': 6},
               'queue_name': 'devel',
```

(continues on next page)

(continued from previous page)

```
               'custom_scheduler_commands': '',
               'max_wallclock_seconds':  1*60*60}

options_eos = Dict(dict=options_eos)

wf_relax = {'film_distance_relaxation': False,
            'force_criterion': 0.049,
            'relax_iter': 5
            }

wf_relax_scf = {'fleur_runmax': 5,
                'serial': False,
                'use_relax_xml': True,
                'itmax_per_run': 50,
                'alpha_mix': 0.015,
                'relax_iter': 25,
                'force_converged': 0.001,
                'force_dict': {'qfix': 2,
                               'forcealpha': 0.75,
                               'forcemix': 'straight'},
                'inpxml_changes': []
                }

wf_relax = Dict(dict=wf_relax)
wf_relax_scf = Dict(dict=wf_relax_scf)

calc_relax = {'comp': {'kmax': 4.0,
                       },
              'kpt': {'div1': 24,
                      'div2': 20,
                      'div3': 1
                      },
              'atom': {'element': 'Pt',
                       'rmt': 2.2,
                       'lmax': 10,
                       'lnonsph': 6,
                       'econfig': '[Kr] 5s2 4d10 4f14 5p6| 5d9 6s1',
                       },
              'atom2': {'element': 'Fe',
                        'rmt': 2.1,
                        'lmax': 10,
                        'lnonsph': 6,
                        'econfig': '[Ne] 3s2 3p6| 3d6 4s2',
                        },
              }

calc_relax = Dict(dict=calc_relax)

options_relax = {'resources': {'num_machines': 1, 'num_mpiprocs_per_machine
↪': 4, 'num_cores_per_mpiproc': 6},
                 'queue_name': 'devel',
                 'custom_scheduler_commands': '',
                 'max_wallclock_seconds':  1*60*60}

inputs = {
    'eos': {
        'scf': {
```

(continues on next page)

(continued from previous page)

```
            'wf_parameters': wf_eos_scf,
            'calc_parameters': calc_eos,
            'options': options_eos,
            'inpgen': inpgen_code,
            'fleur': fleur_code
        },
        'wf_parameters': wf_eos
    },
    'relax': {
        'scf': {
            'wf_parameters': wf_relax_scf,
            'calc_parameters': calc_relax,
            'options': options_relax,
            'inpgen': inpgen_code,
            'fleur': fleur_code
        },
        'wf_parameters': wf_relax,
    },
    'wf_parameters': wf_para
}

res = submit(FleurCreateMagneticWorkChain, **inputs)
```

Magnetic workchains

Force-theorem subgroup

Fleur Spin-Spiral Dispersion workchain

- **Current version**: 0.2.0
- **Class**: *FleurSSDispWorkChain*
- **String to pass to the** `WorkflowFactory()`: `fleur.ssdisp`
- **Workflow type**: Scientific workchain, force-theorem subgroup

Contents

Import Example:

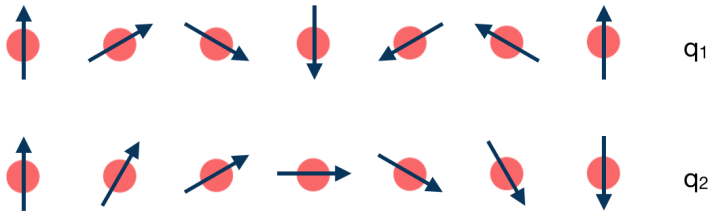
```
from aiida_fleur.workflows.ssdisp_conv import FleurSSDispWorkChain
#or
WorkflowFactory('fleur.ssdisp')
```

Description/Purpose

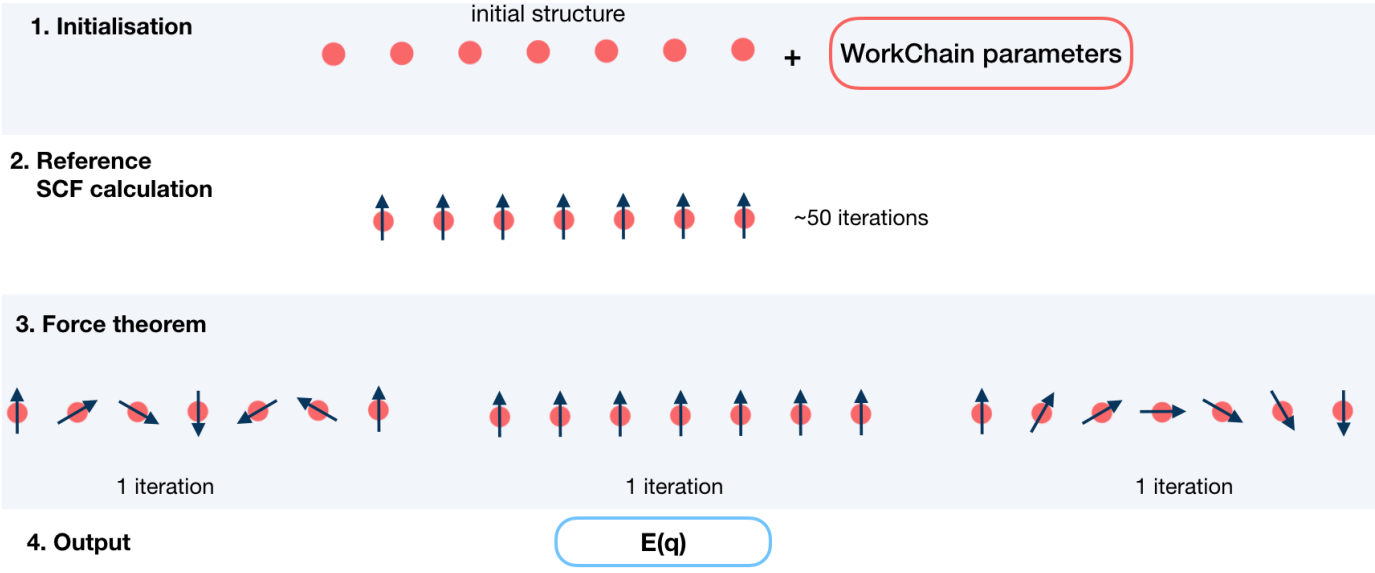
This workchain calculates spin spiral energy dispersion over a given set of q-points. Resulting energies do not contain terms, corresponding to DMI energies. To take into account DMI, see the *Fleur Dzyaloshinskii–Moriya Interaction energy workchain* documentation.

In this workchain the force-theorem is employed which means the workchain converges a reference charge density first and then submits a single FleurCalculation with a `<forceTheorem>` tag. However, it is possible to specify inputs to use external pre-converged charge density to use it as a reference.

The task of the workchain us to calculate the energy difference between two or several structures having a different magnetisation profile:



To do this, the workchain employs the force theorem approach:



As it was mentioned, it is not always necessary to start with a structure. Setting up input parameters correctly (see *Supported input configurations*) one can start from a given FleuinpData, inp.xml or converged/not-fully-converged reference charge density.

Input nodes

The FleurSSDispWorkChain employs exposed feature of the AiiDA, thus inputs for the nested *SCF* workchain should be passed in the namespace `scf`.

name	type	description	required
scf	namespace	inputs for nested SCF WorkChain	no
fleur	`Code`	Fleur code	yes
wf_parameters	`Dict`	Settings of the workchain	no
fleurinp	*`FleurinpData`*	*FLEUR input*	no
remote	`RemoteData`	Remote folder of another calculation	no
options	`Dict`	AiiDA options (computational resources)	no

Only **fleur** input is required. However, it does not mean that it is enough to specify **fleur** only. One *must* keep one of the supported input configurations described in the *Supported input configurations* section.

Workchain parameters and its defaults

`wf_parameters`

`wf_parameters`: `Dict` - Settings of the workflow behavior. All possible keys and their defaults are listed below:

```
# -*- coding: utf-8 -*-
'beta': {'all': 1.57079},           # see description below
'prop_dir': [1.0, 0.0, 0.0],        # sets a propagation direction of a q-vector
'q_vectors': [[0.0, 0.0, 0.0],      # set a set of q-vectors to calculate SSDispersion
              [0.125, 0.0, 0.0],
              [0.250, 0.0, 0.0],
              [0.375, 0.0, 0.0]],
'ref_qss': [0.0, 0.0, 0.0],         # sets a q-vector for the reference calculation
'inpxml_changes': []                # additional changes before the FT step
'serial': False                     # False if use MPI version for the FT calc
'only_even_MPI': False,             # True if suppress parallelisation having odd␣
↪number of MPI
```

beta is a python dictionary containing a `key`: `value` pairs. Each pair sets **beta** parameter in an inp.xml file. `key` specifies the atom label to change, `key` equal to *'all'* sets all atoms groups. For example,

```
'beta' : {'222' : 1.57079}
```

changes

```
<atomGroup species="Fe-1">
  <filmPos label="                222">.0000000000 .0000000000 -11.4075100502</
↪filmPos>
  <force calculate="T" relaxXYZ="TTT"/>
  <nocoParams l_relax="F" alpha=".00000000" beta="0.00000" b_cons_x=".00000000" b_
↪cons_y=".00000000"/>
</atomGroup>
```

to:

```
<atomGroup species="Fe-1">
  <filmPos label="                  222">.0000000000 .0000000000 -11.4075100502</
↪filmPos>
  <force calculate="T" relaxXYZ="TTT"/>
  <nocoParams l_relax="F" alpha=".00000000" beta="1.57079" b_cons_x=".00000000" b_
↪cons_y=".00000000"/>
</atomGroup>
```

Note: **beta** actually sets a beta parameter for a whole atomGroup. It can be that the atomGroup correspond to several atoms and **beta** switches sets beta for atoms that was not intended to change. You must be careful and make sure that several atoms do not correspond to a given specie.

prop_dir is used only to set up a spin spiral propagation direction to `calc_parameters['qss']` which will be passed to SCF workchain and inpgen. It can be used to properly set up symmetry operations in the reference calculation.

options

`options`: `Dict` - AiiDA options (computational resources). Example:

```
'resources': {"num_machines": 1, "num_mpiprocs_per_machine": 1},
'max_wallclock_seconds': 6*60*60,
'queue_name': '',
'custom_scheduler_commands': '',
'import_sys_environment': False,
'environment_variables': {}
```

Output nodes

- `out`: `Dict` - Information of workflow results like success, last result node, list with convergence behavior

```
"energies": [
    0.0,
    0.00044082445345511,
],
"energy_units": "eV",
"errors": [],
"info": [],
"initial_structure": "a75459e5-f83e-4aff-a25d-595d938cb841",
"is_it_force_theorem": true,
"q_vectors": [
    [
        0.0,
        0.0,
        0.0
    ],
    [
        0.125,
        0.125,
        0.0
    ],
],
```

(continues on next page)

(continued from previous page)

```
"warnings": [],
"workflow_name": "FleurSSDispWorkChain",
"workflow_version": "0.1.0"
```

Resulting Spin Spiral energies are sorted according to theirs q-vectors i.e. `energies[N]` corresponds to `q_vectors[N]`.

Supported input configurations

SSDisp workchain has several input combinations that implicitly define the workchain layout. Only **scf**, **fleurinp** and **remote** nodes control the behaviour, other input nodes are truly optional. Depending on the setup of the given inputs, one of three supported scenarios will happen:

1. **scf**:

   SCF workchain will be submitted to converge the reference charge density which will be followed be the force theorem calculation. Depending on the inputs given in the SCF namespace, SCF will start from the structure or FleurinpData or will continue converging from the given remote_data (see details in *SCF WorkChain*).

2. **remote**:

   Files which belong to the **remote** will be used for the direct submission of the force theorem calculation. `inp.xml` file will be converted to FleurinpData and charge density will be used as a reference charge density.

3. **remote** + **fleurinp**:

   Charge density which belongs to **remote** will be used as a reference charge density, however `inp.xml` from the **remote** will be ignored. Instead, the given **fleurinp** will be used. The aforementioned input files will be used for direct submission of the force theorem calculation.

Other combinations of the input nodes **scf**, **fleurinp** and **remote** are forbidden.

Warning: One *must* follow one of the supported input configurations. To protect a user from the workchain misbehaviour, an error will be thrown if one specifies e.g. both **scf** and **remote** inputs because in this case the intention of the user is not clear either he/she wants to converge a new charge density or use the given one.

Error handling

A list of implemented *exit codes*:

Code	Meaning
230	Invalid workchain parameters
231	Invalid input configuration
233	Input codes do not correspond to fleur or inpgen codes respectively.
235	Input file modification failed.
236	Input file was corrupted after modifications
334	Reference calculation failed.
335	Found no reference calculation remote repository.
336	Force theorem calculation failed.

Example usage

```
# -*- coding: utf-8 -*-
from aiida.orm import load_node, Dict
from aiida.engine import submit

from aiida_fleur.workflows.ssdisp import FleurSSDispWorkChain

structure = load_node(STRUCTURE_PK)
fleur_code = load_node(FLEUR_PK)
inpgen_code = load_node(INPGEN_PK)

wf_para = Dict(dict={'beta': {'all': 1.57079},
                     'prop_dir': [0.125, 0.125, 0.0],
                     'q_vectors': [[0.0, 0.0, 0.0],
                                   [0.125, 0.125, 0.0],
                                   [0.250, 0.250, 0.0],
                                   [0.375, 0.375, 0.0],
                                   [0.500, 0.500, 0.0]],
                     'ref_qss': [0.0, 0.0, 0.0],
                     'inpxml_changes': [],
                     'serial': False,
                     'only_even_MPI': False
                     })

options = Dict(dict={'resources': {'num_machines': 1, 'num_mpiprocs_per_
↪machine': 24},
                     'queue_name': 'devel',
                     'custom_scheduler_commands': '',
                     'max_wallclock_seconds':  60*60})


parameters = Dict(dict={'atom': {'element': 'Pt',
                                 'lmax': 8
                                 },
                        'atom2': {'element': 'Fe',
                                  'lmax': 8,
                                  },
                        'comp': {'kmax': 3.8,
                                 },
                        'kpt': {'div1': 20,
                                'div2': 24,
                                'div3': 1
                                }})

wf_para_scf = {'fleur_runmax': 2,
               'itmax_per_run': 120,
               'density_converged': 0.2,
               'serial': False,
               'mode': 'density'
               }

wf_para_scf = Dict(dict=wf_para_scf)

options_scf = Dict(dict={'resources': {'num_machines': 2, 'num_mpiprocs_per_
↪machine': 24},
                         'queue_name': 'devel',
```

(continues on next page)

(continued from previous page)

```
                        'custom_scheduler_commands': '',
                        'max_wallclock_seconds':  60*60})

inputs = {'scf': {'wf_parameters': wf_para_scf,
                  'structure': structure,
                  'calc_parameters': parameters,
                  'options': options_scf,
                  'inpgen': inpgen_code,
                  'fleur': fleur_code
                  },
          'wf_parameters': wf_para,
          'fleur': fleur_code,
          'options': options
          }


res = submit(FleurSSDispWorkChain, **inputs)
```

Fleur Dzyaloshinskii–Moriya Interaction energy workchain

- **Current version**: 0.2.0
- **Class**: *FleurDMIWorkChain*
- **String to pass to the** `WorkflowFactory()`: `fleur.dmi`
- **Workflow type**: Scientific workchain, force theorem sub-group

Contents

Import Example:

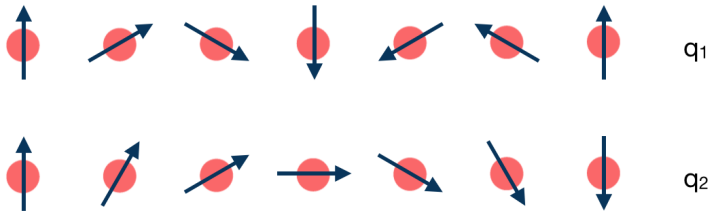
```
from aiida_fleur.workflows.dmi import FleurDMIWorkChain
#or
WorkflowFactory('fleur.dmi')
```

Description/Purpose

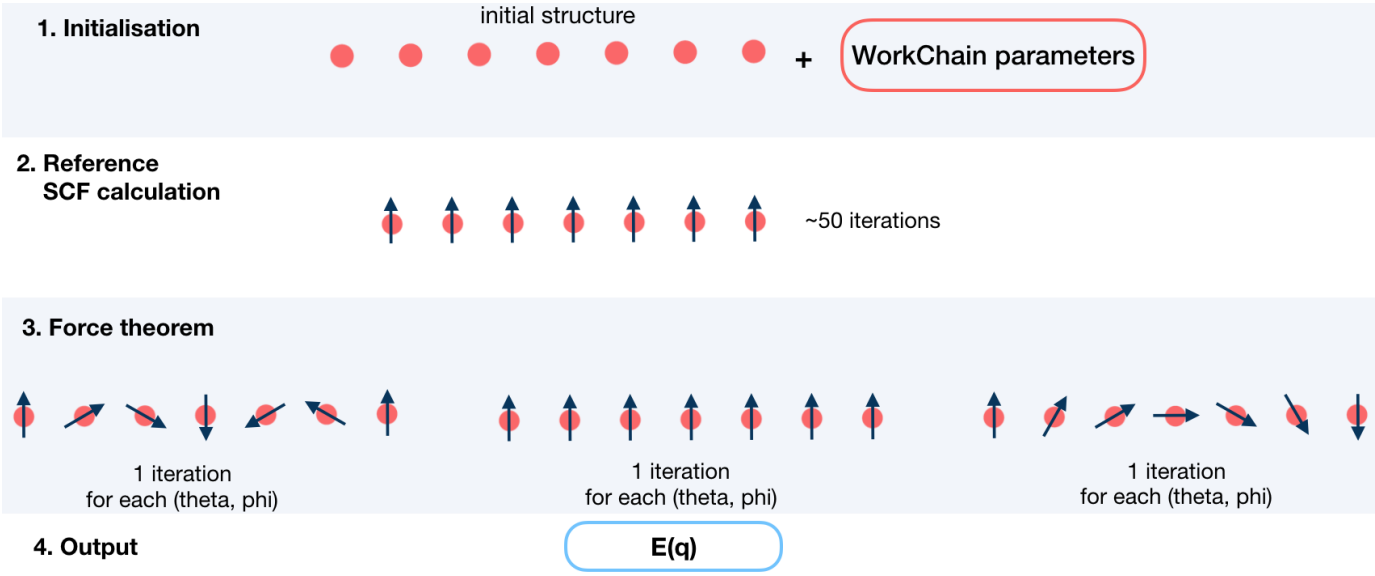
This workchain calculates Dzyaloshinskii–Moriya Interaction energy over a given set of q-points.

In this workchain the force-theorem is employed which means the workchain converges a reference charge density first then it submits a single FleurCalculation with a `<forceTheorem>` tag. However, it is possible to specify inputs to use external pre-converged charge density and use is as a reference.

The task of the workchain us to calculate the energy difference between two or several structures having a different magnetisation profile (theta and phi values can also vary):



To do this, the workchain employs the force theorem approach:



It is not always necessary to start with a structure. Setting up input parameters correctly (see *Supported input configurations*) one can start from a given FleuinpData, inp.xml or converged/not-fully-converged reference charge density.

Input nodes

The FleurSSDispWorkChain employs exposed feature of the AiiDA, thus inputs for the nested *SCF* workchain should be passed in the namespace `scf`.

name	type	description	required
scf	namespace	inputs for nested SCF WorkChain	no
fleur	`Code`	Fleur code	yes
wf_parameters	`Dict`	Settings of the workchain	no
fleurinp	*`FleurinpData`*	*FLEUR input*	no
remote	`RemoteData`	Remote folder of another calculation	no
options	`Dict`	AiiDA options (computational resources)	no

Only **fleur** input is required. However, it does not mean that it is enough to specify **fleur** only. One *must* keep one of the supported input configurations described in the *Supported input configurations* section.

Workchain parameters and its defaults

wf_parameters

`wf_parameters`: `Dict` - Settings of the workflow behavior. All possible keys and their defaults are listed below:

```
# -*- coding: utf-8 -*-
'beta': {'all': 1.57079},                # see description below
'sqas_theta': [0.0, 1.57079, 1.57079],   # a list of theta values for the FT
'sqas_phi': [0.0, 0.0, 1.57079],         # a list of phi values for the FT
'soc_off': [],                           # a list of atom labels to switch off SOC term
'q_vectors': [[0.0, 0.0, 0.0],           # set a set of q-vectors to calculate DMI␣
↪dispersion
             [0.1, 0.1, 0.0]]
'serial': False,                         # False if use MPI version for the FT calc
'only_even_MPI': False,                  # True if suppress parallelisation having odd␣
↪number of MPI
'ref_qss': [0.0, 0.0, 0.0],              # sets a q-vector for the reference␣
↪calculation
'inpxml_changes': [],                    # additional changes before the FT step
```

beta is a python dictionary containing a `key : value` pairs. Each pair sets **beta** parameter in an inp.xml file. `key` specifies the atom label to change, `key` equal to *'all'* sets all atoms groups. For example,

```
'beta' : {'222' : 1.57079}
```

changes

```
<atomGroup species="Fe-1">
  <filmPos label="                222">.0000000000 .0000000000 -11.4075100502</
↪filmPos>
  <force calculate="T" relaxXYZ="TTT"/>
  <nocoParams l_relax="F" alpha=".00000000" beta="0.00000" b_cons_x=".00000000" b_
↪cons_y=".00000000"/>
</atomGroup>
```

to:

```
<atomGroup species="Fe-1">
  <filmPos label="                222">.0000000000 .0000000000 -11.4075100502</
↪filmPos>
  <force calculate="T" relaxXYZ="TTT"/>
  <nocoParams l_relax="F" alpha=".00000000" beta="1.57079" b_cons_x=".00000000" b_
↪cons_y=".00000000"/>
</atomGroup>
```

Note: **beta** actually sets a beta parameter for a whole atomGroup. It can be that the atomGroup correspond to several atoms and **beta** switches sets beta for atoms that was not intended to change. You must be careful and make sure that several atoms do not correspond to a given specie.

---

soc_off is a python list containing atoms labels. SOC is switched off for species, corresponding to the atom with a given label.

Note: It can be that the spice correspond to several atoms and **soc_off** switches off SOC for atoms that was not

intended to change. You must be careful and make sure that several atoms do not correspond to a given specie.

An example of **soc_off** work:

```
'soc_off': ['458']
```

changes

```
<species name="Ir-2" element="Ir" atomicNumber="77" coreStates="17" magMom=".00000000
↪" flipSpin="T">
  <mtSphere radius="2.52000000" gridPoints="747" logIncrement=".01800000"/>
  <atomicCutoffs lmax="8" lnonsphr="6"/>
  <energyParameters s="6" p="6" d="5" f="5"/>
  <prodBasis lcutm="4" lcutwf="8" select="4 0 4 2"/>
  <lo type="SCLO" l="1" n="5" eDeriv="0"/>
</species>
-----
<atomGroup species="Ir-2">
  <filmPos label="                458">1.000/4.000 1.000/2.000 11.4074000502</
↪filmPos>
  <force calculate="T" relaxXYZ="TTT"/>
  <nocoParams l_relax="F" alpha=".00000000" beta=".00000000" b_cons_x=".00000000" b_
↪cons_y=".00000000"/>
</atomGroup>
```

to:

```
<species name="Ir-2" element="Ir" atomicNumber="77" coreStates="17" magMom=".00000000
↪" flipSpin="T">
  <mtSphere radius="2.52000000" gridPoints="747" logIncrement=".01800000"/>
  <atomicCutoffs lmax="8" lnonsphr="6"/>
  <energyParameters s="6" p="6" d="5" f="5"/>
  <prodBasis lcutm="4" lcutwf="8" select="4 0 4 2"/>
  <special socscale="0.0"/>
  <lo type="SCLO" l="1" n="5" eDeriv="0"/>
</species>
```

As you can see, I was careful about "Ir-2" specie and it contained a single atom with a label 458. Please also refer to *Setting up atom labels* section to learn how to set labels up.

sqas_theta and **sqas_phi** are python lists that set SOC theta and phi values.

prop_dir is used only to set up a spin spiral propagation direction to `calc_parameters['qss']` which will be passed to SCF workchain and inpgen. It can be used to properly set up symmetry operations in the reference calculation.

options

`options`: `Dict` - AiiDA options (computational resources). Example:

```
'resources': {"num_machines": 1, "num_mpiprocs_per_machine": 1},
'max_wallclock_seconds': 6*60*60,
'queue_name': '',
'custom_scheduler_commands': '',
'import_sys_environment': False,
'environment_variables': {}
```

Output nodes

- `out`: `Dict` - Information of workflow results like success, last result node, list with convergence behavior


```
"angles": 3,
"energies": [
    0.0
],
"energy_units": "eV",
"errors": [],
"info": [],
"initial_structure": "35e5058d-161c-4cf9-801e-4eca99e7d7be",
"phi": [
    3.1415927,
],
"q_vectors": [
    [
        0.0,
        0.0,
        0.0
    ],
],
"theta": [
    0.0,
],
"warnings": [],
"workflow_name": "FleurDMIWorkChain",
"workflow_version": "0.1.0"
```


Resulting DMI energies are sorted according to theirs q-vector, theta and phi values i.e. `energies[N]` corresponds to `q_vectors[N]`, `phi[N]` and `theta[N]`.

Supported input configurations

DMI workchain has several input combinations that implicitly define the workchain layout. Only **scf**, **fleurinp** and **remote** nodes control the behaviour, other input nodes are truly optional. Depending on the setup of the inputs, one of several supported scenarios will happen:

1. **scf**:

   SCF workchain will be submitted to converge the reference charge density which will be followed be the force theorem calculation. Depending on the inputs given in the SCF namespace, SCF will start from the structure or FleurinpData or will continue converging from the given remote_data (see details in *SCF WorkChain*).

2. **remote**:

   Files which belong to the **remote** will be used for the direct submission of the force theorem calculation. `inp.xml` file will be converted to FleurinpData and charge density will be used as a reference charge density.

3. **remote** + **fleurinp**:

   Charge density which belongs to **remote** will be used as a reference charge density, however `inp.xml` from the **remote** will be ignored. Instead, the given **fleurinp** will be used. The aforementioned input files will be used for direct submission of the force theorem calculation.

Other combinations of the input nodes **scf**, **fleurinp** and **remote** are forbidden.

Warning: One *must* follow one of the supported input configurations. To protect a user from the workchain misbehaviour, an error will be thrown if one specifies e.g. both **scf** and **remote** inputs because in this case the intention of the user is not clear either he/she wants to converge a new charge density or use the given one.

Error handling

A list of implemented *exit codes*:

Code	Meaning
230	Invalid workchain parameters
231	Invalid input configuration
233	Input codes do not correspond to fleur or inpgen codes respectively.
235	Input file modification failed.
236	Input file was corrupted after modifications
334	Reference calculation failed.
335	Found no reference calculation remote repository.
336	Force theorem calculation failed.

Example usage

```
# -*- coding: utf-8 -*-
from aiida.orm import load_node, Dict
from aiida.engine import submit

from aiida_fleur.workflows.dmi import FleurDMIWorkChain

structure = load_node(STRUCTURE_PK)
fleur_code = load_node(FLEUR_PK)
inpgen_code = load_node(INPGEN_PK)

wf_para = Dict(dict={'serial': False,
                     'only_even_MPI': False,
                     'beta': {'all': 1.57079},
                     'sqas_theta': [0.0, 1.57079, 1.57079],
                     'sqas_phi': [0.0, 0.0, 1.57079],
                     'soc_off': [],
                     'q_vectors': [[0.0, 0.0, 0.0],
                                   [0.1, 0.1, 0.0]],
                     'ref_qss': [0.0, 0.0, 0.0],
                     'inpxml_changes': []
                     })

options = Dict(dict={'resources': {'num_machines': 1, 'num_mpiprocs_per_
↪machine': 24},
                     'queue_name': 'devel',
                     'custom_scheduler_commands': '',
                     'max_wallclock_seconds':  60*60})


parameters = Dict(dict={'atom': {'element': 'Pt',
                                 'lmax': 8
```

(continues on next page)

(continued from previous page)

```
                                },
                          'atom2': {'element': 'Fe',
                                    'lmax': 8,
                                    },
                          'comp': {'kmax': 3.8,
                                   },
                          'kpt': {'div1': 20,
                                  'div2': 24,
                                  'div3': 1
                                  }})

wf_para_scf = {'fleur_runmax': 2,
               'itmax_per_run': 120,
               'density_converged': 0.2,
               'serial': False,
               'mode': 'density'
               }

wf_para_scf = Dict(dict=wf_para_scf)

options_scf = Dict(dict={'resources': {'num_machines': 2, 'num_mpiprocs_per_
↪machine': 24},
                         'queue_name': 'devel',
                         'custom_scheduler_commands': '',
                         'max_wallclock_seconds':  60*60})

inputs = {'scf': {'wf_parameters': wf_para_scf,
                  'structure': structure,
                  'calc_parameters': parameters,
                  'options': options_scf,
                  'inpgen': inpgen_code,
                  'fleur': fleur_code
                  },
          'wf_parameters': wf_para,
          'fleur': fleur_code,
          'options': options
          }


res = submit(FleurDMIWorkChain, **inputs)
```

Fleur Magnetic Anisotropy Energy workflow

- **Current version**: 0.2.0
- **Class**: *FleurMaeWorkChain*
- **String to pass to the** `WorkflowFactory()`: `fleur.mae`
- **Workflow type**: Scientific workchain, force-theorem subgroup
- **Aim**: Calculate Magnetic Anisotropy Energies along given spin quantization axes

Contents

Import Example:

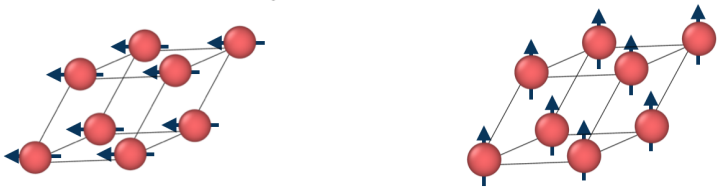
```
from aiida_fleur.workflows.mae import FleurMaeWorkChain
#or
WorkflowFactory('fleur.mae')
```

Description/Purpose

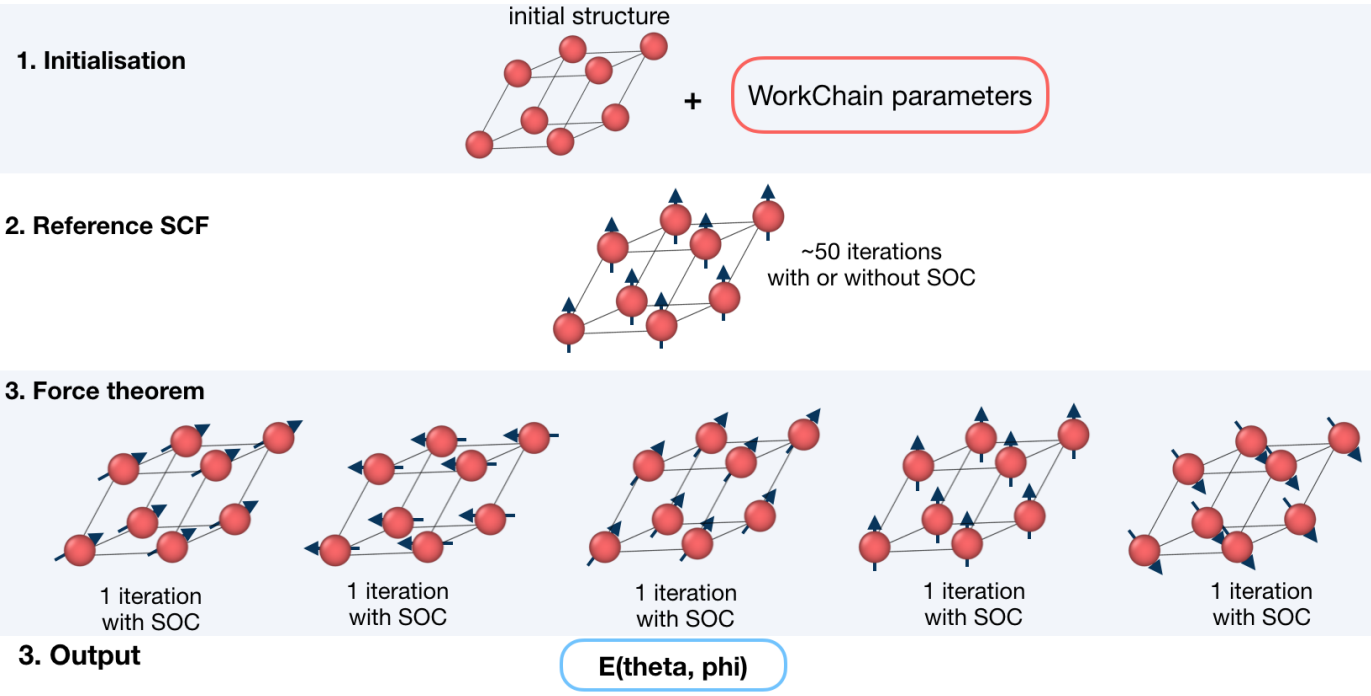
This workchain calculates Magnetic Anisotropy Energy over a given set of spin-quantization axes. The force-theorem is employed which means the workchain converges a reference charge density first then it submits a single FleurCalculation with a <*forceTheorem*> tag.

The task of the workchain us to calculate the energy difference between two or several structures having a different magnetisation profile:

Energies are not the same for:



To do this, the workchain employs the force theorem approach:



It is not always necessary to start with a structure. Setting up input parameters correctly (see *Supported input configurations*) one can start from a given FleuinpData, inp.xml or converged/not-fully-converged reference charge density.

Input nodes

The FleurMaeWorkChain employs exposed feature of the AiiDA, thus inputs for the nested *SCF* workchain should be passed in the namespace `scf`.

name	type	description	required
scf	namespace	inputs for nested SCF WorkChain	no
fleur	`Code`	Fleur code	yes
wf_parameters	`Dict`	Settings of the workchain	no
fleurinp	*`FleurinpData`*	*FLEUR input*	no
remote	`RemoteData`	Remote folder of another calculation	no
options	`Dict`	AiiDA options (computational resources)	no

Only **fleur** input is required. However, it does not mean that it is enough to specify **fleur** only. One *must* keep one of the supported input configurations described in the *Supported input configurations* section.

Workchain parameters and its defaults

`wf_parameters`

`wf_parameters`: `Dict` - Settings of the workflow behavior. All possible keys and their defaults are listed below:

```
# -*- coding: utf-8 -*-
'sqa_ref': [0.7, 0.7],                  # sets theta and phi for the reference calc
'use_soc_ref': False,                   # True if reference calc should use SOC terms
```

(continues on next page)

(continued from previous page)

```
'sqas_theta': [0.0, 1.57079, 1.57079],  # a list of theta values for the FT
'sqas_phi': [0.0, 0.0, 1.57079],        # a list of phi values for the FT
'serial': False,                        # False if use MPI version for the FT calc
'only_even_MPI': False,                 # True if suppress parallelisation having odd␣
↪number of MPI
'soc_off': [],                          # a list of atom labels to switch off SOC term
'inpxml_changes': []                    # additional changes before the FT step
```

soc_off is a python list containing atoms labels. SOC is switched off for species, corresponding to the atom with a given label.

Note: It can be that the specie correspond to several atoms and **soc_off** switches off SOC for atoms that was not intended to change. You must be careful and make sure that several atoms do not correspond to a given specie.

An example of **soc_off** work:

```
'soc_off': ['458']
```

changes

```
<species name="Ir-2" element="Ir" atomicNumber="77" coreStates="17" magMom=".00000000
↪" flipSpin="T">
  <mtSphere radius="2.52000000" gridPoints="747" logIncrement=".01800000"/>
  <atomicCutoffs lmax="8" lnonsphr="6"/>
  <energyParameters s="6" p="6" d="5" f="5"/>
  <prodBasis lcutm="4" lcutwf="8" select="4 0 4 2"/>
  <lo type="SCLO" l="1" n="5" eDeriv="0"/>
</species>
-----
<atomGroup species="Ir-2">
  <filmPos label="                458">1.000/4.000 1.000/2.000 11.4074000502</
↪filmPos>
  <force calculate="T" relaxXYZ="TTT"/>
  <nocoParams l_relax="F" alpha=".00000000" beta=".00000000" b_cons_x=".00000000" b_
↪cons_y=".00000000"/>
</atomGroup>
```

to:

```
<species name="Ir-2" element="Ir" atomicNumber="77" coreStates="17" magMom=".00000000
↪" flipSpin="T">
  <mtSphere radius="2.52000000" gridPoints="747" logIncrement=".01800000"/>
  <atomicCutoffs lmax="8" lnonsphr="6"/>
  <energyParameters s="6" p="6" d="5" f="5"/>
  <prodBasis lcutm="4" lcutwf="8" select="4 0 4 2"/>
  <special socscale="0.0"/>
  <lo type="SCLO" l="1" n="5" eDeriv="0"/>
</species>
```

As you can see, I was careful about "Ir-2" specie and it contained a single atom with a label 458. Please also refer to *Setting up atom labels* section to learn how to set labels up.

sqas_theta and **sqas_phi** are python lists that set SOC theta and phi values.

sqa_ref sets a spin quantization axis [theta, phi] for the reference calculation if SOC terms are switched on by **use_soc_ref**.

options

`options`: `Dict` - AiiDA options (computational resources). Example:

```
'resources': {"num_machines": 1, "num_mpiprocs_per_machine": 1},
'max_wallclock_seconds': 6*60*60,
'queue_name': '',
'custom_scheduler_commands': '',
'import_sys_environment': False,
'environment_variables': {}
```

Output nodes

- `out`: `Dict` - Information of workflow results like success, last result node, list with convergence behavior

```
"errors": [],
"info": [],
"initial_structure": "ac274613-27f5-4c0b-9d42-bae340007ab1",
"is_it_force_theorem": true,
"mae_units": "eV",
"maes": [
    0.0006585155416697,
    0.0048545112659747,
    0.0
],
"phi": [
    0.0,
    0.0,
    1.57079
],
"theta": [
    0.0,
    1.57079,
    1.57079
],
"warnings": [],
"workflow_name": "FleurMaeWorkChain",
"workflow_version": "0.1.0"
```

Resulting Magnetic Anisotropy Directions are sorted according to theirs theta and phi values i.e. `maes[N]` corresponds to `theta[N]` and `phi[N]`.

Supported input configurations

MAE workchain has several input combinations that implicitly define the workchain layout. Only **scf**, **fleurinp** and **remote** nodes control the behaviour, other input nodes are truly optional. Depending on the setup of the inputs, one of several supported scenarios will happen:

1. **scf**:

   SCF workchain will be submitted to converge the reference charge density which will be followed be the force theorem calculation. Depending on the inputs given in the SCF namespace, SCF will start from the structure or FleurinpData or will continue converging from the given remote_data (see details in *SCF WorkChain*).

2. **remote**:

   Files which belong to the **remote** will be used for the direct submission of the force theorem calculation. `inp.xml` file will be converted to FleurinpData and charge density will be used as a reference charge density.

3. **remote** + **fleurinp**:

   Charge density which belongs to **remote** will be used as a reference charge density, however `inp.xml` from the **remote** will be ignored. Instead, the given **fleurinp** will be used. The aforementioned input files will be used for direct submission of the force theorem calculation.

Other combinations of the input nodes **scf**, **fleurinp** and **remote** are forbidden.

Warning: One *must* follow one of the supported input configurations. To protect a user from the workchain misbehaviour, an error will be thrown if one specifies e.g. both **scf** and **remote** inputs because in this case the intention of the user is not clear either he/she wants to converge a new charge density or use the given one.

Error handling

A list of implemented *exit codes*:

Code	Meaning
230	Invalid workchain parameters
231	Invalid input configuration
233	Input codes do not correspond to fleur or inpgen codes respectively.
235	Input file modification failed.
236	Input file was corrupted after modifications
334	Reference calculation failed.
335	Found no reference calculation remote repository.
336	Force theorem calculation failed.

Example usage

```
# -*- coding: utf-8 -*-
from aiida.orm import load_node, Dict
from aiida.engine import submit

from aiida_fleur.workflows.mae import FleurMaeWorkChain

structure = load_node(STRUCTURE_PK)
fleur_code = load_node(FLEUR_PK)
inpgen_code = load_node(INPGEN_PK)

wf_para = Dict(dict={'sqa_ref': [0.7, 0.7],
                     'use_soc_ref': False,
                     'sqas_theta': [0.0, 1.57079, 1.57079],
                     'sqas_phi': [0.0, 0.0, 1.57079],
                     'serial': False,
                     'only_even_MPI': False,
                     'soc_off': [],
                     'inpxml_changes': [],
```

(continues on next page)

(continued from previous page)

```
                    })

options = Dict(dict={'resources': {'num_machines': 1, 'num_mpiprocs_per_
↪machine': 24},
                     'queue_name': 'devel',
                     'custom_scheduler_commands': '',
                     'max_wallclock_seconds':  60*60})


parameters = Dict(dict={'atom': {'element': 'Pt',
                                 'lmax': 8
                                 },
                        'atom2': {'element': 'Fe',
                                  'lmax': 8,
                                  },
                        'comp': {'kmax': 3.8,
                                 },
                        'kpt': {'div1': 20,
                                'div2': 24,
                                'div3': 1
                                }})

wf_para_scf = {'fleur_runmax': 2,
               'itmax_per_run': 120,
               'density_converged': 0.2,
               'serial': False,
               'mode': 'density'
               }

wf_para_scf = Dict(dict=wf_para_scf)

options_scf = Dict(dict={'resources': {'num_machines': 2, 'num_mpiprocs_per_
↪machine': 24},
                         'queue_name': 'devel',
                         'custom_scheduler_commands': '',
                         'max_wallclock_seconds':  60*60})

inputs = {'scf': {'wf_parameters': wf_para_scf,
                  'structure': structure,
                  'calc_parameters': parameters,
                  'options': options_scf,
                  'inpgen': inpgen_code,
                  'fleur': fleur_code
                  },
          'wf_parameters': wf_para,
          'fleur': fleur_code,
          'options': options
          }


res = submit(FleurMaeWorkChain, **inputs)
```

Self-consistent sub-group

Fleur Spin-Spiral Dispersion Converge workchain

- **Current version**: 0.2.0
- **Class**: *FleurSSDispConvWorkChain*
- **String to pass to the** `WorkflowFactory()`: `fleur.ssdisp_conv`
- **Workflow type**: Scientific workchain, self-consistent subgroup
- **Aim**: Calculate spin-spiral energy dispersion over given q-points converging all the q_points.

Contents

Import Example:

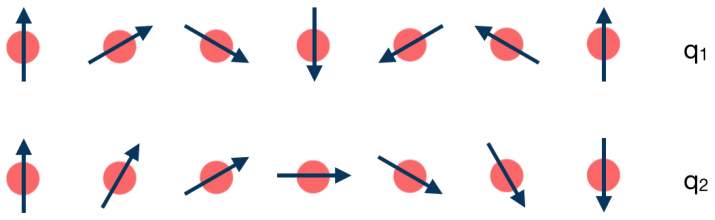
```
from aiida_fleur.workflows.ssdisp_conv import FleurSSDispConvWorkChain
#or
WorkflowFactory('fleur.ssdisp_conv')
```

Description/Purpose

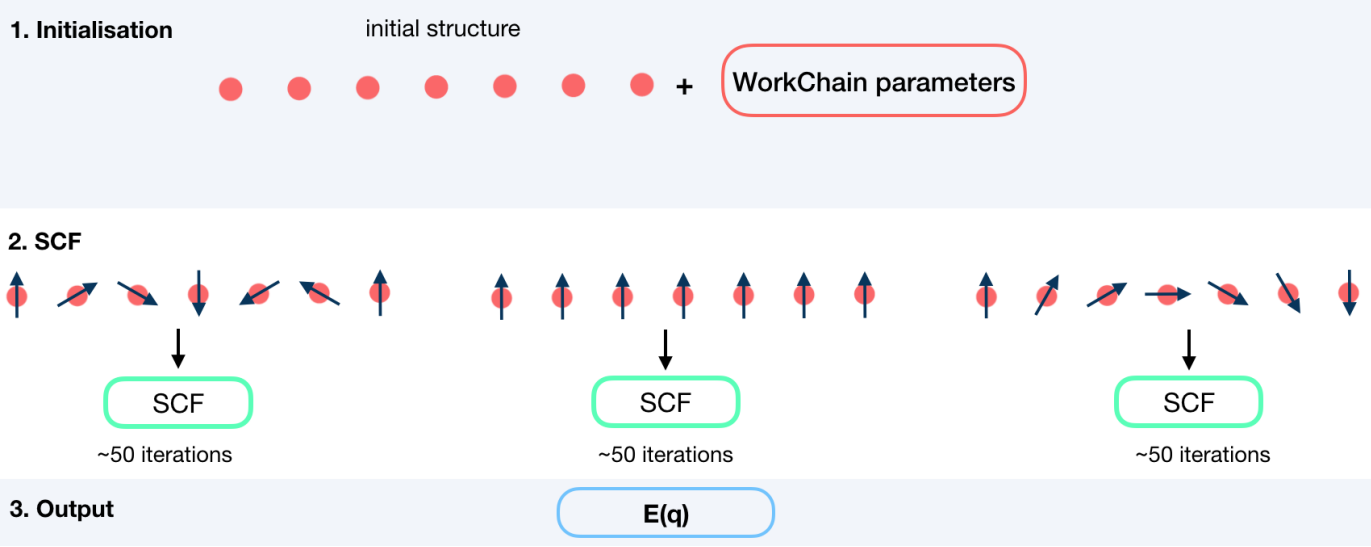
This workchain calculates spin spiral energy dispersion over a given set of q-points. Resulting energies do not contain terms, corresponding to DMI energies. To take into account DMI, see the *Fleur Dzyaloshinskii–Moriya Interaction energy workchain* documentation.

In this workchain the force-theorem is employed which means the workchain converges a reference charge density first and then submits a single FleurCalculation with a `<forceTheorem>` tag. However, it is possible to specify inputs to use external pre-converged charge density to use it as a reference.

The task of the workchain us to calculate the energy difference between two or several structures having a different magnetisation profile:



To do this, the workchain employs the force theorem approach:



Input nodes

The FleurSSDispWorkChain employs exposed feature of the AiiDA, thus inputs for the nested *SCF* workchain should be passed in the namespace `scf`.

name	type	description	required
scf	namespace	inputs for nested SCF WorkChain	yes
wf_parameters	`Dict`	Settings of the workchain	no

Workchain parameters and its defaults

`wf_parameters`

`wf_parameters`: `Dict` - Settings of the workflow behavior. All possible keys and their defaults are listed below:

```
# -*- coding: utf-8 -*-
'beta': {'all': 1.57079},                    # see the description below
'q_vectors': {'label': [0.0, 0.0, 0.0],      # sets q_points to calculate
              'label2': [0.125, 0.0, 0.0]
              }
'suppress_symmetries': False                 # True if use no symmetries
```

beta is a python dictionary containing a `key : value` pairs. Each pair sets **beta** parameter in an inp.xml file. `key` specifies the atom label to change, `key` equal to *'all'* sets all atoms groups. For example,

```
'beta' : {'222' : 1.57079}
```

changes

```
<atomGroup species="Fe-1">
  <filmPos label="                222">.0000000000 .0000000000 -11.4075100502</
↪filmPos>
  <force calculate="T" relaxXYZ="TTT"/>
```

(continues on next page)

(continued from previous page)

```
  <nocoParams l_relax="F" alpha=".00000000" beta="0.00000" b_cons_x=".00000000" b_
↪cons_y=".00000000"/>
</atomGroup>
```

to:

```
<atomGroup species="Fe-1">
  <filmPos label="                222">.0000000000 .0000000000 -11.4075100502</
↪filmPos>
  <force calculate="T" relaxXYZ="TTT"/>
  <nocoParams l_relax="F" alpha=".00000000" beta="1.57079" b_cons_x=".00000000" b_
↪cons_y=".00000000"/>
</atomGroup>
```

Note: **beta** actually sets a beta parameter for a whole atomGroup. It can be that the atomGroup correspond to several atoms and **beta** switches sets beta for atoms that was not intended to change. You must be careful and make sure that several atoms do not correspond to a given specie.

---

q_vectors is a python dictionary (`key: value` pairs). The `key` can be any string which sets a label of the q-vector. `value` must be a list of 3 values: $$q_x, q_y, q_z$$.

Output nodes

- `out`: `Dict` - Information of workflow results like success, last result node, list with convergence behavior


```
{
    "energies": {
        "label": 0.0,
        "label2": 0.014235119451769
    },
    "energy_units": "eV",
    "errors": [],
    "failed_labels": [],
    "info": [],
    "q_vectors": {
        "label": [
            0.0,
            0.0,
            0.0
        ],
        "label2": [
            0.125,
            0.0,
            0.0
        ]
    },
    "warnings": [],
    "workflow_name": "FleurSSDispConvWorkChain",
    "workflow_version": "0.1.0"
}
```


Resulting Spin Spiral energies are listed according to given labels.

Layout

SSDisp converge always starts with a structure and a list of q-vectors to calculate. There is no way to continue from pre-converged charge density.

Error handling

A list of implemented *exit codes*:

Code	Meaning
230	Invalid workchain parameters
340	Convergence SSDisp calculation failed for all q-vectors
341	Convergence SSDisp calculation failed for some q-vectors

Example usage

```
# -*- coding: utf-8 -*-
from aiida.orm import load_node, Dict
from aiida.engine import submit

from aiida_fleur.workflows.ssdisp_conv import FleurSSDispConvWorkChain

fleur_code = load_node(FLEUR_PK)
inpgen_code = load_node(INPGEN_PK)
structure = load_node(STRUCTURE_PK)

wf_para = Dict(dict={'beta': {'all': 1.57079},
                     'q_vectors': {'label': [0.0, 0.0, 0.0],
                                   'label2': [0.125, 0.0, 0.0]
                                   }
                     })


options = Dict(dict={'resources': {'num_machines': 1, 'num_mpiprocs_per_
↪machine': 24},
                     'queue_name': 'devel',
                     'custom_scheduler_commands': '',
                     'max_wallclock_seconds':  60*60})


parameters = Dict(dict={'atom': {'element': 'Pt',
                                 'lmax': 8
                                 },
                        'atom2': {'element': 'Fe',
                                  'lmax': 8,
                                  },
                        'comp': {'kmax': 3.8,
                                 },
                        'kpt': {'div1': 20,
                                'div2': 24,
                                'div3': 1
                                }})
```

(continues on next page)

(continued from previous page)

```
wf_para_scf = {'fleur_runmax': 2,
               'itmax_per_run': 120,
               'density_converged': 0.2,
               'serial': False,
               'mode': 'density'
               }

wf_para_scf = Dict(dict=wf_para_scf)

options_scf = Dict(dict={'resources': {'num_machines': 2, 'num_mpiprocs_per_
↪machine': 24},
                         'queue_name': 'devel',
                         'custom_scheduler_commands': '',
                         'max_wallclock_seconds':  60*60})

inputs = {'scf': {'wf_parameters': wf_para_scf,
                  'structure': structure,
                  'calc_parameters': parameters,
                  'options': options_scf,
                  'inpgen': inpgen_code,
                  'fleur': fleur_code
                  },
          'wf_parameters': wf_para,
          }


res = submit(FleurSSDispConvWorkChain, **inputs)
```

Fleur Magnetic Anisotropy Energy Converge workchain

- **Current version**: 0.2.0
- **Class**: *FleurMaeConvWorkChain*
- **String to pass to the** `WorkflowFactory()`: `fleur.mae_conv`
- **Workflow type**: Scientific workchain, self-consistent subgroup

Contents

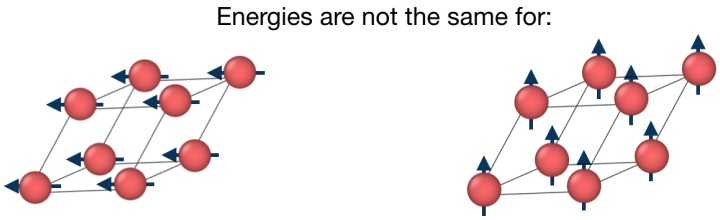
Import Example:

```
from aiida_fleur.workflows.mae_conv import FleurMaeConvWorkChain
#or
WorkflowFactory('fleur.mae_conv')
```

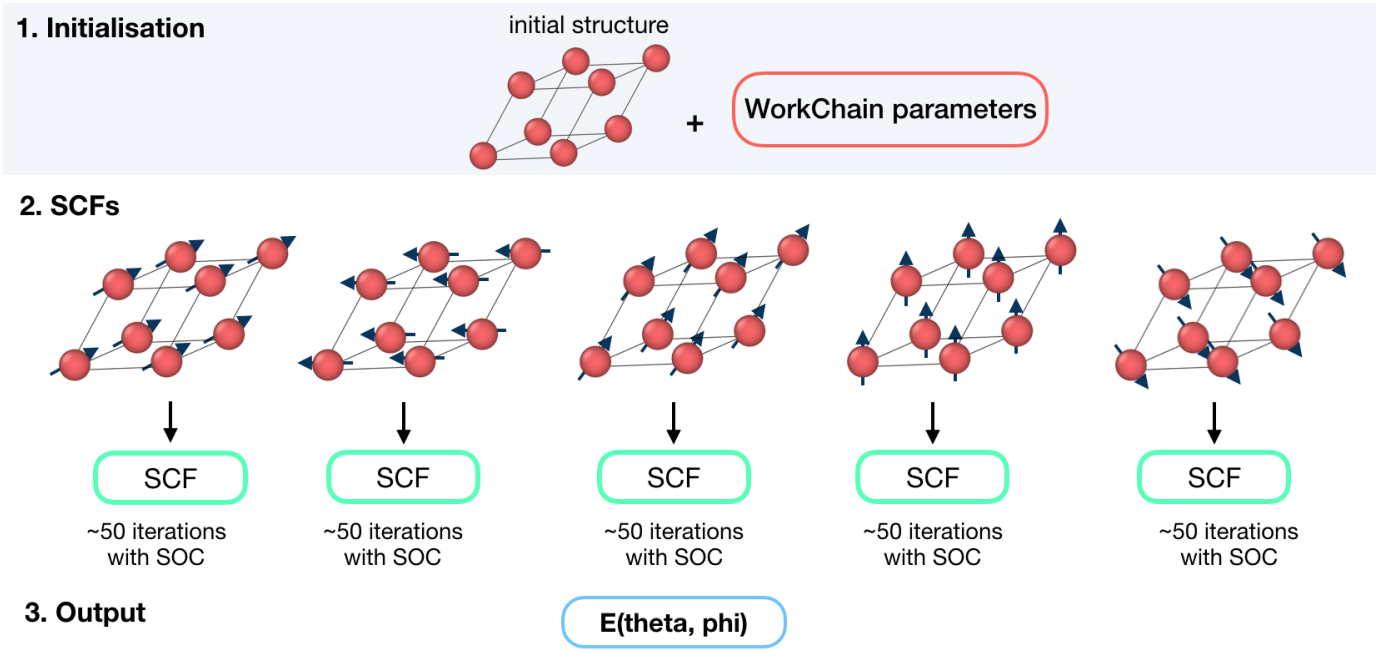
Description/Purpose

This workchain calculates Magnetic Anisotropy Energy over a given set of spin-quantization axes. The force-theorem is employed which means the workchain converges a reference charge density first then it submits a single FleurCalculation with a <*forceTheorem*> tag.

The task of the workchain us to calculate the energy difference between two or several structures having a different magnetisation profile:



To do this, the workchain employs the force theorem approach:



Input nodes

The FleurSSDispWorkChain employs exposed feature of the AiiDA, thus inputs for the nested *SCF* workchain should be passed in the namespace `scf`.

name	type	description	required
scf	namespace	inputs for nested SCF WorkChain	yes
wf_parameters	Dict	Settings of the workchain	no

Workchain parameters and its defaults

wf_parameters

wf_parameters: Dict - Settings of the workflow behavior. All possible keys and their defaults are listed below:

```
# -*- coding: utf-8 -*-
'sqas': {'label': [0.0, 0.0]},     # sets theta, phi pairs to calculate
'soc_off': []                      # a list of atom labels to switch off SOC term
```

soc_off is a python list containing atoms labels. SOC is switched off for species, corresponding to the atom with a given label.

Note: It can be that the specie correspond to several atoms and **soc_off** switches off SOC for atoms that was not intended to change. You must be careful and make sure that several atoms do not correspond to a given specie.

An example of **soc_off** work:

```
'soc_off': ['458']
```

changes

```
<species name="Ir-2" element="Ir" atomicNumber="77" coreStates="17" magMom=".00000000
↪" flipSpin="T">
  <mtSphere radius="2.52000000" gridPoints="747" logIncrement=".01800000"/>
  <atomicCutoffs lmax="8" lnonsphr="6"/>
  <energyParameters s="6" p="6" d="5" f="5"/>
  <prodBasis lcutm="4" lcutwf="8" select="4 0 4 2"/>
  <lo type="SCLO" l="1" n="5" eDeriv="0"/>
</species>
-----
<atomGroup species="Ir-2">
  <filmPos label="                 458">1.000/4.000 1.000/2.000 11.4074000502</
↪filmPos>
  <force calculate="T" relaxXYZ="TTT"/>
  <nocoParams l_relax="F" alpha=".00000000" beta=".00000000" b_cons_x=".00000000" b_
↪cons_y=".00000000"/>
</atomGroup>
```

to:

```
<species name="Ir-2" element="Ir" atomicNumber="77" coreStates="17" magMom=".00000000
↪" flipSpin="T">
  <mtSphere radius="2.52000000" gridPoints="747" logIncrement=".01800000"/>
  <atomicCutoffs lmax="8" lnonsphr="6"/>
  <energyParameters s="6" p="6" d="5" f="5"/>
  <prodBasis lcutm="4" lcutwf="8" select="4 0 4 2"/>
  <special socscale="0.0"/>
```

(continues on next page)

(continued from previous page)

```
  <lo type="SCLO" l="1" n="5" eDeriv="0"/>
</species>
```

As you can see, I was careful about "Ir-2" specie and it contained a single atom with a label 458. Please also refer to *Setting up atom labels* section to learn how to set labels up.

sqas is a python dictionary (`key: value` pairs). The `key` can be any string which sets a label of the SQA. `value` must be a list of 2 values: `[theta, phi]`.

Output nodes

- `out`: `Dict` - Information of workflow results like success, last result node, list with convergence behavior


```
{
    "errors": [],
    "failed_labels": [],
    "info": [],
    "mae": {
        "label": 0.001442720531486,
        "label2": 0.0
    },
    "mae_units": "eV",
    "sqa": {
        "label": [
            0.0,
            0.0
        ],
        "label2": [
            1.57079,
            1.57079
        ]
    },
    "warnings": [],
    "workflow_name": "FleurMaeConvWorkChain",
    "workflow_version": "0.1.0"
}
```


Resulting MAE energies are listed according to given labels.

Layout

MAE converge always starts with a structure and a list of q-vectors to calculate. There is no way to continue from pre-converged charge density.

Error handling

A list of implemented *exit codes*:

Code	Meaning
230	Invalid workchain parameters
342	Convergence MAE calculation failed for all SQAs
343	Convergence MAE calculation failed for all SQAs

Example usage

```
# -*- coding: utf-8 -*-
from aiida.orm import load_node, Dict
from aiida.engine import submit

from aiida_fleur.workflows.mae_conv import FleurMaeConvWorkChain

fleur_code = load_node(FLEUR_PK)
inpgen_code = load_node(INPGEN_PK)
structure = load_node(STRUCTURE_PK)

wf_para = Dict(dict={'sqas': {'label': [0.0, 0.0]},
                     'soc_off': []})


options = Dict(dict={'resources': {'num_machines': 1, 'num_mpiprocs_per_
↪machine': 24},
                     'queue_name': 'devel',
                     'custom_scheduler_commands': '',
                     'max_wallclock_seconds':  60*60})


parameters = Dict(dict={'atom': {'element': 'Pt',
                                 'lmax': 8
                                 },
                        'atom2': {'element': 'Fe',
                                  'lmax': 8,
                                  },
                        'comp': {'kmax': 3.8,
                                 },
                        'kpt': {'div1': 20,
                                'div2': 24,
                                'div3': 1
                                }})

wf_para_scf = {'fleur_runmax': 2,
               'itmax_per_run': 120,
               'density_converged': 0.2,
               'serial': False,
               'mode': 'density'
               }

wf_para_scf = Dict(dict=wf_para_scf)

options_scf = Dict(dict={'resources': {'num_machines': 2, 'num_mpiprocs_per_
↪machine': 24},
                         'queue_name': 'devel',
                         'custom_scheduler_commands': '',
                         'max_wallclock_seconds':  60*60})

inputs = {'scf': {'wf_parameters': wf_para_scf,
                  'structure': structure,
                  'calc_parameters': parameters,
                  'options': options_scf,
                  'inpgen': inpgen_code,
                  'fleur': fleur_code
```

(continues on next page)

(continued from previous page)

```
                    },
            'wf_parameters': wf_para,
            }


res = submit(FleurMaeConvWorkChain, **inputs)
```

3.1.5 Verdi command line extentions

Currently there are no specific verdi commands implemented for AiiDA-FLEUR. If you have any suggestions on some, please post an issue on GitHub.

3.1.6 Tools

here some more information about the tools contained in this package. and how to use them

Manipulation parameterdata:

merger

Getting structure data:

From cif files (ICSD) From COD/TCOD From OQMD From ALFOWLIB From Materials project

3.1.7 Tutorials

sda

3.1.7.1 Basic AiiDA tutorials:

If you are not familiar with the basics of AiiDA yet, you might want to checkout the AiiDA youtube tutorials. The jupyter notebooks from the tutorials you will find here on github, where you can also try them out in binder. Virtual machines for tutorials and tutorial manuals you find here.

3.1.7.2 How calculation plugins work:

Run inpgen calculation tutorial

sorry, not uploaded yet

Run fleur calculation tutorial

sorry, not uploaded yet

3.1.7.3 Running workflows:

Run fleur SCF tutorial

sorry, not uploaded yet

Run fleur eos tutorial

sorry, not uploaded yet

Run fleur bandstructure/dos tutorial

sorry, not uploaded yet

3.1.7.4 Data extraction and evaluation:

General calculation, workflow

Total database

3.1.8 Hints

3.1.8.1 For Users

Common Errors, Traps:

1. Wrong AiiDA datatype/Data does not have function X. Sometimes if the daemon is restarted (and something in the plugin files might have changed) AiiDA will return node of the superclasses, and not the plugin classes, which will be caught by some assert, or some methods will be called that are not implemented in the base-classes. If you are a user, goto the plugin folder and delete all '.pyc' files. And restart the daemon. Restarting jupyter-notebook, might also help. You have to clear the old plugin classes from the cache. If you are a developer, this might also be because there is still some bug in the used class, and the plugin system of AiiDA cannot load it. Therefore check you development environment for simple syntax errors and others. Also checking if the python interpreter runs through on the file, or checking with pylint might help. `$reentry scan aiida` might also help, if plugin code was changed.
2. TypeError: super(type, obj): obj must be an instance or subtype of type. This has a similar reason as 1. The class was changed and was not yet initialize by AiiDA. restart the daemon and clear .pyc files. If this happens for a subworkflow class it might also help to also import the subworkflow in your nodebook/pythonscript.
3. Submission fails. If it is a first calculation to a computer check if the resource is available. Check the log of the calculation. Run verdi computer test. This might also be due to reason 1. if it is a followup simulations that does something with data produced by an other calculation before, but the output had the wrong type.

3.1.8.2 FAQ

to come

3.1.9 Exit codes

AiiDA processes return a special object upon termination - an exit code. Basically, there are two types of exit-codes: non-zero and zero ones. If a process returned a zero exit code it has finished successfully. In contrast, non-zero exit code means there were a problem.

For example, there are 2 processes shown below:

```
(aiidapy)$ verdi process list -a -p 1
  PK  Created    State             Process label             Process status
 ----  ---------  ----------------  ------------------------  ------------------------
↪-----------
  60  3m ago      Finished [0]    FleurCalculation
  68  3m ago      Finished [302]  FleurCalculation
```

The first calculation was successful and the second one failed and threw exit code 302, which means it could not open one of the output files for some reason.

For more detailed information, see AiiDA documentation.

The list of all exit codes implemented in AiiDA-FLEUR:

Exit code	Exit message	Thrown by
230	Invalid workchain parameters	CreateMagnetic
230	Invalid workchain parameters	DMI
230	Invalid workchain parameters	EOS
230	Invalid workchain parameters	MAE
230	Invalid workchain parameters	MAE Conv
230	Invalid workchain parameters	Relax
230	Invalid workchain parameters	SCF
230	Invalid workchain parameters	SSDisp
230	Invalid workchain parameters	SSDisp Conv
231	Invalid input configuration	CreateMagnetic
231	Invalid input configuration	DMI
231	Invalid input configuration	MAE
231	Invalid input configuration	SCF
231	Invalid input configuration	SSDisp
233	Input codes do not correspond to fleur or inpgen codes respectively.	DMI
233	Input codes do not correspond to fleur or inpgen codes respectively.	MAE
233	Input codes do not correspond to fleur or inpgen codes respectively.	SSDisp
235	Input file modification failed.	DMI
235	Input file modification failed.	MAE
235	Input file modification failed	SCF
235	Input file modification failed.	SSDisp
236	Input file was corrupted after modifications	DMI
236	Input file was corrupted after modifications	MAE
236	Input file was corrupted after modifications	SCF
236	Input file was corrupted after modifications	SSDisp
300	No retrieved folder found	FleurCalculatio
300	No retrieved folder found	FleurCalculatio
300	No retrieved folder found	FleurinpgenCal
300	No retrieved folder found	FleurinpgenCal
301	One of the output files can not be opened	FleurCalculatio
301	One of the output files can not be opened	FleurinpgenCal
302	FLEUR calculation failed for unknown reason	FleurCalculatio
303	XML output file was not found	FleurCalculatio
304	Parsing of XML output file failed	FleurCalculatio
305	Parsing of relax XML output file failed	FleurCalculatio
306	XML input file was not found	FleurinpgenCal
310	FLEUR calculation failed due to memory issue	FleurCalculatio

Continued on ne

Table 1 – continued from previous page

311	FLEUR calculation failed because atoms spilled to the vacuum	FleurBase
311	FLEUR calculation failed because atoms spilled to the vacuum	FleurCalculatio
311	FLEUR calculation failed because atoms spilled to the vacuum	Relax
312	FLEUR calculation failed due to MT overlap	FleurCalculatio
313	Overlapping MT-spheres during relaxation	FleurBase
313	Overlapping MT-spheres during relaxation	FleurCalculatio
313	Overlapping MT-spheres during relaxation	Relax
314	Problem with cdn is suspected	Relax
315	Invalid Elements found in the LDA+U density matrix.	FleurCalculatio
315	Invalid Elements found in the LDA+U density matrix.	FleurBase
316	Calculation failed due to time limits.	FleurCalculatio
334	Reference calculation failed.	DMI
334	Reference calculation failed.	MAE
334	Reference calculation failed.	SSDisp
335	Found no reference calculation remote repository.	DMI
335	Found no reference calculation remote repository.	MAE
335	Found no reference calculation remote repository.	SSDisp
336	Force theorem calculation failed.	DMI
336	Force theorem calculation failed.	MAE
336	Force theorem calculation failed.	SSDisp
340	Convergence SSDisp calculation failed for all q-vectors	SSDisp conv
341	Convergence SSDisp calculation failed for some q-vectors	SSDisp conv
343	Convergence MAE calculation failed for all SQAs	MAE conv
344	Convergence MAE calculation failed for some SQAs	MAE conv
350	The workchain execution did not lead to relaxation criterion. Thrown in the very end of the workchain.	Relax
351	SCF Workchains failed for some reason.	Relax
352	Found no relaxed structure info in the output of SCF	Relax
353	Found no SCF output	Relax
354	Force is small, switch to BFGS	Relax
360	Inpgen calculation failed	SCF
361	Fleur calculation failed	SCF
380	Specified substrate is not bcc or fcc, only them are supported	CreateMagnetic
382	Relaxation calculation failed.	CreateMagnetic
383	EOS WorkChain failed.	CreateMagnetic
389	FLEUR calculation failed due to memory issue and it can not be solved for this scheduler	FleurBase
390	check_kpts() suggests less than 60% of node load	FleurBase
399	FleurCalculation failed and FleurBaseWorkChain has no strategy to resolve this	FleurBase
399	FleurRelaxWorkChain failed and FleurBaseRelaxWorkChain has no strategy to resolve this	Relax Base

CHAPTER 4

Developer's Guide

Some things to notice for AiiDA-FLEUR developers. Conventions, programming style, Integrated testing, things that should not be forgotten

4.1 Developer's guide

This is the developers guide for AiiDA-FLEUR

Contents

4.1.1 Package layout

All source code is under 'aiida_fleur/'

Folder name	Content
calculation	Calculation plugin classes. Each within his own file.
cmdline	Verdi command line plugins.
common	BaseRestartWorkChain routines copied from AiiDA-core.
data	Data structure plugins, each with his own file.
fleur_schema	Place of the XML schema files to validate Fleur input files
parsers	Parsers of the package, each has its own source file.
tests	Contineous integration tests
tools	Everything using, common used functions and workfunctions
workflows	All workchain/workflow classes, each has its own file.

The example folder contains currently some small manual examples, tutorials, calculation] and workchain submission tests. Documentation is fully contained within the docs folder. The rest of the files are needed for python packaging or continuous integration things.

4.1.2 Automated tests

Every decent software should have a set of rather fast tests which can be run after every commit. The more complete all code features and code lines are tested the better. Read the unittest design guidelines on the web. Through idealy there should be only one test(set) for one 'unit', to ensure that if something breaks, it stays local in the test result. Tests should be clearly understanble and documented.

You can run the continuous integration tests of aiida-fleur via (for this make sure that postgres 'pg_ctl' command is in your path):

```
cd aiida_fleur/tests/
./run_all_cov.sh
```

the output should look something like this:

```
(env_aiida)% ./run_all.sh
====================================== test session starts␣
↪==================================
platform darwin -- Python 2.7.15, pytest-3.5.1, py-1.5.3, pluggy-0.6.0
rootdir: /home/github/aiida-fleur, inifile: pytest.ini
plugins: cov-2.5.1
collected 166 items

test_entrypoints.py ............                                                    ␣
↪[  7%]
data/test_fleurinp.py ..........................................................
↪[ 63%]
parsers/test_fleur_parser.py ........                                              ␣
↪[ 68%]
tools/test_common_aiida.py .                                                       ␣
↪[ 68%]
tools/test_common_fleur_wf.py ..                                                   ␣
↪[ 69%]
tools/test_common_fleur_wf_util.py ..........                                      ␣
↪[ 75%]
tools/test_element_econfig_list.py .......                                         ␣
↪[ 80%]
tools/test_extract_corelevels.py ...                                               ␣
↪[ 81%]
```

(continues on next page)

(continued from previous page)

```
tools/test_io_routines.py ..                                                 ␣
↪[ 83%]
tools/test_parameterdata_util.py ..                                          ␣
↪[ 84%]
tools/test_read_cif_folder.py .                                              ␣
↪[ 84%]
tools/test_xml_util.py ................                                      ␣
↪[ 94%]
workflows/test_workflows_builder_init.py .........                           ␣
↪[100%]

---------- coverage: platform darwin, python 2.7.15-final-0 ----------
Name                                                 Stmts   Miss  Cover   ␣
↪Missing
--------------------------------------------------------------------------------------
↪------
./aiida_fleur/__init__.py                              2      0   100%
./aiida_fleur/calculation/__init__.py                  1      0   100%
./aiida_fleur/calculation/fleur.py                   305    284     7%   43-221, xxx
./aiida_fleur/calculation/fleurinputgen.py           264    234    11%   40-63, xxx
./aiida_fleur/data/__init__.py                         1      0   100%
./aiida_fleur/data/fleurinp.py                       409    132    68%   85-86, xxx
./aiida_fleur/data/fleurinpmodifier.py               175     69    61%   72, 65, xxx
./aiida_fleur/fleur_schema/__init__.py                 1      0   100%
./aiida_fleur/fleur_schema/schemafile_index.py        14      0   100%
./aiida_fleur/parsers/__init__.py                      4      0   100%
./aiida_fleur/parsers/fleur.py                       461    199    57%   50-61, 68, xxx
./aiida_fleur/parsers/fleur_inputgen.py               52     42    19%   46-55, 65-152
./aiida_fleur/tools/ParameterData_util.py             33      5    85%   48, 50, 70-73
./aiida_fleur/tools/StructureData_util.py            361    312    14%   39-71, 79-84,␣
↪xxx
./aiida_fleur/tools/__init__.py                        1      0   100%
./aiida_fleur/tools/check_existence.py                 7      7     0%   14-149
./aiida_fleur/tools/common_aiida.py                  130     97    25%   53-73, 89-121,␣
↪xxx
./aiida_fleur/tools/common_fleur_wf.py               260    209    20%   39, 47-51, 56-
↪57, xxx
./aiida_fleur/tools/common_fleur_wf_util.py          232    108    53%   24-43, 80-102,␣
↪xxx
xxx
--------------------------------------------------------------------------------------
↪------
TOTAL                                                                7316   5332␣
↪    27%



==================================== 166 passed in 22.53 seconds␣
↪============================
```

If anything (especially a lot of tests) fails it is very likly that your installation is messed up. Maybe some packages are missing (reinstall them by hand and report please). Or the aiida-fleur version you have installed is not compatible with the aiida-core version you are running, since not all aiida-core versions are backcompatible. We try to not break back compability within aiida-fleur itself. Therfore, newer versions of it should still work with older versions of the FLEUR code, but newer FLEUR releases force you to migrate to a newer aiida-fleur version.

The current test coverage of AiiDA-FLEUR has room to improve which is mainly due to the fact that calculations and

workchains are not yet in the CI tests, because this requires more effort. Also most functions that do not depend on AiiDA are moved out of this package.

Parser and fleurinp test:

There are basic parser tests which run for every outputfile (out.xml) in folder 'aiida_fleur/tests/files/outxml/all_test/' If something changes in the FLEUR output or output of a certain feature or codepath, just add such an outputfile to this folder (try to keep the filesize small, if possible).

For input file testing add input files to be tested to the 'aiida_fleur/tests/files/inpxml' folder and subfolders. On these files some basic fleurinpData tests are run.

4.1.3 Plugin development

Read the AiiDA plugin developer guide. In general ensure the provenance and try to reduce complexity and use a minimum number of nodes. Here some questions you should ask yourself:

For calculation plugins:

- What are my input nodes, are they all needed?
- Is it apparent to the user how/where the input is specified?
- What features of the code are supported/unsupported?
- Is the plugin robust, transparent? Keep as simple/dump as possible/neccessary.
- What are usual errors a user will do? Can they be circumvented? At least they should be caught.
- Are AiiDA espected name convention accounted for? Otherwise it won't work.

Parsers:

- Is the parser robust? The parser should never fail.
- Is the parser code modular, easy to read and understand?
- Fully tested? Parsers are rather easy testable, do so!
- Parsers should have a version number. Can one reparse?

For datastructure plugins:

- Do you really need a new Datastructure?
- What is stored in the Database/Attributes?
- Do the names/keys apply with AiiDA conventions?
- Is the ususal information the user is interested easy to query for?
- What is stored in the Repository/Files?
- Is the data code specific or rather general? If general it should become an extra extermal plugin.

4.1.4 Workflow/chain development

Here are some guidelines for writing FLEUR workflows/workchains and workflows in general. Keep in mind that a workflow is **SOFTWARE** which will be used by others and build on top and **NOT** just a script. Also not for every task a workflow is needed. Read the workchain guidelines of AiiDA-core itself and the aiida-quantumespresso package.

4.1.4.1 General Workflow development guidelines:

1. Every workflow needs a clear **documentation** of input, output! Think this through and do not change it later on light hearted, because you will break the code of others! Therefore, invest the time to think about a **clear interface**.
2. Think about the **complete design** of the workflow first, break it into smaller parts. Write a clear, self esplaining 'spec.outline' then implement step for step.
3. **Reuse** as much of previous workflows **code** as possible, use subworkflows. (otherwise your code explodes, is hard to understand again und not reusable)
4. If you think some processing is common or might be useful for something else, make it **modular**, and import the method (goes along with point 3.).
5. Try to keep the workflow **context clean**! (this part will always be saved and visible, there people track what is going on.
6. Give the **user feedback** of what is going on. Write clear report statements in the **workflow report**.
7. Think about **resource management**. i.e if a big system needs to be calculated and the user says use x hundred cores, and in the workflow simulations on very small systems need to be done, it makes no sense to submit a job with the same huge amount of resources. Use resource estimators and check if plausible.
8. **ERROR handling**: Error handling is very important and might take a lot of effort. Write at least an outline (named: inspect_xx, handle_xx), which skeleton for all the errors (treated or not). (look at the AiiDA QE workflows as good example) Now iterative put every time you encounter a 'crash' because something failed (usually variable/node access stuff), the corresponding code in a try block and call your handler. Use the workchain exit methods to clearly terminate the workflow in the case something went wrong and it makes no sense to continue. Keep in mind, your workflow should never:
   - End up in a while true. Check calculation or subworkflow failure cases.
   - Crash at a later point because a calculation or subworkflow failed. The user won't understand easily what happend. Also this makes it impossible to build useful error handling of your workflow on top, if using your workflow as a subworkflow.
9. **Write tests** and provide **easy examples**. Doing so for workchains is not trivial. It helps a lot to keep things modular and certain function seperate for testing.
10. Workflows should have a version number. Everytime the output or input of the workflow changes the version number should increase. (This allows to account for different workflow version handling in data parsing and processing later on. Or ggf)

4.1.4.2 FLEUR specific desgin suggestions, conventions:

1. Output nodes of a workflow has the **naming convention** 'output_wfname_description' i.e 'output_scf_wc_para'
2. Every workflow should give back **one parameter output node named 'output_wfname_para'** which contains all the 'physical results' the workflow is designed to provide, or at least information to access these results directly (if stored in files and so on) further the node should contain valuable information to make sense/judge the quality of the data. Try to design this node in a way that if you take a look at it, you understand the following questions:

- Which workflow was run, what version?
- What came out?
- What was put in, how can I see what was put in?
- Is this valueable or garbage?
- What were the last calculations run?

3. So far **name Fleur workflows/workchains classes: fleur_name_wc** 'Fleur' avoids confusion when working with multi codes because other codes perform similar task and have similar workchains. The '_wc' ending because it makes it clearer on import in you scripts and notebook to know that this in not a simple function.
4. For user friendlyness: add **extras, label, descriptions** to calculations and output nodes. In 'verdi calculation list' the user should be able to what workchain the calculation belongs to and what it runs on. Also if you run many simulations think about creating a group node for all the workflow internal(between) calculations. All these efforts makes it easier to extract results from global queries.
5. Write **base subworkchains**, that take all FLAPW parameters as given, but do their task very well and then write workchains on top of these. Which then can use workchains/functions to optimize the FLEUR FLAPW parameters.
6. Outsource methods to test for calculation failure, that you have only one routine in all workchains, that one can improve

4.1.5 Entrypoints

In order to make AiiDA aware of any classes (plugins) like (calculations, parsers, data, workchains, workflows, commandline) the python entrypoint system is used. Therefore, you have to register any of the above classes as an entrypoint in the 'setup.json' file.

Example:

```
"entry_points" : {
    "aiida.calculations" : [
        "fleur.fleur = aiida_fleur.calculation.fleur:FleurCalculation",
        "fleur.inpgen = aiida_fleur.calculation.fleurinputgen:FleurinputgenCalculation
↪"
    ],
    "aiida.data" : [
            "fleur.fleurinp = aiida_fleur.data.fleurinp:FleurinpData",
            "fleur.fleurinpmodifier = aiida_fleur.data.
↪fleurinpmodifier:FleurinpModifier"
    ],
    "aiida.parsers" : [
            "fleur.fleurparser = aiida_fleur.parsers.fleur:FleurParser",
            "fleur.fleurinpgenparser = aiida_fleur.parsers.fleur_inputgen:Fleur_
↪inputgenParser"
    ],
    "aiida.workflows" : [
        "fleur.scf = aiida_fleur.workflows.scf:fleur_scf_wc",
        "fleur.dos = aiida_fleur.workflows.dos:fleur_dos_wc",
        "fleur.band = aiida_fleur.workflows.band:FleurBandWorkChain",
        "fleur.eos = aiida_fleur.workflows.eos:fleur_eos_wc",
        "fleur.dummy = aida_fleur.workflows.dummy:dummy_wc",
        "fleur.sub_dummy = aida_fleur.workflows.dummy:sub_dummy_wc",
        "fleur.init_cls = aiida_fleur.workflows.initial_cls:fleur_initial_cls_wc",
        "fleur.corehole = aiida_fleur.workflows.corehole:fleur_corehole_wc",
```

(continues on next page)

(continued from previous page)

```
        "fleur.corelevel = aiida_fleur.workflows.corelevel:fleur_corelevel_wc"
    ]}
```

The left handside will be the entry point name. This name has to be used in any FactoryClasses of AiiDA. The convention here is that the name has two parts 'package_name.whatevername'. The package name has to be reserved/registerd in the AiiDA registry, because entry points should be unique. The right handside has the form 'module_path:class_name'.

4.1.6 Other information

Google python guide, doing releases, pypi, packaging, git basics, issues, aiida logs, loglevel, ...

4.1.6.1 Useful to know

1. pip -e is your friend:

```
pip install -e package_dir
```

Always install python packages you are working on with -e, this way the new version is used, if the files are changed, as long as the '.pyc' files are updated.

2. In jupyter/python use the magic:

```
%load_ext autoreload
%autoreload 2
```

This will import your classes everytime anew. Otherwise they are not reimportet if they have already importet. This is very useful for development work.

CHAPTER 5

Module reference (API)

Automatic generated documentation for all modules, classes and functions with reference to the source code. The search is your friend.

5.1 Source code Documentation (API reference)

5.1.1 Fleur input generator plug-in

5.1.1.1 Fleurinputgen Calculation

Input plug-in for the FLEUR input generator 'inpgen'. The input generator for the Fleur code is a preprocessor and should be run locally (with the direct scheduler) or inline, because it does not take many resources.

class aiida_fleur.calculation.fleurinputgen.**FleurinputgenCalculation**(**args*, ***kwargs*)

JobCalculationClass for the inpgen, which is a preprocessor for a FLEUR calculation. For more information about produced files and the FLEUR-code family, go to http://www.flapw.de/.

classmethod define(*spec*)

Define the process specification, including its inputs, outputs and known exit codes.

Parameters **spec** – the calculation job process spec to define.

prepare_for_submission(*folder*)

This is the routine to be called when you want to create the input files for the inpgen with the plug-in.

Parameters **folder** – a aiida.common.folders.Folder subclass where the plugin should put all its files.

aiida_fleur.calculation.fleurinputgen.**conv_to_fortran**(*val*, *quote_strings=True*)

Parameters **val** – the value to be read and converted to a Fortran-friendly string.

aiida_fleur.calculation.fleurinputgen.**get_input_data_text**(*key*, *val*, *value_only*, *mapping=None*)

Given a key and a value, return a string (possibly multiline for arrays) with the text to be added to the input file.

Parameters

- **key** – the flag name
- **val** – the flag value. If it is an array, a line for each element is produced, with variable indexing starting from 1. Each value is formatted using the conv_to_fortran function.
- **mapping** – Optional parameter, must be provided if val is a dictionary. It maps each key of the 'val' dictionary to the corresponding list index. For instance, if `key='magn'`, `val = {'Fe': 0.1, 'O': 0.2}` and `mapping = {'Fe': 2, 'O': 1}`, this function will return the two lines `magn(1) = 0.2` and `magn(2) = 0.1`. This parameter is ignored if 'val' is not a dictionary.

5.1.1.2 Fleurinputgen Parser

This module contains the parser for a inpgen calculation and methods for parsing different files produced by inpgen.

class aiida_fleur.parsers.fleur_inputgen.**Fleur_inputgenParser**(*node*)

This class is the implementation of the Parser class for the FLEUR inpgen. It takes the files received from an inpgen calculation and creates AiiDA nodes for the Database. From the inp.xml file a FleurinpData object is created, also some information from the out file is stored in a ParameterData node.

parse(***kwargs*)

Takes inp.xml generated by inpgen calculation and created an FleurinpData node.

Returns a dictionary of AiiDA nodes to be stored in the database.

5.1.2 Fleur-code plugin

5.1.2.1 Fleur Calculation

This file contains a CalcJob that represents FLEUR calculation.

class aiida_fleur.calculation.fleur.**FleurCalculation**(**args*, ***kwargs*)

A CalcJob class that represents FLEUR DFT calculation. For more information about the FLEUR-code family go to http://www.flapw.de/

classmethod define(*spec*)

Define the process specification, including its inputs, outputs and known exit codes.

Parameters **spec** – the calculation job process spec to define.

prepare_for_submission(*folder*)

This is the routine to be called when you make a FLEUR calculation. This routine checks the inputs and modifies copy lists accordingly. The standard files to be copied are given here.

Parameters **folder** – a aiida.common.folders.Folder subclass where the plugin should put all its files.

5.1.2.2 Fleur Parser

This module contains the parser for a FLEUR calculation and methods for parsing different files produced by FLEUR.

Please implement file parsing routines that they can be executed from outside the parser. Makes testing and portability easier.

class aiida_fleur.parsers.fleur.**FleurParser**(*node*)

This class is the implementation of the Parser class for FLEUR. It parses the FLEUR output if the calculation was successful, i.e checks if all files are there that should be and their condition. Then it parses the out.xml file and returns a (simple) parameterData node with the results of the last iteration. Other files (DOS.x, bands.x, relax.xml, ...) are also parsed if they are retrieved.

get_linkname_outparams()

Returns the name of the link to the output_complex Node contains the Fleur output in a rather complex dictionary.

get_linkname_outparams_complex()

Returns the name of the link to the output_complex Node contains the Fleur output in a rather complex dictionary.

parse(***kwargs*)

Receives in input a dictionary of retrieved nodes. Does all the logic here. Checks presents of files. Calls routines to parse them and returns parameter nodes and success.

Return successful Bool, if overall parsing was successful or not

Return new_nodes_list list of tuples of two (linkname, Dataobject), nodes to be stored by AiiDA

aiida_fleur.parsers.fleur.**convert_frac**(*ratio*)

Converts ratio strings into float, e.g. 1.0/2.0 -> 0.5

aiida_fleur.parsers.fleur.**parse_bands_file**(*bands_lines*)

Parses the returned bands.1 and bands.2 file and returns a complete bandsData object. bands.1 has the form: k value, energy

Parameters **bands_lines** – string of the read in bands file

aiida_fleur.parsers.fleur.**parse_dos_file**(*dos_lines*)

Parses the returned DOS.X files. Structure: (100(1x,e10.3)) e,totdos,interstitial,vac1,vac2, (at(i),i=1,ntype),((q(l,i),l=1,LMAX),i=1,ntype) where e is the energy in eV (= 1/27.2 htr) at(i) is the local DOS of a single atom of the i'th atom-type and q(l,i) is the l-resolved DOS at the i'th atom but has to be multiplied by the number of atoms of this type.

Parameters

- **dos_lines** – string of the read in dos file
- **number_of_atom_types** – integer, number of atom types

aiida_fleur.parsers.fleur.**parse_relax_file**(*rlx*)

This function parsers relax.xml output file and returns a Dict containing all the data given there.

aiida_fleur.parsers.fleur.**parse_xmlout_file**(*outxmlfile*)

Parses the out.xml file of a FLEUR calculation Receives as input the absolute path to the xml output file

Parameters **outxmlfile** – path to out.xml file

Returns xml_data_dict a simple dictionary (QE output like) with parsed data

5.1.3 Fleur input Data structure

5.1.3.1 Fleur input Data structure

In this module is the *FleurinpData* class, and methods for FLEUR input manipulation plus methods for extration of AiiDA data structures.

class aiida_fleur.data.fleurinp.**FleurinpData**(***kwargs*)

AiiDA data object representing everything a FLEUR calculation needs.

It is initialized with an absolute path to an `inp.xml` file or a FolderData node containing `inp.xml`. Other files can also be added that will be copied to the remote machine, where the calculation takes place.

It stores the files in the repository and stores the input parameters of the `inp.xml` file of FLEUR in the database as a python dictionary (as internal attributes). When an `inp.xml` (name important!) file is added to files, FleurinpData searches for a corresponding xml schema file in the PYTHONPATH environment variable. Therefore, it is recommend to have the plug-in source code directory in the python environment. If no corresponding schema file is found an error is raised.

FleurinpData also provides the user with methods to extract AiiDA StructureData and KpointsData nodes.

Remember that most attributes of AiiDA nodes can not be changed after they have been stored in the database! Therefore, you have to use the FleurinpModifier class and its methods if you want to change somthing in the `inp.xml` file. You will retrieve a new FleurinpData that way and start a new calculation from it.

__init__(***kwargs*)

Initialize a FleurinpData object set the files given

del_file(*filename*)

Remove a file from FleurinpData instancefind

Parameters **filename** – name of the file to be removed from FleurinpData instance

files

Returns the list of the names of the files stored

find_schema(*inp_version_number*)

Method which searches for a schema files (.xsd) which correspond to the input xml file. (compares the version numbers)

Parameters **inp_version_number** – a version of `inp.xml` file schema to be found

Returns

A two-element tuple:

1. A list of paths where schema files are located
2. A boolen which shows if the required version schema file was found

get_content(*filename='inp.xml'*)

Returns the content of the single file stored for this data node.

Returns A string of the file content

get_fleur_modes()

Analyses `inp.xml` file to set up a calculation mode. 'Modes' are paths a FLEUR calculation can take, resulting in different output files. This files can be automatically addded to the retrieve_list of the calculation.

Common modes are: scf, jspin2, dos, band, pot8, lda+U, eels, ...

Returns a dictionary containing all possible modes. A mode is activated assigning a non-empty string to the corresponding key.

get_kpointsdata()
This routine returns an AiiDA KpointsData type produced from the inp.xml file. This only works if the kpoints are listed in the in inpxml. This is a calcfunction and keeps the provenance!

Returns KpointsData node

get_kpointsdata_ncf()
This routine returns an AiiDA KpointsData type produced from the inp.xml file. This only works if the kpoints are listed in the in inpxml. This is NOT a calcfunction and does not keep the provenance!

Returns KpointsData node

static get_parameterdata(*fleurinp*)
This routine returns an AiiDA Dict type produced from the inp.xml file. The returned node can be used for inpgen as *calc_parameters*. This is a calcfunction and keeps the provenance!

Returns Dict node

get_parameterdata_ncf()
This routine returns an AiiDA Dict type produced from the inp.xml file. This node can be used for inpgen as *calc_parameters*. This is NOT a calcfunction and does NOT keep the provenance!

Returns Dict node

get_structuredata()
This routine return an AiiDA Structure Data type produced from the inp.xml file. If this was done before, it returns the existing structure data node. This is a calcfunction and therefore keeps the provenance.

Parameters **fleurinp** – a FleurinpData instance to be parsed into a StructureData

Returns StructureData node

get_structuredata_ncf()
This routine returns an AiiDA Structure Data type produced from the inp.xml file. not a calcfunction

Parameters **self** – a FleurinpData instance to be parsed into a StructureData

Returns StructureData node, or None

get_tag(*xpath*)
Tries to evaluate an xpath expression for inp.xml file. If it fails it logs it.

Parameters **xpath** – an xpath expression

Returns A node list retrived using given xpath

inp_dict
Returns the inp_dict (the representation of the inp.xml file) as it will or is stored in the database.

open(*path='inp.xml'*, *mode='r'*, *key=None*)
Returns an open file handle to the content of this data node.

Parameters

- **key** – name of the file to be opened
- **mode** – the mode with which to open the file handle

Returns A file handle in read mode

set_file(*filename*, *dst_filename=None*, *node=None*)
Add a file to the *FleurinpData* instance.

Parameters

- **filename** – absolute path to the file or a filename of node is specified

- **node** – a FolderData node containing the file

set_files (*files*, *node=None*)
Add the list of files to the *FleurinpData* instance. Can by used as an alternative to the setter.

Parameters

- **files** – list of abolute filepaths or filenames of node is specified
- **node** – a FolderData node containing files from the filelist

5.1.3.2 Fleurinp modifier

In this module is the FleurinpModifier class, which is used to manipulate FleurinpData objects in a way which keeps the provernance.

class aiida_fleur.data.fleurinpmodifier.**FleurinpModifier** (*original*)
A class which represents changes to the *FleurinpData* object.

add_num_to_att (*xpathn*, *attributename*, *set_val*, *mode='abs'*, *occ=None*)
Appends a *add_num_to_att()* to the list of tasks that will be done on the FleurinpData.

Parameters

- **xpathn** – an xml path to the attribute to change
- **attributename** – a name of the attribute to change
- **set_val** – a value to be added/multiplied to the previous value
- **mode** – 'abs' if to add set_val, 'rel' if multiply
- **occ** – a list of integers specifying number of occurrence to be set

static apply_modifications (*fleurinp_tree_copy*, *nmmp_lines_copy*, *modification_tasks*, *schema_tree=None*)
Applies given modifications to the fleurinp lxml tree. It also checks if a new lxml tree is validated against schema. Does not rise an error if inp.xml is not validated, simple prints a message about it.

Parameters

- **fleurinp_tree_copy** – a fleurinp lxml tree to be modified
- **n_mmp_lines_copy** – a n_mmp_mat file to be modified
- **modification_tasks** – a list of modification tuples

Returns a modified fleurinp lxml tree and a modified n_mmp_mat file

changes ()
Prints out all changes given in a *FleurinpModifier* instance.

create_tag (*xpath*, *newelement*, *create=False*)
Appends a *create_tag()* to the list of tasks that will be done on the FleurinpData.

Parameters

- **xpathn** – a path where to place a new tag
- **newelement** – a tag name to be created
- **create** – if True and there is no given xpath in the FleurinpData, creates it

delete_att (*xpath*, *attrib*)
Appends a *delete_att()* to the list of tasks that will be done on the FleurinpData.

Parameters

- **xpathn** – a path to the attribute to be deleted
- **attrib** – the name of an attribute

delete_tag(*xpath*)
Appends a *delete_tag()* to the list of tasks that will be done on the FleurinpData.

Parameters **xpathn** – a path to the tag to be deleted

freeze()
This method applies all the modifications to the input and returns a new stored fleurinpData object.

Returns stored *FleurinpData* with applied changes

get_avail_actions()
Returns the allowed functions from FleurinpModifier

replace_tag(*xpath*, *newelement*)
Appends a *replace_tag()* to the list of tasks that will be done on the FleurinpData.

Parameters

- **xpathn** – a path to the tag to be replaced
- **newelement** – a new tag

set_atomgr_att(*attributedict*, *position=None*, *species=None*, *create=False*)
Appends a *change_atomgr_att()* to the list of tasks that will be done on the FleurinpData.

Parameters

- **species_name** – a path to the tag to be replaced
- **attributedict** – attribute dictionary to be set into the atom group
- **create** – if True and there is no given atom group in the FleurinpData, creates it

set_atomgr_att_label(*attributedict*, *atom_label*, *create=False*)
Appends a *change_atomgr_att_label()* to the list of tasks that will be done on the FleurinpData.

Parameters

- **attributedict** – a new tag
- **atom_label** – Atom label which atom group will be set
- **create** – if True and there is no given atom group in the FleurinpData, creates it

set_inpchanges(*change_dict*)
Appends a *set_inpchanges()* to the list of tasks that will be done on the FleurinpData.

Parameters **change_dict** – a dictionary with changes

An example of change_dict:

```
change_dict = {'itmax' : 1,
               'l_noco': True,
               'ctail': False,
               'l_ss': True}
```

set_kpath(*kpath*, *count*, *gamma='F'*)
Appends a *set_kpath()* to the list of tasks that will be done on the FleurinpData.

set_kpointsdata(*kpointsdata_uuid*)
Appends a *set_kpointsdata_f()* to the list of tasks that will be done on the FleurinpData.

Parameters **kpointsdata_uuid** – an aiida.orm.KpointsData or node uuid, since the node is self cannot be be serialized in tasks.

set_nkpts(*count*, *gamma='F'*)
Appends a *set_nkpts()* to the list of tasks that will be done on the FleurinpData.

set_nmmpmat(*species_name*, *orbital*, *spin*, *occStates=None*, *denmat=None*, *phi=None*, *theta=None*)
Appends a *set_nmmpmat()* to the list of tasks that will be done on the FleurinpData.

Parameters

- **species_name** – species on which the density matrix should be set
- **orbital** – orbital on which the density matrix should be set
- **occStates** – list which specifies the diagonal elements of the density matrix
- **denmat** – matrix, which specifies the density matrix
- **phi** – optional angle to rotate density matrix
- **theta** – optional angle to rotate density matrix

set_species(*species_name*, *attributedict*, *create=False*)
Appends a *set_species()* to the list of tasks that will be done on the FleurinpData.

Parameters

- **species_name** – a path to the tag to be replaced
- **attributedict** – attribute dictionary to be set into the specie
- **create** – if True and there is no given specie in the FleurinpData, creates it

set_species_label(*at_label*, *attributedict*, *create=False*)
Appends a *set_species_label()* to the list of tasks that will be done on the FleurinpData.

Parameters

- **at_label** – Atom label which specie will be set
- **attributedict** – attribute dictionary to be set into the specie
- **create** – if True and there is no given specie in the FleurinpData, creates it

shift_value(*change_dict*, *mode='abs'*)
Appends a *shift_value()* to the list of tasks that will be done on the FleurinpData.

Parameters

- **change_dict** – a dictionary with changes
- **mode** – 'abs' if change given is absolute, 'rel' if relative

An example of change_dict:

```
change_dict = {'itmax' : 1, dVac = -2}
```

shift_value_species_label(*label*, *att_name*, *value*, *mode='abs'*)
Appends a *shift_value_species_label()* to the list of tasks that will be done on the FleurinpData.

Parameters

- **label** – a label of an atom
- **att_name** – attrubute name of a specie

- **value** – value to set
- **mode** – 'abs' if change given is absolute, 'rel' if relative

show(*display=True*, *validate=False*)
Applies the modifications and displays/prints the resulting `inp.xml` file. Does not generate a new *FleurinpData* object.

Parameters

- **display** – a boolean that is True if resulting `inp.xml` has to be printed out
- **validate** – a boolean that is True if changes have to be validated

Returns a lxml tree representing inp.xml with applied changes

undo(*revert_all=False*)
Cancels the last change or all of them

Parameters **revert_all** – set True if need to cancel all the changes, False if the last one.

validate()
Extracts the schema-file. Makes a test if all the changes lead to an inp.xml file that is validated against the schema.

Returns a lxml tree representing inp.xml with applied changes

xml_set_all_attribv(*xpathn*, *attributename*, *attribv*, *create=False*)
Appends a *xml_set_all_attribv()* to the list of tasks that will be done on the FleurinpData.

Parameters

- **xpathn** – a path to the attribute
- **attributename** – an attribute name
- **attribv** – an attribute value which will be set
- **create** – if True and there is no given xpath in the FleurinpData, creates it

xml_set_all_text(*xpathn*, *text*, *create=False*)
Appends a *xml_set_all_text()* to the list of tasks that will be done on the FleurinpData.

Parameters

- **xpathn** – a path to the attribute
- **text** – text to be set
- **create** – if True and there is no given xpath in the FleurinpData, creates it

xml_set_attribv_occ(*xpathn*, *attributename*, *attribv*, *occ=None*, *create=False*)
Appends a *xml_set_attribv_occ()* to the list of tasks that will be done on the FleurinpData.

Parameters

- **xpathn** – a path to the attribute
- **attributename** – an attribute name
- **attribv** – an attribute value which will be set
- **occ** – a list of integers specifying number of occurrence to be set
- **create** – if True and there is no given xpath in the FleurinpData, creates it

xml_set_first_attribv(*xpathn*, *attributename*, *attribv*, *create=False*)
Appends a *xml_set_first_attribv()* to the list of tasks that will be done on the FleurinpData.

Parameters

- **xpathn** – a path to the attribute
- **attributename** – an attribute name
- **attribv** – an attribute value which will be set
- **create** – if True and there is no given xpath in the FleurinpData, creates it

xml_set_text(*xpathn*, *text*, *create=False*)
Appends a *xml_set_text()* to the list of tasks that will be done on the FleurinpData.

Parameters

- **xpathn** – a path to the attribute
- **text** – text to be set
- **create** – if True and there is no given xpath in the FleurinpData, creates it

xml_set_text_occ(*xpathn*, *text*, *create=False*, *occ=0*)
Appends a *xml_set_text_occ()* to the list of tasks that will be done on the FleurinpData.

Parameters

- **xpathn** – a path to the attribute
- **text** – text to be set
- **create** – if True and there is no given xpath in the FleurinpData, creates it
- **occ** – an integer specifying number of occurrence to be set

aiida_fleur.data.fleurinpmodifier.**modify_fleurinpdata**(*original*, *modifications*, ***kwargs*)
A CalcFunction that performs the modification of the given FleurinpData and stores the result in a database.

Parameters

- **original** – a FleurinpData to be modified
- **modifications** – a python dictionary of modifications in the form of {'task': ...}
- **kwargs** – dict of other aiida nodes to be linked to the modifications

Returns new_fleurinp a modified FleurinpData that is stored in a database

aiida_fleur.data.fleurinpmodifier.**set_kpointsdata_f**(*fleurinp_tree_copy*, *kpointsdata_uuid*)
This calc function writes all kpoints from a KpointsData node in the inp.xml file as a kpointslist. It replaces kpoints written in the inp.xml file. Currently it is the users responsibility to provide a full KpointsData node with weights.

Parameters

- **fleurinp_tree_copy** – fleurinp_tree_copy
- **kpointsdata_uuid** – node identifier or KpointsData node to be written into inp.xml

Returns modified xml tree

5.1.4 Workflows/Workchains

5.1.4.1 Base: Fleur-Base WorkChain

This module contains the FleurBaseWorkChain. FleurBaseWorkChain is a workchain that wraps the submission of the FLEUR calculation. Inheritance from the BaseRestartWorkChain allows to add scenarios to restart a calculation in an automatic way if an expected failure occurred.

class aiida_fleur.workflows.base_fleur.**FleurBaseWorkChain**(**args*, ***kwargs*)
Workchain to run a FLEUR calculation with automated error handling and restarts

check_kpts()
This routine checks if the total number of requested cpus is a factor of kpts and makes an optimisation.

If suggested number of num_mpiprocs_per_machine is 60% smaller than requested, it throws an exit code and calculation stop withour submission.

validate_inputs()
Validate inputs that might depend on each other and cannot be validated by the spec. Also define dictionary *inputs* in the context, that will contain the inputs for the calculation that will be launched in the *run_calculation* step.

5.1.4.2 SCF: Fleur-Scf WorkChain

In this module you find the workchain 'FleurScfWorkChain' for the self-consistency cycle management of a FLEUR calculation with AiiDA.

class aiida_fleur.workflows.scf.**FleurScfWorkChain**(*inputs=None*, *logger=None*, *runner=None*, *enable_persistence=True*)
Workchain for converging a FLEUR calculation (SCF).

It converges the charge density, total energy or the largest force. Two paths are possible:

(1) Start from a structure and run the inpgen first optional with calc_parameters

(2) Start from a Fleur calculation, with optional remoteData

Parameters

- **wf_parameters** – (Dict), Workchain Specifications
- **structure** – (StructureData), Crystal structure
- **calc_parameters** – (Dict), Inpgen Parameters
- **fleurinp** – (FleurinpData), to start with a Fleur calculation
- **remote_data** – (RemoteData), from a Fleur calculation
- **inpgen** – (Code)
- **fleur** – (Code)

Returns output_scf_wc_para (Dict), Information of workflow results like Success, last result node, list with convergence behavior

change_fleurinp()
This routine sets somethings in the fleurinp file before running a fleur calculation.

condition()
check convergence condition

control_end_wc(*errormsg*)
Controlled way to shutdown the workchain. will initialize the output nodes The shutdown of the workchain will has to be done afterwards

fleurinpgen_needed()
Returns True if inpgen calculation has to be submitted before fleur calculations

get_res()
Check how the last Fleur calculation went Parse some results.

inspect_fleur()
Analyse the results of the previous Calculation (Fleur or inpgen), checking whether it finished successfully or if not, troubleshoot the cause and adapt the input parameters accordingly before restarting, or abort if unrecoverable error was found

return_results()
return the results of the calculations This should run through and produce output nodes even if everything failed, therefore it only uses results from context.

run_fleur()
run a FLEUR calculation

run_fleurinpgen()
run the inpgen

start()
init context and some parameters

validate_input()
validate input and find out which path (1, or 2) to take # return True means run inpgen if false run fleur directly

aiida_fleur.workflows.scf.**create_scf_result_node**(***kwargs*)
This is a pseudo wf, to create the right graph structure of AiiDA. This wokfunction will create the output node in the database. It also connects the output_node to all nodes the information commes from. So far it is just also parsed in as argument, because so far we are to lazy to put most of the code overworked from return_results in here.

5.1.4.3 BandDos: Bandstructure WorkChain

This is the worklfow 'band' for the Fleur code, which calculates a electron bandstructure.

class aiida_fleur.workflows.banddos.**FleurBandDosWorkChain**(*inputs=None*, *logger=None*, *runner=None*, *enable_persistence=True*)
This workflow calculated a bandstructure from a Fleur calculation

Params a Fleurcalculation node

Returns Success, last result node, list with convergence behavior

control_end_wc(*errormsg*)
Controlled way to shutdown the workchain. will initialize the output nodes The shutdown of the workchain will has to be done afterwards

converge_scf()
Converge charge density.

create_new_fleurinp()
create a new fleurinp from the old with certain parameters

get_inputs_scf()
Initialize inputs for scf workflow: wf_param, options, calculation parameters, codes, structure

return_results()
return the results of the calculations

run_fleur()
run a FLEUR calculation

scf_needed()
Returns True if SCF WC is needed.

start()
check parameters, what condictions? complete? check input nodes

aiida_fleur.workflows.banddos.**create_band_result_node**(***kwargs*)
This is a pseudo wf, to create the right graph structure of AiiDA. This wokfunction will create the output node in the database. It also connects the output_node to all nodes the information commes from. So far it is just also parsed in as argument, because so far we are to lazy to put most of the code overworked from return_results in here.

5.1.4.4 DOS: Density of states WorkChain

This is the worklfow 'dos' for the Fleur code, which calculates a density of states (DOS).

class aiida_fleur.workflows.dos.**fleur_dos_wc**(*inputs=None*, *logger=None*, *runner=None*, *enable_persistence=True*)
This workflow calculated a DOS from a Fleur calculation

Params a Fleurcalculation node

Returns Success, last result node, list with convergence behavior

wf_parameters: { 'tria', 'nkpts', 'sigma', 'emin', 'emax'} defaults : tria = True, nkpts = 800, sigma=0.005, emin= -0.3, emax = 0.8

create_new_fleurinp()
create a new fleurinp from the old with certain parameters

return_results()
return the results of the calculations

run_fleur()
run a FLEUR calculation

start()
check parameters, what condictions? complete? check input nodes

5.1.4.5 EOS: Calculate a lattice constant

In this module you find the workflow 'FleurEosWorkChain' for the calculation of of an equation of state

class aiida_fleur.workflows.eos.**FleurEosWorkChain**(*inputs=None*, *logger=None*, *runner=None*, *enable_persistence=True*)
This workflow calculates the equation of states of a structure. Calculates several unit cells with different volumes. A Birch_Murnaghan equation of states fit determines the Bulk modulus and the groundstate volume of the cell.

Params wf_parameters Dict node, optional 'wf_parameters', protocol specifying parameter dict

Params structure StructureData node, 'structure' crystal structure

Params calc_parameters Dict node, optional 'calc_parameters' parameters for inpgen

Params inpgen Code node,

Params fleur Code node,

Return output_eos_wc_para Dict node, contains relevant output information. about general succeed, fit results and so on.

control_end_wc(*errormsg*)
Controlled way to shutdown the workchain. It will initialize the output nodes The shutdown of the workchain will has to be done afterwards

converge_scf()
Launch fleur_scfs from the generated structures

get_inputs_scf()
get and 'produce' the inputs for a scf-cycle

return_results()
return the results of the calculations (scf workchains) and do a Birch-Murnaghan fit for the equation of states

start()
check parameters, what condictions? complete? check input nodes

structures()
Creates structure data nodes with different Volume (lattice constants)

aiida_fleur.workflows.eos.**birch_murnaghan**(*volumes*, *volume0*, *bulk_modulus0*, *bulk_deriv0*)
This evaluates the Birch Murnaghan equation of states

aiida_fleur.workflows.eos.**birch_murnaghan_fit**(*energies*, *volumes*)
least squares fit of a Birch-Murnaghan equation of state curve. From delta project containing in its columns the volumes in A^3/atom and energies in eV/atom # The following code is based on the source code of eos.py from the Atomic # Simulation Environment (ASE) <https://wiki.fysik.dtu.dk/ase/>. :params energies: list (numpy arrays!) of total energies eV/atom :params volumes: list (numpy arrays!) of volumes in A^3/atom

#volume, bulk_modulus, bulk_deriv, residuals = Birch_Murnaghan_fit(data)

aiida_fleur.workflows.eos.**create_eos_result_node**(***kwargs*)
This is a pseudo cf, to create the right graph structure of AiiDA. This calcfunction will create the output nodes in the database. It also connects the output_nodes to all nodes the information comes from. This includes the output_parameter node for the eos, connections to run scfs, and returning of the gs_structure (best scale) So far it is just parsed in as kwargs argument, because we are to lazy to put most of the code overworked from return_results in here.

aiida_fleur.workflows.eos.**eos_structures**(*inp_structure*, *scalelist*)
Calcfunction, which creates many rescaled StructureData nodes out of a given crystal structure. Keeps the provenance in the database

:param StructureData, a StructureData node :param scalelist, AiiDA List, list of floats, scaling factors for the cell

Returns dict of New StructureData nodes with rescalled structure, which are linked to input Structure

aiida_fleur.workflows.eos.**eos_structures_nocf**(*inp_structure*, *scalelist*)
Creates many rescalled StructureData nodes out of a crystal structure. Does NOT keep the provenance in the database.

:param StructureData, a StructureData node (pk, sor uuid) :param scalelist, list of floats, scaling factors for the cell

Returns dict of New StructureData nodes with rescalled structure, key=scale

5.1.4.6 Relax: Relaxation of a Cystalstructure WorkChain

In this module you find the workflow 'FleurRelaxWorkChain' for geometry optimization.

class aiida_fleur.workflows.relax.**FleurRelaxWorkChain**(*inputs=None*, *logger=None*, *runner=None*, *enable_persistence=True*)

This workflow performs structure optimization.

static analyse_relax(*relax_dict*)

This function generates a new fleurinp analysing parsed relax.xml from the previous calculation.

NOT IMPLEMENTED YET

Parameters **relax_dict** – parsed relax.xml from the previous calculation

Return new_fleurinp new FleurinpData object that will be used for next relax iteration

check_failure()

Throws an exit code if scf failed

condition()

Checks if relaxation criteria is achieved.

Returns True if structure is optimized and False otherwise

control_end_wc(*errormsg*)

Controlled way to shutdown the workchain. It will initialize the output nodes The shutdown of the workchain will has to be done afterwards.

converge_scf()

Submits *aiida_fleur.workflows.scf.FleurScfWorkChain*.

generate_new_fleurinp()

This function fetches relax.xml from the previous iteration and calls *analyse_relax()*. New FleurinpData is stored in the context.

get_inputs_final_scf()

Initializes inputs for final scf on relaxed structure.

get_inputs_first_scf()

Initialize inputs for the first iteration.

get_inputs_scf()

Initializes inputs for further iterations.

get_results_final_scf()

Parser some results of final scf

get_results_relax()

Generates results of the workchain. Creates a new structure data node which is an optimized structure.

return_results()

This function stores results of the workchain into the output nodes.

run_final_scf()

Run a final scf for charge convergence on the optimized structure

should_relax()
Should we run a relaxation or only a final scf This allows to call the workchain to run an scf only and makes logic of other higher workflows a lot easier

should_run_final_scf()
Check if a final scf should be run on the optimized structure

start()
Retrieve and initialize paramters of the WorkChain, validate inputs

aiida_fleur.workflows.relax.**create_relax_result_node**(***kwargs*)
This calcfunction assures the right provenance (additional links) for ALL result nodes it takes any nodes as input and return a special set of nodes. All other inputs will be connected in the DB to these ourput nodes

5.1.4.7 initial_cls: Caluclation of inital corelevel shifts

This is the worklfow 'initial_cls' using the Fleur code calculating corelevel shifts with different methods.

aiida_fleur.workflows.initial_cls.**clshifts_to_be**(*coreleveldict*, *reference_dict*)
This methods converts corelevel shifts to binding energies, if a reference is given. These can than be used for plotting.

Example:

```
reference = {'W' : {'4f7/2' : [124],
                    '4f5/2' : [102]},
             'Be' : {'1s': [117]}}
corelevels = {'W' : {'4f7/2' : [0.4, 0.3, 0.4 ,0.1],
                     '4f5/2' : [0, 0.3, 0.4, 0.1]},
              'Be' : {'1s': [0, 0.2, 0.4, 0.1, 0.3]}}
```

aiida_fleur.workflows.initial_cls.**create_initcls_result_node**(***kwargs*)
This is a pseudo wf, to create the rigth graph structure of AiiDA. This wokfunction will create the output node in the database. It also connects the output_node to all nodes the information commes from. So far it is just also parsed in as argument, because so far we are to lazy to put most of the code overworked from return_results in here.

aiida_fleur.workflows.initial_cls.**extract_results**(*calcs*)
Collect results from certain calculation, check if everything is fine, calculate the wanted quantities.

params: calcs : list of scf workchains nodes

aiida_fleur.workflows.initial_cls.**fleur_calc_get_structure**(*calc_node*)
Get the AiiDA data structure from a fleur calculations

class aiida_fleur.workflows.initial_cls.**fleur_initial_cls_wc**(*inputs=None*, *logger=None*, *runner=None*, *enable_persistence=True*)
Turn key solution for the calculation of core level shift

check_input()
Init same context and check what input is given if it makes sence

collect_results()
Collect results from certain calculation, check if everything is fine, calculate the wanted quantities. currently all energies are in hartree (as provided by Fleur)

control_end_wc(*errormsg*)
Controled way to shutdown the workchain. will initalize the output nodes The shutdown of the workchain will has to be done afterwards

find_parameters()
If the same parameters shall be used in the calculations you have to find some that match. For low error on CLS. therefore use the ones enforced or extract from the previous Fleur calculation.

get_references()
To calculate a CLS in initial state approx, we need reference calculations to the Elemental crystals. First it is checked if the user has provided them Second the database is checked, if there are structures with certain extras. Third the COD database is searched for the elemental Cystal structures. If some referneces are not found stop here. Are there already calculation of these 'references', ggf use them. We do not put these calculation in the calculation queue yet because we need specific parameters for them

handle_scf_failure()
In here we handle all failures from the scf workchain

relax()
Do structural relaxation for certain structures.

relaxation_needed()
If the structures should be relaxed, check if their Forces are below a certain threshold, otherwise throw them in the relaxation wf.

return_results()
return the results of the calculations

run_fleur_scfs()
Run SCF-cycles for all structures, calculations given in certain workflow arrays.

run_scfs_ref()
Run SCF-cycles for ref structures, calculations given in certain workflow arrays. parameter nodes should be given

aiida_fleur.workflows.initial_cls.**get_para_from_group**(*element*, *group*)
get structure node for a given element from a given group of structures (quit creedy, done straighforward)

aiida_fleur.workflows.initial_cls.**get_ref_from_group**(*element*, *group*)
Return a structure data node from a given group for a given element. (quit creedy, done straighforward)

params: group: group name or pk params: element: string with the element i.e 'Si'

returns: AiiDA StructureData node

aiida_fleur.workflows.initial_cls.**querry_for_ref_structure**(*element_string*)
This methods finds StructureData nodes with the following extras: extra.type = 'bulk', # Should be done by looking at pbc, but I could not get querry to work. extra.specific = 'reference', 'extra.elemental' = True, extra.structure = element_string

param: element_string: string of an element return: the latest StructureData node that was found

5.1.4.8 corehole: Performance of coreholes calculations

This is the workflow 'corehole' using the Fleur code, which calculates binding energies and corelevel shifts with different methods. 'divide and conquer'

aiida_fleur.workflows.corehole.**create_corehole_result_node**(***kwargs*)
This is a pseudo wf, to create the rigth graph structure of AiiDA. This wokfunction will create the output node in the database. It also connects the output_node to all nodes the information commes from. So far it is just also

parsed in as argument, because so far we are to lazy to put most of the code overworked from return_results in here.

aiida_fleur.workflows.corehole.**extract_results_corehole**(*calcs*)
Collect results from certain calculation, check if everything is fine, calculate the wanted quantities.

params: calcs : list of scf workchains nodes

class aiida_fleur.workflows.corehole.**fleur_corehole_wc**(*inputs=None*, *logger=None*, *runner=None*, *enable_persistence=True*)
Turn key solution for a corehole calculation with the FLEUR code. Has different protocols for different corehole types (valence, charge).

Calculates supercells. Extracts binding energies for certain corelevels from the total energy differences a the calculation with corehole and without.

Documentation: See help for details.

Two paths are possible:

(1) Start from a structure -> workchains run inpgen first (recommended)

(2) Start from a Fleurinp data object

Also it is recommended to provide a calc parameter node for the structure

Parameters

- **wf_parameters** – Dict node, specify, resources and what should be calculated
- **structure** – structureData node, crystal structure
- **calc_parameters** – Dict node, inpgen parameters for the crystal structure
- **fleurinp** – fleurinpData node,
- **inpgen** – Code node,
- **fleur** – Code node,

Returns output_corehole_wc_para Dict node, successful=True if no error

Uses workchains fleur_scf_wc, fleur_relax_wc

Uses calcfunctions supercell, create_corehole_result_node, prepare_struc_corehole_wf

check_input()
init all context parameters, variables. Do some input checks. Further input checks are done in further workflow steps

check_scf()
Check if ref scf was successful, or something needs to be dealt with. If unsuccesful abort, because makes no sense to continue.

collect_results()
Collect results from certain calculation, check if everything is fine, calculate the wanted quantities. currently all energies are in hartree (as provided by Fleur)

control_end_wc(*errormsg*)
Controled way to shutdown the workchain. report errors and always initalize/produce output nodes. But log successful=False

create_coreholes()
Check the input for the corelevel specification, create structure and parameter nodes with all the need coreholes. create the wf_parameter nodes for the scfs. Add all calculations to scfs_to_run.

Layout: # Check what coreholes should be created. # said in the input, look in the original cell # These positions are the same for the supercell. # break the symmetry for the supercells. (make the corehole atoms its own atom type) # create a new species and a corehole for this atom group. # move all the atoms in the cell that impurity is in the origin (0.0, 0.0, 0.0) # use the fleurinp_change feature of scf to create the corehole after inpgen gen in the scf # start the scf with the last charge density of the ref calc? so far no, might not make sense

TODO if this becomes to long split

create_supercell()
create the needed supercell

relax()
Do structural relaxation for certain structures.

relaxation_needed()
If the structures should be relaxed, check if their Forces are below a certain threshold, otherwise throw them in the relaxation wf.

return_results()
return the results of the calculations

run_ref_scf()
Run a scf for the reference super cell

run_scfs()
Run a scf for the all corehole calculations in parallel super cell

supercell_needed()
check if a supercell is needed and what size it should be

aiida_fleur.workflows.corehole.**prepare_struc_corehole_wf**(*base_supercell*, *wf_para*, *para=None*)
calcfunction which does all/some the structure+calcparameter manipulations together (therefore less nodes are produced and proverance is kept) wf_para: Dict node dict: {'site' : sites[8], 'kindname' : 'W1', 'econfig': "[Kr] 5s2 4d10 4f13 | 5p6 5d5 6s2", 'fleurinp_change' : []}

5.1.4.9 MAE: Force-theorem calculation of magnetic anisotropy energies

In this module you find the workflow 'FleurMaeWorkChain' for the calculation of Magnetic Anisotropy Energy via the force theorem.

class aiida_fleur.workflows.mae.**FleurMaeWorkChain**(*inputs=None*, *logger=None*, *runner=None*, *enable_persistence=True*)
This workflow calculates the Magnetic Anisotropy Energy of a structure.

change_fleurinp()
This routine sets somethings in the fleurinp file before running a fleur calculation.

control_end_wc(*errormsg*)
Controlled way to shutdown the workchain. will initialize the output nodes The shutdown of the workchain will has to be done afterwards

converge_scf()
Converge charge density with or without SOC. Submit a single Fleur calculation to obtain a reference for further force theorem calculations.

force_after_scf()
Calculate energy of a system for given SQAs using the force theorem. Converged reference is stored in self.ctx['xyz'].

force_wo_scf()
Submit FLEUR force theorem calculation using input remote

get_inputs_scf()
Initialize inputs for scf workflow: wf_param, options, calculation parameters, codes, structure

get_results()
Generates results of the workchain.

return_results()
This function outputs results of the wc

scf_needed()
Returns True if SCF WC is needed.

start()
Retrieve and initialize paramters of the WorkChain

aiida_fleur.workflows.mae.**save_mae_output_node**(***kwargs*)
This is a pseudo cf, to create the right graph structure of AiiDA. This calcfunction will create the output node in the database. It also connects the output_node to all nodes the information comes from. So far it is just also parsed in as argument, because so far we are to lazy to put most of the code overworked from return_results in here.

5.1.4.10 MAE Conv: Self-consistent calculation of magnetic anisotropy energies

In this module you find the workflow 'FleurMAEWorkChain' for the calculation of Magnetic Anisotropy Energy converging all the directions.

class aiida_fleur.workflows.mae_conv.**FleurMaeConvWorkChain**(*inputs=None*, *logger=None*, *runner=None*, *enable_persistence=True*)
This workflow calculates the Magnetic Anisotropy Energy of a structure.

control_end_wc(*errormsg*)
Controlled way to shutdown the workchain. will initialize the output nodes The shutdown of the workchain will has to be done afterwards

converge_scf()
Converge charge density with or without SOC. Depending on a branch of MAE calculation, submit a single Fleur calculation to obtain a reference for further force theorem calculations or submit a set of Fleur calculations to converge charge density for all given SQAs.

get_inputs_scf()
Initialize inputs for scf workflow

get_results()
Retrieve results of converge calculations

return_results()
Retrieve results of converge calculations

start()
Retrieve and initialize paramters of the WorkChain

aiida_fleur.workflows.mae_conv.**save_output_node**(*out*)
This calcfunction saves the out dict in the db

5.1.4.11 SSDisp: Force-theorem calculation of spin spiral dispersion

In this module you find the workflow 'FleurSSDispWorkChain' for the calculation of spin spiral dispersion using scalar-relatevistic Hamiltonian.

class aiida_fleur.workflows.ssdisp.**FleurSSDispWorkChain**(*inputs=None*, *logger=None*, *runner=None*, *enable_persistence=True*)

This workflow calculates spin spiral dispersion of a structure.

change_fleurinp()
This routine sets somethings in the fleurinp file before running a fleur calculation.

control_end_wc(*errormsg*)
Controlled way to shutdown the workchain. It will initialize the output nodes The shutdown of the workchain will has to be done afterwards

converge_scf()
Converge charge density for collinear case which is a reference for futher spin spiral calculations.

force_after_scf()
This routine uses the force theorem to calculate energies dispersion of spin spirals. The force theorem calculations implemented into the FLEUR code. Hence a single iteration FLEUR input file having <forceTheorem> tag has to be created and submitted.

force_wo_scf()
Submit FLEUR force theorem calculation using input remote

get_inputs_scf()
Initialize inputs for the scf cycle

get_results()
Generates results of the workchain.

return_results()
This function outputs results of the wc

scf_needed()
Returns True if SCF WC is needed.

start()
Retrieve and initialize paramters of the WorkChain

aiida_fleur.workflows.ssdisp.**save_output_node**(*out*)
This calcfunction saves the out dict in the db

5.1.4.12 SSDisp Conv: Self-consistent calculation of spin spiral dispersion

In this module you find the workflow 'FleurSSDispConvWorkChain' for the calculation of Spin Spiral energy Dispersion converging all the directions.

class aiida_fleur.workflows.ssdisp_conv.**FleurSSDispConvWorkChain**(*inputs=None*, *logger=None*, *runner=None*, *enable_persistence=True*)

This workflow calculates the Spin Spiral Dispersion of a structure.

control_end_wc(*errormsg*)
Controlled way to shutdown the workchain. will initialize the output nodes The shutdown of the workchain will has to be done afterwards

converge_scf()
Converge charge density with or without SOC. Depending on a branch of Spiral calculation, submit a single Fleur calculation to obtain a reference for further force theorem calculations or submit a set of Fleur calculations to converge charge density for all given SQAs.

get_inputs_scf()
Initialize inputs for scf workflow: wf_param, options, calculation parameters, codes, structure

get_results()
Retrieve results of converge calculations

return_results()
Retrieve results of converge calculations

start()
Retrieve and initialize paramters of the WorkChain

aiida_fleur.workflows.ssdisp_conv.**save_output_node**(*out*)
This calcfunction saves the out dict in the db

5.1.4.13 DMI: Force-theorem calculation of Dzjaloshinskii-Moriya interaction energy dispersion

In this module you find the workflow 'FleurDMIWorkChain' for the calculation of DMI energy dispersion.

class aiida_fleur.workflows.dmi.**FleurDMIWorkChain**(*inputs=None*, *logger=None*, *runner=None*, *enable_persistence=True*)
This workflow calculates DMI energy dispersion of a structure.

change_fleurinp()
This routine sets somethings in the fleurinp file before running a fleur calculation.

control_end_wc(*errormsg*)
Controlled way to shutdown the workchain. will initialize the output nodes The shutdown of the workchain will has to be done afterwards

converge_scf()
Converge charge density for collinear case which is a reference for futher spin spiral calculations.

force_after_scf()
This routine uses the force theorem to calculate energies dispersion of spin spirals. The force theorem calculations implemented into the FLEUR code. Hence a single iteration FLEUR input file having <forceTheorem> tag has to be created and submitted.

force_wo_scf()
Submit FLEUR force theorem calculation using input remote

get_inputs_scf()
Initialize inputs for the scf cycle

get_results()
Generates results of the workchain.

return_results()
This function outputs results of the wc

scf_needed()
Returns True if SCF WC is needed.

start()
Retrieve and initialize paramters of the WorkChain

aiida_fleur.workflows.dmi.**save_output_node**(*out*)
This calcfunction saves the out dict in the db

5.1.5 Fleur tools/utility

5.1.5.1 Dealing with XML Schema files

This file is just were to hardcode some schema file paths

5.1.5.2 Structure Data util

Collection of utility routines dealing with StructureData objects

aiida_fleur.tools.StructureData_util.**abs_to_rel**(*vector*, *cell*)
Converts a position vector in absolute coordinates to relative coordinates.

Parameters

- **vector** – list or np.array of length 3, vector to be converted
- **cell** – Bravais matrix of a crystal 3x3 Array, List of list or np.array

Returns list of legth 3 of scaled vector, or False if vector was not length 3

aiida_fleur.tools.StructureData_util.**abs_to_rel_f**(*vector*, *cell*, *pbc*)
Converts a position vector in absolute coordinates to relative coordinates for a film system.

Parameters

- **vector** – list or np.array of length 3, vector to be converted
- **cell** – Bravais matrix of a crystal 3x3 Array, List of list or np.array
- **pbc** – Boundary conditions, List or Tuple of 3 Boolean

Returns list of legth 3 of scaled vector, or False if vector was not length 3

aiida_fleur.tools.StructureData_util.**adjust_calc_para_to_structure**(*parameter*, *structure*, *add_atom_base_lists=True*, *write_new_kind_names=False*)
Adjust calculation parameters for inpgen to a given structure with several kinds

Rules: 1. Only atom lists are changed in the parameter node 2. If at least one atomlist of a certain element is in parameter all kinds with this elements will have atomlists in the end 3. For a certain kind which has no atom list yet and at least one list with such an element exists it gets the parameters from the atom list with the lowest number (while atom<atom0<atom1) 4. Atom lists with ids are preserved

Parameters

- **parameter** – aiida.orm.Dict node containing calc parameters
- **structure** – aiida.orm.StructureData node containing a crystal structure
- **add_atom_base_lists** – Bool (default True), if the atom base lists should be added or not

Returns new aiida.orm.Dict with new calc_parameters

aiida_fleur.tools.StructureData_util.**adjust_film_relaxation**(*structure*, *suggestion*, *scale_as=None*, *bond_length=None*, *hold_layers=3*)

Tries to optimize interlayer distances. Can be used before RelaxWC to improve its behaviour. This function only works if USER_API_KEY was set.

For now only binary structures are analysed to ensure the closest contact between two elements of the interest. In case of trinary systems (like ABC) I can not not guarantee that A and C will be the nearest neighbours.

The same is true for interlayer distances of the same element. To ensure the nearest-neighbour condition I use unary compounds.

Parameters

- **structure** – ase film structure which will be adjusted
- **suggestion** – dictionary containing average bond length between different elements, is is basically the result of *request_average_bond_length()*
- **scale_as** – an element name, for which the El-El bond length will be enforced. It is can be helpful to enforce the same interlayer distance in the substrate, i.e. adjust deposited film interlayer distances only.
- **bond_length** – a float that sets the bond length for scale_as element
- **hold_layers** – this parameters sets the number of layers that will be marked via the certain label. The label is reserved for future use in the relaxation WC: all the atoms marked with the label will not be relaxed.

aiida_fleur.tools.StructureData_util.**break_symmetry**(*structure*, *atoms=None*, *site=None*, *pos=None*, *new_kinds_names=None*, *add_atom_base_lists=True*, *parameterdata=None*)

This routine introduces different 'kind objects' in a structure and names them that inpgen will make different species/atomgroups out of them. If nothing specified breaks ALL symmetry (i.e. every atom gets their own kind)

Parameters

- **structure** – StructureData
- **atoms** – python list of symbols, exp: ['W', 'Be']. This would make for all Be and W atoms their own kinds.
- **site** – python list of integers, exp: [1, 4, 8]. This would create for atom 1, 4 and 8 their own kinds.
- **pos** – python list of tuples of 3, exp [(0.0, 0.0, -1.837927), ...]. This will create a new kind for the atom at that position. Be carefull the number given has to match EXACTLY the position in the structure.
- **parameterdata** – Dict node, containing calculation_parameters, however, this only works well if you prepare already a node for containing the atom lists from the symmetry breaking, or lists without ids.
- **add_atom_base_lists** – Bool (default True), if the atom base lists should be added or not

Returns StructureData, a AiiDA crystal structure with new kind specification.

Returns DictData, a AiiDA dict with new parameters for inpgen.

aiida_fleur.tools.StructureData_util.**break_symmetry_wf**(*structure*, *wf_para*, *parameterdata=None*)

This is the calcfunction of the routine break_symmetry, which introduces different 'kind objects' in a structure and names them that inpgen will make different species/atomgroups out of them. If nothing specified breaks ALL symmetry (i.e. every atom gets their own kind)

Parameters

- **structure** – StructureData
- **wf_para** – ParameterData which contains the keys atoms, sites, pos (see below)

  'atoms': python list of symbols, exp: ['W', 'Be']. This would make for all Be and W atoms their own kinds.

  'site': python list of integers, exp: [1, 4, 8]. This would create for atom 1, 4 and 8 their own kinds.

  'pos': python list of tuples of 3, exp [(0.0, 0.0, -1.837927), ...]. This will create a new kind for the atom at that position. Be carefull the number given has to match EXACTLY the position in the structure.
- **parameterdata** – AiiDa ParameterData

Returns StructureData, a AiiDA crystal structure with new kind specification.

aiida_fleur.tools.StructureData_util.**center_film**(*structure*)

Centers a film at z=0

Parameters **structure** – AiiDA structure

Returns AiiDA structure

aiida_fleur.tools.StructureData_util.**center_film_wf**(*structure*)

Centers a film at z=0, keeps the provenance in the database

Parameters **structure** – AiiDA structure

Returns AiiDA structure

aiida_fleur.tools.StructureData_util.**check_structure_para_consistent**(*parameter*, *structure*, *verbose=True*)

Check if the given calculation parameters for inpgen match to a given structure

If parameter contains atom lists which do not fit to any kind in the structure, false is returned This knows how the FleurinputgenCalculation prepares structures.

Parameters

- **parameter** – aiida.orm.Dict node containing calc parameters
- **structure** – aiida.orm.StructureData node containing a crystal structure

Returns Boolean, True if parameter is consistent to structure

aiida_fleur.tools.StructureData_util.**create_all_slabs**(*initial_structure*, *miller_index*, *min_slab_size_ang*, *min_vacuum_size=0*, *bonds=None*, *tol=0.001*, *max_broken_bonds=0*, *lll_reduce=False*, *center_slab=False*, *primitive=False*, *max_normal_search=1*, *symmetrize=False*)

Returns a dictionary of structures

aiida_fleur.tools.StructureData_util.**create_manual_slab_ase**(*lattice='fcc'*, *miller=None*, *host_symbol='Fe'*, *latticeconstant=4.0*, *size=(1, 1, 5)*, *replacements=None*, *decimals=10*, *pop_last_layers=0*, *inverse=False*)

Wraps ase.lattice lattices generators to create a slab having given lattice vectors directions.

Parameters

- **lattice** – 'fcc' and 'bcc' are supported. Set the host lattice of a slab.
- **miller** – a list of directions of lattice vectors
- **symbol** – a string specifying the atom type
- **latticeconstant** – the lattice constant of a structure
- **size** – a 3-element tuple that sets supercell size. For instance, use (1,1,5) to set 5 layers of a slab.
- **decimals** – sets the rounding of atom positions. See numpy.around.
- **pop_last_layers** – specifies how many bottom layers to remove. Sometimes one does not want to use the integer number of unit cells along z, extra layers can be removed.

Return structure an ase-lattice representing a slab with replaced atoms

aiida_fleur.tools.StructureData_util.**create_slap**(*initial_structure*, *miller_index*, *min_slab_size*, *min_vacuum_size=0*, *lll_reduce=False*, *center_slab=False*, *primitive=False*, *max_normal_search=1*, *reorient_lattice=True*)

wraps the pymatgen slab generator

aiida_fleur.tools.StructureData_util.**find_equi_atoms**(*structure*)

This routine uses spglib and ASE to provide informations of all equivivalent atoms in the cell.

Parameters **structure** – AiiDA StructureData

Returns equi_info_symbol, list of lists ['element': site_indexlist, ...] len(equi_info_symbol) = number of symmetryatomtypes and n_equi_info_symbol, dict {'element': numberequiatomstypes}

aiida_fleur.tools.StructureData_util.**find_primitive_cell**(*structure*)
uses spglib find_primitive to find the primitive cell

Parameters **sructure** – AiiDA structure data

Returns list of new AiiDA structure data

aiida_fleur.tools.StructureData_util.**find_primitive_cell_wf**(*structure*)
uses spglib find_primitive to find the primitive cell :param structure: AiiDa structure data

Returns list of new AiiDa structure data

aiida_fleur.tools.StructureData_util.**find_primitive_cells**(*uuid_list*)
uses spglib find_primitive to find the primitive cell :param uuid_list: list of structureData uuids, or pks

Returns list of new AiiDa structure datas

aiida_fleur.tools.StructureData_util.**get_all_miller_indices**(*structure*, *highestindex*)
wraps the pymatgen function get_symmetrically_distinct_miller_indices for an AiiDa structure

aiida_fleur.tools.StructureData_util.**get_layers**(*structure*, *decimals=10*)
Extracts atom positions and their types belonging to the same layer

Parameters

- **structure** – ase lattice or StructureData which represents a slab
- **number** – the layer number. Note, that layers will be sorted according to z-position
- **decimals** – sets the tolerance of atom positions determination. See more in numpy.around.

Return layer, layer_z_positions layer is a list of tuples, the first element of which is atom positions and the second one is atom type. layer_z_position is a sorted list of all layer positions

aiida_fleur.tools.StructureData_util.**get_spacegroup**(*structure*)

Parameters **structure** – AiiDA StructureData

Returns the spacegroup (spglib class) of a given AiiDA structure

aiida_fleur.tools.StructureData_util.**is_primitive**(*structure*)
Checks if a structure is primitive or not, :param structure: AiiDA StructureData :return: True if the structure can not be anymore refined. prints False if the structure can be futher refined.

aiida_fleur.tools.StructureData_util.**is_structure**(*structure*)
Test if the given input is a StructureData node, by object, id, or pk :param structure: AiiDA StructureData :return: if yes returns a StructureData node in all cases, if no returns None

aiida_fleur.tools.StructureData_util.**magnetic_slab_from_relaxed**(*relaxed_structure*, *orig_structure*, *total_number_layers*, *num_relaxed_layers*, *tolerance_decimals=10*)
Transforms a structure that was used for interlayer distance relaxation to a structure that can be further used for magnetic calculations.

Usually one uses a slab having z-reflection symmetry e.g. A-B1-B2-B3-B2-B1-A where A is a magnetic element (Fe, Ni, Co, Cr) and B is a substrate. However, further magnetic calculations are done using assymetric slab A-B1-B2-B3-B4-B5-B6-B7-B8. The function uses A-B1, B1-B2 etc. iterlayer distances for constraction of assymetric relaxed film.

The function works as follows: it constructs a new StructureData object taking x and y positions from the orig_structure and z positions from relax_structure for first num_relaxed_interlayers. Then it appends orig_structure slab to the bottom it a way the total number of layers is total_number_layers.

Parameters

- **relaxed_structure** – Structure which is the output of Relax WorkChain. In thin function it is assumed to have inversion or at least z-reflection symmetry.
- **orig_structure** – The host structure slab having the lattice perioud corresponding to the bulk structure of the substrate.
- **total_number_layers** – the total number of layers to produce
- **num_relaxed_layers** – the number of top layers to adjust according to **relaxed_struct**
- **tolerance_decimals** – sets the rounding of atom positions. See numpy.around.

Return magn_structure Resulting assymetric structure with adjusted interlayer distances for several top layers.

aiida_fleur.tools.StructureData_util.**move_atoms_incell**(*structure*, *vector*)
moves all atoms in a unit cell by a given vector

Parameters

- **structure** – AiiDA structure
- **vector** – tuple of 3, or array

Returns AiiDA structure

aiida_fleur.tools.StructureData_util.**move_atoms_incell_wf**(*structure*, *wf_para*)
moves all atoms in a unit cell by a given vector

Parameters

- **structure** – AiiDA structure
- **wf_para** – AiiDA Dict node with vector: tuple of 3, or array (currently 3 AiiDA Floats to make it a wf, In the future maybe a list or vector if AiiDa basetype exists)

Returns AiiDA stucture

aiida_fleur.tools.StructureData_util.**rel_to_abs**(*vector*, *cell*)
Converts a position vector in internal coordinates to absolute coordinates in Angstrom.

Parameters

- **vector** – list or np.array of length 3, vector to be converted
- **cell** – Bravais matrix of a crystal 3x3 Array, List of list or np.array

Returns list of legth 3 of scaled vector, or False if vector was not lenth 3

aiida_fleur.tools.StructureData_util.**rel_to_abs_f**(*vector*, *cell*)
Converts a position vector in internal coordinates to absolute coordinates in Angstrom for a film structure (2D).

aiida_fleur.tools.StructureData_util.**request_average_bond_length**(*main_elements*, *sub_elements*, *user_api_key*)
Requests MaterialsProject to estimate thermal average bond length between given elements. Also requests information about lattice constants of fcc and bcc structures.

Parameters

- **main_elements** – element list to calculate the average bond length only combinations of AB, AA and BB are calculated, where A belongs to main_elements, B belongs to sub_elements.
- **sub_elements** – element list, see main_elements

Returns bond_data, a dict containing obtained lattice constants.

aiida_fleur.tools.StructureData_util.**request_average_bond_length_store**(*main_elements*, *sub_elements*, *user_api_key*)

Requests MaterialsProject to estimate thermal average bond length between given elements. Also requests information about lattice constants of fcc and bcc structures. Stores the result in the Database. Notice that this is not a calcfunction! Therefore, the inputs are not stored and the result node is unconnected.

Parameters

- **main_elements** – element list to calculate the average bond length only combinations of AB, AA and BB are calculated, where A belongs to main_elements, B belongs to sub_elements.
- **sub_elements** – element list, see main_elements

Returns bond_data, a dict containing obtained lattice constants.

aiida_fleur.tools.StructureData_util.**rescale**(*inp_structure*, *scale*)

Rescales a crystal structures Volume, atoms stay at their same relative postions, therefore the absolute postions change. Keeps the provenance in the database.

Parameters

- **inp_structure** – a StructureData node (pk, or uuid)
- **scale** – float scaling factor for the cell

Returns New StructureData node with rescalled structure, which is linked to input Structure and None if inp_structure was not a StructureData

aiida_fleur.tools.StructureData_util.**rescale_nowf**(*inp_structure*, *scale*)

Rescales a crystal structures Volume, atoms stay at their same relative postions, therefore the absolute postions change. DOES NOT keep the provenance in the database.

Parameters

- **inp_structure** – a StructureData node (pk, or uuid)
- **scale** – float scaling factor for the cell

Returns New StructureData node with rescalled structure, which is linked to input Structure and None if inp_structure was not a StructureData

aiida_fleur.tools.StructureData_util.**sort_atoms_z_value**(*structure*)

Resorts the atoms in a structure by there Z-value

Parameters **structure** – AiiDA structure

Returns AiiDA structure

aiida_fleur.tools.StructureData_util.**supercell**(*inp_structure*, *n_a1*, *n_a2*, *n_a3*)

Creates a super cell from a StructureData node. Keeps the provenance in the database.

Parameters

- **StructureData** – a StructureData node (pk, or uuid)
- **scale** – tuple of 3 AiiDA integers, number of cells in a1, a2, a3, or if cart =True in x,y,z

Returns StructureData Node with supercell

aiida_fleur.tools.StructureData_util.**supercell_ncf**(*inp_structure*, *n_a1*, *n_a2*, *n_a3*)
Creates a super cell from a StructureData node. Does NOT keeps the provenance in the database.

Parameters

- **StructureData** – a StructureData node (pk, or uuid)
- **scale** – tuple of 3 AiiDA integers, number of cells in a1, a2, a3, or if cart=True in x,y,z

Returns StructureData Node with supercell

5.1.5.3 XML utility

In this module contains useful methods for handling xml trees and files which are used by the Fleur code and the fleur plugin.

aiida_fleur.tools.xml_util.**add_num_to_att**(*xmltree*, *xpathn*, *attributename*, *set_val*, *mode='abs'*, *occ=None*)
Routine adds something to the value of an attribute in the xml file (should be a number here) This is a lower-level version of *shift_value()* which allows one to specife an arbitrary xml path.

Param an etree a xpath from root to the attribute and the attribute value

Parameters

- **xpathn** – an xml path to the attribute to change
- **attributename** – a name of the attribute to change
- **set_val** – a value to be added/multiplied to the previous value
- **mode** – 'abs' if to add set_val, 'rel' if multiply
- **occ** – a list of integers specifying number of occurrence to be set

Comment: Element.set will add the attribute if it does not exist, xpath expression has to exist

example: add_num_to_add(tree, '/fleurInput/bzIntegration', 'valenceElectrons', '1')
add_num_to_add(tree, '/fleurInput/bzIntegration', 'valenceElectrons', '1.1', mode='rel')

aiida_fleur.tools.xml_util.**change_atomgr_att**(*fleurinp_tree_copy*, *attributedict*, *position=None*, *species=None*)
Method to set parameters of an atom group of the fleur inp.xml file.

Parameters

- **fleurinp_tree_copy** – xml etree of the inp.xml
- **attributedict** – a python dict specifying what you want to change.
- **position** – position of an atom group to be changed. If equals to 'all', all species will be changed
- **species** – atom groups, corresponding to the given specie will be changed
- **create** – bool, if species does not exist create it and all subtags?

Return fleurinp_tree_copy xml etree of the new inp.xml

attributedict is a python dictionary containing dictionaries that specify attributes to be set inside the certain specie. For example, if one wants to set a beta noco parameter it can be done via:

```
'attributedict': {'nocoParams': [('beta', val)]}
```

`force` and `nocoParams` keys are supported. To find possible keys of the inner dictionary please refer to the FLEUR documentation flapw.de

aiida_fleur.tools.xml_util.**change_atomgr_att_label**(*fleurinp_tree_copy*, *attributedict*, *at_label*)

This method calls *change_atomgr_att()* method for a certain atom specie that corresponds to an atom with a given label.

Parameters

- **fleurinp_tree_copy** – xml etree of the inp.xml
- **at_label** – string, a label of the atom which specie will be changed. 'all' to change all the species
- **attributedict** – a python dict specifying what you want to change.

Return fleurinp_tree_copy xml etree of the new inp.xml

attributedict is a python dictionary containing dictionaries that specify attributes to be set inside the certain specie. For example, if one wants to set a beta noco parameter it can be done via:

```
'attributedict': {'nocoParams': [('beta', val)]}
```

`force` and `nocoParams` keys are supported. To find possible keys of the inner dictionary please refer to the FLEUR documentation flapw.de

aiida_fleur.tools.xml_util.**clear_xml**(*tree*)

Removes comments and executes xinclude tags of an xml tree.

Parameters **tree** – an xml-tree which will be processes

Return cleared_tree an xml-tree without comments and with replaced xinclude tags

aiida_fleur.tools.xml_util.**convert_ev_to_htr**(*value*, *parser_info_out=None*)

Divides the value given with the Hartree factor (converts htr to eV)

aiida_fleur.tools.xml_util.**convert_fleur_lo**(*loelements*)

Converts lo xml elements from the inp.xml file into a lo string for the inpgen

aiida_fleur.tools.xml_util.**convert_from_fortran_bool**(*stringbool*)

Converts a string in this case ('T', 'F', or 't', 'f') to True or False

Parameters **stringbool** – a string ('t', 'f', 'F', 'T')

Returns boolean (either True or False)

aiida_fleur.tools.xml_util.**convert_htr_to_ev**(*value*, *parser_info_out=None*)

Multiplies the value given with the Hartree factor (converts htr to eV)

aiida_fleur.tools.xml_util.**convert_to_float**(*value_string*, *parser_info_out=None*, *suc_return=True*)

Tries to make a float out of a string. If it can't it logs a warning and returns True or False if convertion worked or not.

Parameters **value_string** – a string

Returns value the new float or value_string: the string given

Returns True or False

aiida_fleur.tools.xml_util.**convert_to_fortran_bool**(*boolean*)

Converts a Boolean as string to the format defined in the input

Parameters **boolean** – either a boolean or a string ('True', 'False', 'F', 'T')

Returns a string (either 't' or 'f')

aiida_fleur.tools.xml_util.**convert_to_fortran_string**(*string*)
converts some parameter strings to the format for the inpgen :param string: some string :returns: string in right format (extra "")

aiida_fleur.tools.xml_util.**convert_to_int**(*value_string*, *parser_info_out=None*, *suc_return=True*)
Tries to make a int out of a string. If it can't it logs a warning and returns True or False if convertion worked or not.

Parameters **value_string** – a string

Returns value the new int or value_string: the string given

Returns True or False, if suc_return=True

aiida_fleur.tools.xml_util.**create_tag**(*xmlnode*, *xpath*, *newelement*, *create=False*, *place_index=None*, *tag_order=None*)
This method evaluates an xpath expresion and creates tag in an xmltree under the returned nodes. If the path does exist things will be overwritten, or created. Per default the new element is appended to the elements, but it can also be inserted in a certain position or after certain other tags.

Parameters

- **xmlnode** – an xmltree that represents inp.xml
- **xpathn** – a path where to place a new tag
- **newelement** – a tag name to be created
- **create** – if True and there is no given xpath in the FleurinpData, creates it
- **place_index** – defines the place where to put a created tag
- **tag_order** – defines a tag order

aiida_fleur.tools.xml_util.**delete_att**(*xmltree*, *xpath*, *attrib*)
Deletes an xml tag in an xmletree.

Parameters

- **xmltree** – an xmltree that represents inp.xml
- **xpathn** – a path to the attribute to be deleted
- **attrib** – the name of an attribute

aiida_fleur.tools.xml_util.**delete_tag**(*xmltree*, *xpath*)
Deletes an xml tag in an xmletree.

Parameters

- **xmltree** – an xmltree that represents inp.xml
- **xpathn** – a path to the tag to be deleted

aiida_fleur.tools.xml_util.**eval_xpath**(*node*, *xpath*, *parser_info=None*)
Tries to evalutate an xpath expression. If it fails it logs it. If seferal paths are found, return a list. If only one - returns the value.

:param root node of an etree and an xpath expression (relative, or absolute) :returns either nodes, or attributes, or text

aiida_fleur.tools.xml_util.**eval_xpath2**(*node*, *xpath*, *parser_info=None*)
Tries to evalutate an xpath expression. If it fails it logs it. Always return a list.

:param root node of an etree and an xpath expression (relative, or absolute) :returns a node list

aiida_fleur.tools.xml_util.**eval_xpath3**(*node*, *xpath*, *create=False*, *place_index=None*, *tag_order=None*)
Tries to evalutate an xpath expression. If it fails it logs it. If create == True, creates a tag

:param root node of an etree and an xpath expression (relative, or absolute) :returns always a node list

aiida_fleur.tools.xml_util.**get_inpgen_para_from_xml**(*inpxmlfile*, *inpgen_ready=True*)
This routine returns an python dictionary produced from the inp.xml file, which can be used as a calc_parameters node by inpgen. Be aware that inpgen does not take all information that is contained in an inp.xml file

Parameters

- **inpxmlfile** – and xml etree of a inp.xml file
- **inpgen_ready** – Bool, return a dict which can be inputed into inpgen while setting atoms

Return new_parameters A Dict, which will lead to the same inp.xml (in case if other defaults, which can not be controlled by input for inpgen, were changed)

aiida_fleur.tools.xml_util.**get_inpgen_paranode_from_xml**(*inpxmlfile*)
This routine returns an AiiDA Parameter Data type produced from the inp.xml file, which can be used by inpgen.

Returns ParameterData node

aiida_fleur.tools.xml_util.**get_inpxml_file_structure**()
This routine returns the structure/layout of the 'inp.xml' file.

Basicly the plug-in should know from this routine, what things are allowed to be set and where, i.e all attributes and their xpaths. As a developer make sure to use this routine always of you need information about the inp.xml file structure. Therefore, this plug-in should be easy to adjust to other codes with xml files as input files. Just rewrite this routine.

For now the structure of the xmlinp file for fleur is hardcoded. If big changes are in the 'inp.xml' file, maintain this routine. TODO: Maybe this is better done, by reading the xml schema datei instead. And maybe it should also work without the schema file, do we want this?

Parameters **Nothing** – TODO xml schema

Return all_switches_once list of all switches ('T' or 'F') which are allowed to be set

Return all_switches_several list of all switches ('T' or 'F') which are allowed to be set

Return other_attributes_once list of all attributes, which occur just once (can be tested)

Return other_attributes_several list of all attributes, which can occur more then once

Return all_text list of all text of tags, which can be set

Return all_attrib_xpath dictonary (attrib, xpath), of all possible attributes with their xpath expression for the xmp inp

Return expertkey keyname (should not be in any other list), which can be used to set anything in the file, by hand, (for experts, and that plug-in does not need to be directly maintained if xmlinp gets a new switch)

aiida_fleur.tools.xml_util.**get_xml_attribute**(*node*, *attributename*, *parser_info_out=None*)
Get an attribute value from a node.

Params node a node from etree

Params attributename a string with the attribute name.

Returns either attributevalue, or None

aiida_fleur.tools.xml_util.**inpxml_todict**(*parent*, *xmlstr*)
Recursive operation which transforms an xml etree to python nested dictionaries and lists. Decision to add a list is if the tag name is in the given list tag_several

Parameters

- **parent** – some xmltree, or xml element
- **xmlstr** – structure/layout of the xml file in xmlstr is tags_several: a list of the tags, which should be converted to a list, not a dictionary(because they are known to occur more often, and want to be accessed in a list later.

Returns a python dictionary

aiida_fleur.tools.xml_util.**is_sequence**(*arg*)
Checks if arg is a sequence

aiida_fleur.tools.xml_util.**replace_tag**(*xmltree*, *xpath*, *newelement*)
replaces a xml tag by another tag on an xmletree in place

Parameters

- **xmltree** – an xmltree that represents inp.xml
- **xpathn** – a path to the tag to be replaced
- **newelement** – a new tag

aiida_fleur.tools.xml_util.**set_dict_or_not**(*para_dict*, *key*, *value*)
setter method for a dictionary that will not set the key, value pair. if the key is [] or None.

aiida_fleur.tools.xml_util.**set_inpchanges**(*fleurinp_tree_copy*, *change_dict*)
Makes given changes directly in the inp.xml file. Afterwards updates the inp.xml file representation and the current inp_userchanges dictionary with the keys provided in the 'change_dict' dictionary.

Parameters

- **fleurinp_tree_copy** – a lxml tree that represents inp.xml
- **change_dict** – a python dictionary with the keys to substitute. It works like dict.update(), adding new keys and overwriting existing keys.

Returns new_tree a lxml tree with applied changes

An example of change_dict:

```
change_dict = {'itmax' : 1,
               'l_noco': True,
               'ctail': False,
               'l_ss': True}
```

A full list of supported keys in the change_dict can be found in *get_inpxml_file_structure()*:

```
'comment': '/fleurInput/comment',
'relPos': '/fleurInput/atomGroups/atomGroup/relPos',
'filmPos': '/fleurInput/atomGroups/atomGroup/filmPos',
'absPos': '/fleurInput/atomGroups/atomGroup/absPos',
'qss': '/fleurInput/calculationSetup/nocoParams/qss',
'l_ss': '/fleurInput/calculationSetup/nocoParams',
'row-1': '/fleurInput/cell/bulkLattice/bravaisMatrix',
```

(continues on next page)

(continued from previous page)

```
'row-2': '/fleurInput/cell/bulkLattice/bravaisMatrix',
'row-3': '/fleurInput/cell/bulkLattice/bravaisMatrix',
'a1': '/fleurInput/cell/filmLattice/a1',  # switches once
'dos': '/fleurInput/output',
'band': '/fleurInput/output',
'secvar': '/fleurInput/calculationSetup/expertModes',
'ctail': '/fleurInput/calculationSetup/coreElectrons',
'frcor': '/fleurInput/calculationSetup/coreElectrons',
'l_noco': '/fleurInput/calculationSetup/magnetism',
'l_J': '/fleurInput/calculationSetup/magnetism',
'swsp': '/fleurInput/calculationSetup/magnetism',
'lflip': '/fleurInput/calculationSetup/magnetism',
'off': '/fleurInput/calculationSetup/soc',
'spav': '/fleurInput/calculationSetup/soc',
'l_soc': '/fleurInput/calculationSetup/soc',
'soc66': '/fleurInput/calculationSetup/soc',
'pot8': '/fleurInput/calculationSetup/expertModes',
'eig66': '/fleurInput/calculationSetup/expertModes',
'l_f': '/fleurInput/calculationSetup/geometryOptimization',
'gamma': '/fleurInput/calculationSetup/bzIntegration/kPointMesh',
'gauss': '',
'tria': '',
'invs': '',
'zrfs': '',
'vchk': '/fleurInput/output/checks',
'cdinf': '/fleurInput/output/checks',
'disp': '/fleurInput/output/checks',
'vacdos': '/fleurInput/output',
'integ': '/fleurInput/output/vacuumDOS',
'star': '/fleurInput/output/vacuumDOS',
'iplot': '/fleurInput/output/plotting',
'score': '/fleurInput/output/plotting',
'plplot': '/fleurInput/output/plotting',
'slice': '/fleurInput/output',
'pallst': '/fleurInput/output/chargeDensitySlicing',
'form66': '/fleurInput/output/specialOutput',
'eonly': '/fleurInput/output/specialOutput',
'bmt': '/fleurInput/output/specialOutput',
'relativisticCorrections': '/fleurInput/xcFunctional',
'calculate': '/fleurInput/atomGroups/atomGroup/force',
'flipSpin': '/fleurInput/atomSpecies/species',
'Kmax': '/fleurInput/calculationSetup/cutoffs',
'Gmax': '/fleurInput/calculationSetup/cutoffs',
'GmaxXC': '/fleurInput/calculationSetup/cutoffs',
'numbands': '/fleurInput/calculationSetup/cutoffs',
'itmax': '/fleurInput/calculationSetup/scfLoop',
'minDistance': '/fleurInput/calculationSetup/scfLoop',
'maxIterBroyd': '/fleurInput/calculationSetup/scfLoop',
'imix': '/fleurInput/calculationSetup/scfLoop',
'alpha': '/fleurInput/calculationSetup/scfLoop',
'spinf': '/fleurInput/calculationSetup/scfLoop',
'kcrel': '/fleurInput/calculationSetup/coreElectrons',
'jspins': '/fleurInput/calculationSetup/magnetism',
'theta': '/fleurInput/calculationSetup/soc',
'phi': '/fleurInput/calculationSetup/soc',
'gw': '/fleurInput/calculationSetup/expertModes',
'lpr': '/fleurInput/calculationSetup/expertModes',
```

(continues on next page)

(continued from previous page)

```
'isec1': '/fleurInput/calculationSetup/expertModes',
'forcemix': '/fleurInput/calculationSetup/geometryOptimization',
'forcealpha': '/fleurInput/calculationSetup/geometryOptimization',
'force_converged': '/fleurInput/calculationSetup/geometryOptimization',
'qfix': '/fleurInput/calculationSetup/geometryOptimization',
'epsdisp': '/fleurInput/calculationSetup/geometryOptimization',
'epsforce': '/fleurInput/calculationSetup/geometryOptimization',
'valenceElectrons': '/fleurInput/calculationSetup/bzIntegration',
'mode': '/fleurInput/calculationSetup/bzIntegration',
'fermiSmearingEnergy': '/fleurInput/calculationSetup/bzIntegration',
'nx': '/fleurInput/calculationSetup/bzIntegration/kPointMesh',
'ny': '/fleurInput/calculationSetup/bzIntegration/kPointMesh',
'nz': '/fleurInput/calculationSetup/bzIntegration/kPointMesh',
'count': '/fleurInput/calculationSetup/kPointCount',
'ellow': '/fleurInput/calculationSetup/energyParameterLimits',
'elup': '/fleurInput/calculationSetup',
'filename': '/fleurInput/cell/symmetryFile',
'scale': '/fleurInput/cell/bulkLattice',
'ndir': '/fleurInput/output/densityOfStates',
'minEnergy': '/fleurInput/output/densityOfStates',
'maxEnergy': '/fleurInput/output/densityOfStates',
'sigma': ' /fleurInput/output/densityOfStates',
'layers': '/fleurInput/output/vacuumDOS',
'nstars': '/fleurInput/output/vacuumDOS',
'locx1': '/fleurInput/output/vacuumDOS',
'locy1': '/fleurInput/output/vacuumDOS',
'locx2': '/fleurInput/output/vacuumDOS',
'locy2': '/fleurInput/output/vacuumDOS',
'nstm': '/fleurInput/output/vacuumDOS',
'tworkf': '/fleurInput/output/vacuumDOS',
'numkpt': '/fleurInput/output/chargeDensitySlicing',
'minEigenval': '/fleurInput/output/chargeDensitySlicing',
'maxEigenval': '/fleurInput/output/chargeDensitySlicing',
'nnne': '/fleurInput/output/chargeDensitySlicing',
'dVac': '/fleurInput/cell/filmLattice',
'dTilda': '/fleurInput/cell/filmLattice',
'xcFunctional': '/fleurInput/xcFunctional/name',  # other_attributes_more
'name': {'/fleurInput/constantDefinitions', '/fleurInput/xcFunctional',
        '/fleurInput/atomSpecies/species'},
'value': '/fleurInput/constantDefinitions',
'element': '/fleurInput/atomSpecies/species',
'atomicNumber': '/fleurInput/atomSpecies/species',
'coreStates': '/fleurInput/atomSpecies/species',
'magMom': '/fleurInput/atomSpecies/species',
'radius': '/fleurInput/atomSpecies/species/mtSphere',
'gridPoints': '/fleurInput/atomSpecies/species/mtSphere',
'logIncrement': '/fleurInput/atomSpecies/species/mtSphere',
'lmax': '/fleurInput/atomSpecies/species/atomicCutoffs',
'lnonsphr': '/fleurInput/atomSpecies/species/atomicCutoffs',
's': '/fleurInput/atomSpecies/species/energyParameters',
'p': '/fleurInput/atomSpecies/species/energyParameters',
'd': '/fleurInput/atomSpecies/species/energyParameters',
'f': '/fleurInput/atomSpecies/species/energyParameters',
'type': '/fleurInput/atomSpecies/species/lo',
'l': '/fleurInput/atomSpecies/species/lo',
'n': '/fleurInput/atomSpecies/species/lo',
'eDeriv': '/fleurInput/atomSpecies/species/lo',
```

(continues on next page)

(continued from previous page)

```
'species': '/fleurInput/atomGroups/atomGroup',
'relaxXYZ': '/fleurInput/atomGroups/atomGroup/force'
```

aiida_fleur.tools.xml_util.**set_kpath**(*fleurinp_tree_copy*, *kpath*, *count*, *gamma*)

Sets a k-path directly into inp.xml

Parameters

- **fleurinp_tree_copy** – a lxml tree that represents inp.xml
- **kpath** – a dictionary with kpoint name as key and k point coordinate as value
- **count** – number of k-points
- **gamma** – a fortran-type boolean that controls if the gamma-point should be included in the k-point mesh

Returns new_tree a lxml tree with applied changes

aiida_fleur.tools.xml_util.**set_nkpts**(*fleurinp_tree_copy*, *count*, *gamma*)

Sets a k-point mesh directly into inp.xml

Parameters

- **fleurinp_tree_copy** – a lxml tree that represents inp.xml
- **count** – number of k-points
- **gamma** – a fortran-type boolean that controls if the gamma-point should be included in the k-point mesh

Returns new_tree a lxml tree with applied changes

aiida_fleur.tools.xml_util.**set_species**(*fleurinp_tree_copy*, *species_name*, *attributedict*, *create=False*)

Method to set parameters of a species tag of the fleur inp.xml file.

Parameters

- **fleurinp_tree_copy** – xml etree of the inp.xml
- **species_name** – string, name of the specie you want to change
- **attributedict** – a python dict specifying what you want to change.
- **create** – bool, if species does not exist create it and all subtags?

Raises **ValueError** – if species name is non existent in inp.xml and should not be created. also if other given tags are garbage. (errors from eval_xpath() methods)

Return fleurinp_tree_copy xml etree of the new inp.xml

attributedict is a python dictionary containing dictionaries that specify attributes to be set inside the certain specie. For example, if one wants to set a MT radius it can be done via:

```
attributedict = {'mtSphere' : {'radius' : 2.2}}
```

Another example:

```
'attributedict': {'special': {'socscale': 0.0}}
```

that switches SOC terms on a sertain specie. `mtSphere`, `atomicCutoffs`, `energyParameters`, `lo`, `electronConfig`, `nocoParams`, `ldaU` and `special` keys are supported. To find possible keys of the inner dictionary please refer to the FLEUR documentation flapw.de

aiida_fleur.tools.xml_util.**set_species_label**(*fleurinp_tree_copy*, *at_label*, *attributedict*, *create=False*)

This method calls *set_species()* method for a certain atom specie that corresponds to an atom with a given label

Parameters

- **fleurinp_tree_copy** – xml etree of the inp.xml
- **at_label** – string, a label of the atom which specie will be changed. 'all' to change all the species
- **attributedict** – a python dict specifying what you want to change.
- **create** – bool, if species does not exist create it and all subtags?

aiida_fleur.tools.xml_util.**shift_value**(*fleurinp_tree_copy*, *change_dict*, *mode='abs'*)

Shifts numertical values of some tags directly in the inp.xml file.

Parameters

- **fleurinp_tree_copy** – a lxml tree that represents inp.xml
- **change_dict** – a python dictionary with the keys to shift.
- **mode** – 'abs' if change given is absolute, 'rel' if relative

Returns new_tree a lxml tree with shifted values

An example of change_dict:

```
change_dict = {'itmax' : 1, 'dVac': -0.123}
```

aiida_fleur.tools.xml_util.**shift_value_species_label**(*fleurinp_tree_copy*, *at_label*, *attr_name*, *value_given*, *mode='abs'*)

Shifts value of a specie by label if at_label contains 'all' then applies to all species

Parameters

- **fleurinp_tree_copy** – xml etree of the inp.xml
- **at_label** – string, a label of the atom which specie will be changed. 'all' if set up all species
- **attr_name** – name of the attribute to change
- **value_given** – value to add or to multiply by
- **mode** – 'rel' for multiplication or 'abs' for addition

aiida_fleur.tools.xml_util.**write_new_fleur_xmlinp_file**(*inp_file_xmltree*, *fleur_change_dic*, *xmlinpstructure*)

This modifies the xml-inp file. Makes all the changes wanted by the user or sets some default values for certain modes

Params inp_file_xmltree xml-tree of the xml-inp file

Params fleur_change_dic dictionary {attrib_name : value} with all the wanted changes.

Returns an etree of the xml-inp file with changes.

aiida_fleur.tools.xml_util.**xml_set_all_attribv**(*xmltree*, *xpathn*, *attributename*, *attribv*, *create=False*)

Routine sets the value of an attribute in the xml file on all places it occurs

Parameters

- **xmltree** – an xmltree that represents inp.xml
- **xpathn** – a path to the attribute
- **attributename** – an attribute name
- **attribv** – an attribute value which will be set
- **create** – if True and there is no given xpath in the FleurinpData, creates it

Returns None, or an etree

Comment: Element.set will add the attribute if it does not exist, xpath expression has to exist

example: xml_set_first_attribv(tree, '/fleurInput/atomGroups/atomGroup/force', 'relaxXYZ', 'TTF')
xml_set_first_attribv(tree, '/fleurInput/atomGroups/atomGroup/force', 'calculate', 'F')

aiida_fleur.tools.xml_util.**xml_set_all_text**(*xmltree*, *xpathn*, *text*, *create=False*, *tag_order=None*)

Routine sets the text of a tag in the xml file

Parameters

- **xmltree** – an xmltree that represents inp.xml
- **xpathn** – a path to the attribute
- **text** – text to be set
- **create** – if True and there is no given xpath in the FleurinpData, creates it
- **place_index** – if create=True, defines the place where to put a created tag
- **tag_order** – if create=True, defines a tag order

aiida_fleur.tools.xml_util.**xml_set_attribv_occ**(*xmltree*, *xpathn*, *attributename*, *attribv*, *occ=None*, *create=False*)

Routine sets the value of an attribute in the xml file on only the places specified in occ

Parameters

- **xmltree** – an xmltree that represents inp.xml
- **xpathn** – a path to the attribute
- **attributename** – an attribute name
- **attribv** – an attribute value which will be set
- **occ** – a list of integers specifying number of occurrence to be set
- **create** – if True and there is no given xpath in the FleurinpData, creates it

Comment: Element.set will add the attribute if it does not exist, xpath expression has to exist

example: xml_set_first_attribv(tree, '/fleurInput/calculationSetup', 'band', 'T')
xml_set_first_attribv(tree, '/fleurInput/calculationSetup', 'dos', 'F')

aiida_fleur.tools.xml_util.**xml_set_first_attribv**(*xmltree*, *xpathn*, *attributename*, *attribv*, *create=False*)

Routine sets the value of the first found attribute in the xml file

Parameters

- **xmltree** – an xmltree that represents inp.xml

- **xpathn** – a path to the attribute
- **attributename** – an attribute name
- **attribv** – an attribute value which will be set
- **create** – if True and there is no given xpath in the FleurinpData, creates it

Returns None, or an etree

Comment: Element.set will add the attribute if it does not exist, xpath expression has to exist

example: xml_set_first_attribv(tree, '/fleurInput/calculationSetup', 'band', 'T')
xml_set_first_attribv(tree, '/fleurInput/calculationSetup', 'dos', 'F')

aiida_fleur.tools.xml_util.**xml_set_text**(*xmltree*, *xpathn*, *text*, *create=False*, *place_index=None*, *tag_order=None*)

Routine sets the text of a tag in the xml file

Parameters

- **xmltree** – an xmltree that represents inp.xml
- **xpathn** – a path to the attribute
- **text** – text to be set
- **create** – if True and there is no given xpath in the FleurinpData, creates it
- **place_index** – if create=True, defines the place where to put a created tag
- **tag_order** – if create=True, defines a tag order

example:

xml_set_text(tree, '/fleurInput/comment', 'Test Fleur calculation for AiiDA plug-in')

but also coordinates and Bravais Matrix!:

xml_set_text(tree, '/fleurInput/atomGroups/atomGroup/relPos','1.20000 PI/3 5.1-MYCrazyCostant')

aiida_fleur.tools.xml_util.**xml_set_text_occ**(*xmltree*, *xpathn*, *text*, *create=False*, *occ=0*, *place_index=None*, *tag_order=None*)

Routine sets the text of a tag in the xml file

Parameters

- **xmltree** – an xmltree that represents inp.xml
- **xpathn** – a path to the attribute
- **text** – text to be set
- **create** – if True and there is no given xpath in the FleurinpData, creates it
- **occ** – an integer that sets occurrence number to be set
- **place_index** – if create=True, defines the place where to put a created tag
- **tag_order** – if create=True, defines a tag order

5.1.5.4 Utility for LDA+U density matrix files

This module contains useful methods for initializing or modifying a n_mmp_mat file for LDA+U via the FleurinpModifier

aiida_fleur.tools.set_nmmpmat.**fac**(*n*)
Returns the factorial of n

aiida_fleur.tools.set_nmmpmat.**get_wigner_matrix**(*l*, *phi*, *theta*)
Produces the wigner rotation matrix for the density matrix

Parameters

- **l** – int, orbital quantum number
- **phi** – float, angle (radian) corresponds to euler angle alpha
- **theta** – float, angle (radian) corresponds to euler angle beta

aiida_fleur.tools.set_nmmpmat.**set_nmmpmat**(*fleurinp_tree_copy*, *nmmp_lines_copy*, *species_name*, *orbital*, *spin*, *occStates=None*, *denmat=None*, *phi=None*, *theta=None*)
Routine sets the block in the n_mmp_mat file specified by species_name, orbital and spin to the desired density matrix

Parameters

- **fleurinp_tree_copy** – an xmltree that represents inp.xml
- **nmmp_lines_copy** – list of lines in the n_mmp_mat file
- **species_name** – string, name of the species you want to change
- **orbital** – integer, orbital quantum number of the LDA+U procedure to be modified
- **spin** – integer, specifies which spin block should be modified
- **occStates** – list, sets the diagonal elements of the density matrix and everything else to zero
- **denmat** – matrix, specify the density matrix explicitely
- **phi** – float, optional angle (radian), by which to rotate the density matrix before writing it
- **theta** – float, optional angle (radian), by which to rotate the density matrix before writing it

Raises

- **ValueError** – If something in the input is wrong
- **KeyError** – If no LDA+U procedure is found on a species

aiida_fleur.tools.set_nmmpmat.**validate_nmmpmat**(*fleurinp_tree*, *nmmp_lines*)
Checks that the given nmmp_lines is valid with the given fleurinp_tree

Checks that the number of blocks is as expected from the inp.xml and each block does not contain non-zero elements outside their size given by the orbital quantum number in the inp.xml. Additionally the occupations, i.e. diagonal elements are checked that they are in between 0 and the maximum possible occupation

Parameters

- **fleurinp_tree_copy** – an xmltree that represents inp.xml
- **nmmp_lines_copy** – list of lines in the n_mmp_mat file

Raises **ValueError** – if any of the above checks are violated.

5.1.5.5 Parameter utility

General Parameter

This contains code snippets and utility useful for dealing with parameter data nodes commonly used by the fleur plugin and workflows

aiida_fleur.tools.dict_util.**clean_nones**(*dict_to_clean*)

Recursively remove all keys which values are None from a nested dictionary return the cleaned dictionary

Parameters **dict_to_clean** – (dict): python dictionary to remove keys with None as value

Returns dict, cleaned dictionary

aiida_fleur.tools.dict_util.**dict_merger**(*dict1*, *dict2*)

Merge recursively two nested python dictionaries and if key is in both digionaries tries to add the entries in both dicts. (merges two subdicts, adds lists, strings, floats and numbers together!)

Parameters

- **dict1** – dict
- **dict2** – dict

Return dict Merged dict

aiida_fleur.tools.dict_util.**extract_elementpara**(*parameter_dict*, *element*)

Parameters

- **parameter_dict** – python dict, parameter node for inpgen
- **element** – string, i.e 'W'

Returns python dictionary, parameter node which contains only the atom parameters for the given element

aiida_fleur.tools.dict_util.**recursive_merge**(*left: Dict[str, Any], right: Dict[str, Any]*) → Dict[str, Any]

Recursively merge two dictionaries into a single dictionary.

keys in right override keys in left!

Parameters

- **left** – first dictionary.
- **right** – second dictionary.

Returns the recursively merged dictionary.

Merge Parameter

This module, contains a method to merge Dict nodes used by the FLEUR inpgen. This might also be of interest for other all-electron codes

aiida_fleur.tools.merge_parameter.**merge_parameter**(*Dict1*, *Dict2*, *overwrite=True*, *merge=True*)

Merges two Dict nodes. Additive: uses all namelists of both. If they have a namelist in common. Dict2 will overwrite the namelist of Dict. If this is not wanted. set overwrite = False. Then attributes of both will be added, but attributes from Dict1 won't be overwritten.

Parameters

- **Dict1** – AiiDA Dict Node
- **Dict2** – AiiDA Dict Node
- **overwrite** – bool, default True
- **merge** – bool, default True

returns: AiiDA Dict Node

#TODO be more carefull how to merge ids in atom namelists, i.e species labels

aiida_fleur.tools.merge_parameter.**merge_parameter_cf**(*Dict1*, *Dict2*, *overwrite=None*)
calcfunction of merge_parameters

aiida_fleur.tools.merge_parameter.**merge_parameters**(*DictList*, *overwrite=True*)
Merge together all parameter nodes in the given list.

5.1.5.6 Corehole/level utility

Contains helper functions to create core-holes in Fleur input files from AiiDA data nodes.

aiida_fleur.tools.create_corehole.**create_corehole_para**(*structure*, *kind*, *econfig*, *species_name='corehole'*, *parameterdata=None*)
This methods sets of electron configurations for a kind or position given, make sure to break the symmetry for this position/kind beforehand, otherwise you will create several coreholes.

Parameters

- **structure** – StructureData
- **kind** – a string with the kind_name (TODO: alternative the kind object)
- **econfig** – string, e.g. econfig = "[Kr] 5s2 4d10 4f13 | 5p6 5d5 6s2" to set, i.e. the corehole

Returns a Dict node

In this module you find methods to parse/extract corelevel shifts from an out.xml file of FLEUR.

aiida_fleur.tools.extract_corelevels.**clshifts_to_be**(*coreleveldict*, *reference_dict*, *warn=False*)
This methods converts corelevel shifts to binding energies, if a reference is given. These can than be used for plotting.

Params reference_dict An example:

```
reference_dict = {'W' : {'4f7/2' : [124],
                         '4f5/2' : [102]},
                  'Be' : {'1s': [117]}}
```

Params coreleveldict An example:

```
coreleveldict = {'W' : {'4f7/2' : [0.4, 0.3, 0.4 ,0.1],
                        '4f5/2' : [0, 0.3, 0.4, 0.1]},
                 'Be' : {'1s': [0, 0.2, 0.4, 0.1, 0.3]}
```

aiida_fleur.tools.extract_corelevels.**convert_to_float**(*value_string*, *parser_info=None*)
Tries to make a float out of a string. If it can't it logs a warning and returns True or False if convertion worked or not.

Parameters **value_string** – a string

Returns value the new float or value_string: the string given

Retruns True or False

aiida_fleur.tools.extract_corelevels.**extract_corelevels**(*outxmlfile*, *options=None*)

Extracts corelevels out of out.xml files

Params outxmlfile path to out.xml file

Parameters **options** – A dict: 'iteration' : X/'all'

Returns corelevels A list of the form:

```
[atomtypes][spin][dict={atomtype : '', corestates : list_of_corestates}]
[atomtypeNumber][spin]['corestates'][corestate number][attribute]
get corelevel energy of first atomtype, spin1, corelevels[0][0]['corestates'][i][
↪'energy']
```

Example of output

```
[[{'atomtype': '     1',
'corestates': [{'energy': -3.6489930627,
                'j': ' 0.5',
                'l': ' 0',
                'n': ' 1',
                'weight': 2.0}],
'eigenvalue_sum': '     -7.2979861254',
'kin_energy': '     13.4757066163',
'spin': '1'}],
[{'atomtype': '     2',
'corestates': [{'energy': -3.6489930627,
                'j': ' 0.5',
                'l': ' 0',
                'n': ' 1',
                'weight': 2.0}],
'eigenvalue_sum': '     -7.2979861254',
'kin_energy': '     13.4757066163',
'spin': '1'}]]
```

aiida_fleur.tools.extract_corelevels.**parse_state_card**(*corestateNode*, *iteration_node*, *parser_info=None*)

Parses the ONE core state card

Params corestateNode an etree element (node), of a fleur output corestate card

Params iteration_node an etree element, iteration node

Params jspin integer 1 or 2

Returns a pythondict of type:

```
{'eigenvalue_sum' : eigenvalueSum,
 'corestates': states,
 'spin' : spin,
 'kin_energy' : kinEnergy,
 'atomtype' : atomtype}
```

You find the usual binding_energy for all elements in the periodic table.

aiida_fleur.tools.element_econfig_list.**convert_fleur_config_to_econfig**(*fleurconf_str*, *keep_spin=False*)

'[Kr] (4d3/2) (4d5/2) (4f5/2) (4f7/2)' -> '[Kr] 4d10 4f14', or '[Kr] 4d3/2 4d5/2 4f5/2 4f7/2'

for now only use for coreconfig, it will fill all orbitals, since it has no information on the filling.

aiida_fleur.tools.element_econfig_list.**econfigstr_hole**(*econfigstr*, *corelevel*, *highesunoccp*, *htype='valence'*)

'1s2 | 2s2' , '1s2' , '2p0' -> '1s1 | 2s2 2p1'

Param string

Param string

Param string

Returns string

aiida_fleur.tools.element_econfig_list.**get_coreconfig**(*element*, *full=False*)

returns the econfiguration as a string of an element.

Param element string

Param full, bool (econfig without [He]...)

Returns string

Note Be careful with base strings...

aiida_fleur.tools.element_econfig_list.**get_econfig**(*element*, *full=False*)

returns the econfiguration as a string of an element.

Params element element string

Params full a bool (econfig without [He]...)

Returns a econfig string

aiida_fleur.tools.element_econfig_list.**get_spin_econfig**(*fulleconfigstr*)

converts and econfig string to a full spin econfig 1s2 2s2 2p6' - > '1s1/2 2s1/2 2p1/2 2p3/2'

aiida_fleur.tools.element_econfig_list.**get_state_occ**(*econfigstr*, *corehole=''*, *valence=''*, *ch_occ=1.0*)

finds out all not full occupied states and returns a dictionary of them return a dict i.e corehole '4f 5/2' ch_occ full or fractional corehole occupation? valence: orbital sting '5d', is to adjust the charges for fractional coreholes To that orbital occupation ch_occ - 1 will be added.

aiida_fleur.tools.element_econfig_list.**highest_unocc_valence**(*econfigstr*)

returns the highest not full valence orbital. If all are full, it returns '' #maybe should be advanced to give back the next highest unocc

aiida_fleur.tools.element_econfig_list.**rek_econ**(*econfigstr*)

recursive routine to return a full econfig '[Xe] 4f14 | 5d10 6s2 6p4' -> '1s 2s ... 4f14 | 5d10 6s2 6p4'

5.1.5.7 Common aiida utility

In here we put all things util (methods, code snipets) that are often useful, but not yet in AiiDA itself. So far it contains:

export_extras import_extras delete_nodes (FIXME) delete_trash (FIXME) create_group

aiida_fleur.tools.common_aiida.**create_group**(*name*, *nodes*, *description=None*, *add_if_exist=False*)

Creates a group for a given node list.

!!! Now aiida-core has these functionality, use it from there instead!!! So far this is only an AiiDA verdi command.

Params name string name for the group

Params nodes list of AiiDA nodes, pks, or uuids

Params description optional string that will be stored as description for the group

Returns the group, AiiDa group

Usage example:

```
group_name = 'delta_structures_gustav'
nodes_to_group_pks =[2142, 2084]
create_group(group_name, nodes_to_group_pks,
             description='delta structures added by hand. from Gustavs inpgen␣
↪files')
```

aiida_fleur.tools.common_aiida.**export_extras**(*nodes*, *filename='node_extras.txt'*)

Writes uuids and extras of given nodes to a json-file. This is useful for import/export because currently extras are lost. Therefore this can be used to save and restore the extras via *import_extras()*.

Param nodes: list of AiiDA nodes, pks, or uuids

Param filename, string where to store the file and its name

example use: .. code-block:: python

node_list = [120,121,123,46] export_extras(node_list)

aiida_fleur.tools.common_aiida.**get_nodes_from_group**(*group*, *return_format='uuid'*)

Returns a list of pk or uuid of a nodes in a given group. Since 1.1.0, this function does !!! Now aiida-core has these functionality, use it from there instead!!!

not load a group using the label or any other identification. Use Group.objects.get(filter=ID) to pre-load this, available filters are: id, uuid, label, type_string, time, description, user_id.

aiida_fleur.tools.common_aiida.**import_extras**(*filename*)

Reads in node uuids and extras from a file (most probably generated by *export_extras()*) and applies them to nodes in the DB.

This is useful for import/export because currently extras are lost. Therefore this can be used to save and restore the extras on the nodes.

Param filename, string what file to read from (has to be json format)

example use: import_extras('node_extras.txt')

5.1.5.8 Reading in Cif files

In this module you find a method (read_cif_folder) to read in all .cif files from a folder and store the structures in the database.

aiida_fleur.tools.read_cif_folder.**read_cif_folder**(*path='/home/docs/checkouts/readthedocs.org/user_builds/aiida-fleur/checkouts/v1.1.3/docs/source'*, *recursive=True*, *store=False*, *log=False*, *comments=''*, *extras=''*, *logfile_name='read_cif_folder_logfile'*)

Method to read in cif files from a folder and its subfolders. It can convert them into AiiDA structures and store them.

defaults input parameter values are: path=”.”, recursive=True, store=False, log=False, comments=”, extras=”

Params path: Path to the dictionary with the files (default, where this method is called)

Params recursive: bool, If True: looks aso in subfolders, if False: just given dir

Params store: bool, if True: stores structures in database

Params log: bool, if True, writes a logfile with information (pks, and co)

Params comments: string: comment to add to the structures

Params extras: dir/string/arb: extras added to the structures stored in the db

5.1.5.9 IO routines

Here we collect IO routines and their utility, for writting certain things to files, or post process files. For example collection of data or database evaluations, for other people.

5.1.5.10 Common utitlity for fleur workchains

In here we put all things (methods) that are common to workflows AND depend on AiiDA classes, therefore can only be used if the dbenv is loaded. Util that does not depend on AiiDA classes should go somewhere else.

aiida_fleur.tools.common_fleur_wf.**calc_time_cost_function**(*natom*, *nkpt*, *kmax*, *nspins=1*)

Estimates the cost of simulating a single iteration of a system

aiida_fleur.tools.common_fleur_wf.**calc_time_cost_function_total**(*natom*, *nkpt*, *kmax*, *niter*, *nspins=1*)

Estimates the cost of simulating a all iteration of a system

aiida_fleur.tools.common_fleur_wf.**cost_ratio**(*total_costs*, *walltime_sec*, *ncores*)

Estimates if simulation cost matches resources

aiida_fleur.tools.common_fleur_wf.**determine_favorable_reaction**(*reaction_list*, *workchain_dict*)

Finds out with reaction is more favorable by simple energy standpoints

TODO check physics reaction list: list of reaction strings workchain_dict = {‘Be12W’ : uuid_wc or output, ‘Be2W’ : uuid, . . . }

return dictionary that ranks the reactions after their enthalpy

TODO: refactor aiida part out of this, leaving an aiida independent part and one more universal

aiida_fleur.tools.common_fleur_wf.**find_last_submitted_calcjob**(*restart_wc*)

Finds the last CalcJob submitted in a higher-level workchain and returns it’s uuid

aiida_fleur.tools.common_fleur_wf.**find_last_submitted_workchain**(*restart_wc*)

Finds the last WorkChain submitted in a higher-level workchain and returns it’s uuid

aiida_fleur.tools.common_fleur_wf.**get_inputs_fleur**(*code*, *remote*, *fleurinp*, *options*, *label=”*, *description=”*, *settings=None*, *serial=False*, *only_even_MPI=False*)

Assembles the input dictionary for Fleur Calculation. Does not check if a user gave correct input types, it is the work of FleurCalculation to check it.

Parameters

- **code** – FLEUR code of Code type
- **remote** – remote_folder from the previous calculation of RemoteData type
- **fleurinp** – FleurinpData object representing input files
- **options** – calculation options that will be stored in metadata
- **label** – a string setting a label of the CalcJob in the DB
- **description** – a string setting a description of the CalcJob in the DB
- **settings** – additional settings of Dict type
- **serial** – True if run a calculation in a serial mode

Example of use:

```
inputs_build = get_inputs_inpgen(structure, inpgencode, options, label,
                                 description, params=params)
future = self.submit(inputs_build)
```

aiida_fleur.tools.common_fleur_wf.**get_inputs_inpgen**(*structure*, *inpgencode*, *options*, *label=''*, *description=''*, *settings=None*, *params=None*, ***kwargs*)

Assembles the input dictionary for Fleur Calculation.

Parameters

- **structure** – input structure of StructureData type
- **inpgencode** – inpgen code of Code type
- **options** – calculation options that will be stored in metadata
- **label** – a string setting a label of the CalcJob in the DB
- **description** – a string setting a description of the CalcJob in the DB
- **params** – input parameters for inpgen code of Dict type

Example of use:

```
inputs_build = get_inputs_inpgen(structure, inpgencode, options, label,
                                 description, params=params)
future = self.submit(inputs_build)
```

aiida_fleur.tools.common_fleur_wf.**get_kpoints_mesh_from_kdensity**(*structure*, *kpoint_density*)

params: structuredata, Aiida structuredata params: kpoint_density

returns: tuple (mesh, offset) returns: kpointsdata node

aiida_fleur.tools.common_fleur_wf.**get_mpi_proc**(*resources*)

Determine number of total processes from given resource dict

aiida_fleur.tools.common_fleur_wf.**is_code**(*code*)

Test if the given input is a Code node, by object, id, uuid, or pk if yes returns a Code node in all cases if no returns None

aiida_fleur.tools.common_fleur_wf.**optimize_calc_options**(*nodes*, *mpi_per_node*, *omp_per_mpi*, *use_omp*, *mpi_omp_ratio*, *fleurinpData=None*, *kpts=None*, *sacrifice_level=0.9*, *only_even_MPI=False*)

Makes a suggestion on parallelisation setup for a particular fleurinpData. Only the total number of k-points is analysed: the function suggests ideal k-point parallelisation + OMP parallelisation (if required). Note: the total number of used CPUs per node will not exceed mpi_per_node * omp_per_mpi.

Sometimes perfect parallelisation is terms of idle CPUs is not what used wanted because it can harm MPI/OMP ratio. Thus the function first chooses first top parallelisations in terms of total CPUs used (bigger than sacrifice_level * maximal_number_CPUs_possible). Then a parallelisation which is the closest to the MPI/OMP ratio is chosen among them and returned.

Parameters

- **nodes** – maximal number of nodes that can be used
- **mpi_per_node** – an input suggestion of MPI tasks per node
- **omp_per_mpi** – an input suggestion for OMP tasks per MPI process
- **use_omp** – False if OMP parallelisation is not needed
- **mpi_omp_ratio** – requested MPI/OMP ratio
- **fleurinpData** – FleurinpData to extract total number of kpts from
- **kpts** – the total number of kpts
- **sacrifice_level** – sets a level of performance sacrifice that a user can afford for better MPI/OMP ratio.

Parm only_even_MPI if set to True, the function does not set MPI to an odd number (if possible)

Returns nodes, MPI_tasks, OMP_per_MPI, message first three are parallelisation info and the last one is an exit message.

aiida_fleur.tools.common_fleur_wf.**performance_extract_calcs**(*calcs*)

Extracts some runtime and system data from given fleur calculations

Params calcs list of calculation nodes/pks/or uuids. Fleur calc specific

Returns data_dict dictionary, dictionary of arrays with the same length, from with a panda frame can be created.

Note: Is not the fastest for many calculations > 1000.

aiida_fleur.tools.common_fleur_wf.**test_and_get_codenode**(*codenode*, *expected_code_type*, *use_exceptions=False*)

Pass a code node and an expected code (plugin) type. Check that the code exists, is unique, and return the Code object.

Parameters

- **codenode** – the name of the code to load (in the form label@machine)
- **expected_code_type** – a string with the plugin that is expected to be loaded. In case no plugins exist with the given name, show all existing plugins of that type
- **use_exceptions** – if True, raise a ValueError exception instead of calling sys.exit(1)

Returns a Code object

In here we put all things (methods) that are common to workflows AND DO NOT depend on AiiDA classes, therefore can be used without loading the dbenv. Util that does depend on AiiDA classes should go somewhere else.

aiida_fleur.tools.common_fleur_wf_util.**balance_equation**(*equation_string*, *allow_negativ=False*, *allow_zero=False*, *eval_linear=True*)

Method that balances a chemical equation.

param equation_string: string (with '->') param allow_negativ: bool, default False, allows for negative coefficients for the products.

return string: balanced equation

balance_equation("C7H16+O2 -> CO2+H2O")) balance_equation("Be12W->Be22W+Be12W") balance_equation("Be12W->Be12W")

have to be intergers everywhere in the equation, factors and formulas

1*C7H16+11*O2 ->7* CO2+8*H2O None 1*Be12W->1*Be12W #TODO The solver better then what we need. Currently if system is over #"Be12W->Be2W+W+Be" solves to {a: 24, b: -d/2 + 144, c: d/2 - 120}-> FAIL-> None # The code fails in the later stage, but this solution should maybe be used.

code adapted from stack exchange (the messy part): https://codegolf.stackexchange.com/questions/8728/balance-chemical-equations

aiida_fleur.tools.common_fleur_wf_util.**calc_stoi**(*unitcellratios*, *formulas*, *error_ratio=None*)

Calculate the Stoichiometry with errors from a given unit cell ratio, formulas.

Example: calc_stoi([10, 1, 7], ['Be12Ti', 'Be17Ti2', 'Be2'], [0.1, 0.01, 0.1]) ({'Be': 12.583333333333334, 'Ti': 1.0}, {'Be': 0.12621369924887876, 'Ti': 0.0012256517540566825}) calc_stoi([10, 1, 7], ['Be12Ti', 'Be17Ti2', 'Be2']) ({'Be': 12.583333333333334, 'Ti': 1.0}, {})

aiida_fleur.tools.common_fleur_wf_util.**check_eos_energies**(*energylist*)

Checks if there is an abnormality in the total energies from the Equation of states. i.e. if one point has a larger energy then its two neighbors

Parameters **energylist** – list of floats

Returns nnormalies integer

aiida_fleur.tools.common_fleur_wf_util.**convert_eq_to_dict**(*equationstring*)

Converts an equation string to a dictionary convert_eq_to_dict('1*Be12Ti->10*Be+1*Be2Ti+5*Be') -> {'products': {'Be': 15, 'Be2Ti': 1}, 'educts': {'Be12Ti': 1}}

aiida_fleur.tools.common_fleur_wf_util.**convert_formula_to_formula_unit**(*formula*)

Converts a formula to the smallest chemical formula unit 'Be4W2' -> 'Be2W'

aiida_fleur.tools.common_fleur_wf_util.**convert_frac_formula**(*formula*, *max_digits=3*)

Converts a formula with fractions to a formula with integer factors only

Be0.5W0.5 -> BeW

Parameters

- **formula** – str, crystal formula i.e. Be2W, Be0.2W0.7
- **max_digits** – int default=3, number of digits after which fractions will be cut off

Returns string

aiida_fleur.tools.common_fleur_wf_util.**determine_convex_hull**(*formation_en_grid*)
Wraps the pyhull packge implementing the qhull algo for our purposes. For now only for 2D phase diagrams Adds the points [1.0, 0.0] and [0.0, 1.0], because in material science these are always there.

Params formation_en_grid: list of points in phase space [[x, formation_energy]]

Returns a hul datatype

aiida_fleur.tools.common_fleur_wf_util.**determine_formation_energy**(*struc_te_dict*, *ref_struc_te_dict*)
This method determines the formation energy. E_form = E(A_xB_y) - x*E(A) - y*E(B)

Params struc_te_dict python dictionary in the form of {'formula' : total_energy} for the compound(s)

Params ref_struc_te_dict python dictionary in the form of {'formula' : total_energy per atom, or per unit cell} for the elements (if the formula of the elements contains a number the total energy is divided by that number)

Returns list of floats, dict {formula : eform, ..} units energy/per atom, energies have some unit as energies given

aiida_fleur.tools.common_fleur_wf_util.**determine_reactions**(*formula*, *available_data*)
Determines and balances theoretical possible reaction. Stoichiometry 'Be12W', [Be12W, Be2W, Be, W, Be22W] -> [[Be22W+Be2W], [Be12W], [Be12+W],...]

Params formula string, given educts (left side of equation)

Params available_data list of strings of compounds (products), from which all possibilities will be constructed

aiida_fleur.tools.common_fleur_wf_util.**get_atomprocent**(*formula*)
This converts a formula to a dictionary with element : atomprocent example converts 'Be24W2' to {'Be': 24/26, 'W' : 2/26}, also BeW to {'Be' : 0.5, 'W' : 0.5} :params: formula: string :returns: a dict, element : atomprocent

Todo alternative with structuredata

aiida_fleur.tools.common_fleur_wf_util.**get_enhalpy_of_equation**(*reaction*, *formenergydict*)
calculate the enthalpy per atom of a given reaction from the given data.

param reaction: string param fromenergydict: dictionary that contains the {compound: formationenergy per atom}

TODO check if physics is right

aiida_fleur.tools.common_fleur_wf_util.**get_natoms_element**(*formula*)
Converts 'Be24W2' to {'Be': 24, 'W' : 2}, also BeW to {'Be' : 1, 'W' : 1}

aiida_fleur.tools.common_fleur_wf_util.**inpgen_dict_set_mesh**(*inpgendict*, *mesh*)
params: python dict, used for inpgen parameterdata node params: mesh either as returned by kpointsdata or tuple of 3 integers

returns: python dict, used for inpgen parameterdata node

aiida_fleur.tools.common_fleur_wf_util.**powerset**(*L*)
Constructs the power set, 'potenz Menge' of a given list.

return list: of all possible subsets

aiida_fleur.tools.common_fleur_wf_util.**ucell_to_atompr**(*ratio*, *formulas*, *element*, *error_ratio=None*)
Converts unit cell ratios into atom ratios.

len(ratio) == len(formulas) (== len(error_ratio)) ucell_to_atompr([10, 1, 7], [‘Be12Ti’, ‘Be17Ti2’, ‘Be2’], element=’Be’, [0.1, 0.1, 0.1])

CHAPTER 6

Indices and tables

- genindex
- modindex
- search

Python Module Index

a

Index

Symbols

A

B

C

D

E

F

G

H

I

M

O

P

Q

R

S