# CONTENTS

1 Acknowledgments: .......................................................... 3

2 User support .................................................................. 5

3 Features, Illustrations, Usage examples: .......................... 7

4 Basic overview ............................................................... 11
  4.1 Requirements to use this code: .................................. 11
  4.2 AiiDA-package Layout: .............................................. 11

5 User’s Guide .................................................................. 13
  5.1 User’s guide ............................................................... 13
    5.1.1 Getting started .................................................... 13
      5.1.1.1 Installation of AiiDA-FLEUR ......................... 13
      5.1.1.2 AiiDA setup ................................................. 14
    5.1.2 AiiDA-FLEUR Data Plugins ................................. 18
      5.1.2.1 FleurinpData ............................................... 19
      5.1.2.2 FleurinpModifier ........................................ 22
    5.1.3 AiiDA-FLEUR Calculations ................................ 28
      5.1.3.1 Fleur input generator plugin ......................... 28
      5.1.3.2 FLEUR code plugin .................................... 31
    5.1.4 AiiDA-FLEUR WorkChains ................................ 38
      5.1.4.1 General design ........................................... 38
      5.1.4.2 Workchain classification .............................. 39
      5.1.4.3 Basic (Technical) Workchains ...................... 40
      5.1.4.4 More advanced (Scientific) Workchains .......... 77
    5.1.5 The command line interface (CLI) ......................... 129
      5.1.5.1 General information .................................... 129
      5.1.5.2 Overview of the main commands ................. 130
      5.1.5.3 Confirm proper setup .................................. 131
      5.1.5.4 Prepare options nodes ................................ 132
      5.1.5.5 Launching Calculations and workchains ....... 132
      5.1.5.6 Executing inpgen ........................................ 132
      5.1.5.7 Executing Fleur ........................................... 133
      5.1.5.8 Executing higher workflows ......................... 133
      5.1.5.9 (DFT) code interoperability ......................... 134
      5.1.5.10 Common workflows .................................. 134
      5.1.5.11 Commandline versus python work ............... 134
    5.1.6 Tools .................................................................. 134
    5.1.7 Tutorials ............................................................ 135
The AiiDA-FLEUR python package enables the use of the all-electron Density Functional Theory (DFT) code FLEUR with the AiiDA framework.

It is open source under the MIT license and available on github. The package is developed mainly at the Forschungszentrum Jülich GmbH, (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 and checkout JuDFT for further information on other IAS-1 made simulation software.
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 also acknowledge support by the Joint Lab Virtual Materials Design (JLVMD) of the Forschungszentrum Jülich. We thank the AiiDA team for their help and work. Also the vial exchange with developers of AiiDA packages for other codes was inspiring.

If you use this package please cite:

- The plugin and workflows:
  

- The FLEUR code: http://www.flapw.de
Chapter 1. Acknowledgments:
You can post any questions in the Fleur user forum
For bugs, feature requests and further issues please use the issue tracker on github of the aiida-fleur repository.
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 over 4000 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.

![Diagram showing Fleur simulations can be chained with other codes through AiiDA StructureData nodes.]

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)
4.1 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

4.2 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
10. Force-theorem Spin Spiral Dispersion
11. Force-theorem *Dzialoshinskii-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.
5.1 User’s guide

This is the AiiDA-FLEUR user’s guide.

5.1.1 Getting started

5.1.1.1 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:
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 .
```

### 5.1.1.2 AiiDA setup

Once AiiDA-FLEUR is installed, it is 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 aida_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
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 your system. Particularly in my case, I need to set proper library paths. Hence my prepend text looks like:

```
Note: The default expected Input generator version is >=32 i.e. >=MaX5.1. If you want to install older version of the input generator you **must** specify a ‘version’ key under the ‘extras’ of the code node i.e for example code.set_extra(‘version’, 31).
```

```
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%]
(continues on next page)
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 AiDA-FLEUR version you have installed is not compatible with the AiDA version you are running, since not all AiDA versions are back-compatible.

### 5.1.2 AiDA-FLEUR Data Plugins

AiDA-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.
5.1.2.1 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:**

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

F = FleurinpData(files=['path_to_inp.xml_file', <other files>])
```

(continues on next page)
# 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.
- **inp_version**: Returns the version of the stored inp.xml
- **parser_info**: Returns errors, warnings and information encountered while constructing the inp_dict from the inp.xml

Note: FleurinpData will use the masci-tools library to parse the inp.xml. This library contains the schema files for the fleur input and output XML files for many of the fleur releases starting from version 0.27. If a version is encountered that is not yet stored in the installed version of the masci-tools library, the latest available version is used.

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 the inp.xml and get a dictionary with the corresponding calculation modes (Noco, SOC, ...)
- **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. If multiple k-point sets are defined (Fleur release MaX5 or later) a dictionary of KpointsData types is returned file. This only works if the kpoints are listed in the inp.xml.
- **get_parameterdata()** - A CalcFunction that extracts a Dict node containing FLAPW parameters. This node can be used as an input for inpgen.
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:

```python
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 Pt atoms will get default labels, but ‘Fe’ atom will get 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:

```python
inpxml_changes = [('set_species_label', {'at_label': '123',
                                          'attributedict': {
                                            'special': {
                                              'socscale': 0.0
                                            },
                                            'create': True
                                          }},
                         'create': True
                       ]

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

Contents

- FleurinpModifier
  - Description
  - Usage
  - User Methods
    * General methods
    * Modification registration methods
  - Modifying the density matrix for LDA+U calculations
  - Usage in Workflows
    * Explicit definition
    * Using inpxml_changes()

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:

```python
from aiida_fleur.data.fleurinpmodifier import FleurinpModifier

F = FleurinpData(files=['inp.xml'])
fm = FleurinpModifier(F)  # Initialise FleurinpModifier
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 \texttt{FleurinpModifier} class.

1. Initialise \texttt{FleurinpModifier} object:

\begin{verbatim}
from aiida_fleur.data.fleurinpmodifier import FleurinpModifier
modification = FleurinpModifier(fleurinp)
\end{verbatim}

2. Register modifications:

\begin{verbatim}
modification.set_inpchanges({'\phi': 0.33079})
modification.set_inpchanges({'\theta': 1.57079})
modification.set_species('W-1', {'mtSphere': {'radius': 3.5}})
modification.undo()
\end{verbatim}

3. Apply modifications:

\begin{verbatim}
modified_fleurinp = modification.freeze()
\end{verbatim}

**User Methods**

**General methods**

- \texttt{validate()}: Tests if the changes in the given list are validated.
- \texttt{freeze()}: Applies all the changes in the list, calls \texttt{modify_fleurinpdata()} and returns a new \texttt{FleurinpData} object.
- \texttt{changes()}: Displays the current list of changes.
- \texttt{undo()}: Remove the last registered change or all registered changes
- \texttt{show()}: Applies the modifications and displays/prints the resulting \texttt{inp.xml} file. Does not generate a new \texttt{FleurinpData} object.

**Modification registration methods**

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

- \texttt{xml_set_attrib_value_no_create()}: Set an attribute on the specified xml elements to the specified value(s). The \texttt{occurrences} argument can be used to select, which occurrences to modify
- \texttt{xml_set_text_no_create()}: Set the text on the specified xml elements to the specified value(s). The \texttt{occurrences} argument can be used to select, which occurrences to modify
- \texttt{xml_create_tag()}: Insert an xml element in the xml tree.
- \texttt{xml_replace_tag()}: Replace an xml element in the xml tree.
- \texttt{xml_delete_tag()}: Delete an xml element in the xml tree.
- \texttt{xml_delete_att()}: Delete an attribute on a xml element in the xml tree.
On the other hand, there are shortcut methods that already know some paths:

- `set_species()`: Specific user-friendly method to change species parameters.
- `set_atomgroup()`: 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_atomgroup_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. *(Only for Max4 and earlier)*
- `set_kpath()`: Specific method to set a kpoint path for bandstructures *(Only for Max4 and earlier)*
- `set_kpointlist()`: Specific method to set the used kpoints via a array of coordinates and weights
- `set_kpointsdata()`: User-friendly method used to writes kpoints of a KpointsData node to the inp.xml file. It replaces old kpoints for MaX4 versions and older. for MaX5 and later the kpoints are entered as a new kpoint list
- `switch_kpointset()`: Specific method to switch the used kpoint set. *(Only for Max5 and later)*
- `set_attrib_value()`: user-friendly method for setting attributes in the xml file by specifying their name
- `set_first_attrib_value()`: user-friendly method for setting the first occurrence of an attribute in the xml file by specifying its name
- `add_number_to_attrib()`: user-friendly method for adding to or multiplying values of attributes in the xml file by specifying their name
- `add_number_to_first_attrib()`: user-friendly method for adding to or multiplying values of the first occurrence of the attribute in the xml file by specifying their name
- `set_text()`: user-friendly method for setting text on xml elements in the xml file by specifying their name
- `set_first_text()`: user-friendly method for setting the text on the first occurrence of an xml element in the xml file by specifying its name
- `set_simple_tag()`: user-friendly method for creating and setting attributes on simple xml elements (only attributes) in the xml file by specifying its name
- `set_complex_tag()`: user-friendly method for creating complex tags in the xml file by specifying its name
- `create_tag()`: User-friendly method for inserting a tag in the right place by specifying it’s name
- `delete_tag()`: User-friendly method for delete a tag by specifying it’s name
- `delete_att()`: User-friendly method for deleting an attribute from a tag by specifying it’s name
- `replace_tag()`: User-friendly method for replacing a tag by another by specifying its name
- `set_nmpmat()`: Specific method for initializing or modifying the density matrix file for a LDA+U calculation (details see below)
- `rotate_nmpmat()`: Specific method for rotating a block of the density matrix file for a LDA+U calculation (details see below) in real space

In addition there are methods for manipulating the stored files on the FleurinpData instance directly:

- `set_file()`: Set a file on the Fleurinpdata instance
- **del_file()**: Delete a file on the Fleurinpdata instance

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

<table>
<thead>
<tr>
<th>XML methods</th>
<th>Shortcuts</th>
</tr>
</thead>
<tbody>
<tr>
<td>Total number of iterations</td>
<td></td>
</tr>
<tr>
<td><code>xml_set_attrib_value_no_create('/fleurInput/calculationSetup/scfLoop', 'itmax', 29)</code></td>
<td><code>set_inpchanges({'itmax': 29})</code></td>
</tr>
<tr>
<td><strong>Muffin tin radius</strong></td>
<td></td>
</tr>
<tr>
<td><code>xml_set_attrib_value_no_create('/fleurInput/atomSpecies/', 'species@name = &quot;W-1&quot;)/mtSphere', 'radius', 2.2, occurrences=0)</code></td>
<td><code>set_species('W-1', {'mtSphere': {'radius': 2.2}})</code></td>
</tr>
<tr>
<td><strong>beta noco parameter</strong></td>
<td></td>
</tr>
<tr>
<td><code>xml_set_attrib_value_no_create('/fleurInput/atomGroups/atomGroup/', 'atomGroup@species = &quot;W-1&quot;)/nocoParams', 'beta', 1.57, occurrences=0)</code></td>
<td><code>set_atomgroup({'nocoParams': {'beta': 1.57}}, species='W-1')</code></td>
</tr>
</tbody>
</table>

**Warning**: Deprecated XML modification methods

After the `aiida-fleur` release 1.1.4 the FleurinpModifier was restructured to enhance its capabilities and to make it more robust. During this process several modification functions were renamed or deprecated. Even though all the old usage is still supported it is encouraged to switch to the new method names and behaviours:

- `xml_set_attribv_occ()`, `xml_set_all_attribv()` and `xml_set_first_attribv()` are unified in the method `xml_set_attrib_value_no_create()`. However, these functions can no longer create missing subtags.
- `xml_set_text_occ()`, `xml_set_all_text()` and `xml_set_text()` are unified in the method `xml_set_text_no_create()`. However, these functions can no longer create missing subtags.
- `create_tag()` is now a higher-level function. The old function requiring an xpath is now called `xml_create_tag()`.
- `replace_tag()` is now a higher-level function. The old function requiring an xpath is now called `xml_replace_tag()`.
- `delete_tag()` is now a higher-level function. The old function requiring an xpath is now called `xml_delete_tag()`.
- `delete_att()` is now a higher-level function. The old function requiring an xpath is now called `xml_delete_att()` an xml element in the xml tree.
- `add_num_to_att()` was renamed to `add_number_to_attrib()` or `add_number_to_first_attrib()`. However, these are also higher-level functions now longer requiring a concrete xpath.
- `set_atomgr_att()` and `set_atomgr_att_label()` were renamed to `set_atomgroup()` and `set_atomgroup_label()`. These functions now also take the changes in the form `attributedict={'nocoParams': {'beta': val}}` instead of `attributedict={'nocoParams': [{'beta': val}]}`.

5.1. User's guide
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.

```python
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
    'l': 3, 'U': 6.76, 'J': 0.76, 'l_amf': 'F'}})
fm.set_nmmpmat('Nd-1', orbital=3, spin=1, occStates=[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.

Usage in Workflows

The `FleurinpModifier` class can be used nicely to explicitly modify the `FleurinpData` instances in scripts. However, when a `inp.xml` file should be modified during the run of a aiida-fleur workflow this class cannot be used directly. Each workflow specifies a `wf_parameters` dictionary input (potentially more in sub workflows), which contains a `inpxml_changes` entry. This entry can be used to modify the used inputs inside the workchain at points defined by the workflow itself. The syntax for the `inpxml_changes` entry is as follows:
**Explicit definition**

```python
wf_parameters = {
    'inpxml_changes': [
        ('set_inpchanges', {'changes': {'dos': True}}),
        ('set_species', {'species_name': 'Fe-1', 'changes': {'electronConfig': {'flipspins': True}}})
    ]
}
```

is equivalent to

```python
from aiida_fleur.data.fleurinpmodifier import FleurinpModifier

F = FleurinpData(files=['inp.xml'])
fm = FleurinpModifier(F)
fm.set_inpchanges({'dos': True})
fm.set_species('Fe-1', {'electronConfig': {'flipspins': True}})
```

**Using inpxml_changes()**

As can be seen from the above example, the syntax for the `inpxml_changes` entry is quite verbose, especially if compared with the more compact formulation using the `FleurinpModifier` directly. For this reason a helper function `inpxml_changes()` is implemented to construct the `inpxml_changes` entry with the exact same syntax as the `FleurinpModifier`

It is used as a context manager, which behaves exactly like the Modifier inside its with block and enters a `inpxml_changes` entry into the dictionary passed to this function after the with block is terminated.

```python
from aiida_fleur.data.fleurinpmodifier import inpxml_changes

parameters = {}
with inpxml_changes(parameters) as fm:
    fm.set_inpchanges({'dos': True})
    fm.set_species('Fe-1', {'electronConfig': {'flipspins': True}})

print(parameters)
```

```python
from aiida_fleur.data.fleurinpmodifier import inpxml_changes

from aiida import plugins

FleurBandDOS = plugins.WorkflowFactory('fleur.banddos')
inputs = FleurBandDOS.get_builder()

with inpxml_changes(inputs) as fm:
    fm.set_inpchanges({'dos': True})
    fm.set_species('Fe-1', {'electronConfig': {'flipspins': True}})
```
5.1.3 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.

5.1.3.1 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:

<table>
<thead>
<tr>
<th>name</th>
<th>type</th>
<th>description</th>
<th>required</th>
</tr>
</thead>
<tbody>
<tr>
<td>code</td>
<td>Code</td>
<td>Inpgen code</td>
<td>yes</td>
</tr>
<tr>
<td>structure</td>
<td>StructureData</td>
<td>Structure data node</td>
<td>yes</td>
</tr>
<tr>
<td>parameters</td>
<td>Dict</td>
<td>FLAPW parameters</td>
<td>no</td>
</tr>
<tr>
<td>metadata.options</td>
<td>Dict</td>
<td>computational resources</td>
<td>yes</td>
</tr>
<tr>
<td>metadata.label</td>
<td>string</td>
<td>computational resources</td>
<td>yes</td>
</tr>
<tr>
<td>metadata.description</td>
<td>string</td>
<td>computational resources</td>
<td>yes</td>
</tr>
</tbody>
</table>

- **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:

```python
# -*- coding: utf-8 -*-
Cd = Dict(dict={
    'atom': {'element': 'Cd', 'rmt': 2.5, 'jri': 981, 'lmax': 12,
             'lnonsph': 6, 'lo': '4d',
             'ecfin': '[Ar] 4s2 3d10 4p6 | 4d10 5s2'},
    'comp': {'lmax': 24, 'gmaxxc': 12.0, 'gmax': 14.0},
    'kpt': {'div1': 4.7, 'div2': 17, 'div3': 17, 'tkb': 0.0005}})
# Magnetism and spin orbit coupling
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': {'lmax': 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', 'ecfin', 'bmu', 'lo', 'element', 'name']
- 'comp': ['jspins', 'frcor', 'ctail', 'kcrel', 'gmax', 'gmaxxc', 'kmax']
- 'exco': ['xctyp', 'relxc'],
- 'film': ['dvac', 'dtild'],
- 'soc': ['theta', 'phi'],

(continues on next page)
'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`:

<table>
<thead>
<tr>
<th>name</th>
<th>type</th>
<th>comment</th>
</tr>
</thead>
<tbody>
<tr>
<td>fleurinp</td>
<td>FleurinpData</td>
<td>represents inp.xml</td>
</tr>
<tr>
<td>remote_folder</td>
<td>FolderData</td>
<td>represents calculation folder</td>
</tr>
<tr>
<td>retrieved</td>
<td>FolderData</td>
<td>represents retrieved folder</td>
</tr>
</tbody>
</table>

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

- **fleurinp**: 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.

<table>
<thead>
<tr>
<th>Exit code</th>
<th>Reason</th>
</tr>
</thead>
<tbody>
<tr>
<td>251</td>
<td>Input parameters for inpgen contain unknown keys</td>
</tr>
<tr>
<td>253</td>
<td>Fleur lattice needs atom positions as input</td>
</tr>
<tr>
<td>254</td>
<td>Excessive input parameters were specified</td>
</tr>
<tr>
<td>300</td>
<td>No retrieved folder found</td>
</tr>
<tr>
<td>301</td>
<td>One of the output files can not be opened</td>
</tr>
<tr>
<td>306</td>
<td>XML input file was not found</td>
</tr>
<tr>
<td>307</td>
<td>Some required files were not retrieved</td>
</tr>
</tbody>
</table>
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'],
}
```

5.1.3.2 FLEUR code plugin

Description

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

<table>
<thead>
<tr>
<th>name</th>
<th>type</th>
<th>description</th>
<th>required</th>
</tr>
</thead>
<tbody>
<tr>
<td>code</td>
<td>Code</td>
<td>Fleur code</td>
<td>yes</td>
</tr>
<tr>
<td>fleurinp</td>
<td>FleurinpData</td>
<td>Object representing inp.xml</td>
<td>no</td>
</tr>
<tr>
<td>parent_folder</td>
<td>RemoteData</td>
<td>Remote folder of another calculation</td>
<td>no</td>
</tr>
<tr>
<td>settings</td>
<td>Dict</td>
<td>special settings</td>
<td>no</td>
</tr>
<tr>
<td>metadata.options</td>
<td>Dict</td>
<td>computational resources</td>
<td>yes</td>
</tr>
</tbody>
</table>

- **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**:

<table>
<thead>
<tr>
<th>name</th>
<th>type</th>
<th>comment</th>
</tr>
</thead>
<tbody>
<tr>
<td>output_parameters</td>
<td>Dict</td>
<td>contains parsed <code>out.xml</code></td>
</tr>
<tr>
<td>remote_folder</td>
<td>FolderData</td>
<td>represents calculation folder</td>
</tr>
<tr>
<td>retrieved</td>
<td>FolderData</td>
<td>represents retrieved folder</td>
</tr>
</tbody>
</table>

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:

```python
# -*- 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': -99592.98569309,
    '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': [2.7677822875, 2.47601e-05],
}
```

(continues on next page)
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,
6.7734197578,
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',
}
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:

```bash
(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 finish successfully.
  --> 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:

```json

'time': '10:38:24',
'sum_of_eigenvalues': -99751.687121869,
'title': 'A Fleur input generator calulation with aiida',
'unparsed': [],
'walltime': 7923,
'walltime_units': 'seconds',
'warnings': {
  'debug': {},
  'error': {},
  'info': {},
  'warning': {}
}
```
<table>
<thead>
<tr>
<th>Exit code</th>
<th>Reason</th>
</tr>
</thead>
<tbody>
<tr>
<td>300</td>
<td>One of output files can not be opened</td>
</tr>
<tr>
<td>301</td>
<td>No retrieved folder found</td>
</tr>
<tr>
<td>302</td>
<td>FLEUR calculation failed for unknown reason</td>
</tr>
<tr>
<td>303</td>
<td>XML output file was not found</td>
</tr>
<tr>
<td>304</td>
<td>Parsing of XML output file failed</td>
</tr>
<tr>
<td>305</td>
<td>Parsing of relax XML output file failed</td>
</tr>
<tr>
<td>310</td>
<td>FLEUR calculation failed due to memory issue</td>
</tr>
<tr>
<td>311</td>
<td>FLEUR calculation failed because atoms spilled to the vacuum</td>
</tr>
<tr>
<td>312</td>
<td>FLEUR calculation failed due to MT overlap</td>
</tr>
<tr>
<td>313</td>
<td>FLEUR calculation failed due to MT overlap during relaxation</td>
</tr>
<tr>
<td>314</td>
<td>Problem with cdn is suspected</td>
</tr>
<tr>
<td>315</td>
<td>Invalid Elements found in the LDA+U density matrix.</td>
</tr>
<tr>
<td>316</td>
<td>Calculation failed due to time limits.</td>
</tr>
</tbody>
</table>

**Parallelization options**

For parallel FLEUR calculations the input under metadata.options can be used. In higher level workchains this input might be present as a plain options input, but it is completely equivalent to the metadata.options input.

```python
inputs.metadata.options = {
    'resources': {
        'num_machines': 2, #Number of computing nodes
        'num_mpiracs_per_machine': 4, #Number of MPI processes per node
        'num_cpus_per_mpiproc': 12, #Number of OMP threads per MPI process
    },
    'withmpi': True, #This flag makes sure that the process is submitted using MPI
    'max_wallclock_seconds': 3600, #Maximum wallclock time in seconds
}
```

This will result in setting the following slurm Parallelization variables in the submit script.

```
#SBATCH --nodes=1
#SBATCH --ntasks-per-node=6
#SBATCH --cpus-per-task=8
#SBATCH --time=01:00:00
#... Further configuration options unrelated to parallelization...
'srun '/path/to/fleur/" '<further FLEUR cmdline flags, e.g. -last_extra>
```

Note, that the srun command is computer specific and is configured in verdi computer setup with the Mpirun command option.
Additional advanced features

In general see the FLEUR documentation.

While the input link with name fleurinp 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:

\[
\text{calc.use_settings(Dict(dict=\text{settings_dict}))}
\]

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:

\[
\text{settings_dict} = \{
\quad \text{'cmdline': ['-hdf', '-magma'],}
\}
\]

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

\[
\text{'mpirun' '-np' 'XX' 'path_to_fleur_executable' '-wtime' 'XXX' < 'inp.xml' > 'shell.out' 2> 'out.error'}
\]

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:

\[
\text{'mpirun' '-np' 'XX' 'path_to_fleur_executable' '-last_extra' '-wtime' 'XXX' < 'inp.xml' > 'shell.out' 2> 'out.error'}
\]

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):

\[
\text{settings_dict} = \{
\quad \text{'additional_retrieve_list': ['testfile.txt'],}
\}
\]
**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 too 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`):

```python
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`):

```python
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`):

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

### 5.1.4 AiiDA-FLEUR WorkChains

#### 5.1.4.1 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:** The `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:

```python
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.

**5.1.4.2 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.

5.1.4.3 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

• Fleur base restart workchain
  – Description/Purpose
  – check_kpts() method
  – Errors

Import Example:

```python
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()` function is used with `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_mipiprocs_per_machine" and omp_per_mpi is taken from "num_cores_per_mpioprooc" 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_mipiprocs_per_machine" / "num_cores_per_mpioprooc" and number of available CPUs per node is calculated as "num_mipiprocs_per_machine" * "num_cores_per_mpioprooc". In other case, when "num_cores_per_mpioprooc" is not given, use_omp is set to `False` and the number of available CPUs per node is assumed to be equal to "num_mipiprocs_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:

```python
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 Reference of Exit codes.

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

Exit codes duplicating FleurCalculation exit codes:

<table>
<thead>
<tr>
<th>Exit code</th>
<th>Reason</th>
</tr>
</thead>
<tbody>
<tr>
<td>311</td>
<td>FleurCalculation failed because an atom spilled to the vacuum during relaxation.</td>
</tr>
<tr>
<td>312</td>
<td>FleurCalculation failed due to MT overlap.</td>
</tr>
<tr>
<td>399</td>
<td>FleurCalculation failed and FleurBaseWorkChain has no strategy to resolve this</td>
</tr>
</tbody>
</table>

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

- Fleur self-consistency field workflow
  - Description/Purpose
  - Input nodes
    * Workchain parameters and its defaults
  - Returns nodes
  - Layout
  - Error handling
  - Plot_fleur visualization
  - Database Node graph
  - Example usage

Import Example:

```python
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.

<table>
<thead>
<tr>
<th>name</th>
<th>type</th>
<th>description</th>
<th>required</th>
</tr>
</thead>
<tbody>
<tr>
<td>fleur</td>
<td>Code</td>
<td>Fleur code</td>
<td>yes</td>
</tr>
<tr>
<td>inpgen</td>
<td>Code</td>
<td>Inpgen code</td>
<td>no</td>
</tr>
<tr>
<td>wf_parameters</td>
<td>Dict</td>
<td>Settings of the workchain</td>
<td>no</td>
</tr>
<tr>
<td>structure</td>
<td>StructureData</td>
<td>Structure data node</td>
<td>no</td>
</tr>
<tr>
<td>calc_parameters</td>
<td>Dict</td>
<td>inpgen parameters</td>
<td>no</td>
</tr>
<tr>
<td>fleurinp</td>
<td>FleurinpData</td>
<td>FLEUR input</td>
<td>no</td>
</tr>
<tr>
<td>remote_data</td>
<td>RemoteData</td>
<td>Remote folder of another calculation</td>
<td>no</td>
</tr>
<tr>
<td>options</td>
<td>Dict</td>
<td>AiiDA options (computational resources)</td>
<td>no</td>
</tr>
<tr>
<td>settings</td>
<td>Dict</td>
<td>Special settings for Fleur calculation</td>
<td>no</td>
</tr>
</tbody>
</table>
Only \texttt{fleur} input is required. However, it does not mean that it is enough to specify \texttt{fleur} only. One \textit{must} keep one of the supported input configurations described in the \textit{Layout} section.

\section*{Workchain parameters and its defaults}

\begin{itemize}
\item \textbf{wf\_parameters:} \texttt{Dict} - Settings of the workflow behavior. All possible keys and their defaults are listed below:

\begin{verbatim}
# -*- 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'
'
'add_comp_para': {
    'only_even_MPI': \texttt{False}, # True if suppress parallelisation having odd
    'number of MPI
    'max_queue_nodes': 20, # Max number of nodes allowed (used by
    automatic error fix)
    'max_queue_wallclock_sec': 86400 # Max number of walltime allowed (used by
    automatic error fix)
},
'itmax_per_run': 30, # Maximum iterations run for one

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

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

\textbf{Note:} Only one of \texttt{density\_converged}, \texttt{energy\_converged} or \texttt{force\_converged} is used by the workchain that corresponds to the \texttt{'mode'}. The other two are ignored. Exception: force mode uses both \texttt{density\_converged} and \texttt{force\_converged} because \textit{FLEUR} code always converges density before forces.

\item \textbf{options:} \texttt{Dict} - \textit{AiiDA} options (computational resources). Also see \textit{Parallelization options} section. Example:

\begin{verbatim}
'resources': {'num_machines': 1, 'num_mpiprocs_per_machine': 1},
'max_wallclock_seconds': 6*60*60,
'queue_name': '',
'custom_scheduler_commands': '',
'import_sys_environment': \texttt{False},
'environment_variables': {}\end{verbatim}
\end{itemize}
Returns nodes

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

<table>
<thead>
<tr>
<th>name</th>
<th>type</th>
<th>comment</th>
</tr>
</thead>
<tbody>
<tr>
<td>output_scf_wc_para</td>
<td>Dict</td>
<td>results of the workchain</td>
</tr>
<tr>
<td>fleurinp</td>
<td>FleurinpData</td>
<td>FleurinpData that was used (after all modifications)</td>
</tr>
<tr>
<td>last_calc</td>
<td>Namespace</td>
<td>Link to all output nodes (out dict, retrieved) of last Fleur calculation</td>
</tr>
</tbody>
</table>

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_calc**: namespace - A link to the output nodes of the last Fleur calculation. This includes the retrieved files, remote folder and output dictionary

- **output_scf_wc_para**: *Dict* - Main results of the workchain. Contains errors, warnings, convergence history and other information. An example:

```python
# -*- coding: utf-8 -*-
{
    'conv_mode': 'density',
    'distance_charge': 0.1406279038,
    'distance_charge_all': [
        61.1110641131,
        43.7556515683,
        ...
    ],
    'distance_charge_units': 'me/bohr^3',
    'errors': [],
    'force_diff_last': 'can not be determined',
    'force_largest': 0.0,
    'info': [],
    'iterations_total': 23,
    '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.

4. **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**:

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

```python
from aiida_fleur.tools.plot import plot_fleur
plot_fleur(50816)
```

Multi node

```python
from aiida_fleur.tools.plot import plot_fleur
plot_fleur(scf_pk_list)
```
**Database Node graph**

```python
from aiida_fleur.tools.graph_fleur import draw_graph
draw_graph(50816)
```

**Example usage**

```python
# -*- 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,
    'add_comp_para': {
```
only_even_MPI: False,
'max_queue_nodes': 20,
'max_queue_wallclock_sec': 86400
}}

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

- **Fleur equation of states (eos) workflow**
  - Description/Purpose
  - Input nodes
  - Returns nodes
  - Layout
  - Database Node graph
  - Plot_fleur visualization
  - Example usage
  - Output node example
  - Error handling
- Exit codes

Import Example:

```python
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.

<table>
<thead>
<tr>
<th>name</th>
<th>type</th>
<th>description</th>
<th>required</th>
</tr>
</thead>
<tbody>
<tr>
<td>scf</td>
<td>namespace</td>
<td>inputs for nested SCF WorkChain. structure input is excluded</td>
<td>no</td>
</tr>
<tr>
<td>wf_parameters</td>
<td>Dict</td>
<td>Settings of the workchain</td>
<td>no</td>
</tr>
<tr>
<td>structure</td>
<td>StructureData</td>
<td>input structure</td>
<td>no</td>
</tr>
</tbody>
</table>

Returns nodes

<table>
<thead>
<tr>
<th>name</th>
<th>type</th>
<th>comment</th>
</tr>
</thead>
<tbody>
<tr>
<td>output_eos_wc_para</td>
<td>Dict</td>
<td>results of the workchain</td>
</tr>
<tr>
<td>output_eos_wc_structure</td>
<td>StructureData</td>
<td>Crystal structure with the volume of the lowest total energy</td>
</tr>
</tbody>
</table>

Layout
Database Node graph

```python
from aiida_fleur.tools.graph_fleur import draw_graph
draw_graph(49670)
```

Plot_fleur visualization

Single node

```python
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
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,  
                          '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': [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,
    16.22601543666224,
    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:

<table>
<thead>
<tr>
<th>Code</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>230</td>
<td>Invalid workchain parameters</td>
</tr>
</tbody>
</table>

Fleur structure optimization Base workchain

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

Import Example:

```python
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

- Fleur structure optimization workchain
  - Description/Purpose
  - Input nodes
    - Workchain parameters and its defaults
  - Output nodes
  - Layout
  - Output nodes
  - Error handling
  - Example usage

Import Example:

```python
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.

<table>
<thead>
<tr>
<th>name</th>
<th>type</th>
<th>description</th>
<th>required</th>
</tr>
</thead>
<tbody>
<tr>
<td>scf</td>
<td>namespace</td>
<td>inputs for nested SCF WorkChain</td>
<td>yes</td>
</tr>
<tr>
<td>final_scf</td>
<td>namespace</td>
<td>inputs for a final SCF WorkChain</td>
<td>no</td>
</tr>
<tr>
<td>wf_parameters</td>
<td>Dict</td>
<td>Settings of the workchain</td>
<td>no</td>
</tr>
</tbody>
</table>

Workchain parameters and its defaults

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

```python
# -*- 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**: FleurinpData - 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

<table>
<thead>
<tr>
<th>name</th>
<th>type</th>
<th>comment</th>
</tr>
</thead>
<tbody>
<tr>
<td>output_relax_wc_para</td>
<td>Dict</td>
<td>results of the workchain</td>
</tr>
<tr>
<td>optimized_structure</td>
<td>FleurinpData</td>
<td>FleurinpData that was used (after all modifications)</td>
</tr>
</tbody>
</table>

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

<table>
<thead>
<tr>
<th>Exit code</th>
<th>Reason</th>
</tr>
</thead>
<tbody>
<tr>
<td>311</td>
<td>FLEUR calculation failed because atoms spilled to the vacuum</td>
</tr>
<tr>
<td>313</td>
<td>Overlapping MT-spheres during relaxation</td>
</tr>
</tbody>
</table>

Example usage

```python
# -*- 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
}
```

(continues on next page)
```python
import aiiada

wf_relax_scf = {
    'fleur_runmax': 5,
    '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_mpi_procs_per_machine': 4,
        'num_cores_per_mpi_proc': 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,
        'inp': inpgen_code,
        'fleur': fleur_code
    }
}
```

(continues on next page)
Fleur dos/band workflow

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

- **Class:** `FleurBandDosWorkChain`
- **String to pass to the WorkflowFactory():** `fleur.banddos`
- **Workflow type:** Workflow (lvl 1)
- **Aim:** Calculate a density of states. Calculate a band structure.
- **Computational demand:** 1 Fleur Job calculation + 1 (optional) Fleur SCF workflow
- **Database footprint:** Outputnode with information, full provenance, ~ 10 nodes (more if SCF workflow is included)
- **File repository footprint:** The JobCalculation run, plus the DOS or Bandstructure files

**Contents**

- **Fleur dos/band workflow**
  - **Description/Purpose**
  - **Input nodes**
  - **Returns nodes**
    - Workchain parameters and its defaults
      - `wf_parameters`
      - `options`
  - **Supported input configurations**
  - **Database Node graph**
  - **Plot_fleur visualization**
  - **Example usage**
  - **Output node example**
  - **Error handling**

Import Example:

```python
from aiida_fleur.workflows.banddos import FleurBandDosWorkChain
# or
WorkflowFactory('fleur.banddos')
```
Description/Purpose

Calculates an electronic band structure on top of a given Fleur calculation (converged or not). It can be started from the crystal structure utilizing the FleurSCFWorkchain as a subworkchain.

This workflow prepares/changes the Fleur input with respect to the kpoint set and bandstructure/DOS related parameters and manages one Fleur calculation.

Input nodes:

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

<table>
<thead>
<tr>
<th>name</th>
<th>type</th>
<th>description</th>
<th>required</th>
</tr>
</thead>
<tbody>
<tr>
<td>scf</td>
<td>namespace</td>
<td>inputs for nested SCF WorkChain</td>
<td>no</td>
</tr>
<tr>
<td>fleur</td>
<td>Code</td>
<td>Fleur code</td>
<td>yes</td>
</tr>
<tr>
<td>wf_parameters</td>
<td>Dict</td>
<td>Settings of the workchain</td>
<td>no</td>
</tr>
<tr>
<td>fleurinp</td>
<td>FleurinpData</td>
<td>FLEUR input</td>
<td>no</td>
</tr>
<tr>
<td>remote</td>
<td>RemoteData</td>
<td>Remote folder of another calculation</td>
<td>no</td>
</tr>
<tr>
<td>kpoints</td>
<td>KpointsData</td>
<td>Kpoint-set to use</td>
<td>no</td>
</tr>
<tr>
<td>options</td>
<td>Dict</td>
<td>AiiDA options (computational resources)</td>
<td>no</td>
</tr>
</tbody>
</table>

Only the 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.

Returns nodes

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

<table>
<thead>
<tr>
<th>name</th>
<th>type</th>
<th>comment</th>
</tr>
</thead>
<tbody>
<tr>
<td>output_banddos_wc_para</td>
<td>Dict</td>
<td>results of the workchain</td>
</tr>
<tr>
<td>last_calc_retrieved</td>
<td>FolderData</td>
<td>Link to last FleurCalculation retrieved files</td>
</tr>
</tbody>
</table>

Workchain parameters and its defaults

**wf_parameters**

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

```python
# -*- coding: utf-8 -*-
'
'mode': 'band',
'kpath': 'auto',  #seek (aiida), fleur (only Max4) or string to pass to ase
'klistname': 'path-3',
'kpoints_number': None,
'kpoints_distance': None,
'kpoints_explicit': None,  #dictionary containing a list of kpoints, weights
#and additional arguments to pass to set_kpointlist
'sigma': 0.005,
'emin': -0.50,
```

(continues on next page)
'emax': 0.90,
'add_comp_para': {
    'only_even_MPI': False,
    'max_queue_nodes': 20,
    'max_queue_wallclock_sec': 86400
},
'inxml_changes': [],

**mode** is a string (either `band` (default) or `dos`). Determines, whether a bandstructure or density of states calculation is performed. This sets the `band` and `dos` switches in the output section of the input file accordingly.

**kpath** is only used if `mode='band'` to determine the kpath to use. There are 5 different options here:

- **auto** Will use the default bandpath in fleur for both Max4 or Max5. If `klistname` is given the corresponding kpoint path is used for Max5 version or later
- A **dictionary** specifying the special points and their coordinates. Only available for versions before Max5. Will generate a kpath with `kpoints_number` points
- **seek** will use `get_explicit_kpoints_path()` to generate a kpath with the given `kpoints_distance`.

**Warning:** This functionality only works for standardized primitive unit cells.

- **skip** nothing is done
- **all other strings** are used to generate a k-path using `bandpath()` for example GMKGALHA. This option supports both `kpoints_number` and `kpoints_distance` for specifying the number of points

**kpoints_number** integer specifying the number of kpoints in the k-path (depending on the kpath option)

**kpoints_distance** float specifying the distance between kpoints in the k-path (depending on the kpath option)

**kpoints_explicit** dictionary, which is used to create a new kpointlist in the input. The dictionary is unpacked and used as the argument for the `set_kpointlist()` function

**klistname** str, will be used to switch the used `kPointList` for fleur versions after Max5 (if `kpath='auto'` or `mode='dos'``)

**sigma, emin, emax** floats specifying the energy grid for DOS calculations

**options**

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

```python
'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': {}}```
Supported input configurations

The FleurBandDosWorkChain 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 charge density which will be followed by the band-structure or DOS 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 band/DOS calculation. inp.xml file will be converted to FleurinpData and the charge density will be used as the charge density used in this calculation.

3. **remote + fleurinp**:
   Charge density which belongs to remote will be used as the charge density used in the band/DOS calculation, however the 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 band/DOS 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.

Database Node graph

```python
from aiida_fleur.tools.graph_fleur import draw_graph
draw_graph(76867)
```

Plot_fleur visualization

Single node

```python
from aiida_fleur.tools.plot import plot_fleur

# DOS calc
plot_fleur(76867)
```

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

```python
from aiida_fleur.tools.plot import plot_fleur

plot_fleur(dos_pk_list)
```
Fig. 1: For the bandstructure visualization it depends on the File produced. Old bandstructure file:

![Density of states](image1)

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

![Bandstructure](image2)
Example usage

```python
# -*- coding: utf-8 -*-
from aiida_fleur.workflows.banddos import FleurBandDosWorkChain
from aiida.orm import Dict, load_node
from aiida.engine import submit

def submit_workchain(fleur_code, inpenger_code, structure, options):
    wf_para = Dict(dict=
        {'mode': 'band',
         'kpath': 'auto',  # seek (aiida), fleur (only Max4) or string to pass to ase
         'klistname': 'path-3',
         'kpoints_number': None,
         'kpoints_distance': None,
         'kpoints_explicit': None,  # dictionary containing a list of kpoints,
         'weights':
             # and additional arguments to pass to set_kpointlist
             'sigma': 0.005,
             'emin': -0.50,
             'emax': 0.90,
             'add_comp_para': {
                 'only_even_MPI': False,
                 'max_queue_nodes': 20,
                 'max_queue_wallclock_sec': 86400
             },
             'inpxml_changes': [],
        })

    wf_para_scf = Dict(dict=
        {'fleur_runmax': 3,
         'density_converged': 0.001,
         'mode': 'density',
         'itmax_per_run': 30,
         'add_comp_para': {
             'only_even_MPI': False,
             'max_queue_nodes': 20,
             'max_queue_wallclock_sec': 86400
         }
        })

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

options_scf = Dict(dict={
    'resources': {
        'num_machines': 1,
        'num_mpiprocs_per_machine': 2
    },
    'withmpi': True,
    'max_wallclock_seconds': 600
})

calc_parameters = Dict(dict={
    'kpt': {
        'nkpts': 500,
        'path': 'default'}
})

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

banddos_workchain = submit(FleurBandDosWorkChain, **inputs)

### Output node example

### Error handling

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

<table>
<thead>
<tr>
<th>Exit code</th>
<th>Reason</th>
</tr>
</thead>
<tbody>
<tr>
<td>230</td>
<td>Invalid workchain parameters, please check input configuration</td>
</tr>
<tr>
<td>231</td>
<td>Invalid input configuration and fleur code nodes</td>
</tr>
<tr>
<td>233</td>
<td>Invalid code node specified, check inpgen and fleur code nodes</td>
</tr>
<tr>
<td>235</td>
<td>Input file modification failed</td>
</tr>
<tr>
<td>236</td>
<td>Input file was corrupted after modifications</td>
</tr>
<tr>
<td>334</td>
<td>SCF calculation failed</td>
</tr>
<tr>
<td>335</td>
<td>Found no SCF remote repository.</td>
</tr>
</tbody>
</table>

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.
Fleur orbital occupation control workflow

- **Current version**: 0.2.0
- **Class**: FleurOrbControlWorkChain
- **String to pass to the WorkflowFactory()**: fleur.orbcontrol
- **Workflow type**: Technical
- **Aim**: Find LDA+U groundstate density matrix
- **Computational demand**: Corresponding to several FleurSCFWorkChain (Can be large depending on number of configurations)
- **Database footprint**: Output node with information, full provenance, ~ 10+10*FLEUR Jobs nodes (Can be large depending on number of configurations)
- **File repository footprint**: no addition to the CalcJob runs

**Contents**

- Fleur orbital occupation control workflow
  - Description/Purpose
  - Input nodes
    - Workchain parameters and its defaults
  - Returns nodes
  - Layout
  - Error handling
  - Plot_fleur visualization

**Import Example:**

```python
from aiida_fleur.workflows.orbcontrol import FleurOrbControlWorkChain
#or
WorkflowFactory('fleur.orbcontrol')
```

**Description/Purpose**

Converges the given system with the FleurSCFWorkChain with different starting configurations for the LDA+U density matrix. Each calculation starts with a fixed density matrix which is used for a configurable number of iterations. After these calculations the density matrix can relax until the system is converged by the FleurSCFWorkChain

This workflow can be started from either a structure or a already converged calculation **without LDA+U**. The used configurations can either be provided explicitely or be generated from the given occupations of the orbital treated with LDA+U.
Input nodes

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

<table>
<thead>
<tr>
<th>name</th>
<th>type</th>
<th>description</th>
<th>required</th>
</tr>
</thead>
<tbody>
<tr>
<td>scf_no_ldau</td>
<td>namespace</td>
<td>Inputs for SCF calculation before adding LDA+U</td>
<td>no</td>
</tr>
<tr>
<td>remote</td>
<td>RemoteData</td>
<td>Remote folder to start the calculations from</td>
<td>no</td>
</tr>
<tr>
<td>fleurinp</td>
<td>FleurinpData</td>
<td>FLEUR input</td>
<td>no</td>
</tr>
<tr>
<td>structure</td>
<td>StructureData</td>
<td>Structure to start from without SCF</td>
<td>no</td>
</tr>
<tr>
<td>calc_parameters</td>
<td>Dict</td>
<td>Parameters for Inpgen calculation</td>
<td>no</td>
</tr>
<tr>
<td>scf_with_ldau</td>
<td>namespace</td>
<td>Inputs for SCF calculations with LDA+U</td>
<td>yes</td>
</tr>
<tr>
<td>fleur</td>
<td>Code</td>
<td>Fleur code</td>
<td>yes</td>
</tr>
<tr>
<td>inpgen</td>
<td>Code</td>
<td>Inpgen Code</td>
<td>no</td>
</tr>
<tr>
<td>wf_parameters</td>
<td>Dict</td>
<td>Settings of the workchain</td>
<td>no</td>
</tr>
<tr>
<td>options</td>
<td>Dict</td>
<td>AiiDA options (computational resources)</td>
<td>no</td>
</tr>
<tr>
<td>options_inpgen</td>
<td>Dict</td>
<td>AiiDA options (computational resources) for the inpgen calculation</td>
<td>no</td>
</tr>
<tr>
<td>settings</td>
<td>Dict</td>
<td>Special settings for Fleur calculation</td>
<td>no</td>
</tr>
<tr>
<td>settings_inpgen</td>
<td>Dict</td>
<td>Special settings for INpgen calculation</td>
<td>no</td>
</tr>
</tbody>
</table>

Only fleur and scf_with_ldau input is required. However, it does not mean that it is enough to specify these 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:

```python
# coding: utf-8

#Number of iterations to run with fixed density matrices
'iterations_fixed': 30,

#Specifications of the LDA+U parameters to add
'ldau_dict': {'all-Nd': {'l': 3, 'U': 6.7},

#Note that the input has to be without LDA+U

(continues on next page)```
J: 0.7, # for this workflow to work

<table>
<thead>
<tr>
<th>consistently</th>
<th>'l_amf': False</th>
</tr>
</thead>
</table>

| 'use_orbital_occupation': False, # If True the obtained |
| configurations are used for the |
| spherical harmonics |

| fixed_occupations': {'all-Nd': {3: (4,0)}}, # Specifies the occupations for each |
| LDA+U orbital |

| possible configurations from |
| fixed_configurations': None, # Alternative way to specify density |
| matrix configurations |

| configurations to use |

Note: Only one of fixed_occupations or fixed_configurations can be used

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

```python
'resources': {'num_machines': 1, 'num_mpi_procs_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.

<table>
<thead>
<tr>
<th>name</th>
<th>type</th>
<th>comment</th>
</tr>
</thead>
<tbody>
<tr>
<td>output_orbcontrol_wc_para</td>
<td>Dict</td>
<td>results of the workchain</td>
</tr>
<tr>
<td>output_orbcontrol_wc_gs_scf</td>
<td>Dict</td>
<td>results of the SCF workchain with the lowest total energy</td>
</tr>
<tr>
<td>output_orbcontrol_wc_gs_fleurinp</td>
<td>FleurinpData</td>
<td>FleurinpData corresponding to the calculation with the lowest total energy</td>
</tr>
</tbody>
</table>

More details:

- output_orbcontrol_wc_gs_fleurinp: FleurinpData - A FleurinpData that was actually used for the groundstate FleurScfWorkChain calculation. It differs from the input FleurinpData because there are some hard-coded modifications in the SCF workchain and the used LDA+U density matrix is included with the file n_mmp_mat.

- output_orbcontrol_wc_para: Dict - Main results of the workchain. Contains errors, warnings, convergence history and other information. An example:

```python
# -*- coding: utf-8 -*-
{
    'configurations': {'all-Nd-3': [[[1,1,1,1,0,0,0], # Lists of used
         'configurations' |
```
(continued from previous page)

\[
(1,1,1,0,1,0,0), \quad \# \text{for all LDA+U orbitals,}
\]

\[
\rightarrow \text{and spin}
\]

\[
..., \quad (0,0,0,0,0,0,0), \quad (0,0,0,0,0,0,0), \quad ...
\]

\[
'\text{total}\_\text{energy}': [ 
  -38166.542950054, \quad 
  -38166.345602746, \quad ...
],
\]

\[
'\text{total}\_\text{energy}\_\text{units}': '\text{Htr}',
\]

\[
'\text{distance}\_\text{charge}': [
  0.000001, 
  0.0000023, 
  ...
],
\]

\[
'\text{distance}\_\text{charge}\_\text{units}': '\text{me/bohr}^3',
\]

\[
'successful\_\text{configs}': [0,1,2,3,...], \quad \# \text{Which configurations}
\]

\[
'non\_\text{successful}\_\text{configs}': [], \quad \# \text{Which configurations}
\]

\[
'did\_\text{not}\_\text{converge}
\]

\[
'\text{failed}\_\text{configs}': [], \quad \# \text{Which configurations}
\]

\[
'\text{for another reason}
\]

\[
'\text{info}': [],
\]

\[
'\text{warnings}': [],
\]

\[
'\text{errors}': [],
\]

\[
'\text{workflow}\_\text{name}': 'FleurOrbControlWorkChain',
\]

\[
'\text{workflow}\_\text{version}': '0.1.0'
\]
The initial structure is used to generate a *FleuringData* object via the input generator. This is used to start the LDA+U calculations without a SCF workchain, directly starting with the fixed LDA+U density matrices.

5. **scf_no_ldau**:

   A *FleurSCFWorkChain* is started with the input in the **scf_no_ldau** namespace and the output is used as a starting point for the LDA+U calculations.

   **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.

**Error handling**

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

<table>
<thead>
<tr>
<th>Exit Code</th>
<th>Reason</th>
</tr>
</thead>
<tbody>
<tr>
<td>230</td>
<td>Invalid workchain parameters</td>
</tr>
<tr>
<td>231</td>
<td>Invalid input configuration</td>
</tr>
<tr>
<td>233</td>
<td>Invalid code node specified, check fleur code nodes</td>
</tr>
<tr>
<td>235</td>
<td>Input file modification failed</td>
</tr>
<tr>
<td>236</td>
<td>Input file was corrupted after modifications</td>
</tr>
<tr>
<td>342</td>
<td>Some of the LDA+U calculations failed</td>
</tr>
<tr>
<td></td>
<td>This is expected for many situations</td>
</tr>
<tr>
<td>343</td>
<td>All of the LDA+U calculations failed</td>
</tr>
<tr>
<td>360</td>
<td>The inpren calculation failed</td>
</tr>
<tr>
<td>450</td>
<td>SCF calculation without LDA+U failed</td>
</tr>
</tbody>
</table>

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**

```python
from aiida_fleur.tools.plot import plot_fleur
plot_fleur(50816)
```
5.1.4.4 More advanced (Scientific) Workchains

Advanced workchains can be divided into categories according to subject.

XPS workchains:

- **Current version**: 0.4.0

**Fleur initial core-level shifts workflow**

Class name, import from:

```
from aiida_fleur.workflows.initial_cls import FleurInitialCLSWorkChain
```

**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

[Diagram of workflow process]

### Input nodes

<table>
<thead>
<tr>
<th>name</th>
<th>type</th>
<th>description</th>
<th>required</th>
</tr>
</thead>
<tbody>
<tr>
<td>inpgen</td>
<td>Code</td>
<td>Inpgen code</td>
<td>yes</td>
</tr>
<tr>
<td>fleur</td>
<td>Code</td>
<td>Fleur code</td>
<td>yes</td>
</tr>
<tr>
<td>wf_parameters</td>
<td>Dict</td>
<td>Settings of the workchain</td>
<td>no</td>
</tr>
<tr>
<td>fleurinp</td>
<td>FleurinpData</td>
<td><em>FLEUR input</em></td>
<td>no</td>
</tr>
<tr>
<td>structure</td>
<td>StructureData</td>
<td>Crystal structure</td>
<td>no</td>
</tr>
<tr>
<td>calc_parameters</td>
<td>Dict</td>
<td>FLAPW parameters for given structure</td>
<td>no</td>
</tr>
<tr>
<td>options</td>
<td>Dict</td>
<td>AiiDA options (computational resources)</td>
<td>no</td>
</tr>
</tbody>
</table>

More details:
- *fleur (aiida.orm.Code)*: Fleur code using the *fleur.fleur* plugin
• inpren (aiida.orm.Code): Inpren code using the fleur.inpren 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:

```python
# -*- 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
'references': {}             # Dict to provide the elemental references
```

options

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

```python
'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.

<table>
<thead>
<tr>
<th>name</th>
<th>type</th>
<th>comment</th>
</tr>
</thead>
<tbody>
<tr>
<td>output_initial_cls_wc_para</td>
<td>Dict</td>
<td>Link to last FleurCalculation output dict</td>
</tr>
</tbody>
</table>

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:

5.1. User's guide
```json

# -*- 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)
```
},
    '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,
            -420.4428292264,
            -370.725949483,
            -101.1391871143,
            -92.5627547497,
            -81.8114542005,
            -20.7351164096,
            -67.3928879745,
            -65.0551729884,
            -17.1796863155,
            -14.6884205438,
            -2.7326665018,
            -8.8548575156,
            -8.3959093745,
            -1.0995895461,
            1.073114662
        ]
    },
},
    'reference_corelevel_energies_units': 'htr',
    'reference_fermi_energy': [0.6026436555

5.1. User's guide
```json
],
'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**

```python
from aiida_fleur.tools.plot import plot_fleur
plot_fleur(50816)
```

### Example usage

```python
# -*- 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}}

# 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:

```python
from aida_fleur.workflows.corehole import FleurCoreholeWorkChain
# 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

<table>
<thead>
<tr>
<th>name</th>
<th>type</th>
<th>description</th>
<th>required</th>
</tr>
</thead>
<tbody>
<tr>
<td>inpgen</td>
<td>Code</td>
<td>Inpgen code</td>
<td>yes</td>
</tr>
<tr>
<td>fleur</td>
<td>Code</td>
<td>Fleur code</td>
<td>yes</td>
</tr>
<tr>
<td>wf_parameters</td>
<td>Dict</td>
<td>Settings of the workchain</td>
<td>no</td>
</tr>
<tr>
<td>fleurinp</td>
<td>FleurinpData</td>
<td>FLEUR input</td>
<td>no</td>
</tr>
<tr>
<td>structure</td>
<td>StructureData</td>
<td>Crystal structure</td>
<td>no</td>
</tr>
<tr>
<td>calc_parameters</td>
<td>Dict</td>
<td>FLAPW parameters for given structure</td>
<td>no</td>
</tr>
<tr>
<td>options</td>
<td>Dict</td>
<td>AiiDA options (computational resources)</td>
<td>no</td>
</tr>
</tbody>
</table>

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:

```python
# -*- coding: utf-8 -*-
'method': 'valence',  # what method to use, default for valence to highest open
'shell孔 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]
'para_group': None,  # use parameter nodes from a parameter group
'relax': False,  # relax the unit cell first?
'relax_mode': 'Fleur',  # what relaxation 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...
'magnetic': True  # jspins=2, makes a difference for coreholes
```

**options**

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

```python
'options': {'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:

```python
# -*- 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)': ['5.99999999', '.00000000']}, 'valenceconfig': '(2s1/2) (2p1/2)',
    {'atomic_number': 4, 'coreconfig': '[He]', 'element': 'Be', 'natoms': 1,
    'species': 'Be-2', 'stateOccupation': [(2p1/2): ['0.00000000', '.00000000']]],
)]]
```

(continues on next page)
'valenceconfig': '(2s1/2) (2p1/2)'
},
 {'atomic_number': 4, 'coreconfig': '[:He]', 'element': 'Be', 'natoms': 1,
 'species': 'Be-2', 'stateOccupation': [{(2p1/2)': ['.0000000', '.0000000']},
'valenceconfig': '(2s1/2) (2p1/2)'
},
 {'atomic_number': 4, 'coreconfig': '[:He]', 'element': 'Be', 'natoms': 1,
 'species': 'Be-2', 'stateOccupation': [{(2p1/2)': ['.0000000', '.0000000']},
'valenceconfig': '(2s1/2) (2p1/2)'
}

```
from aiida_fleur.tools.graph_fleur import draw_graph

draw_graph(30528)
```

### Database Node graph
**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**

```python
# -*- 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')
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={
    'atoms': ['Be'],
    'supercell_size': [2, 2, 2],
    'corelevel': ['1s'],
    'hole_charge': 1.0,
    'magnetic': True,
    'method': 'valence'})

# 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

Magnetic workchains:
Magnetic workchains can be divided into two subgroups the Force-theorem subgroup and the self-consistent sub-group.

The Force-theorem subgroup contains:
- ssdisp_wc
- dmi_wc
- mae_wc

The self-consistent sub-group contains:
- ssdisp_conv_wc
Fleur Spin-Spiral Dispersion workchain

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

Contents

- **Fleur Spin-Spiral Dispersion workchain**
  - Description/Purpose
  - Input nodes
  - Output nodes
  - Supported input configurations
  - Error handling
  - Example usage

Import Example:

```python
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 is to calculate the energy difference between two or several structures having a different magnetisation profile:

![Diagram of magnetisation profiles](image)

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.

<table>
<thead>
<tr>
<th>name</th>
<th>type</th>
<th>description</th>
<th>required</th>
</tr>
</thead>
<tbody>
<tr>
<td>scf</td>
<td>namespace</td>
<td>inputs for nested SCF WorkChain</td>
<td>no</td>
</tr>
<tr>
<td>fleur</td>
<td>Code</td>
<td>Fleur code</td>
<td>yes</td>
</tr>
<tr>
<td>wf_parameters</td>
<td>Dict</td>
<td>Settings of the workchain</td>
<td>no</td>
</tr>
<tr>
<td>fleurinp</td>
<td>FleurinpData</td>
<td>FLEUR input</td>
<td>no</td>
</tr>
<tr>
<td>remote</td>
<td>RemoteData</td>
<td>Remote folder of another calculation</td>
<td>no</td>
</tr>
<tr>
<td>options</td>
<td>Dict</td>
<td>AiiDA options (computational resources)</td>
<td>no</td>
</tr>
</tbody>
</table>

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:

```plaintext
# -*- 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],
               [0.125, 0.0, 0.0],
               [0.250, 0.0, 0.0],
               [0.375, 0.0, 0.0]],  # set a set of q-vectors to calculate SSDispersion
```

(continues on next page)
'ref_qss': [0.0, 0.0, 0.0],  # sets a q-vector for the reference calculation
'inpxml_changes': [],  # additional changes before the FT step
'add_comp_para': {
    'only_even_MPI': False,  # True if suppress parallelisation having odd number of MPI
    'max_queue_nodes': 20,  # Max number of nodes allowed (used by automatic error fix)
    'max_queue_wallclock_sec': 86400  # Max number of walltime allowed (used by automatic error fix)
},

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

\begin{verbatim}
'beta' : {'222' : 1.57079}
\end{verbatim}

to:

\begin{verbatim}
<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>
\end{verbatim}

\textbf{Note:} \texttt{beta} actually sets a beta parameter for a whole atomGroup. It can be that the atomGroup correspond to several atoms and \texttt{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.

\texttt{prop_dir} is used only to set up a spin spiral propagation direction to \texttt{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.

\textbf{options}

\textbf{options: Dict - AiiDA options (computational resources). Example:}

\begin{verbatim}
'options': {'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': {}}
\end{verbatim}
Output nodes

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

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

Resulting Spin Spiral energies are sorted according to their 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*:

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

**Example usage**

```python
# -*- 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': [],
        'add_comp_para': {
            'only_even_MPI': False,
            'max_queue_nodes': 20,
            'max_queue_wallclock_sec': 86400
        }
    }
)
```
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, '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,
    }
}
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

• Fleur Dzyaloshinskii–Moriya Interaction energy workchain
  – Description/Purpose
  – Input nodes
  – Output nodes
  – Supported input configurations
  – Error handling
  – Example usage

Import Example:

```python
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:

1. **Initialisation**
   - Initial structure + WorkChain parameters

2. **Reference SCF calculation**
   - ~50 iterations

3. **Force theorem**
   - 1 iteration for each (theta, phi)

4. **Output**
   - \( E(q) \)

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`.

<table>
<thead>
<tr>
<th>name</th>
<th>type</th>
<th>description</th>
<th>required</th>
</tr>
</thead>
<tbody>
<tr>
<td>scf</td>
<td>namespace</td>
<td>inputs for nested SCF WorkChain</td>
<td>no</td>
</tr>
<tr>
<td>fleur</td>
<td>Code</td>
<td>Fleur code</td>
<td>yes</td>
</tr>
<tr>
<td>wf_parameters</td>
<td>Dict</td>
<td>Settings of the workchain</td>
<td>no</td>
</tr>
<tr>
<td>fleurinp</td>
<td>FleurinpData</td>
<td>FLEUR input</td>
<td>no</td>
</tr>
<tr>
<td>remote</td>
<td>RemoteData</td>
<td>Remote folder of another calculation</td>
<td>no</td>
</tr>
<tr>
<td>options</td>
<td>Dict</td>
<td>AiiDA options (computational resources)</td>
<td>no</td>
</tr>
</tbody>
</table>

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**

The `wf_parameters` dictionary contains settings of the workflow behavior. All possible keys and their defaults are listed below:

```python
# -*- 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
'pic_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
              [0.1, 0.1, 0.0]],
'add_comp_para': {  # True if suppress parallelisation having odd
  'only_even_MPI': False,  # number of MPI
  'max_queue_nodes': 20,  # Max number of nodes allowed (used by automatic
  'max_queue_wallclock_sec': 86400  # Max number of walltime allowed (used by
# automatic error fix)
},
'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 the `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}
```

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 species 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:

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

changes

```xml
<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" l nonsphr="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:

```xml
<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" l nonsphr="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*:

<table>
<thead>
<tr>
<th>Code</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>230</td>
<td>Invalid workchain parameters</td>
</tr>
<tr>
<td>231</td>
<td>Invalid input configuration</td>
</tr>
<tr>
<td>233</td>
<td>Input codes do not correspond to fleur or inpgen codes respectively.</td>
</tr>
<tr>
<td>235</td>
<td>Input file modification failed.</td>
</tr>
<tr>
<td>236</td>
<td>Input file was corrupted after modifications</td>
</tr>
<tr>
<td>334</td>
<td>Reference calculation failed.</td>
</tr>
<tr>
<td>335</td>
<td>Found no reference calculation remote repository.</td>
</tr>
<tr>
<td>336</td>
<td>Force theorem calculation failed.</td>
</tr>
</tbody>
</table>
Example usage

```python
# -*- 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={'add_comp_para': {
    'only_even_MPI': False,
    'max_queue_nodes': 20,
    'max_queue_wallclock_sec': 86400
},
    '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},
                        '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,
               'mode': 'density'}
```

(continues on next page)
AiiDa-FLEUR Documentation, Release 2.0.0

(continued from previous page)

```python
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

- **Fleur Magnetic Anisotropy Energy workflow**
  - Description/Purpose
  - Input nodes
  - Output nodes
  - Supported input configurations
  - Error handling
  - Example usage

Import Example:
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 is 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:

1. Initialisation

   ![Initialisation Diagram](image)

2. Reference SCF

   ![Reference SCF Diagram](image)

3. Force theorem

   ![Force theorem Diagram](image)

4. Output

   ![Output Diagram](image)

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.

<table>
<thead>
<tr>
<th>name</th>
<th>type</th>
<th>description</th>
<th>required</th>
</tr>
</thead>
<tbody>
<tr>
<td>scf</td>
<td>namespace</td>
<td>inputs for nested SCF WorkChain</td>
<td>no</td>
</tr>
<tr>
<td>fleur</td>
<td>Code</td>
<td>Fleur code</td>
<td>yes</td>
</tr>
<tr>
<td>wf_parameters</td>
<td>Dict</td>
<td>Settings of the workchain</td>
<td>no</td>
</tr>
<tr>
<td>fleurinp</td>
<td>FleurinpData</td>
<td>FLEUR input</td>
<td>no</td>
</tr>
<tr>
<td>remote</td>
<td>RemoteData</td>
<td>Remote folder of another calculation</td>
<td>no</td>
</tr>
<tr>
<td>options</td>
<td>Dict</td>
<td>AiiDA options (computational resources)</td>
<td>no</td>
</tr>
</tbody>
</table>

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** is 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
'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
'add_comp_para': {
    'only_even_MPI': False, # True if suppress parallelisation having odd number of MPI
    'max_queue_nodes': 20, # Max number of nodes allowed (used by automatic error fix)
    'max_queue_wallclock_sec': 86400, # Max number of walltime allowed (used by automatic error fix)
},
'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
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:

```python
'options': {'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

```json
"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. $mae[N]$ 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*:

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

**Example usage**

```python
# -*- 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],
                     'add_comp_para': {
                         'only_even_MPI': False,
                         'max_queue_nodes': 20,
                         'max_queue_wallclock_sec': 86400,
                     },
                     'soc_off': [],
                     'inpxml_changes': []},
                 }

options = Dict(dict={'resources': {'num_machines': 1, 'num_mpiprocs_per_machine': 24},
                      'queue_name': 'devel',
                      'custom_scheduler_commands': ''},
```
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, 'mode': 'density'}

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)
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**

- Fleur Spin-Spiral Dispersion Converge workchain
  - Description/Purpose
  - Input nodes
  - Output nodes
  - Layout
  - Error handling
  - Example usage

**Import Example:**

```python
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.

<table>
<thead>
<tr>
<th>name</th>
<th>type</th>
<th>description</th>
<th>required</th>
</tr>
</thead>
<tbody>
<tr>
<td>scf</td>
<td>namespace</td>
<td>inputs for nested SCF WorkChain</td>
<td>yes</td>
</tr>
<tr>
<td>wf_parameters</td>
<td>Dict</td>
<td>Settings of the workchain</td>
<td>no</td>
</tr>
</tbody>
</table>

Workchain parameters and its defaults

**wf_parameters**

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

```python
# -*- coding: utf-8 -*-
'
'beta': {'all': 1.57079},
'q_vectors': {'label': [0.0, 0.0, 0.0],
              'label2': [0.125, 0.0, 0.0]}
'suppress_symmetries': False
```

**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

```xml
<atomGroup species="Fe-1">
  <filmPos label="222">0.000000000 0.000000000 11.4075100502</filmPos>
  <force calculate="T" relaxXYZ="TTT"/>
  <nocoParams l_relax="F" alpha=".0000000" beta="0.00000" b_cons_x=".00000000" b_cons_y="" b_cons_z=""/>
</atomGroup>
```
to:

```xml
<atomGroup species="Fe-1">
  <filmPos label="222">.00000000 0.00000000 -11.4075100502</filmPos>
  <force calculate="T" relaxXYZ="TTT"/>
  <nocoParams l_relax="F" alpha="0.00000000" beta="1.57079" b_cons_x="0.00000000" 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

```python
{
  "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:

<table>
<thead>
<tr>
<th>Code</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>230</td>
<td>Invalid workchain parameters</td>
</tr>
<tr>
<td>340</td>
<td>Convergence SSDisp calculation failed for all q-vectors</td>
</tr>
<tr>
<td>341</td>
<td>Convergence SSDisp calculation failed for some q-vectors</td>
</tr>
</tbody>
</table>

Example usage

```python
# -*- 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},
                         }
```

(continues on next page)
```python
wf_para_scf = {'fleur_runmax': 2,
               'itmax_per_run': 120,
               'density_converged': 0.2,
               '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**

- Fleur Magnetic Anisotropy Energy Converge workchain
  - Description/Purpose
  - Input nodes
    - Workchain parameters and its defaults
      - wf_parameters
  - Output nodes
import Example:

```python
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:

1. **Initialisation**
   - Initial structure
   - WorkChain parameters

2. **SCFs**
   - SCF
   - ~50 iterations with SOC

3. **Output**
   - $E(\theta, \phi)$
### Input nodes

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

<table>
<thead>
<tr>
<th>name</th>
<th>type</th>
<th>description</th>
<th>required</th>
</tr>
</thead>
<tbody>
<tr>
<td>scf</td>
<td>namespace</td>
<td>inputs for nested SCF WorkChain</td>
<td>yes</td>
</tr>
<tr>
<td>wf_parameters</td>
<td>Dict</td>
<td>Settings of the workchain</td>
<td>no</td>
</tr>
</tbody>
</table>

### Workchain parameters and its defaults

#### wf_parameters

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

```python
# -*- 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:

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

changes

```xml
<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="SCL0" l="1" n="5" eDeriv="0"/>
</species>

<atomGroup species="Ir-2">
<filmPos label="458">1.000/4.000 1.000/2.000 11.407400502</filmPos>
<force calculate="T" relaxXYZ="TTT"/>
<nocoParams l_relax="F" alpha=".00000000" beta=".00000000" b_cons_x=".00000000" b_cons_y=".00000000"/>
</atomGroup>
```

to:

---
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.

`sqa` 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

```python
{
    "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*:

<table>
<thead>
<tr>
<th>Code</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>230</td>
<td>Invalid workchain parameters</td>
</tr>
<tr>
<td>342</td>
<td>Convergence MAE calculation failed for all SQAs</td>
</tr>
<tr>
<td>343</td>
<td>Convergence MAE calculation failed for all SQAs</td>
</tr>
</tbody>
</table>

Example usage

```python
# -*- 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}})
```

(continues on next page)
wf_para_scf = {'fleur_runmax': 2,
               'itmax_per_run': 120,
               'density_converged': 0.2,
               '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(FleurMaeConvWorkChain, **inputs)

And other workflows like the create_magnetic_wc.

**Fleur Create Magnetic Film workchain**

- **Current version:** 0.2.0
- **Class:** FleurCreateMagneticWorkChain
- **String to pass to the WorkflowFactory():** fleur.create_magnetic
- **Workflow type:** Scientific workchain

**Contents**

- **Fleur Create Magnetic Film workchain**
  - Description/Purpose
  - Input nodes
  - Output nodes
  - Supported input configurations
  - Error handling
  - Structures with known AFM structures
Example usage

Import Example:

```python
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 following magnetic workchains: DMI, MAE or SSDisp.

The inputs include information about the substrate (structure type, miller indices of surfaces or vectors forming the primitive unit cell, chemical elements) 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 lattices are supported. Note, the algorithm always uses conventional unit cell e.g. one gets 4 atoms in the unit cell for fcc lattice (see the figure above).

After the EOS step 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.

Finally, using the result of the relaxation workchain, a magnetic structure having no z-reflection symmetry is constructed. For this the workchain takes first `num_relaxed_layers` layers from the relaxed structure and attaches so many substrate layers so there are `total_number_layers` layers. 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.

<table>
<thead>
<tr>
<th>name</th>
<th>type</th>
<th>description</th>
<th>required</th>
</tr>
</thead>
<tbody>
<tr>
<td>eos</td>
<td>namespace</td>
<td>inputs for nested EOS WC. structure input is excluded.</td>
<td>no</td>
</tr>
<tr>
<td>relax</td>
<td>namespace</td>
<td>inputs for nested Relax WC. structure input of SCF sub-namespace is excluded</td>
<td>no</td>
</tr>
<tr>
<td>wf_parameters</td>
<td>Dict</td>
<td>Settings of the workchain</td>
<td>no</td>
</tr>
<tr>
<td>eos_output</td>
<td>Dict</td>
<td>EOS output dictionary</td>
<td>no</td>
</tr>
<tr>
<td>optimized_structure</td>
<td>StructureData</td>
<td>relaxed film structure</td>
<td>no</td>
</tr>
<tr>
<td>distance_suggestion</td>
<td>Dict</td>
<td>interatomic distance suggestion, output of the request_average_bond_length_store()</td>
<td>no</td>
</tr>
</tbody>
</table>

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:

```python
# -*- coding: utf-8 -*-
'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': {1: 'Fe', -1: 'Fe'},  # sets the layer number to be replaced by another element
'
----
'decimals': 10,  # set the accuracy of writing atom positions
'z_coordinate_window': 2,  # set how z-coordinates will be rounded before grouping in layers
'pop_last_layers': 1,  # number of bottom layers to be removed before relaxation
'hold_layers': None,  # a list of layer numbers to be held during the relaxation
'AFM_name': 'FM',  # a name of hardcoded AFM structure: FM, AFM_x, AFM_y, AFM_xy
'magnetic_layers': 1,  # the total number of magnetic layers (for symmetric films divide by 2)
'magnetic_layers_positions': None,  # a suggestion of initial layer z coordinates, used if AFM structures are known
```

(continues on next page)
Some of the parameters, which can be set in the workchain parameter dictionary, control how the structure will be created for the relaxation step. The following procedure is used to construct a film for relaxation:

1. Create a slab using ASE methods. For this following parameters are used: lattice, miller or directions, host_symbol, size, and latticeconstant (or lattice constant from distance_suggestion input node). For more details refer to ase documentation.

2. Remove pop_last_layers last layers. This step can help one to ensure symmetrical film.

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 will be 8 layers in the slab since there are 2 layers in the unit cell. That means the x, y positions of the atom in the first layer are equal to (0.0, 0.0) and the 8th layer 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.

3. Replace atom layers according to replacements dictionary. The dictionary should consist of INT: STRING pairs, where INT defines the layer number to be replaced (counting from the lowest layers, INT=1 for the first layer and INT=-1 for the last) and STRING defines the element name.

4. Adjust interlayer distances using distance_suggestion, first_layer_factor and last_layer_factor. If the input structure has z-reflection symmetry, then first_layer_factor is ignored and the last_layer_factor controls both surface layers. If AFM_layer_positions is given and AFM structures are known for the input lattice and directions, then the adjusting procedure will not do it automatically, but simply enforce z-coordinates from AFM_layer_positions. Read more in the section Structures with known AFM structures.

Warning: Adjusting of interlayer distances for non-symmetric films work well only if substrate is positioned above magnetic elements (z-coordinate of substrate atoms are higher than magnetic ones). This can be achieved by using replacements: {1: 'Fe'} instead of replacements: {-1: 'Fe'}.

1. Mark fixed layers according to hold_layers. hold_layers is a list of layer number to be marked as fixed during the relaxation step. Similarly to replacements, the 1st layer corresponds to number 1 and the last to -1.

After the structure is relaxed, the final magnetic non-symmetrical structure is constructed. For this total_number_layers and num_relaxed_layers setting the total number of layers of the number of layers extracted from the relaxed structure respectively.
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*:

<table>
<thead>
<tr>
<th>Code</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>230</td>
<td>Invalid workchain parameters</td>
</tr>
<tr>
<td>231</td>
<td>Invalid input configuration</td>
</tr>
<tr>
<td>380</td>
<td>Specified substrate is not bcc or fcc, only them are supported</td>
</tr>
<tr>
<td>382</td>
<td>Relaxation calculation failed.</td>
</tr>
<tr>
<td>383</td>
<td>EOS WorkChain failed.</td>
</tr>
</tbody>
</table>

5.1. User’s guide
Structures with known AFM structures

**Warning:** The workchain uses `define_AFM_structures` method to mark spin-up and spin-down atoms. It does not actually set initial moment, a user is responsible for adding `inxml_changes` changing the initial spin. For instance, for MaX4 FLEUR one can use following `inxml_changes` to set initial moments for AFM structure:

```
'inxml_changes': [
    ('set_species_label', {'at_label': '49990',
                         'attributedict': {'magMom': 4.0},
                         'create': True}),
    ('set_species_label', {'at_label': '49991',
                         'attributedict': {'magMom': -4.0},
                         'create': True})]
```

Spin-up atoms are marked by label '49990' and spin-down atoms are marked by '49991'. Please make sure that these labels are not overwritten, for example by fixing atoms label.

1. FCC(110) surfaces, 1 or 2 magnetic layers.

   In this case one must use following parameters in the `wf_parameters`:

   ```
   'lattice': 'fcc'
   'directions': [[-1, 1, 0], [0, 0, 1], [1, 1, 0]]
   'magnetic_layers': 1 or 2
   'AFM_name': 'FM', 'AFM_x', 'AFM_y' or 'AFM_xy'
   ```

   Note that for 'AFM_xy' structure unit vectors forming a computational cell are changed to 
   
   ```
   [[-1, 1, 2], [1, -1, 2], [1, 1, 0]].
   ```

1. BCC(110) surfaces, 1 or 2 magnetic layers.

   In this case one must use following parameters in the `wf_parameters`:

   ```
   'lattice': 'bcc'
   'directions': [[1, -1, 1], [1, -1, -1], [1, 1, 0]]
   'magnetic_layers': 1 or 2
   'AFM_name': 'FM', 'AFM_x', 'AFM_y' or 'AFM_xy'
   ```

   Note that for 'AFM_x' and 'AFM_y' structure unit vectors forming a computational cell are changed to 
   
   ```
   [[0, 0, 1], [1, -1, 0], [1, 1, 0]].
   ```

**Warning:** The check of input lattice vector directions is hardcoded, which means that the code will not recognize 'bcc' and '[[1, -1, -1], [-1, 1, 1], [1, 1, 0]]' despite it produces the same structure. In this case the workchain will be excepted.

There is a possibility to enforce layer z-coordinates for generating AFM structures. This allows one to reuse the relaxed FM structured for following AFM calculation to save computational resources because it is expected that the FM relaxed structure is a better initial guess for an AFM one rather than a structure, proposed by automatic adjusting function. To do this, one should make sure that the length of `AFM_layer_positions` is the same as the total number of layers of the AFM structure and `magnetic_layers` is correctly initialised.

Figures below illustrate known AFM structures for FCC(110) and BCC(110) structures. The number on each atoms shows to which layer atom belongs (first or second) and the color corresponds to spin orientation (up or down). Note that the input computational unit cell will be changed (shown on the figures), hence you might want to adjust the k-mesh correspondingly.
Example usage

```python
# -*- 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',
    'directions': [[-1, 1, 0],
                   [0, 0, 1],
                   [1, 1, 0]],
    'host_symbol': 'Pt',
    'latticeconstant': 4.0,
}
```

(continues on next page)
following values:

```python
wf_para = Dict(dict=wf_para)
wf_eos = Dict(dict=wf_eos)
wf_eos_scf = Dict(dict=wf_eos_scf)
wf_eos = Dict(dict=wf_eos)
wf_relax = Dict(dict=wf_relax)
wf_relax_scf = Dict(dict=wf_relax_scf)
```

...
'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': {
            'wf_parameters': wf_eos_scf,
            'calc_parameters': calc_eos,
            'options': options_eos,
            'inpgen': inpgen_code,
            'fleur': fleur_code
        },
        'wf_parameters': wf_eos
    },
    'relax': {

Fleur crystal field workflow

- Current version: 0.2.0
- Class: FleurCFCoeffWorkChain
- String to pass to the WorkflowFactory(): fleur.cfcoeff
- Workflow type: Scientific workchain
- Aim: Calculate 4f Crystal field coefficients

Contents

- Fleur crystal field workflow
  - Description/Purpose
  - Input nodes
    * Workchain parameters and its defaults
  - Returns nodes
  - Layout
  - Error handling
  - Plot_fleur visualization
  - Database Node graph
  - Example usage

Import Example:

```python
from aiida_fleur.workflows.cfcoeff import FleurCFCoeffWorkChain

# or
WorkflowFactory('fleur.cfcoeff')
```
Description/Purpose

Calculates the 4f crystal field coefficients for a given structure using the method described in C.E. Patrick, J.B. Staunton: J. Phys.: Condens. Matter 31, 305901 (2019).

This method boils down to the formula

\[ B_{lm} = \sqrt{\frac{2l + 1}{4\pi}} \int_{R_{MT}} dr r^2 V_{lm}(r)n_{4f}(r) \]

where \( V_{lm}(r) \) is the potential of the surroundings of the 4f site and \( n_{4f}(r) \) is the spherical charge density of the 4f state. The potential is calculated using one of two options:

1. Calculate the potential of an analogue structure, where the 4f atom is replaced by a yttrium atom.
2. Calculate the potential from the system including the 4f atom directly.

This is done by first calculating the converged charge density for the 4f structure and if used the analogue structure with the FleurScfWorkChain. Then a subsequent calculation is done to extract the potentials/charge density. The calculation of the formula above is done after with the CFCalculation tool in masci-tools.

Input nodes

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

<table>
<thead>
<tr>
<th>name</th>
<th>type</th>
<th>description</th>
<th>required</th>
</tr>
</thead>
<tbody>
<tr>
<td>scf</td>
<td>namespace</td>
<td>Inputs for the SCF workchain including the 4f atom</td>
<td>no</td>
</tr>
<tr>
<td>orbcontrol</td>
<td>namespace</td>
<td>Inputs for the Orbcontrol workchain including the 4f atom</td>
<td>no</td>
</tr>
<tr>
<td>scf_rare_earth_analogue</td>
<td>namespace</td>
<td>Inputs for the SCF workchain with the 4f atom replaced with the analogue</td>
<td>no</td>
</tr>
<tr>
<td>wf_parameters</td>
<td>Dict</td>
<td>Settings of the workchain</td>
<td>no</td>
</tr>
</tbody>
</table>

One of the scf or orbcontrol input nodes is required.

Workchain parameters and its defaults

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

```python
# -*- coding: utf-8 -*-
'
# determines for which element to calculate
# the crystal field coefficients
'element': '',

'rare_earth_analogue': True,  # True if analogue calculation should be used
'analogue_element': 'Y',       # Which element to use for the analogue
'structure'            
'replace_all': True,          # Whether to replace all atoms for the analogue
'analogue in one structure
'soc_off': True,             # if True the socscale is set to 0.0 for the 4f site
'the 4f site'
```

(continues on next page)
Returns nodes

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

<table>
<thead>
<tr>
<th>name</th>
<th>type</th>
<th>comment</th>
</tr>
</thead>
<tbody>
<tr>
<td>output_cfcoeff_wc_para</td>
<td>Dict</td>
<td>results of the workchain</td>
</tr>
<tr>
<td>output_cfcoeff_wc_potentials</td>
<td>XyData</td>
<td>XyData with the calculated potentials</td>
</tr>
<tr>
<td>output_cfcoeff_wc_charge_densities</td>
<td>XyData</td>
<td>XyData with the calculated charge densities</td>
</tr>
</tbody>
</table>

Layout

TODO

Error handling

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

<table>
<thead>
<tr>
<th>Exit code</th>
<th>Reason</th>
</tr>
</thead>
<tbody>
<tr>
<td>230</td>
<td>Invalid workchain parameters</td>
</tr>
<tr>
<td>231</td>
<td>Invalid input configuration</td>
</tr>
<tr>
<td>235</td>
<td>Input file modification failed.</td>
</tr>
<tr>
<td>236</td>
<td>Input file was corrupted after modifications</td>
</tr>
<tr>
<td>345</td>
<td>SCF workchain failed</td>
</tr>
<tr>
<td>451</td>
<td>Orbcontrol workchain failed</td>
</tr>
<tr>
<td>452</td>
<td>FleurBaseWorkChain for CF calculation failed</td>
</tr>
</tbody>
</table>

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

TODO
5.1.5 The command line interface (CLI)

Or how-to manually work or script from the terminal.

Besides the python API, aïda-fleur comes with a builtin command line interface (CLI) `aïda-fleur`, which exposes functionalities of aïda-fleur on the command line, similar to the `verdi` commands of `aïda-core`. This interface is built using the `click` library and supports tab-completion. Of course everything you can do through the CLI and much more you can also do through the python API.

Here you will learn how to use this CLI. Everything in a code block with a “$” in front can be executed in a shell, if not otherwise indicated. Expected output is displayed below the command. If a code block if a “$” contains “<>” it means that you have to replace it with what stands inside. For example `<scf-wc_pk>` means you have to type in the “pk/id” of the SCF workflow which was run.

5.1.5.1 General information

To enable tab-completion, add the following to your shell loading script, e.g. the `.bashrc` or virtual environment activate script, or execute:

```bash
eval "$(source _AIIDA_FLEUR_COMPLETE=source aïda-fleur)"
```

In general, to learn about a command you can execute every command with the `-h/++help` option to see its help string. This will show you what the command does and what arguments, options and defaults it has. If it is a command group it will show you all sub-commands.

Example command group:

```bash
$ aïda-fleur -h
Usage: aïda-fleur [OPTIONS] COMMAND [ARGS]...

  CLI for the `aïda-fleur` plugin.

Options:
  -p, ++profile PROFILE    Execute the command for this profile instead of the default profile.
  -h, ++help               Show this message and exit.

Commands:
  data    Commands to create and inspect data nodes.
  launch  Commands to launch workflows and calcjobs of aïda-fleur.
  plot    Invoke the plot_fleur command on given nodes.
  workflow Commands to inspect aïda-fleur workchains.
```

Example for a command:
$ aiida-fleur launch scf -h
Usage: aiida-fleur launch scf [OPTIONS]

Launch a scf workchain

Options:
-s, ++structure STRUCTUREFILE StructureData node, given by pk or uuid or
file in any for mat which will be converted.
[default: (dynamic)]
-i, ++inpgen CODE A code node or label for an inpgen
executable. [default: (dynamic)]
-calc_p, ++calc-parameters DATA Dict with calculation (FLAPW) parameters to
build, which will be given to inpgen.
-set, ++settings DATA Settings node for the calcjob.
-inp, ++fleurinp DATA FleurinpData node for the fleur calculation.
-f, ++fleur CODE A code node or label for a fleur executable.
[default: (dynamic)]
-wf, ++wf-parameters DATA Dict containing parameters given to the
workchain.
-P, ++parent-folder DATA The PK of a parent remote folder (for
restarts).
-d, ++daemon Submit the process to the daemon instead of
running it locally. [default: False]
-set, ++settings DATA Settings node for the calcjob.
-opt, ++option-node DATA Dict, an option node for the workchain.
-h, ++help Show this message and exit.

For the full automatic documentation of all commands checkout the Commandline Interface (CLI) section in the module
guide.

5.1.5.2 Overview of the main commands

The main commands groups of aiida-fleur are data, launch, plot and workflow.

The data group contains commands to create and inspect data nodes, for utility which is more specific to aiida-fleur and not covered by the `verdi data` commands of aiida-core. Sub-commands of aiida-fleur data include:

```
fleurinp   Commands to handle `FleurinpData` nodes.
parameter  Commands to create and inspect `Dict` nodes containing FLAPW parameters
structure  Commands to create and inspect `StructureData` nodes.
```

The launch group contains commands to launch workflows/workchains and calcjobs of aiida-fleur from the shell. Sub-commands of aiida-fleur launch include:
important options out most launch commands include: the -s option to provide a crystal structure. this can be either a pk or uuid from a structuredata node in the database or any file on disk in a format ase can read a structure from. this includes:

the plot command invokes the plot_fleur command of aiida-fleur on given nodes. the plot_fleur command can visualize the output of a lot of aiida-fleur workchains.

the workflow command group has sub commands to inspect aiida-fleur workchains and prepare inputs.

for example to launch an scf workchain on a given structure execute:

\[
\text{
$ aiida-fleur launch scf -i <inpgenpk> -f <fleurpk> -s <structurepk> $
}
\]

the command can also process structures in any format ase can handle, this includes Cif, xsf and poscar files. In such a case simply parse the path to the file:

\[
\text{
$ aiida-fleur launch scf -i <inpgenpk> -f <fleurpk> -s ./structure/Cu.cif $
}
\]

5.1.5.3 Confirm proper setup

Quickly confirm that you have a computer and a code setup within your database.

\[
\text{
$ verdi computer list -a
$ verdi code list -a $
}
\]

should display some configured computer and codes like this (notice the “*”s):

\[
\text{
Info: List of configured computers  
Info: Use 'verdi computer show COMPUTERNAME' to display more detailed information  
* localhost  
* iffslurm
}
\]

\[
\text{
# (use 'verdi code show CODEID' to see the details)  
# List of configured codes:  
* pk 149 - fleur_MPI_MaXR5_AMD@iffslurm  
* pk 150 - inpgen_MaXR5_AMD@iffslurm  
* pk 151 - inpgen_MaXR5_th1@iffslurm  
* pk 148 - fleur_MPI_MaXR5_th1@iffslurm
}
\]

5.1. User’s guide
5.1.5.4 Prepare options nodes

Usually, when submitting calculations or workchains to a computer you have to provide an options node in which you specify the queue to submit to and what computational resources the scheduler should allocate. If the default option node is enough, or if the options for the default queue stored in the ‘extras’ of a code node, you do not need to provide this node.

To submit simulations to the th1 queue with one node and run with two mpi processes execute:

```bash
aiida-fleur data options create -q 'th1' -N 1 -M 2
```

To submit simulations to the th1-2020-32 queue with one node and run with two mpi processes execute:

```bash
aiida-fleur data options create -q 'th1-2020-32' -N 1 -M 2
```

You should see some output this:

```
Success: Created and stored Options node <290> <99f79d2e-04aa-4aaf-9b5f-9eabad8142d8>
{
"max_wallclock_seconds": 1800,
"queue_name": "th1-2020-32",
"resources": {
  "num_machines": 1,
  "num_mpiprocs_per_machine": 2
}
}
```

Remember these pks (further named opt_th1_pk and opt_amd_pk) we need them further for launching workchains. To display the contents of any aiida.orm.Dict node you can execute verdi data dict show <pk>.

5.1.5.5 Launching Calculations and workchains

5.1.5.6 Executing inpgen

First we run a simple inpgen calculation from the command line on a Si structure provided by some cif file.

```bash
$ aiida-fleur launch inpgen -i inpgen_MaXR5_th1 -s Si.cif -q th1
```

The structure is provided via the -s option, which can either be an identifier of a StructureData node or any supported format by ase.io (see https://wiki.fysik.dtu.dk/ase/ase/io/io.html?highlight=formats) Among many others this includes: `cif, poscar, xsf, xyz, concar, outcar, xtd, xsd` One should be cautious when dealing with film and magnetic structures, because one has to make sure that the setup is as fleur needs it, and that all the magnetic information is preserved. One could use this command to convert most formats to fleur input, or with ++dry-run one can get an input file for the input generator without storing anything in the database. Also the execution above will block the interpreter until the job is finished and you see the logged output. If the job is finished look at output of the process with

```bash
$ verdi calcjob show <inpgen_calc_pk>
$ verdi process report <inpgen_calc_pk>
$ verdi outputls <inpgen_calc_pk>
```

will show you all files retrieved and stored in the aiida_repository by aiida.
$ verdi calcjob gotocomputer <inpgen_calc_pk>

you can go to the remote computer to the directory where the job was executed (execute there exit or logout to logout from the remote computer.). To see print the inp.xml file or any other retrieved output file execute

$ verdi cajcjob outputcat <inpgen_calc_pk>

to see the input file for the inpgen calculation execute:

$ verdi calcjob inputcat <inpgen_calc_pk>

5.1.5.7 Executing Fleur

Launch fleur calculation works in the same way, per default the base_fleur workchain is launched, which has some basic error handlers for fleur calculations. On the resulting FleurinpData from the inpgen calculation above we now launch a fleur calculation.

$ aiida-fleur launch fleur ++fleur fleur_MaXR5_th1 -inp <fleurinp_pk>

5.1.5.8 Executing higher workflows

The interface to launch other workflows is very similar to the interface and options of the base calculations. This time for each command we execute we add the -d option to submit the workflow to the daemon, executing them in the background instead of blocking the interpreter. You can launch directly workflows like this

$ aiida-fleur launch scf -d -s Si.cif -i inpgen_MaXR5_th1 ++fleur fleur_MaXR5_th1 -opt -<opt_th1_pk>
$ aiida-fleur launch relax -d -s Si.cif -i inpgen_MaXR5_th1 ++fleur fleur_MaXR5_th1 -opt -<opt_th1_pk>

launch an equation of states in the background to a different queue as for the other workflows

$ aiida-fleur launch eos -s Si.cif -i inpgen_MaXR5_th1 ++fleur fleur_MaXR5_AMD -opt <opt_amd_pk>

Check with

verdi process list -p1

what the status of the workflows is while they execute. When they are finished we can visualize the results using the aiida-fleur plot command, which visualizes workchain results statically with matplotlib or interactive with bokeh.

$ aiida-fleur plot <scf-wc_pk>
$ aiida-fleur plot <eos_wc_pk>

To easily display inputs and result dictionaries of aiida-fleur workchains you can utilize the workflow sub-commands.

$ aiida-fleur workflow inputcat <scf_wc_pk>
$ aiida-fleur workflow res ++info <scf_wc_pk>

Congratulations, you finished the aiida-fleur command line tutorial! Thanks you! If you have any feedback, suggestions, feature requests, contact a developer or write an issue in the aiida-fleur git repository: https://github.com/JuDFTteam/aiida-fleur.
# Further comments, where to go from here:

## 5.1.5.9 (DFT) code inter operability

You can now run a kkr scf with this relaxed structure as inputs over the similar aiida-kkr CLI. For example: For this first look at the output from the fleur relax workflow above and identify the pk of the optimized output structure

```bash
$ verdi node show <relax_wc_pk>
```

```bash
$ aiida-kkr launch scf -S <optimized_structure_pk> ++kkr <kkr_code>++voro <voronoi_code>
```

For more on this checkout the aiida-kkr tutorials.

## 5.1.5.10 Common workflows

There is also work going on for common workflow interfaces between DFT codes. For this checkout the aiida-common-workflow repository (https://github.com/aiidateam/aiida-common-workflows). This is per default installed with all codes on quantum mobile, not here on iffaiida. These common workflows use protocols ('moderate', 'fast', 'precise'), which are codes specific, but which allow to execute the same type of workflow on otherwise the same input for example to following lines would execute an equation of states workflow with different codes on quantum mobile (otherwise needs more inputs):

```bash
aiida-common-workflows launch eos -S Fe -p moderate fleur
aiida-common-workflows launch eos -S Fe -p fast quantum_espresso
aiida-common-workflows launch eos -S Fe -p precise siesta
aiida-common-workflows launch eos -S Fe cp2k
```

Other useful commandline interfaces:

- ASE: (https://wiki.fysik.dtu.dk/ase/cmdline.html)

## 5.1.5.11 Commandline versus python work

Work on the commandline is rather interactive, if you do not write a bash script to execute the commands you may lose information on the execution and maybe how to find things, if you have not logged something. The same if true for working with ipython. For testing and small projects the command line interface is really useful and fast. For large projects we still suggest strongly to use the python interface, because there you have the full functionality of aiida-fleur making it easier to execute a sequence of workflows which depend on each other.

## 5.1.6 Tools

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

In general if you are looking for something which does not depend on AiiDA and may not be FLEUR specific, the place to look for is the masci-tools repository and its documentation. This includes:

- parsers for files
- utility for xml
- plot methods

Tools and utility which does depend on AiiDA but is not FLEUR specific you find in the aiida-jutools repository. This includes:
• StructureData curation
• Meta AiiDA database analysis

Since so far we lack a in detail documentation of tools and utility within aiida-fleur you are referenced to the automatic documentation of tools and utility.

5.1.7 Tutorials

Here we link you to some tutorial resources for aiida-fleur and related topics. In general you find hands-on tutorial material under https://github.com/JuDFTteam/judft_tutorials and in the examples folder of the aiida-fleur package (the examples lack sometimes behind).

5.1.7.1 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.

An introduction video into AiiDA you find here or other videos under the Materials cloud channel.

5.1.7.2 AiiDA-FLEUR tutorials

Lectures: - Introduction into AiiDA-fleur - Introduction into AiiDA-fleur workflows

5.1.7.3 FLEUR & FLAPW tutorials

In general for new, documentation and tutorials for the FLEUR program checkout www.flapw.de.

Videos and pdfs from a tutorial in 2021: https://www.flapw.de/MaX-5.1/video/

5.1.8 Hints/FAQ

5.1.8.1 For Users

<table>
<thead>
<tr>
<th>Common Errors, Traps:</th>
</tr>
</thead>
<tbody>
<tr>
<td>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 <code>.pyc</code> 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.</td>
</tr>
</tbody>
</table>
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.

### 5.1.8.2 FAQ

to come

### 5.1.9 Reference of 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:

<table>
<thead>
<tr>
<th>Exit code</th>
<th>Exit message</th>
<th>Thrown by</th>
</tr>
</thead>
<tbody>
<tr>
<td>230</td>
<td>Invalid workchain parameters</td>
<td>CreateMagnetic</td>
</tr>
<tr>
<td>230</td>
<td>Invalid workchain parameters</td>
<td>DMI</td>
</tr>
<tr>
<td>230</td>
<td>Invalid workchain parameters</td>
<td>EOS</td>
</tr>
<tr>
<td>230</td>
<td>Invalid workchain parameters</td>
<td>MAE</td>
</tr>
<tr>
<td>230</td>
<td>Invalid workchain parameters</td>
<td>MAE Conv</td>
</tr>
<tr>
<td>230</td>
<td>Invalid workchain parameters</td>
<td>Relax</td>
</tr>
<tr>
<td>230</td>
<td>Invalid workchain parameters</td>
<td>SCF</td>
</tr>
<tr>
<td>230</td>
<td>Invalid workchain parameters</td>
<td>SSDisp</td>
</tr>
<tr>
<td>230</td>
<td>Invalid workchain parameters</td>
<td>SSDisp Conv</td>
</tr>
<tr>
<td>230</td>
<td>Invalid workchain parameters</td>
<td>BandDos</td>
</tr>
<tr>
<td>231</td>
<td>Invalid input configuration</td>
<td>CreateMagnetic</td>
</tr>
<tr>
<td>231</td>
<td>Invalid input configuration</td>
<td>DMI</td>
</tr>
<tr>
<td>231</td>
<td>Invalid input configuration</td>
<td>MAE</td>
</tr>
<tr>
<td>231</td>
<td>Invalid input configuration</td>
<td>SCF</td>
</tr>
<tr>
<td>231</td>
<td>Invalid input configuration</td>
<td>SSDisp</td>
</tr>
<tr>
<td>231</td>
<td>Invalid input configuration</td>
<td>BandDos</td>
</tr>
<tr>
<td>233</td>
<td>Input codes do not correspond to fleur or inpgen codes respectively.</td>
<td>DMI</td>
</tr>
<tr>
<td>Page</td>
<td>Section</td>
<td>Description</td>
</tr>
<tr>
<td>------</td>
<td>---------</td>
<td>-------------</td>
</tr>
<tr>
<td>233</td>
<td>Input codes do not correspond to fleur or inpgen codes respectively.</td>
<td>MAE</td>
</tr>
<tr>
<td>233</td>
<td>Input codes do not correspond to fleur or inpgen codes respectively.</td>
<td>SSDisp</td>
</tr>
<tr>
<td>233</td>
<td>Input codes do not correspond to fleur or inpgen codes respectively.</td>
<td>BandDos</td>
</tr>
<tr>
<td>235</td>
<td>Input file modification failed.</td>
<td>DMI</td>
</tr>
<tr>
<td>235</td>
<td>Input file modification failed.</td>
<td>MAE</td>
</tr>
<tr>
<td>235</td>
<td>Input file modification failed.</td>
<td>SCF</td>
</tr>
<tr>
<td>235</td>
<td>Input file modification failed.</td>
<td>SSDisp</td>
</tr>
<tr>
<td>235</td>
<td>Input file modification failed.</td>
<td>BandDos</td>
</tr>
<tr>
<td>236</td>
<td>Input file was corrupted after modifications</td>
<td>DMI</td>
</tr>
<tr>
<td>236</td>
<td>Input file was corrupted after modifications</td>
<td>MAE</td>
</tr>
<tr>
<td>236</td>
<td>Input file was corrupted after modifications</td>
<td>SCF</td>
</tr>
<tr>
<td>236</td>
<td>Input file was corrupted after modifications</td>
<td>SSDisp</td>
</tr>
<tr>
<td>236</td>
<td>Input file was corrupted after modifications</td>
<td>BandDos</td>
</tr>
<tr>
<td>300</td>
<td>No retrieved folder found</td>
<td>FleurCalculation</td>
</tr>
<tr>
<td>300</td>
<td>No retrieved folder found</td>
<td>FleurinpgenCalculation</td>
</tr>
<tr>
<td>300</td>
<td>No retrieved folder found</td>
<td>FleurinpgenCalculation</td>
</tr>
<tr>
<td>301</td>
<td>One of the output files can not be opened</td>
<td>FleurCalculation</td>
</tr>
<tr>
<td>301</td>
<td>One of the output files can not be opened</td>
<td>FleurinpgenCalculation</td>
</tr>
<tr>
<td>302</td>
<td>FLEUR calculation failed for unknown reason</td>
<td>FleurCalculation</td>
</tr>
<tr>
<td>303</td>
<td>XML output file was not found</td>
<td>FleurCalculation</td>
</tr>
<tr>
<td>304</td>
<td>Parsing of XML output file failed</td>
<td>FleurCalculation</td>
</tr>
<tr>
<td>305</td>
<td>Parsing of relax XML output file failed</td>
<td>FleurCalculation</td>
</tr>
<tr>
<td>306</td>
<td>XML input file was not found</td>
<td>FleurinpgenCalculation</td>
</tr>
<tr>
<td>310</td>
<td>FLEUR calculation failed due to memory issue</td>
<td>FleurCalculation</td>
</tr>
<tr>
<td>311</td>
<td>FLEUR calculation failed because atoms spilled to the vacuum</td>
<td>FleurBase</td>
</tr>
<tr>
<td>311</td>
<td>FLEUR calculation failed because atoms spilled to the vacuum</td>
<td>FleurCalculation</td>
</tr>
<tr>
<td>311</td>
<td>FLEUR calculation failed because atoms spilled to the vacuum</td>
<td>Relax</td>
</tr>
<tr>
<td>312</td>
<td>FLEUR calculation failed due to MT overlap</td>
<td>FleurCalculation</td>
</tr>
<tr>
<td>313</td>
<td>Overlapping MT-spheres during relaxation</td>
<td>FleurBase</td>
</tr>
<tr>
<td>313</td>
<td>Overlapping MT-spheres during relaxation</td>
<td>FleurCalculation</td>
</tr>
<tr>
<td>313</td>
<td>Overlapping MT-spheres during relaxation</td>
<td>Relax</td>
</tr>
<tr>
<td>314</td>
<td>Problem with cdn is suspected</td>
<td>Relax</td>
</tr>
<tr>
<td>316</td>
<td>Calculation failed due to time limits.</td>
<td>FleurCalculation</td>
</tr>
<tr>
<td>318</td>
<td>Calculation failed due to a missing dependency</td>
<td>FleurCalculation</td>
</tr>
<tr>
<td>334</td>
<td>Reference calculation failed.</td>
<td>DMI</td>
</tr>
<tr>
<td>334</td>
<td>Reference calculation failed.</td>
<td>MAE</td>
</tr>
<tr>
<td>334</td>
<td>Reference calculation failed.</td>
<td>SSDisp</td>
</tr>
<tr>
<td>334</td>
<td>Reference calculation failed.</td>
<td>BandDos</td>
</tr>
<tr>
<td>335</td>
<td>Found no reference calculation remote repository.</td>
<td>DMI</td>
</tr>
<tr>
<td>335</td>
<td>Found no reference calculation remote repository.</td>
<td>MAE</td>
</tr>
<tr>
<td>335</td>
<td>Found no reference calculation remote repository.</td>
<td>SSDisp</td>
</tr>
<tr>
<td>335</td>
<td>Found no reference calculation remote repository.</td>
<td>BandDos</td>
</tr>
<tr>
<td>336</td>
<td>Force theorem calculation failed.</td>
<td>DMI</td>
</tr>
<tr>
<td>336</td>
<td>Force theorem calculation failed.</td>
<td>MAE</td>
</tr>
<tr>
<td>336</td>
<td>Force theorem calculation failed.</td>
<td>SSDisp</td>
</tr>
<tr>
<td>340</td>
<td>Convergence SSDisp calculation failed for all q-vectors</td>
<td>SSDisp conv</td>
</tr>
<tr>
<td>341</td>
<td>Convergence SSDisp calculation failed for some q-vectors</td>
<td>SSDisp conv</td>
</tr>
<tr>
<td>343</td>
<td>Convergence MAE calculation failed for all SQAs</td>
<td>MAE conv</td>
</tr>
<tr>
<td>344</td>
<td>Convergence MAE calculation failed for some SQAs</td>
<td>MAE conv</td>
</tr>
</tbody>
</table>

continues on next page
<table>
<thead>
<tr>
<th>Page</th>
<th>Description</th>
<th>Workchain Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>350</td>
<td>The workchain execution did not lead to relaxation criterion. Thrown in the very end of the workchain.</td>
<td>Relax</td>
</tr>
<tr>
<td>351</td>
<td>SCF Workchains failed for some reason.</td>
<td>Relax</td>
</tr>
<tr>
<td>352</td>
<td>Found no relaxed structure info in the output of SCF</td>
<td>Relax</td>
</tr>
<tr>
<td>353</td>
<td>Found no SCF output</td>
<td>Relax</td>
</tr>
<tr>
<td>354</td>
<td>Force is small, switch to BFGS</td>
<td>Relax</td>
</tr>
<tr>
<td>360</td>
<td>Inpgen calculation failed</td>
<td>SCF</td>
</tr>
<tr>
<td>360</td>
<td>Inpgen calculation failed</td>
<td>SCF</td>
</tr>
<tr>
<td>361</td>
<td>Fleur calculation failed</td>
<td>SCF</td>
</tr>
<tr>
<td>380</td>
<td>Specified substrate is not bcc or fcc, only them are supported</td>
<td>CreateMagnetic</td>
</tr>
<tr>
<td>382</td>
<td>Relaxation calculation failed</td>
<td>CreateMagnetic</td>
</tr>
<tr>
<td>383</td>
<td>EOS WorkChain failed</td>
<td>CreateMagnetic</td>
</tr>
<tr>
<td>388</td>
<td>Fleur Calculation failed due to time limits and it cannot be resolved (e.g because of no cdn file)</td>
<td>FleurBase</td>
</tr>
<tr>
<td>389</td>
<td>FLEUR calculation failed due to memory issue and it can not be solved for this scheduler</td>
<td>FleurBase</td>
</tr>
<tr>
<td>390</td>
<td>check_kpts() suggests less than 60% of node load</td>
<td>FleurBase</td>
</tr>
<tr>
<td>399</td>
<td>FleurCalculation failed and FleurBaseWorkChain has no strategy to resolve this</td>
<td>FleurBase</td>
</tr>
<tr>
<td>399</td>
<td>FleurRelaxWorkChain failed and FleurBaseRelaxWorkChain has no strategy to resolve this</td>
<td>Relax Base</td>
</tr>
</tbody>
</table>
Some things to notice for AiiDA-FLEUR developers. Conventions, programming style, Integrated testing, things that should not be forgotten

6.1 Developer’s guide

This is the developers guide for AiiDA-FLEUR

Contents

- Developer’s guide
  - Package layout
  - Automated tests
  - Plugin development
  - Workflow/chain development
    - General Workflow development guidelines:
    - FLEUR specific design suggestions, conventions:
  - Entrypoints
  - Documentation
  - Other information
    - Useful to know

6.1.1 Package layout

All source code is under ‘aiida_fleur/’
6.1.2 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 ideally 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 understandable 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 .................................................................[ 93%]
parsers/test_fleur_parser.py ........ [ 63%]
tools/test_common_aiida.py . [ 69%]
tools/test_common_fleur_wf.py .. [ 69%]
tools/test_common_fleur_wf_util.py ........ [ 75%]
tools/test_element_econfig_list.py ....... [ 80%]
```

(continues on next page)
If anything (especially a lot of tests) fails it is very likely 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 compatibility within aiida-fleur itself. Therefore, newer versions of it should still work with older versions of the FLEUR.

6.1. Developer's guide
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 output file (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 output file 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.

### 6.1.3 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/necessary.
- 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 usual information the user is interested easy to query for?
6.1.4 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.

### 6.1.4.1 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 explaining ‘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 and 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 separate 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 )
6.1.4.2 FLEUR specific design 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 valuable 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

6.1.5 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 'pyproject.toml' file.

Example:

```
"entry_points" : {
   "aidda.calculations" : [
      "fleur.fleur = aiida_fleur.calculation.fleur:FleurCalculation",
      "fleur.inpgen = aiida_fleur.calculation.fleurinputgen:FleurinputgenCalculation"
   ],
   "aidda.data" : [
      "fleur.fleurinp = aiida_fleur.data.fleurinp:FleurinpData",
      "fleur.fleurinpmodifier = aiida_fleur.data.fleurinpmodifier:FleurinpModifier"
   ],
   "aidda.parsers" : [
      "fleur.fleurfparser = aiida_fleur.parsers.fleur:FleurParser",
      "fleur.fleurinpgenparser = aiida_fleur.parsers.fleur_inputgen:Fleur_inputgenParser"
   ],
   "aidda.workflows" : [
```

(continues on next page)
"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",
"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’.

### 6.1.6 Documentation

Since a lot of the documentation is auto generated it is important that you give every module, class and function proper doc strings.

For the documentation we use sphinx <https://www.sphinx-doc.org/en/master/usage/restructuredtext/basics.html>, which is based on restructured text, also see <https://sublime-and-sphinx-guide.readthedocs.io/en/latest/index.html>. And we build and upload the documentation to readthedocs <https://docs.readthedocs.io/en/stable/index.html>. Also in restructured text headings are marked with some underlining, while the order is arbitrary and sphinx determines it on occurrence. To make the whole documentation consistent it is important that you stay to the conventions of underlying.

<table>
<thead>
<tr>
<th>Heading level</th>
<th>underline with</th>
<th>Comment</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>#</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>*</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>=</td>
<td>usual start here</td>
</tr>
<tr>
<td>3</td>
<td>+</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>–</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>^</td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>‘</td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>’</td>
<td></td>
</tr>
<tr>
<td>8</td>
<td>.</td>
<td></td>
</tr>
</tbody>
</table>
6.1.7 Other information

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

6.1.7.1 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.
7.1 Source code Documentation (API reference)

7.1.1 Fleur input generator plug-in

7.1.1.1 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.

```python
class aiida_fleur.calculation.fleurinputgen.FleurinputgenCalculation(*args: Any, **kwargs: Any)
```

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/](http://www.flapw.de/).

```python
classmethod define(spec)
```

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

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

```python
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.

```python
aiida_fleur.calculation.fleurinputgen.write_inpgen_file_aiida_struct(structure, file, input_params=None, settings=None)
```

Wraps around masci_tools write inpgen_file, unpacks aiida structure.
7.1.1.2 Fleurinputgen Parser

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

```python
class aida_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.

```python
def 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.

7.1.2 Fleur-code plugin

7.1.2.1 Fleur Calculation

This file contains a CalcJob that represents FLEUR calculation.

```python
class aida_fleur.calculation.fleur.FleurCalculation(*args: Any, **kwargs: Any)
```

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

```python
classmethod define(spec)
```

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

Ports are added to the metadata input namespace (inherited from the base Process), and a code input Port, a remote_folder output Port and retrieved folder output Port are added.

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

```python
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.

7.1.2.2 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.

```python
class aida_fleur.parsers.fleur.FleurParser(node: CalcJobNode)
```

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.

```python
def 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.parse_relax_file(relax_file, schema_dict)

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

7.1.3 Fleur input Data structure

7.1.3.1 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(files: Optional[list[str]] = None, node:
Optional[Union[aiida.orm.nodes.node.Node, str, int]] = None, **kwargs: Any)

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, parsed
into a XML tree and validated against the XML schema file for the given file version. These XML schemas are
provided by the masci-tools library

FleurinpData also provides the user with methods to extract AiiDA StructureData, KpointsData nodes and Dict
nodes with LAPW parameters.

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__(files: Optional[list[str]] = None, node: Optional[Union[aiida.orm.nodes.node.Node, str, int]] =
None, **kwargs: Any) → None

Initialize a FleurinpData object set the files given

convert_inpxml(to_version: aiida.orm.nodes.data.str.Str) → aiida_fleur.data.fleurinp.FleurinpData

Convert the fleurinp data node to a different inp.xml version and return a clone of the node

Note: If the Fleurinp contains included xml trees the resulting FleurinpData will contain only the combined
inp.xml

convert_inpxml_ncf(to_version: str) → aiida_fleur.data.fleurinp.FleurinpData

Convert the fleurinp data node to a different inp.xml version and return a clone of the node
Note: If the Fleurinp contains included xml trees the resulting FleurinpData will contain only the combined inp.xml

del_file(filename: str) → None
Remove a file from FleurinpData instance

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

property files: list[str]
Returns the list of the names of the files stored

get_content(filename: str = 'inp.xml') → str
Returns the content of the single file stored for this data node.

Returns
A string of the file content

get_fleur_modes() → dict[str, Any]
Analyses inp.xml file to set up a calculation mode. ‘Modes’ are paths a FLEUR calculation can take, resulting in different output. This files can be automatically added 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.

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(name: Optional[str] = None, index: Optional[int] = None, only_used: bool = False) → aiida.orm.nodes.data.array.kpoints.KpointsData | dict[str, aiida.orm.nodes.data.array.kpoints.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 NOT a calcfunction and does not keep the provenance!

Parameters
• name – str, optional, if given only the kpoint set with the given name is returned

• index – int, optional, if given only the kpoint set with the given index is returned

Returns KpointsData node

get_nkpts() → int
This routine returns the number of kpoints used in the fleur calculation defined in this input

Returns
int with the number of kPoints

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**(inpgen_ready: bool = True, write_ids: bool = True) →
    aida.orm.nodes.data.dict.Dict

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**(normalize_kind_name: Optional[aiida.orm.nodes.data.bool.Bool] = None) →
    aida.orm.nodes.data.structure.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**(normalize_kind_name: bool = True) →
    aida.orm.nodes.data.structure.StructureData

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

**property inp_dict**: dict[str, Any]

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

**property inp_version**: str | None

Returns the version string corresponding to the inp.xml file

**load_inpxml**(validate_xml_schema: bool = True, return_included_tags: bool = False, **kwargs: Any) →
    tuple[lxml.etree._ElementTree,
          masci_tools.io.parsers.fleur_schema.schema_dict.InputSchemaDict]
    tuple[lxml.etree._ElementTree,
          masci_tools.io.parsers.fleur_schema.schema_dict.InputSchemaDict, set[str]]

Returns the lxml etree and the schema dictionary corresponding to the version. If validate_xml_schema=True the file will also be validated against the schema

Keyword arguments are passed on to the parser

**open**(path: str = 'inp.xml', mode: str = 'r') → ContextManager[BinaryIO | TextIO]

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

**property parser_info**: dict[str, Any]

Dict property, with the info and warnings from the inpxml_parser

**set_file**(filename: str, dst_filename: Optional[str] = None, node: Optional[Union[aiida.orm.nodes.node.Node, str, int]] = None) → 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: list[str], node: Optional[Union[aiida.orm.nodes.node.Node, str, int]] = None) → None`

Add the list of files to the FleurinpData instance. Can by used as an alternative to the setter.

Parameters

• files – list of absolute filepaths or filenames of node is specified
• node – a FolderData node containing files from the filelist

`aiida_fleur.data.fleurinp.convert_inpxml(fleurinp: aiida_fleur.data.fleurinp.FleurinpData, to_version: aiida.orm.nodes.data.str.Str) → aiida_fleur.data.fleurinp.FleurinpData`

Convert the fleurinp data node to a different inp.xml version and return a clone of the node

Note: If the Fleurinp contains included xml trees the resulting FleurinpData will contain only the combined inp.xml

`aiida_fleur.data.fleurinp.get_fleurinp_from_folder_data(folder_node: aiida.orm.nodes.data.folder.FolderData, store: bool = False, additional_files: Optional[list[str]] = None) → aiida_fleur.data.fleurinp.FleurinpData`

Create FleurinpData object from the given RemoteData object

Parameters

• remote_node – RemoteData to use for the generation of the FleurinpData
• store – bool, if True the FleurinpData object will be stored after generation

Returns FleurinpData object with the input xml files from the retrieved folder of the calculation associated RemoteData

`aiida_fleur.data.fleurinp.get_fleurinp_from_folder_data_cf(folder_node: aiida.orm.nodes.data.folder.FolderData, additional_files: aiida.orm.nodes.data.list.List | None = None) → aiida_fleur.data.fleurinp.FleurinpData`

Create FleurinpData object from the given FolderData object

Parameters remote_node – FolderData to use for the generation of the FleurinpData

Returns FleurinpData object with the input xml files from the FolderData

`aiida_fleur.data.fleurinp.get_fleurinp_from_remote_data(remote_node: aiida.orm.nodes.data.remote.base.RemoteData, store: bool = False, additional_files: Optional[list[str]] = None) → aiida_fleur.data.fleurinp.FleurinpData`

Create FleurinpData object from the given RemoteData object

Parameters

• remote_node – RemoteData to use for the generation of the FleurinpData
• store – bool, if True the FleurinpData object will be stored after generation
Returns FleurinpData object with the input xml files from the retrieved folder of the calculation associated RemoteData

aiida_fleur.data.fleurinp.get_fleurinp_from_remote_data_cf(remote_node: aiida.orm.nodes.data.remote.base.RemoteData, additional_files: aiida.orm.nodes.data.list.List | None = None) → aiida_fleur.data.fleurinp.FleurinpData

Create FleurinpData object from the given RemoteData object

Parameters

• remote_node – RemoteData to use for the generation of the FleurinpData
• store – bool, if True the FleurinpData object will be stored after generation

Returns FleurinpData object with the input xml files from the retrieved folder of the calculation associated RemoteData

aiida_fleur.data.fleurinp.get_kpointsdata(fleurinp: aiida_fleur.data.fleurinp.FleurinpData, name: aiida.orm.nodes.data.str.Str | None = None, index: aiida.orm.nodes.data.int.Int | None = None, only_used: aiida.orm.nodes.data.bool.Bool | None = None) → aiida.orm.nodes.data.array.kpoints.KpointsData | dict[str, aiida.orm.nodes.data.array.kpoints.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 or dict of KpointsData

aiida_fleur.data.fleurinp.get_parameterdata(fleurinp: aiida_fleur.data.fleurinp.FleurinpData, inpgen_ready: aiida.orm.nodes.data.bool.Bool | None = None, write_ids: aiida.orm.nodes.data.bool.Bool | None = None) → aiida.orm.nodes.data.dict.Dict

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

aiida_fleur.data.fleurinp.get_structuredata(fleurinp: aiida_fleur.data.fleurinp.FleurinpData, normalize_kind_name: aiida.orm.nodes.data.bool.Bool | None = None) → aiida.orm.nodes.data.structure.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
7.1.3.2 Fleurinp modifier

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

class aiida_fleur.data.fleurinpmodifier.FleurinpModifier(validate_signatures=True)

A class which represents changes to the FleurinpData object.

add_number_to_attrib(*args: Any, **kwargs: Any) → None

Appends a add_number_to_attrib() to the list of tasks that will be done on the xmltree.

Adds a given number to the attribute value in a xmltree specified by the name of the attribute and optional further specification If there are no nodes under the specified xpath an error is raised

Parameters

- name – the attribute name to change
- number_to_add – number to add/multiply with the old attribute value
- complex_xpath – an optional xpath to use instead of the simple xpath for the evaluation
- filters – Dict specifying constraints to apply on the xpath. See XPathBuilder for details
- mode – str (either rel/relative or abs/absolute). rel/relative multiplies the old value with number_to_add abs/absolute adds the old value and number_to_add
- occurrences – int or list of int. Which occurrence of the node to set. By default all are set.

Kwargs:

- param tag_name str, name of the tag where the attribute should be parsed
- param contains str, this string has to be in the final path
- param not_contains str, this string has to NOT be in the final path
- param exclude list of str, here specific types of attributes can be excluded valid values are: settable, settable_contains, other

add_number_to_first_attrib(*args: Any, **kwargs: Any) → None

Appends a add_number_to_first_attrib() to the list of tasks that will be done on the xmltree.

Adds a given number to the first occurrence of an attribute value in a xmltree specified by the name of the attribute and optional further specification If there are no nodes under the specified xpath an error is raised

Parameters

- name – the attribute name to change
- number_to_add – number to add/multiply with the old attribute value
- complex_xpath – an optional xpath to use instead of the simple xpath for the evaluation
- mode – str (either rel/relative or abs/absolute). rel/relative multiplies the old value with number_to_add abs/absolute adds the old value and number_to_add
- filters – Dict specifying constraints to apply on the xpath. See XPathBuilder for details

Kwargs:
**add_task_list** *(task_list: list[tuple[str, dict[str, Any]]]) → None*

Add a list of tasks to be added.

**Parameters**
- **task_list** – list of tuples first index is the name of the method second is defining the arguments by keyword in a dict

**align_nmmpmat_to_sqa** *(*args: Any, **kwargs: Any) → None*

Appends a `align_nmmpmat_to_sqa()` to the list of tasks that will be done on the xmltree.

Align the density matrix with the given SQA of the associated species.

**Parameters**
- **species_name** – string, name of the species you want to change
- **orbital** – integer or string (‘all’), orbital quantum number of the LDA+U procedure to be modified
- **phi_before** – float or list of floats, angle (radian), values for phi for the previous alignment of the density matrix
- **theta_before** – float or list of floats, angle (radian), values for theta for the previous alignment of the density matrix
- **filters** – Dict specifying constraints to apply on the xpath. See `XPathBuilder` for details

**Raises**
- **ValueError** – If something in the input is wrong
- **KeyError** – If no LDA+U procedure is found on a species

**classmethod apply_fleurinp_modifications** *(new_fleurinp: aida_fleur.data.fleurinp.FleurinpData, modification_tasks: list[masci_tools.io.fleurxmlmodifier.ModifierTask]) → None*

Apply the modifications working directly on the cloned FleurinpData instance. The functions will warn the user if one of the methods executed here are after XML modifications in the task list, since this method will implicitly reorder the order of the execution.

**Warning:** These should be performed BEFORE the XML Modification methods in any of the functions doing modifications (`FleurinpModifier.show()`, `FleurinpModifier.validate()`, `FleurinpModifier.freeze()`).

**Parameters**
- **new_fleurinp** – The Fleurinpdata instance cloned from the original
- **modification_tasks** – a list of modification tuples
class \texttt{apply}\_modifications}(xmltree: lxml.etree._ElementTree, nmmp\_lines: list[str] | None, modification\_tasks: list[masci\_tools.io.fleurxmlmodifier.ModifierTask], validate\_changes: bool = True, adjust\_version\_for\_dev\_version: bool = True) → tuple[lxml.etree._ElementTree, list[str] | 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**

- \texttt{xmltree} – a lxml tree to be modified (IS MODIFIED INPLACE)
- \texttt{nmmp\_lines} – a n_mmp_mat file to be modified (IS MODIFIED INPLACE)
- \texttt{modification\_tasks} – a list of modification tuples
- \texttt{validate\_changes} – bool optional (default True), if True after all tasks are performed both the xmltree and nmmp_lines are checked for consistency
- \texttt{adjust\_version\_for\_dev\_version} – bool optional (default True), if True and the schema\_dict and file version differ, e.g. a development version is used the version is temporarily modified to swallow the validation error that would occur

**Returns** a modified lxml tree and a modified n_mmp_mat file

\texttt{changes()} → list[masci\_tools.io.fleurxmlmodifier.ModifierTask]

Prints out all changes currently registered on this instance

\texttt{clone\_species(*args: Any, **kwargs: Any) → None}

Appends a \texttt{clone\_species()} to the list of tasks that will be done on the xmltree.

Method to create a new species from an existing one with evtl. modifications

For reference of the changes dictionary look at \texttt{set\_species()}

**Parameters**

- \texttt{species\_name} – string, name of the specie you want to clone Has to correspond to one single species (no ‘all’/’all-<search_string>’)
- \texttt{new\_name} – new name of the cloned species
- \texttt{changes} – a optional python dict specifying what you want to change.

\texttt{create\_tag(*args: Any, **kwargs: Any) → None}

Appends a \texttt{create\_tag()} to the list of tasks that will be done on the xmltree.

**Parameters**

- \texttt{tag} – str of the tag to create
- \texttt{complex\_xpath} – an optional xpath to use instead of the simple xpath for the evaluation
- \texttt{create\_parents} – bool optional (default False), if True and the given xpath has no results the the parent tags are created recursively
- \texttt{occurrences} – int or list of int. Which occurrence of the parent nodes to create a tag. By default all nodes are used.

**Kwargs:**

- \texttt{param contains} str, this string has to be in the final path
- \texttt{param not\_contains} str, this string has to NOT be in the final path
del_file(filename: str) → None

Appends a `del_file()` to the list of tasks that will be done on the FleurinpData instance.

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

delete_att(*args: Any, **kwargs: Any) → None

Appends a `delete_att()` to the list of tasks that will be done on the xmltree.

This method deletes a attribute with a uniquely identified xpath.

**Parameters**
- **name** – str of the attribute to delete
- **complex_xpath** – an optional xpath to use instead of the simple xpath for the evaluation
- **filters** – Dict specifying constraints to apply on the xpath. See `XPathBuilder` for details
- **occurrences** – int or list of int. Which occurrence of the parent nodes to delete a attribute. By default all nodes are used.

**Kwargs:**
- **tag_name** str, name of the tag where the attribute should be parsed
- **contains** str, this string has to be in the final path
- **not_contains** str, this string has to NOT be in the final path
- **exclude** list of str, here specific types of attributes can be excluded valid values are: settable, settable_contains, other

delete_tag(*args: Any, **kwargs: Any) → None

Appends a `delete_tag()` to the list of tasks that will be done on the xmltree.

This method deletes a tag with a uniquely identified xpath.

**Parameters**
- **tag** – str of the tag to delete
- **complex_xpath** – an optional xpath to use instead of the simple xpath for the evaluation
- **filters** – Dict specifying constraints to apply on the xpath. See `XPathBuilder` for details
- **occurrences** – int or list of int. Which occurrence of the parent nodes to delete a tag. By default all nodes are used.

**Kwargs:**
- **contains** str, this string has to be in the final path
- **not_contains** str, this string has to NOT be in the final path

freeze() → aiida_fleur.data.fleurinp.FleurinpData

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

**Returns**
- stored `FleurinpData` with applied changes
classmethod fromList(task_list: list[tuple[str, dict[str, Any]]], *args: Any, **kwargs: Any) → masci_tools.io.fleurxmlmodifier.FleurXMLModifier

Instantiate the FleurXMLModifier from a list of tasks to be added immediately

Parameters

task_list – list of tuples first index is the name of the method second is defining the arguments by keyword in a dict

Other arguments are passed on to the __init__ method

Returns class with the task list instantiated

get_avail_actions() → dict[str, Callable]

Returns the allowed functions from FleurinpModifier

replace_tag(*args: Any, **kwargs: Any) → None

Deprecation layer for replace_tag if there are slashes in the first positional argument or xpath is is in kwargs. We know that it is the old usage.

Appends a replace_tag() to the list of tasks that will be done on the xmltree.

Parameters

• tag – str of the tag to replace
• newelement – a new tag
• complex_xpath – an optional xpath to use instead of the simple xpath for the evaluation
• occurrences – int or list of int. Which occurrence of the parent nodes to replace a tag. By default all nodes are used.

Kwars:

param contains str, this string has to be in the final path

param not contains str, this string has to NOT be in the final path

rotate_numpmat(*args: Any, **kwargs: Any) → None

Appends a rotate_numpmat() to the list of tasks that will be done on the xmltree.

Rotate the density matrix with the given angles phi and theta

Parameters

• species_name – string, name of the species you want to change
• orbital – integer or string (‘all’), orbital quantum number of the LDA+U procedure to be modified
• phi – float, angle (radian), by which to rotate the density matrix
• theta – float, angle (radian), by which to rotate the density matrix
• filters – Dict specifying constraints to apply on the xpath. See XPathBuilder for de-
tails

Raises

• ValueError – If something in the input is wrong
• KeyError – If no LDA+U procedure is found on a species
set_atomgroup(*args: Any, **kwargs: Any) → None

Appends a set_atomgroup() to the list of tasks that will be done on the xmltree.

Method to set parameters of an atom group of the fleur inp.xml file.

**Parameters**

- **changes** – 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 species will be changed
- **filters** – Dict specifying constraints to apply on the xpath. See XPathBuilder for details

`changes` 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:

```
'changes': {'nocoParams': {'beta': val}}
```

set_atomgroup_label(*args: Any, **kwargs: Any) → None

Appends a set_atomgroup_label() to the list of tasks that will be done on the xmltree.

This method calls set_atomgroup() method for a certain atom species that corresponds to an atom with a given label.

**Parameters**

- **atom_label** – string, a label of the atom which specie will be changed. ‘all’ to change all the species
- **changes** – a python dict specifying what you want to change.

`changes` 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:

```
'changes': {'nocoParams': {'beta': val}}
```

set_attrib_value(*args: Any, **kwargs: Any) → None

Appends a set_attrib_value() to the list of tasks that will be done on the xmltree.

Sets an attribute in a xmltree to a given value, specified by its name and further specifications. If there are no nodes under the specified xpath a tag can be created with `create=True`. The attribute values are converted automatically according to the types of the attribute with `convert_to_xml()` if they are not `str` already.

**Parameters**

- **name** – the attribute name to set
- **value** – value or list of values to set
- **complex_xpath** – an optional xpath to use instead of the simple xpath for the evaluation
- **filters** – Dict specifying constraints to apply on the xpath. See XPathBuilder for details
- **occurrences** – int or list of int. Which occurrence of the node to set. By default all are set.
- **create** – bool optional (default False), if True the tag is created if is missing
Kwargs:

- **param tag_name**  str, name of the tag where the attribute should be parsed
- **param contains**  str, this string has to be in the final path
- **param not_contains**  str, this string has to NOT be in the final path
- **param exclude**  list of str, here specific types of attributes can be excluded valid values are:
  - settable, settable_contains, other

`set_complex_tag(*args: Any, **kwargs: Any) → None`

Appends a `set_complex_tag()` to the list of tasks that will be done on the xmltree.

Function to correctly set tags/attributes for a given tag. Goes through the attributedict and decides based
on the schema_dict, how the corresponding key has to be handled. The tag is specified via its name and
evtl. further specification

Supports:

- attributes
- tags with text only
- simple tags, i.e. only attributes (can be optional single/multiple)
- complex tags, will recursively create/modify them

Parameters

- **tag_name**  – name of the tag to set
- **changes**  – Keys in the dictionary correspond to names of tags and the values are the mod-
difications to do on this tag (attributename, subdict with changes to the subtag, …)
- **complex_xpath**  – an optional xpath to use instead of the simple xpath for the evaluation
- **filters**  – Dict specifying constraints to apply on the xpath. See `XPathBuilder` for de-
tails
- **create**  – bool optional (default False), if True and the path, where the complex tag is set
does not exist it is created

Kwargs:

- **param contains**  str, this string has to be in the final path
- **param not_contains**  str, this string has to NOT be in the final path

`set_file(filename: str, dst_filename: Optional[str] = None, node: Optional[Union[int, str,
aiida.orm.nodes.data.data.Data]] = None) → None`

Appends a `set_file()` to the list of tasks that will be done on the FleurinpData instance.

Parameters

- **filename**  – absolute path to the file or a filename of node is specified
- **dst_filename**  – str of the filename, which should be used instead of the real filename to
  store it
- **node**  – a `FolderData` node containing the file
**set_first_attrib_value**(*args: Any, **kwargs: Any) → None

Appends a `set_first_attrib_value()` to the list of tasks that will be done on the xmltree.

Sets the first occurrence of an attribute in a xmltree to a given value, specified by its name and further specifications. If there are no nodes under the specified xpath a tag can be created with `create=True`. The attribute values are converted automatically according to the types of the attribute with `convert_to_xml()` if they are not `str` already.

**Parameters**

- **name** – the attribute name to set
- **value** – value or list of values to set
- **complex_xpath** – an optional xpath to use instead of the simple xpath for the evaluation
- **filters** – Dict specifying constraints to apply on the xpath. See **XPathBuilder** for details
- **create** – bool optional (default False), if True the tag is created if is missing

**Kwargs:**

- **tag_name** str, name of the tag where the attribute should be parsed
- **contains** str, this string has to be in the final path
- **not_contains** str, this string has to NOT be in the final path
- **exclude** list of str, here specific types of attributes can be excluded valid values are: settable, settable_contains, other

**set_first_text**(*args: Any, **kwargs: Any) → None

Appends a `set_first_text()` to the list of tasks that will be done on the xmltree.

Sets the text the first occurrence of a tag in a xmltree to a given value, specified by the name of the tag and further specifications. By default the text will be set on all nodes returned for the specified xpath. If there are no nodes under the specified xpath a tag can be created with `create=True`. The text values are converted automatically according to the types with `convert_to_xml()` if they are not `str` already.

**Parameters**

- **tag_name** – str name of the tag, where the text should be set
- **text** – value or list of values to set
- **complex_xpath** – an optional xpath to use instead of the simple xpath for the evaluation
- **filters** – Dict specifying constraints to apply on the xpath. See **XPathBuilder** for details
- **create** – bool optional (default False), if True the tag is created if is missing

**Kwargs:**

- **contains** str, this string has to be in the final path
- **not_contains** str, this string has to NOT be in the final path
set_inpchanges(*args: Any, **kwargs: Any) → None

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

This method sets all the attribute and texts provided in the change_dict.

The first occurrence of the attribute/tag is set

**Parameters**

- **changes** – dictionary {attrib_name : value} with all the wanted changes.
- **path_spec** – dict, with ggf. necessary further specifications for the path of the attribute

An example of changes:

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

set_kpath(*args: Any, **kwargs: Any) → None

Appends a set_kpath() to the list of tasks that will be done on the xmltree.

Sets a k-path directly into inp.xml as an alternative kpoint set with purpose ‘bands’

**Warning:** This method is only supported for input versions before the Max5 release

**Parameters**

- **kpath** – a dictionary with kpoint name as key and k point coordinate as value
- **count** – number of k-points
- **gamma** – bool that controls if the gamma-point should be included in the k-point mesh

set_kpointlist(*args: Any, **kwargs: Any) → None

Appends a set_kpointlist() to the list of tasks that will be done on the xmltree.

Explicitly create a kPointList from the given kpoints and weights. This routine will add the specified kPointList with the given name.

**Warning:** For input versions Max4 and older all keyword arguments are not valid (name, kpoint_type, special_labels, switch and overwrite)

**Parameters**

- **kpoints** – list or array containing the relative coordinates of the kpoints
- **weights** – list or array containing the weights of the kpoints
- **name** – str for the name of the list, if not given a default name is generated
- **kpoint_type** – str specifying the type of the kPointList (‘path’, ‘mesh’, ‘spex’, ‘tria’, …)
- **special_labels** – dict mapping indices to labels. The labels will be inserted for the kpoints corresponding to the given index
- **switch** – bool, if True the kPoint list will be used by Fleur when starting the next calculation
- **overwrite** – bool, if True and a kPoint list with the given name already exists it will be overwritten

```python
def set_kpointmesh(*args: Any, **kwargs: Any) -> None
```

Appends a `set_kpointmesh()` to the list of tasks that will be done on the xmltree.

Create a kpoint mesh using spglib

for details see `get_stabilized_reciprocal_mesh()`

**Parameters**

- **mesh** – list-like woth three elements, giving the size of the kpoint set in each direction
- **use_symmetry** – bool if True the available symmetry operations in the inp.xml will be used to reduce the kpoint set otherwise only the identity matrix is used
- **name** – Name of the created kpoint list. If not given a name is generated
- **switch** – bool if True the kpoint list is directly set as the used set
- **overwrite** – if True and a kpoint list of the given name already exists it will be overwritten
- **shift** – shift the center of the kpoint set
- **time_reversal** – bool if True time reversal symmetry will be used to reduce the kpoint set
- **map_to_first_bz** – bool if True the kpoints are mapped into the [0,1] interval

```python
def set_kpointpath(*args: Any, **kwargs: Any) -> None
```

Appends a `set_kpointpath()` to the list of tasks that will be done on the xmltree.

Create a kpoint list for a bandstructure calculation (using ASE kpath generation)

The path can be defined explicitly (see `bandpath()`) or derived from the unit cell

**Parameters**

- **path** – str, list of str or None defines the path to interpolate (for syntax `bandpath()`)
- **nkpts** – int number of kpoints in the path
- **density** – float number of kpoints per Angstroem
- **name** – Name of the created kpoint list. If not given a name is generated
- **switch** – bool if True the kpoint list is directly set as the used set
- **overwrite** – if True and a kpoint list of the given name already exists it will be overwritten
- **special_points** – dict mapping names to coordinates for special points to use

```python
def set_kpointsdata(kpointsdata_uuid: int | str | aiida.orm.nodes.data.array.kpoints.KpointsData, name: Optional[str] = None, switch: bool = False, kpoint_type: str = 'path') -> None
```

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

**Parameters**

- **kpointsdata_uuid** – node identifier or `KpointsData` node to be written into inp.xml
- **name** – str name to give the newly entered kpoint list (only MaX5 or later)
- **switch** – bool if True the entered kpoint list will be used directly (only Max5 or later)
- **kpoint_type** – str of the type of kpoint list given (mesh, path, etc.) only Max5 or later
set_nkpts(*args: Any, **kwargs: Any) → None

Appends a set_nkpts() to the list of tasks that will be done on the xmltree.
Sets a k-point mesh directly into inp.xml

**Warning:** This method is only supported for input versions before the Max5 release

**Parameters**

- **count** – number of k-points
- **gamma** – bool that controls if the gamma-point should be included in the k-point mesh

set_nmmpmat(*args: Any, **kwargs: Any) → None

Appends a set_nmmpmat() to the list of tasks that will be done on the xmltree.
Routine sets the block in the n_mmp_mat file specified by species_name, orbital and spin to the desired density matrix

**Parameters**

- **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
- **state_occupations** – list, sets the diagonal elements of the density matrix and everything else to zero
- **denmat** – matrix, specify the density matrix explicitly
- **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
- **filters** – Dict specifying constraints to apply on the xpath. See XPathBuilder for details

**Raises**

- **ValueError** – If something in the input is wrong
- **KeyError** – If no LDA+U procedure is found on a species

set_simple_tag(*args: Any, **kwargs: Any) → None

Appends a set_simple_tag() to the list of tasks that will be done on the xmltree.
Sets one or multiple simple tag(s) in an xmltree. A simple tag can only hold attributes and has no subtags.
The tag is specified by its name and further specification If the tag can occur multiple times all existing tags are DELETED and new ones are written. If the tag only occurs once it will automatically be created if its missing.

**Parameters**

- **tag_name** – str name of the tag to modify/set
- **changes** – list of dicts or dict with the changes. Elements in list describe multiple tags. Keys in the dictionary correspond to {‘attributename’: attributevalue}
- **complex_xpath** – an optional xpath to use instead of the simple xpath for the evaluation
• **filters** – Dict specifying constraints to apply on the xpath. See *XPathBuilder* for details
• **create_parents** – bool optional (default False), if True and the path, where the simple tags are set does not exist it is created

**Kwargs:**

- **param contains** str, this string has to be in the final path
- **param not contains** str, this string has to NOT be in the final path

```python
set_species(*args: Any, **kwargs: Any) → None
```

Appends a `set_species()` to the list of tasks that will be done on the xmltree.

Method to set parameters of a species tag of the fleur inp.xml file.

**Parameters**

- **species_name** – string, name of the specie you want to change. Can be name of the species, ‘all’ or ‘all-<string>’ (sets species with the string in the species name)
- **changes** – a python dict specifying what you want to change.
- **create** – bool, if species does not exist create it and all subtags?
- **filters** – Dict specifying constraints to apply on the xpath. See *XPathBuilder* for details

**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 xmltree** xml etree of the new inp.xml

**changes** 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:

```python
changes = {'mtSphere': {'radius': 2.2}}
```

Another example:

```python
'changes': {'special': {'socscale': 0.0}}
```

that switches SOC terms on a certain 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

```python
set_species_label(*args: Any, **kwargs: Any) → None
```

Appends a `set_species_label()` to the list of tasks that will be done on the xmltree.

This method calls `set_species()` method for a certain atom species that corresponds to an atom with a given label

**Parameters**

- **atom_label** – string, a label of the atom which specie will be changed. ‘all’ to change all the species
- **changes** – a python dict specifying what you want to change.
- **create** – bool, if species does not exist create it and all subtags?
**set_text**(*args: Any, **kwargs: Any) → None

Appends a `set_text()` to the list of tasks that will be done on the xmltree.

Sets the text on tags in an xmltree to a given value, specified by the name of the tag and further specifications. By default the text will be set on all nodes returned for the specified xpath. If there are no nodes under the specified xpath a tag can be created with `create=True`. The text values are converted automatically according to the types with `convert_to_xml()` if they are not `str` already.

**Parameters**

- **tag_name** – str name of the tag, where the text should be set
- **text** – value or list of values to set
- **complex_xpath** – an optional xpath to use instead of the simple xpath for the evaluation
- **filters** – Dict specifying constraints to apply on the xpath. See `XPathBuilder` for details
- **occurrences** – int or list of int. Which occurrence of the node to set. By default all are set.
- **create** – bool optional (default False), if True the tag is created if is missing

**Kwargs:**

- **param contains** str, this string has to be in the final path
- **param not_contains** str, this string has to NOT be in the final path

**set_xcfunctional**(*args: Any, **kwargs: Any) → None

Appends a `set_xcfunctional()` to the list of tasks that will be done on the xmltree.

Set the Exchange Correlation potential tag

Setting a inbuilt XC functional .. code-block:: python

```
set_xcfunctional(xmltree, schema_dict, 'vwn')
```

Setting a LibXC XC functional .. code-block:: python

```
set_xcfunctional(xmltree, schema_dict, {'exchange': 'lda_x', 'correlation':"lda_c_xalpha"}, libxc=True)
```

**Parameters**

- **xc_functional** – str or dict. If str it is the name of a inbuilt XC functional. If it is a dict it specifies either the name or id for LibXC functionals for the keys ‘exchange’, ‘correlation’, ‘etot_exchange’ and ‘etot_correlation’
- **xc_functional_options** – dict with further general changes to the `xcFunctional` tag
- **libxc** – bool if True the functional is a LibXC functional

**shift_value**(*args: Any, **kwargs: Any) → None

Appends a `shift_value()` to the list of tasks that will be done on the xmltree.

Shifts numerical values of attributes directly in the inp.xml file.

The first occurrence of the attribute is shifted

**Parameters**

- **changes** – a python dictionary with the keys to shift and the shift values.
• **mode** – str (either `rel/relative` or `abs/absolute`). `rel/relative` multiplies the old value with the given value `abs/absolute` adds the old value and the given value

• **path_spec** – dict, with ggf. necessary further specifications for the path of the attribute

An example of changes:

```
changes = {'itmax': 1, 'dVac': -0.123}
```

**shift_value_species_label**(*args: Any, **kwargs: Any) → None

Appends a `shift_value_species_label()` to the list of tasks that will be done on the xmltree.

Shifts the value of an attribute on a species by label if `atom_label` contains ‘all’ then applies to all species

**Parameters**

• **atom_label** – string, a label of the atom which specie will be changed. ‘all’ if set up all species

• **attribute_name** – name of the attribute to change

• **number_to_add** – value to add or to multiply by

• **mode** – str (either `rel/relative` or `abs/absolute`). `rel/relative` multiplies the old value with `number_to_add abs/absolute` adds the old value and `number_to_add`

**Kwargs if the attribute_name does not correspond to a unique path:**

• **param contains** str, this string has to be in the final path

• **param not_contains** str, this string has to NOT be in the final path

**show**(*display: bool = True, validate: bool = False) → lxml.etree._ElementTree

Applies the modifications and displays/prin ts 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

**switch_kpointset**(*args: Any, **kwargs: Any) → None

Appends a `switch_kpointset()` to the list of tasks that will be done on the xmltree.

Switch the used k-point set

**Warning:** This method is only supported for input versions after the Max5 release

**Parameters**

• **list_name** – name of the kPoint set to use

**switch_species**(*args: Any, **kwargs: Any) → None

Appends a `switch_species()` to the list of tasks that will be done on the xmltree.

Method to switch the species of an atom group of the fleur inp.xml file.

**Parameters**

• **new_species_name** – name of the species to switch to
• **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 species will be changed

• **clone** – if True and the new species name does not exist and it corresponds to changing from one species the species will be cloned with `clone_species()`

• **changes** – changes to do if the species is cloned

• **filters** – Dict specifying constraints to apply on the xpath. See XPathBuilder for details

```python
switch_species_label(*args: Any, **kwargs: Any) → None
```

Appends a `switch_species_label()` to the list of tasks that will be done on the xmltree.

Method to switch the species of an atom group of the fleur inp.xml file based on a label of a contained atom

**Parameters**

• **atom_label** – string, a label of the atom which group will be changed. ‘all’ to change all the groups

• **new_species_name** – name of the species to switch to

• **clone** – if True and the new species name does not exist and it corresponds to changing from one species the species will be cloned with `clone_species()`

• **changes** – changes to do if the species is cloned

```python
property task_list: list[tuple[str, dict[str, Any]]]
```

Return the current changes in a format accepted by `add_task_list()` and `fromList()`

```python
undo(revert_all: bool = False) → list[masci_tools.io.fleurxmlmodifier.ModifierTask]
```

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.

```python
validate() → lxml.etree._ElementTree
```

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

```python
xml_create_tag(*args: Any, **kwargs: Any) → None
```

Appends a `xml_create_tag()` to the list of tasks that will be done on the xmltree.

**Parameters**

• **xpath** – a path where to place a new tag

• **element** – a tag name or etree Element to be created

• **place_index** – defines the place where to put a created tag

• **tag_order** – defines a tag order

• **occurrences** – int or list of int. Which occurrence of the parent nodes to create a tag. By default all nodes are used.

```python
xml_delete_att(*args: Any, **kwargs: Any) → None
```

Appends a `xml_delete_att()` to the list of tasks that will be done on the xmltree.

Deletes an attribute in the XML tree

**Parameters**
• **xpath** – a path to the attribute to be deleted

• **name** – the name of an attribute to delete

• **occurrences** – int or list of int. Which occurrence of the parent nodes to create a tag. By default all nodes are used.

```python
xml_delete_tag(*args: Any, **kwargs: Any) -> None
```

Appears a `xml_delete_tag()` to the list of tasks that will be done on the xmltree.

Deletes a tag in the XML tree.

**Parameters**

• **xpath** – a path to the tag to be deleted

• **occurrences** – int or list of int. Which occurrence of the parent nodes to create a tag. By default all nodes are used.

```python
xml_replace_tag(*args: Any, **kwargs: Any) -> None
```

Appears a `xml_replace_tag()` to the list of tasks that will be done on the xmltree.

**Parameters**

• **xpath** – a path to the tag to be replaced

• **element** – a new tag

• **occurrences** – int or list of int. Which occurrence of the parent nodes to create a tag. By default all nodes are used.

```python
xml_set_attrib_value_no_create(*args: Any, **kwargs: Any) -> None
```

Appears a `xml_set_attrib_value_no_create()` to the list of tasks that will be done on the xmltree.

Sets an attribute in a xmltree to a given value. By default the attribute will be set on all nodes returned for the specified xpath.

**Parameters**

• **xpath** – a path where to set the attributes

• **name** – the attribute name to set

• **value** – value or list of values to set (if not str they will be converted with `str(value)`)  

• **occurrences** – int or list of int. Which occurrence of the node to set. By default all are set.

**Raises** `ValueError` – If the lengths of attribv or occurrences do not match number of nodes

```python
xml_set_text_no_create(*args: Any, **kwargs: Any) -> None
```

Appears a `xml_set_text_no_create()` to the list of tasks that will be done on the xmltree.

Sets the text of a tag in a xmltree to a given value. By default the text will be set on all nodes returned for the specified xpath.

**Parameters**

• **xpath** – a path where to set the text

• **text** – value or list of values to set (if not str they will be converted with `str(value)`)  

• **occurrences** – int or list of int. Which occurrence of the node to set. By default all are set.

**Raises** `ValueError` – If the lengths of text or occurrences do not match number of nodes

Context manager to construct an inpxml_changes entry in the given dictionary

Usage:

```python
with inpxml_changes(parameters) as fm:
    #parameters is a dict, which can also already contain an inpxml_changes entry
    fm.set_inpxml_changes({'l_noco': True, 'ctail': False})
    fm.set_species('all-Nd', {'electronConfig': {'flipspins': True}})

print(parameters) #The parameters now also contains the tasks defined in the with block
```

Example for usage with a Builder:

```python
from aiida import plugins
FleurBandDOS = plugins.WorkflowFactory('fleur.banddos')
inputs = FleurBandDOS.get_builder()

with inpxml_changes(inputs) as fm:
    fm.set_inpxml_changes({'l_noco': True, 'ctail': False})
    fm.set_species('all-Nd', {'electronConfig': {'flipspins': True}})

print(inputs.wf_parameters['inpxml_changes']) #The wf_parameters in the root level namespace are now set

with inpxml_changes(inputs.scf) as fm:
    fm.switch_kpoints('my-awesome-kpoints')

print(inputs.scf.wf_parameters['inpxml_changes']) #The wf_parameters in the scf namespace are now set
```

Parameters

- **wf_parameters** – dict or aiida Dict (no stored) into which to enter the changes
- **append** – bool if True the tasks are appended behind any evtl. already defined. For False the tasks are added in front
- **builder_entry** – name of the entry for the inp.xml changes inside the parameters dictionary
- **builder_replace_stored** – if True and a ProcessBuilder is given and the wf_parameters input is given and already stored the produced changes will replace the node

aiida_fleur.data.fleurinpmodifier.modify_fleurinpdata

aiida_fleur.data.fleurinpmodifier.modify_fleurinpdata(original: aiida_fleur.data.fleurinp.FleurinpData, modifications: aiida.orm.nodes.data.dict.Dict, **kwargs: aiida.orm.nodes.node.Node) → aiida_fleur.data.fleurinp.FleurinpData
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

### 7.1.4 Workflows/Workchains

#### 7.1.4.1 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.

```python
class aiida_fleur.workflows.base_fleur.FleurBaseWorkChain(*args: Any, **kwargs: Any)
```

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 without submission.

**classmethod define(spec)**

Define the process specification.

**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.

#### 7.1.4.2 SCF: Fleur-Scf WorkChain

In this module you find the workchain ‘FleurScfWorkChain’ for the self-consistency cycle management of a FLEUR calculation with AiiDA.

```python
class aiida_fleur.workflows.scf.FleurScfWorkChain(*args: Any, **kwargs: Any)
```

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

**classmethod define**(spec)  
Define the specification of the process, including its inputs, outputs and known exit codes.  
A *metadata* input namespace is defined, with optional ports that are not stored in the database.

**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

**reset_straight_mixing**()  
Turn off the straight mixing features again

**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.

7.1.4.3 BandDos: Bandstructure WorkChain

This is the workflow ‘band’ for the Fleur code, which calculates a electron bandstructure.

```python
class aiida_fleur.workflows.banddos.FleurBandDosWorkChain(*args: Any, **kwargs: Any)

This workflow calculated a bandstructure from a Fleur calculation

   Params a Fleurcalculation node
   Returns Success, last result node, list with convergence behavior

banddos_after_scf()
   This method submits the BandDOS calculation after the initial SCF calculation

banddos_wo_scf()
   This method submits the BandDOS calculation without a previous SCF calculation

change_fleurinp()
   create a new fleurinp from the old with certain parameters

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.

classmethod define(spec)
   Define the specification of the process, including its inputs, outputs and known exit codes.
   A metadata input namespace is defined, with optional ports that are not stored in the database.

get_inputs_scf()
   Initialize inputs for scf workflow: wf_param, options, calculation parameters, codes, structure

return_results()
   return the results of the calculations

scf_needed()
   Returns True if SCF WC is needed.

start()
   check parameters, what condictions? complete? check input nodes

aiida_fleur.workflows.banddos.create_aiida_bands_data(fleurinp, retrieved)

Creates aiida.orm.BandsData object containing the kpoints and eigenvalues from the banddos.hdf file of the calculation

Parameters
   • fleurinp – FleurinpData for the calculation
   • retrieved – aiida.orm.FolderData for the bandstructure calculation
```
Returns `aiida.orm.BandsData` for the bandstructure calculation

Raises ExitCode 300, banddos.hdf file is missing

Raises ExitCode 310, banddos.hdf reading failed

Raises ExitCode 320, reading kpointsdata from Fleurinp failed

`aiida_fleur.workflows.banddos.create_aiida_dos_data(retrieved)`

Creates `aiida.orm.XyData` object containing the standard DOS components from the `banddos.hdf` file of the calculation

Parameters `retrieved` – `aiida.orm.FolderData` for the DOS calculation

Returns `aiida.orm.XyData` containing all standard DOS components

Raises ExitCode 300, banddos.hdf file is missing

Raises ExitCode 310, banddos.hdf reading failed

`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.

### 7.1.4.4 DOS: Density of states WorkChain

This is the workflow ‘dos’ for the Fleur code, which calculates a density of states (DOS).

```python
class aiida_fleur.workflows.dos.fleur_dos_wc(*args: Any, **kwargs: Any)
```

DEPRECATED: Use FleurBandDosWorkChain instead (entrypoint fleur.banddos) This workflow calculated a DOS from a Fleur calculation

Params a Fleur calculation 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

```python
create_new_fleurinp()
```

create a new fleurinp from the old with certain parameters

```python
class_method define(spec)
```

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

A metadata input namespace is defined, with optional ports that are not stored in the database.

```python
return_results()
```

return the results of the calculations

```python
run_fleur()
```

run a FLEUR calculation

```python
start()
```

check parameters, what condictions? complete? check input nodes
7.1.4.5 EOS: Calculate a lattice constant

In this module you find the workflow ‘FleurEosWorkChain’ for the calculation of of an equation of state

```python
class aiida_fleur.workflows.eos.FleurEosWorkChain(*args: Any, **kwargs: Any)
    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.
```

```python
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
```

```python
classmethod define(spec)
    Define the specification of the process, including its inputs, outputs and known exit codes.

    A metadata input namespace is defined, with optional ports that are not stored in the database.

get_inputs_scf()
    get and ‘produce’ the inputs for a scf-cycle

get_inputs_scf_first()
    get and ‘produce’ the inputs for a scf-cycle
```

```python
inspect_first()
    Check if the first calculation failed and

return_results()
    return the results of the calculations (scf workchains) and do a Birch-Murnaghan fit for the equation of states

run_first()
    Launch the first fleur SCF workchain
```

```python
start()
    check parameters, what condictions? complete? check input nodes

structures()
    Creates structure data nodes with different Volume (lattice constants)
```

```python
aiida_fleur.workflows.eos.birch_murnaghan(volumes, volume0, bulk_modulus0, bulk_deriv0)
    This evaluates the Birch Murnaghan equation of states
```
least squares fit of a Birch-Murnaghan equation of state curve. From delta project containing in its columns the volumes in \( \text{A}^3/\text{atom} \) 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 (numpay arrays!) of total energies eV/\text{atom} :params volumes: list (numpay arrays!) of volumes in \( \text{A}^3/\text{atom} \)

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

Returns dict of New StructureData nodes with rescaled structure, which are linked to input Structure

aiida_fleur.workflows.eos.eos_structures_nocf(inp_structure, scalelist)

Creates many rescaled StructureData nodes out of a crystal structure. Does NOT keep the provenance in the database.

Returns dict of New StructureData nodes with rescaled structure, key=scale

7.1.4.6 Relax: Relaxation of a Crystalstructure WorkChain

In this module you find the workflow ‘FleurRelaxWorkChain’ for geometry optimization.

class aiida_fleur.workflows.relax.FleurRelaxWorkChain(*args: Any, **kwargs: Any)

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.

classmethod define(spec)
Define the specification of the process, including its inputs, outputs and known exit codes.
A metadata input namespace is defined, with optional ports that are not stored in the database.

generate_new_fleurinp()
This function fetches relax.xml from the previous iteration and calls analyse_relax(). New FleurinpData is stored in the context.

generate_inputs_final_scf()
Initializes inputs for final scf on relaxed structure.

generate_inputs_first_scf()
Initializes inputs for the first iteration.

generate_inputs_scf()
Initializes inputs for further iterations.

generate_results_final_scf()
Parser some results of final scf

generate_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_new_scf()
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 parameters of the WorkChain, validate inputs

aiida_fleur.workflows.relax.create_relax_result_node(**kwargs)
This calc function 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 output nodes

7.1.4.7 initial_cls: Calculation of initial core-level shifts
This is the workflow FleurInitialCLSWorkChain ‘initial_cls’ using the Fleur code calculating core-level shifts with different methods.

class aiida_fleur.workflows.initial_cls.FleurInitialCLSWorkChain(*args: Any, **kwargs: Any)
Turn key solution for the calculation of core level shift

check_input()
Init same context and check what input is given if it makes sense
**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)**
Controlled way to shutdown the workchain. will initialize the output nodes The shutdown of the workchain will has to be done afterwards

**classmethod define(spec)**
Define the specification of the process, including its inputs, outputs and known exit codes.
A metadata input namespace is defined, with optional ports that are not stored in the database.

**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 Crystal structures. If some references 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.clshifts_to_be(coreleveldict, reference_dict)**
This methods converts corelevel shifts to binding energies, if a reference is given. These can then be used for plotting.

Example:

```python
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]}}
```
This is a pseudo wf, to create the right graph structure of AiiDA. This workflow 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 too lazy to put most of the code overworked from return_results in here.

This method finds StructureData nodes with the following extras: extra.type = ‘bulk’, # Should be done by looking at pbc, but I could not get query 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

7.1.4.8 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’

class aiida_fleur.workflows.corehole.FleurCoreholeWorkChain(*args: Any, **kwargs: Any)

Turn key solution for a corehole calculation with the FLEUR code. Has different protocols for different core-hole 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 inpgenre 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, inpgenre 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 unsuccessful 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)**
Controlled way to shutdown the workchain. report errors and always initialize/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

**classmethod define(spec)**
Define the specification of the process, including its inputs, outputs and known exit codes.

A metadata input namespace is defined, with optional ports that are not stored in the database.

**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.create_corehole_result_node(**kwargs)`
This is a pseudo wf, to create the right graph structure of AiiDA. This workfunction 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.

`aiida_fleur.workflows.corehole.extract_results_corehole(calcs)`
Collect results from certain calculation, check if everything is fine, calculate the wanted quantities.

```python
params: calcs : list of scf workchains nodes
```

`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 provenance is kept) wf_para: Dict node dict: {'site': sites[8], 'kindname': 'W1', 'econfig': "[Kr] 5s2 4d10 4f13 | 5p6 5d5 6s2", 'fleurinp_change': []}

### 7.1.4.9 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(*args: Any, **kwargs: Any)**
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.

**classmethod define(spec)**
Define the specification of the process, including its inputs, outputs and known exit codes.
A *metadata* input namespace is defined, with optional ports that are not stored in the database.

**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 parameters 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.

7.1.4.10 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(*args: Any, **kwargs: Any)
   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.

classmethod define(spec)
   Define the specification of the process, including its inputs, outputs and known exit codes.
   A metadata input namespace is defined, with optional ports that are not stored in the database.

get_inputs_scf(sqa)
   Initialize inputs for scf workflow

get_results()
   Retrieve results of converge calculations

return_results()
   Retrieve results of converge calculations

start()
   Retrieve and initialize parameters of the WorkChain

aiida_fleur.workflows.mae_conv.save_output_node(out)
   This calcfunction saves the out dict in the db
7.1.4.11 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-relativistic Hamiltonian.

```python
class aiida_fleur.workflows.ssdisp.FleurSSDispWorkChain(*args: Any, **kwargs: Any):
    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.

    classmethod define(spec)
    Define the specification of the process, including its inputs, outputs and known exit codes.
    
    A metadata input namespace is defined, with optional ports that are not stored in the database.

    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
```
7.1.4.12 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.

```python
class aiida_fleur.workflows.ssdisp_conv.FleurSSDispConvWorkChain(*args: Any, **kwargs: Any)
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.

classmethod define(spec)
Define the specification of the process, including its inputs, outputs and known exit codes.

A metadata input namespace is defined, with optional ports that are not stored in the database.

get_inputs_scf(qss)
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 parameters of the WorkChain

aiida_fleur.workflows.ssdisp_conv.save_output_node(out)
This calcfunction saves the out dict in the db
```

7.1.4.13 DMI: Force-theorem calculation of Dzialoshinskii-Moriya interaction energy dispersion

In this module you find the workflow `FleurDMIWorkChain` for the calculation of DMI energy dispersion.

```python
class aiida_fleur.workflows.dmi.FleurDMIWorkChain(*args: Any, **kwargs: Any)
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 further spin spiral calculations.

classmethod define(spec)
Define the specification of the process, including its inputs, outputs and known exit codes.

A metadata input namespace is defined, with optional ports that are not stored in the database.
```
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 parameters of the WorkChain

aiida_fleur.workflows.dmi.save_output_node(out)
This calc function saves the out dict in the db

7.1.4.14 OrbControl: Self-consistent calculation of groundstate density matrix with LDA+U

In this module you find the workflow ‘FleurOrbControlWorkChain’ for finding the groundstate in a DFT+U calculation.

class aiida_fleur.workflows.orbcontrol.FleurOrbControlWorkChain(*args: Any, **kwargs: Any)
Workchain for determining the groundstate density matrix in an DFT+U calculation. This is done in 2 or 3 steps:
1. Converge the system without DFT+U (a converged calculation can be provided to skip this step)
2. A fixed number of iterations is run with fixed density matrices either generated as all distinct permutations for the given occupations or the explicitly given configurations
3. The system and density matrix is relaxed

Parameters

- **wf_parameters** – (Dict), Workchain Specifications
- **scf_no_ldau** – (Dict), Inputs to a FleurScfWorkChain providing the initial system either converged or staring from a structure
- **scf_with_ldau** – (Dict), Inputs to a FleurScfWorkChain. Only the wf_parameters are valid
- **fleurinp** – (FleurinpData) FleurinpData to start from if no SCF should be done
- **remote** – (RemoteData) RemoteData to start from if no SCF should be done
- **structure** – (StructureData) Structure to start from if no SCF should be done
- **calc_parameters** – (Dict), Inpgen Parameters
- **settings** – (Dict), additional settings for e.g retrieving files
- **options** – (Dict), Options for the submission of the jobs
• **inpgen** – (Code)
• **fleur** – (Code)

**control_end_wc**(*errmsg*)
Controlled way to shutdown the workchain. It will initialize the output nodes. The shutdown of the workchain will have to be done afterwards.

**converge_scf**()
Launch fleur.scf after the fixed density matrix calculations to relax the density matrix.

**converge_scf_no_ldau**()
Launch fleur.scf for the system without LDA+U.

**create_configurations**()
Creates the configurations for the initial density matrices.
If `fixed_occupations` was provided, the density matrices are constructed as having the given occupations and constructing all distinct permutations.
If `fixed_configurations` was provided, only the given configurations are taken.

**classmethod define**(spec)
Define the specification of the process, including its inputs, outputs and known exit codes.
A `metadata` input namespace is defined, with optional ports that are not stored in the database.

**classmethod get_builder_continue_fixed**(node)
Get a Builder prepared with inputs to continue from the charge densities of a already finished MagRotate-WorkChain.

Parameters

`node` – Instance, from which the calculation should be continued.

**classmethod get_builder_continue_relaxed**(node, allow_nonconverged=True)
Get a Builder prepared with inputs to continue from the charge densities of a already finished MagRotate-WorkChain.

Parameters

`node` – Instance, from which the calculation should be continued.

**get_inputs_fixed_configurations**(index, config)
Sets up the input for the fixed density matrix calculation.

**get_inputs_scf**()
Get the input for the scf workchain after the fixed density matrix calculations to relax the density matrix.

**get_inputs_scf_no_ldau**()
Get the inputs for the scf workchain without LDA+U.

**inpgen_needed**()
Returns whether the inpgen should be run directly by this workchain.

**return_results**()
Return the results of the relaxed DFT+U calculations (scf workchains).

**run_fixed_calculations**()
Returns whether to run frozen density matrix calculations.

**run_fleur_fixed**()
Launches fleur.base with `l_linMix=T` and `mixParam=0.0`, i.e. with a fixed density matrix for all configurations.
run_inpgen()
Run the input generator

scf_no_ldau_needed()
Returns whether to run an additional scf workchain before adding LDA+U

start()
init context and some parameters

validate_input()
validate input

aiida_fleur.workflows.orbcontrol.create_orbcontrol_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 orbcontrol, connections to run scfs, and returning of the gs_calculation (best initial density matrix) 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.orbcontrol.extract_nmmp_file(folder)
Extract the density matrix file from the given folder data

Raises ExitCode 300, No density matrix file found

aiida_fleur.workflows.orbcontrol.generate_density_matrix_configurations(occupations=None, configurations=None)
Generate all the necessary density matrix configurations from either the occupations or the explicitly given configurations for each species/orbital

Both arguments are expected as dictionaries in the form d[species][orbital], with the orbital key holding the specification for the current LDA+U procedure

Parameters
• occupations – specifying the occupations for each procedure
• configurations – specifying a explicit list of configurations that should be calculated

Returns list of dictionaries with all the possible starting configurations for the whole system

7.1.4.15 CFCoeff: Calculation of 4f crystal field coefficients
In this module you find the workflow ‘FleurCFCoeffWorkChain’ for calculating the 4f crystal field coefficients

class aiida_fleur.workflows.cffcoff.FleurCFCoeffWorkChain(*args: Any, **kwargs: Any)
Workflow for calculating rare-earth crystal field coefficients

check_cf_calculation(calc_name)
Check that the CFCalculation finished successfully

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

classmethod define(spec)
Define the specification of the process, including its inputs, outputs and known exit codes.
A metadata input namespace is defined, with optional ports that are not stored in the database.
return_results()
    Return results for cf calculation
start()
    init context and some parameters
validate_input()
    validate input

aiida_fleur.workflows.cfcoeff.calculate_cf_coefficients(cf_cdn_folder: aiida.orm.nodes.data.folder.FolderData, cf_pot_folder: aiida.orm.nodes.data.folder.FolderData, convert: aiida.orm.nodes.data.bool.Bool = None, atomTypes: aiida.orm.nodes.data.list.List = None) → aiida.orm.nodes.data.dict.Dict

Calculate the crystal field coefficients using the tool from the masci-tools package

Parameters

• cf_cdn_folder – FolderData for the retrieved files for the charge density data
• cf_pot_folder – FolderData for the retrieved files for the potential data

Raises ExitCode 300, CFData.hdf file is missing
Raises ExitCode 310, CFdata.hdf reading failed
Raises ExitCode 320, Crystal field calculation failed

aiida_fleur.workflows.cfcoeff.create_cfcoeff_results_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 orbcontrol, connections to run scfs, and returning of the gs_calculation (best initial density matrix) 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.cfcoeff.reconstruct_cfcalculation(charge_densities, potentials, atomtype, **kwargs)

Reconstruct the CFCalculation instance from the outputs of the FleurCFCoeffWorkChain

aiida_fleur.workflows.cfcoeff.reconstruct_cfcoeffcients(output_dict, atomtype=None)

Reconstruct the CFCoefficient list from the output dictionary of the FleurCFCoeffWorkChain

Parameters

• output_dict – output dictionary node or the corresponding dictionary
• atomtype – int of the atomtype to reconstruct the coefficients for
7.1.5 Commandline interface (CLI)

7.1.5.1 aiida-fleur

CLI for the aiida-fleur plugin.

```
aiida-fleur [OPTIONS] COMMAND [ARGS]...
```

Options

- **-p, --profile** <profile>
  Execute the command for this profile instead of the default profile.

- **-v, --version**
  Show the version and exit.

**data**

Commands to create and inspect data nodes.

```
aiida-fleur data [OPTIONS] COMMAND [ARGS]...
```

**fleurinp**

Commands to handle FleurinpData nodes.

```
aiida-fleur data fleurinp [OPTIONS] COMMAND [ARGS]...
```

**cat**

Dumb the content of a file contained in given fleurinpdata, per default dump inp.xml

```
aiida-fleur data fleurinp cat [OPTIONS] NODE
```

Options

- **-f, --filename** <filename>
  Display the file content of the given filename.
  Default inp.xml
Arguments

NODE
   Required argument

extract-inpgen

Write out a inpgen input file, that most closely reproduces the input file in the node when run through the inpgen

```
aiida-fleur data fleurinp extract-inpgen [OPTIONS] NODE
```

Options

-o, --output-filename <output_filename>
   Name of the file to write out.
   
   Default

--para, --no-para
   Add additional LAPW parameters to output file
   
   Default True

Arguments

NODE
   Required argument

list

List stored FleurinpData in the database with additional information

```
aiida-fleur data fleurinp list [OPTIONS] [--]
```

Options

-G, --groups <groups>
   One or multiple groups identified by their ID, UUID or label.

-p, --past-days <PAST_DAYS>
   Only include entries created in the last PAST_DAYS number of days.

-A, --all-users
   Include all entries regardless of the owner.

-r, --raw
   Display only raw query results, without any headers or footers.
--uuid, --no-uuid
    Display uuid of nodes.
    Default False

--ctime, --no-ctime
    Display ctime of nodes.
    Default False

--extras, --no-extras
    Display extras of nodes.
    Default True

--strucinfo, --no-strucinfo
    Prepare additional information on the crystal structure to show. This slows down the query.
    Default False

open

opens the inp.xml in some editor, readonly. inp.xml this way looking at xml might be more convenient.

aiida-fleur data fleurinp open [OPTIONS] NODE

Options

-f, --filename <filename>
    Open the file of the given filename.
    Default inp.xml

-s, --save
    Write out the changed content

-o, --output-filename <output_filename>
    Filename of the output
    Default

Arguments

NODE
    Required argument
options

Commands to create and inspect `Dict` nodes containing options.

```
aiida-fleur data options [OPTIONS] COMMAND [ARGS]...
```

create

Command to create options dict nodes

```
aiida-fleur data options create [OPTIONS]
```

Options

- \(-N, --max-num-machines\) `<max_num_machines>`
  
The maximum number of machines (nodes) to use for the calculations.
  
  Default 1

- \(-M, --num-mpiprocs-per-machine\) `<num_mpiprocs_per_machine>`
  
  Run the simulation with so many num-mpi-procs-per-machine.
  
  Default 2

- \(-W, --max-wallclock-seconds\) `<max_wallclock_seconds>`
  
  The maximum wallclock time in seconds to set for the calculations.
  
  Default 1800

- \(-q, --queue\) `<queue>`
  
  The queue name to submit to.
  
  Default

- \(-n, --dry-run\)
  
  Perform a dry run.

- \(--show, --no-show\)
  
  Print the contents from the options dict.
  
  Default True

parameter

Commands to create and inspect `Dict` nodes containing FLAPW parameters (`calc_parameters`).

```
aiida-fleur data parameter [OPTIONS] COMMAND [ARGS]...
```
import

Extract FLAPW parameters from a Fleur input file and store as Dict in the db.
FILENAME is the name/path of the inp.xml file to use.

```
aiida-fleur data parameter import [OPTIONS] FILENAME
```

Options

--fleurinp, --no-fleurinp
Store also the fleurinp and the extractor calcfunction in the db.

Default False

--show, --no-show
Print the contents from the extracted dict.

Default True

-n, --dry-run
Perform a dry run.

Arguments

FILENAME
Required argument

structure

Commands to create and inspect StructureData nodes.

```
aiida-fleur data structure [OPTIONS] COMMAND [ARGS]...
```

import

Import a StructureData from a Fleur input file.
FILENAME is the name/path of the inp.xml file to use.
If you want to import a structure from any file type you can use ‘verdi data structure import -ase <filename>’ instead.

```
aiida-fleur data structure import [OPTIONS] FILENAME
```
Options

--fleurinp, --no-fleurinp
Store also the fleurinp and the extractor calcfunction in the db.

  Default False

-n, --dry-run
Perform a dry run.

Arguments

FILENAME
Required argument

launch

Commands to launch workflows and calcjobs of aiida-fleur.

```
aiida-fleur launch [OPTIONS] COMMAND [ARGS]...
```

banddos

Launch a banddos workchain

```
aiida-fleur launch banddos [OPTIONS]
```

Options

--fleurinp <fleurinp>
FleurinpData node for the fleur calculation.

-f, --fleur <fleur>
A code node or label for a fleur executable.

  Default <function get_fleur at 0x7f396cfb6e60>

-wf, --wf-parameters <wf_parameters>
Dict containing parameters given to the workchain.

-P, --parent-folder <parent_folder>
The PK of a parent remote folder (for restarts).

-d, --daemon
Submit the process to the daemon instead of running it locally.

  Default False

-set, --settings <settings>
Settings node for the calcjob.

-opt, --option-node <option_node>
Dict, an option node for the workchain.
**corehole**

Launch a corehole workchain

```bash
aiida-fleur launch corehole [OPTIONS]
```

**Options**

-`-s, --structure <structure>`
  StructureData node, given by pk or uuid or file in any for mat which will be converted.
  
  **Default** `<function get_si_bulk_structure at 0x7f396cfb6cb0>`

-`-i, --inpgen <inpgen>`
  A code node or label for an inpgen executable.
  
  **Default** `<function get_inpgen at 0x7f396cfb6dd0>`

-`-calc_p, --calc-parameters <calc_parameters>`
  Dict with calculation (FLAPW) parameters to build, which will be given to inpgen.

-`-inp, --fleurinp <fleurinp>`
  FleurinpData node for the fleur calculation.

-`-f, --fleur <fleur>`
  A code node or label for a fleur executable.
  
  **Default** `<function get_fleur at 0x7f396cfb6e60>`

-`-wf, --wf-parameters <wf_parameters>`
  Dict containing parameters given to the workchain.

-`-d, --daemon`
  Submit the process to the daemon instead of running it locally.
  
  **Default** False

-`-set, --settings <settings>`
  Settings node for the calcjob.

-`-opt, --option-node <option_node>`
  Dict, an option node for the workchain.

**create_magnetic**

Launch a create_magnetic workchain

```bash
aiida-fleur launch create_magnetic [OPTIONS]
```
Options

-i, --inpgen <inpgen>
A code node or label for an inpgen executable.

    Default <function get_inpgen at 0x7f396cfb6dd0>

-calc_p, --calc-parameters <calc_parameters>
Dict with calculation (FLAPW) parameters to build, which will be given to inpgen.

-f, --fleur <fleur>
A code node or label for a fleur executable.

    Default <function get_fleur at 0x7f396cfb6e60>

-wf, --wf-parameters <wf_parameters>
    Required Dict containing parameters given to the workchain.

eos, --eos-parameters <eos_parameters>
Dict containing wf parameters given to the sub EOS workchains.

-scf, --scf-parameters <scf_parameters>
Dict containing parameters given to the sub SCF workchains.

-relax, --relax-parameters <relax_parameters>
Dict containing wf parameters given to the sub relax workchains.

-d, --daemon
Submit the process to the daemon instead of running it locally.

    Default False

-opt, --option-node <option_node>
Dict, an option node for the workchain.

dmi
Launch a dmi workchain

aiida-fleur launch dmi [OPTIONS]

Options

-s, --structure <structure>
StructureData node, given by pk or uuid or file in any for mat which will be converted.

    Default <function get_fept_film_structure at 0x7f396cfb6d40>

-i, --inpgen <inpgen>
A code node or label for an inpgen executable.

    Default <function get_inpgen at 0x7f396cfb6dd0>

-calc_p, --calc-parameters <calc_parameters>
Dict with calculation (FLAPW) parameters to build, which will be given to inpgen.
-f, --fleur <fleur>
A code node or label for a fleur executable.
  Default <function get_fleur at 0x7f396cfb6e60>
-wf, --wf-parameters <wf_parameters>
  Required Dict containing parameters given to the workchain.
-scf, --scf-parameters <scf_parameters>
  Dict containing parameters given to the sub SCF workchains.
-d, --daemon
  Submit the process to the daemon instead of running it locally.
    Default False
-opt, --option-node <option_node>
  Dict, an option node for the workchain.

eos
Launch a eos workchain

aiida-fleur launch eos [OPTIONS]

Options
-s, --structure <structure>
  StructureData node, given by pk or uuid or file in any for mat which will be converted.
  Default <function get_si_bulk_structure at 0x7f396cfb6cb0>
-i, --inpgen <inpgen>
  A code node or label for an inpgen executable.
  Default <function get_inpgen at 0x7f396cfb6dd0>
-calc_p, --calc-parameters <calc_parameters>
  Dict with calculation (FLAPW) parameters to build, which will be given to inpgen.
-f, --fleur <fleur>
  A code node or label for a fleur executable.
  Default <function get_fleur at 0x7f396cfb6e60>
-wf, --wf-parameters <wf_parameters>
  Dict containing parameters given to the workchain.
-scf, --scf-parameters <scf_parameters>
  Dict containing parameters given to the sub SCF workchains.
-d, --daemon
  Submit the process to the daemon instead of running it locally.
    Default False


- **set**, **--settings** `<settings>`
  Settings node for the calcjob.

- **opt**, **--option-node** `<option_node>`
  Dict, an option node for the workchain.

### fleur

Launch a base_fleur workchain. If launch_base is False launch a single fleur calcjob instead.

```plaintext
aiida-fleur launch fleur [OPTIONS]
```

### Options

- **inp**, **--fleurinp** `<fleurinp>`
  FleurinpData node for the fleur calculation.

- **f**, **--fleur** `<fleur>`
  A code node or label for a fleur executable.

  Default `<function get_fleur at 0x7f396cfb6e60>`

- **P**, **--parent-folder** `<parent_folder>`
  The PK of a parent remote folder (for restarts).

- **set**, **--settings** `<settings>`
  Settings node for the calcjob.

- **d**, **--daemon**
  Submit the process to the daemon instead of running it locally.

  Default False

- **N**, **--max-num-machines** `<max_num_machines>`
  The maximum number of machines (nodes) to use for the calculations.

  Default 1

- **W**, **--max-wallclock-seconds** `<max_wallclock_seconds>`
  The maximum wallclock time in seconds to set for the calculations.

  Default 1800

- **M**, **--num-mpi-procs-per-machine** `<num_mpiprocs_per_machine>`
  Run the simulation with so many num-mpi-procs-per-machine.

  Default 2

- **opt**, **--option-node** `<option_node>`
  Dict, an option node for the workchain.

- **I**, **--with-mpi**
  Run the calculations with MPI enabled.

  Default False
-q, --queue <queue>
   The queue name to submit to.
   
   Default

--launch_base, --no-launch_base
   Run the base_fleur workchain, which also handles errors instead of a single fleur calcjob.
   
   Default True

**init_cls**

Launch an init_cls workchain

```bash
aiida-fleur launch init_cls [OPTIONS]
```

**Options**

-s, --structure <structure>
   StructureData node, given by pk or uuid or file in any for mat which will be converted.
   
   Default <function get_si_bulk_structure at 0x7f396cfb6cb0>

-i, --inpgen <inpgen>
   A code node or label for an inpgen executable.
   
   Default <function get_inpgen at 0x7f396cfb6dd0>

-calc_p, --calc-parameters <calc_parameters>
   Dict with calculation (FLAPW) parameters to build, which will be given to inpgen.

-inp, --fleurinp <fleurinp>
   FleurinpData node for the fleur calculation.

-f, --fleur <fleur>
   A code node or label for a fleur executable.
   
   Default <function get_fleur at 0x7f396cfb6e60>

-wf, --wf-parameters <wf_parameters>
   Dict containing parameters given to the workchain.

-d, --daemon
   Submit the process to the daemon instead of running it locally.
   
   Default False

-set, --settings <settings>
   Settings node for the calcjob.

-opt, --option-node <option_node>
   Dict, an option node for the workchain.
**inpgen**

Launch an inpgen calcjob on given input

If no code is given it queries the DB for inpgen codes and uses the one with the newest creation time.

Either structure or anysource_structure can be specified. Default structure is Si bulk.

```
aiida-fleur launch inpgen [OPTIONS]
```

**Options**

- `s`, `--structure <structure>`
  StructureData node, given by pk or uuid or file in any format which will be converted.

  **Default** `<function get_si_bulk_structure at 0x7f396cfb6cb0>`

- `i`, `--inpgen <inpgen>`
  A code node or label for an inpgen executable.

  **Default** `<function get_inpgen at 0x7f396cfb6dd0>`

- `calc_p`, `--calc-parameters <calc_parameters>`
  Dict with calculation (FLAPW) parameters to build, which will be given to inpgen.

- `set`, `--settings <settings>`
  Settings node for the calcjob.

- `d`, `--daemon`
  Submit the process to the daemon instead of running it locally.

  **Default** False

- `opt`, `--option-node <option_node>`
  Dict, an option node for the workchain.

- `q`, `--queue <queue>`
  The queue name to submit to.

  **Default**

**mae**

Launch a mae workchain

```
aiida-fleur launch mae [OPTIONS]
```
Options

- **--structure** <structure>
  StructureData node, given by pk or uuid or file in any for mat which will be converted.

  Default: `<function get_fept_film_structure at 0x7f396cfb6d40>`

- **--inpgen** <inpgen>
  A code node or label for an inpgen executable.

  Default: `<function get_inpgen at 0x7f396cfb6dd0>`

- **--calc-parameters** <calc_parameters>
  Dict with calculation (FLAPW) parameters to build, which will be given to inpgen.

- **--fleurinp** <fleurinp>
  FleurinpData node for the fleur calculation.

- **--fleur** <fleur>
  A code node or label for a fleur executable.

  Default: `<function get_fleur at 0x7f396cfb6e60>`

- **--wf-parameters** <wf_parameters>
  Dict containing parameters given to the workchain.

- **--scf-parameters** <scf_parameters>
  Dict containing parameters given to the sub SCF workchains.

- **--parent-folder** <parent_folder>
  The PK of a parent remote folder (for restarts).

- **--daemon**
  Submit the process to the daemon instead of running it locally.

  Default: False

- **--settings** <settings>
  Settings node for the calcjob.

- **--option-node** <option_node>
  Dict, an option node for the workchain.

relax

Launch a base relax workchain

# TODO final scf input

```bash
aiida-fleur launch relax [OPTIONS]
```
Options

- **s**, **--structure** **<structure>**
  StructureData node, given by pk or uuid or file in any for mat which will be converted.
  Default <function get_si_bulk_structure at 0x7f396cfb6cb0>

- **i**, **--inpgen** **<inpgen>**
  A code node or label for an inpgen executable.
  Default <function get_inpgen at 0x7f396cfb6dd0>

- **-calc_p, --calc-parameters** **<calc_parameters>**
  Dict with calculation (FLAPW) parameters to build, which will be given to inpgen.

- **-f**, **--fleur** **<fleur>**
  A code node or label for a fleur executable.
  Default <function get_fleur at 0x7f396cfb6e60>

- **-wf, --wf-parameters** **<wf_parameters>**
  Dict containing parameters given to the workchain.

- **-scf, --scf-parameters** **<scf_parameters>**
  Dict containing parameters given to the sub SCF workchains.

- **-d**, **--daemon**
  Submit the process to the daemon instead of running it locally.
  Default False

- **-set, --settings** **<settings>**
  Settings node for the calcjob.

- **-opt, --option-node** **<option_node>**
  Dict, an option node for the workchain.

**scf**

Launch a scf workchain

```
aiida-fleur launch scf [OPTIONS]
```

Options

- **s**, **--structure** **<structure>**
  StructureData node, given by pk or uuid or file in any for mat which will be converted.
  Default <function get_si_bulk_structure at 0x7f396cfb6cb0>

- **i**, **--inpgen** **<inpgen>**
  A code node or label for an inpgen executable.
  Default <function get_inpgen at 0x7f396cfb6dd0>

- **-calc_p, --calc-parameters** **<calc_parameters>**
  Dict with calculation (FLAPW) parameters to build, which will be given to inpgen.
-set, --settings <settings>
Settings node for the calcjob.

-inp, --fleurinp <fleurinp>
FleurinpData node for the fleur calculation.

-f, --fleur <fleur>
A code node or label for a fleur executable.

 Default <function get_fleur at 0x7f396cfb6e60>

-wf, --wf-parameters <wf_parameters>
Dict containing parameters given to the workchain.

-P, --parent-folder <parent_folder>
The PK of a parent remote folder (for restarts).

-d, --daemon
Submit the process to the daemon instead of running it locally.

 Default False

-set, --settings <settings>
Settings node for the calcjob.

-opt, --option-node <option_node>
Dict, an option node for the workchain.

**ssdisp**

Launch a ssdisp workchain

```
aiida-fleur launch ssdisp [OPTIONS]
```

**Options**

-s, --structure <structure>
StructureData node, given by pk or uuid or file in any for mat which will be converted.

 Default <function get_fept_film_structure at 0x7f396cfb6d40>

-i, --inpgen <inpgen>
A code node or label for an inpgen executable.

 Default <function get_inpgen at 0x7f396cfb6dd0>

-calc_p, --calc-parameters <calc_parameters>
Dict with calculation (FLAPW) parameters to build, which will be given to inpgen.

-f, --fleur <fleur>
A code node or label for a fleur executable.

 Default <function get_fleur at 0x7f396cfb6e60>

-wf, --wf-parameters <wf_parameters>
Required Dict containing parameters given to the workchain.
-scf, --scf-parameters <scf_parameters>
Dict containing parameters given to the sub SCF workchains.

-d, --daemon
Submit the process to the daemon instead of running it locally.
   Default False

-opt, --option-node <option_node>
Dict, an option node for the workchain.

plot

Invoke the plot_fleur command on given nodes

```
aiida-fleur plot [OPTIONS] [NODES]...
```

Options

-\( f \) <filename>

--save <save>
   Should the result of plot_fleur be saved to a files.
   Default False

--show, --no-show
   Show the output of plot_fleur.
   Default True

--show_dict, --no-show_dict
   Show the output of plot_fleur.
   Default False

--bokeh

--matplotlib

Arguments

NODES
   Optional argument(s)
**workflow**

Commands to inspect aiida-fleur workchains.

```
aiida-fleur workflow [OPTIONS] COMMAND [ARGS]...
```

**inputdict**

Print data from Dict nodes input into any fleur process.

```
aiida-fleur workflow inputdict [OPTIONS] [--] PROCESS
```

**Options**

```
--info, --no-info
    Print an info header above each node.
-l, --label <label>
    Print only output dicts with a certain link_label.
--show, --no-show
    Show the main output of the command.
    Default True
-k, --keys <keys>
    Filter the output by one or more keys.
-f, --format <fmt>
    The format of the output data.
    Options json+date | yaml | yaml_expanded
```

**Arguments**

```
PROCESS
    Required argument
```

**res**

Print data from Dict nodes returned or created by any fleur process.

```
aiida-fleur workflow res [OPTIONS] [--] PROCESS
```
## Options

--info, --no-info  
Print an info header above each node.

-l, --label <label>  
Print only output dicts with a certain link_label.

--show, --no-show  
Show the main output of the command.

  Default True

-k, --keys <keys>  
Filter the output by one or more keys.

-f, --format <fmt>  
The format of the output data.

  Options json+date | yaml | yaml_expanded

## Arguments

PROCESS  
Required argument

### 7.1.6 Fleur tools/utility

#### 7.1.6.1 Structure Data util

Collection of utility routines dealing with StructureData objects

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,  
  last_layer_factor=0.85,  
  first_layer_factor=0.85)
Tries to optimize interlayer distances. Can be used before RelaxWC to improve its behaviour. Works only for films having no z-reflection symmetry, for other films check out the adjust_sym_film_relaxation

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.
- **last_layer_factor** – a float factor to which interlayer distance between last and second last layers is multiplied
- **first_layer_factor** – a float factor to which interlayer distance between first and second layers is multiplied

```python
aiida_fleur.tools.StructureData_util.adjust_sym_film_relaxation(structure, suggestion, scale_as=None, bond_length=None, last_layer_factor=0.85, ILD=None)
```

Tries to optimize interlayer distances. Can be used before RelaxWC to improve its behaviour. Works only for films having z-reflection symmetry, for other films check out the adjust_film_relaxation

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.
- **last_layer_factor** – a float factor to which interlayer distance between last and second last layers is multiplied

```python
aiida_fleur.tools.StructureData_util.adjust_film_relaxation(structure, suggestion, scale_as=None, bond_length=None, last_layer_factor=0.85, ILD=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 careful 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 careful 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, 
directions=None, host_symbol=’Fe’, 
latticeconstant=4.0, size=(1, 1, 5), 
replacements=None, decimals=8, 
pop_last_layers=0)

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 planes forming the primitive unit cell
• directions – 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.
• replacements – a dict of type [INT: STRING], where INT is the layer number to be replaced (counting from lowest z-coordinate layers, INT=1 for the first layer INT=-1 for the last one) and STRING is the element name.
• decimals – sets the rounding of atom positions. See numpy.around.
• pop_last_layers – specifies how many layers to remove. Sometimes one does not want to use the integer number of unit cells along z, extra layers can be removed. Layers are removed in order from highest to lowest z-coordinate.

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

7.1. Source code Documentation (API reference) 209
`aiida_fleur.tools.StructureData_util.define_AFM_structures`:

```python
define_AFM_structures(structure, lattice, directions, host_symbol, replacements, latticeconstant, size, decimals=8, pop_last_layers=0, AFM_name='FM', magnetic_layers=1, sym_film=False)
```

Create

`aiida_fleur.tools.StructureData_util.find_equi_atoms(structure)`

This routine uses spglib and ASE to provide informations of all equivalence 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’: numbequiatomstypes}

`aiida_fleur.tools.StructureData_util.find_primitive_cell(structure)`

uses spglib find_primitive to find the primitive cell

**Parameters**
- `structure` – 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

**Parameters**
- `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

**Parameters**
- `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_atomtype_site_symmetry(struct)`

Get the local site symmetry symbols for each atomtype

Uses pymatgen SpaceGroupAnalyzer

**Parameters**
- `struct` – StructureData to analyse

**Returns**
- list of the site symmetry symbols for each atomtype (In the order they appear in the StructureData)

`aiida_fleur.tools.StructureData_util.get_layers(structure, z_coordinate_window=8)`

Extracts atom positions and their types belonging to the same layer Removes any information related to kind specie.

**Parameters**
- `structure` – ase lattice or StructureData which represents a slab
- `number` – the layer number. Note, that layers will be sorted according to z-position
- `z_coordinate_window` – sets the maximal difference between 2 atoms that will be considered in the same layer. It is an integer, which sets how z-coordinates will be rounded. For instance, z_coordinate_window = 2 means that first z-coordinates will be rounded up to 2 digits after the dot and then grouped.

**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.has_z_reflection(structure)

Checks if a structure has z-reflection symmetry

aiida_fleur.tools.StructureData_util.is_primitive(structure)

Checks if a structure is primitive or not.

aiida_fleur.tools.StructureData_util.is_structure(structure)

Test if the given input is a StructureData node, by object, id, or pk

aiida_fleur.tools.StructureData_util.magnetic_slab_from_relaxed(relaxed_structure, orig_structure, total_number_layers, num_relaxed_layers, z_coordinate_window=3, shift=(0, 0))

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. interlayer 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 period 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.mark_atoms(structure, condition, kind_id=’999999’)

Marks atom where sites fullfill the given condition with a given id The resulting kind name for these atoms is element-kind_id

condition is a callable taking the site and kind as arguments

aiida_fleur.tools.StructureData_util.mark_fixed_atoms(structure, hold_layers=None)

Marks atom in layers, that should be fixed in the relaxation. Uses reserved 49999 label
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.replace_element(inp_structure, replace_dict, replace_all=None)

Replaces the given element with the element_replacement, but keeps the structure the same. If there are more than onesitetheyareeitherallreplacedoralistwithonereplacementatatimeisreturned. Keeps the provenance in the database.

**Parameters**
- `inp_structure` – a StructureData node (pk, or uuid)
- `replace_dict` – Dict of elementstoreplace. Replacement is done according to the symbols
- `replace_all` – bool determines whether to replace all occurrences of the element at once Otherwise a list, with one occurrence replaced at a time

**Returns** Dict with new StructureData nodes with replaced elements, which is/are linked to input Structure and None if inp_structure was not a StructureData

**Example usage:** This example replaces all Neodymium atoms with Yttrium replace_element(structure, Dict(dict={'Nd':'Y'}), replace_all=Bool(True))

aiida_fleur.tools.StructureData_util.replace_elementf(inp_structure, replace_dict, replace_all)

Replaces the site according to replace_dict (symbols), but keeps the structure the same. If there are more than one site they are either all replaced or a list with one replacement at a time is returned. DOES NOT keep the provenance in the database.

**Parameters**
- `inp_structure` – a StructureData node (pk, or uuid)
- `replace_dict` – Dict of elementstoreplace. Replacement is done according to the symbols
- `replace_all` – bool determines whether to replace all occurrences of the element at once Otherwise a list, with one occurrence replaced at a time

**Returns** New StructureData node or list of new StructureData nodes with replaced elements, which is/are linked to input Structure and None if inp_structure was not a StructureData

aiida_fleur.tools.StructureData_util.request_average_bond_length(first_bin, second_bin, user_api_key, ignore_second_bin=False)
Requests MaterialsProject to estimate thermal average bond length between given elements. Also requests information about lattice constants of fcc and bcc structures.

**Parameters**

- **first_bin** – element list to calculate the average bond length only combinations of AB are calculated, where A belongs to first_bin, B belongs to second_bin.
- **second_bin** – element list, see main_elements
- **user_api_key** – user API key from materialsproject
- **ignore_second_bin** – if True, the second bin is ignored and all possible combinations from the first one are constructed.

**Returns** bond_data, a dict containing obtained lattice constants.

```python
aiida_fleur.tools.StructureData_util.request_average_bond_length_store(first_bin, second_bin, user_api_key, ignore_second_bin=False)
```

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**

- **first_bin** – element list to calculate the average bond length only combinations of AB, AA and BB are calculated, where A belongs to first_bin, B belongs to second_bin.
- **second_bin** – element list, see main_elements
- **user_api_key** – user API key from materialsproject
- **ignore_second_bin** – if True, the second bin is ignored and all possible combinations from the first one are constructed.

**Returns** bond_data, a dict containing obtained lattice constants.

```python
aiida_fleur.tools.StructureData_util.request_average_bond_length_store(first_bin, second_bin, user_api_key, ignore_second_bin=False)
```

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

```python
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. 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

```python
aiida_fleur.tools.StructureData_util.rescale_nowf(inp_structure, scale)
```
simplify_kind_name(kind_name)
Simplifies the kind name string. Example: “W-1” -> “W”, “Iron (Fe)” -> “Fe”

sort_atoms_z_value(structure)
Resorts the atoms in a structure by their Z-value

Parameters
- **structure** – AiiDA structure

Returns
- AiiDA structure

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

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

## 7.1.6.2 XML utility

This module defines XML modifying functions, that require an aiida node as input

set_kpointsdata_f(xmltree: Union[lxml.etree._Element, lxml.etree._ElementTree],
schema_dict:
masci_tools.io.parsers.fleur_schema.schema_dict.InputSchemaDict,
kpointsdata_uuid: aiida.orm.nodes.data.array.kpoints.KpointsData
| int | str, name: Optional[str] = None,
switch: bool = False, kpoint_type:
Literal['path', 'mesh', 'tria', 'tria-bulk', 'spex-mesh'] = 'path') →
Union[lxml.etree._Element,
lxml.etree._ElementTree]

This function creates a kpoint list in the inp.xml from a KpointsData Node If no weights are given the weight is distributed equally along the kpoints

Parameters
- **xmltree** – an xmltree that represents inp.xml
- **schema_dict** – InputSchemaDict containing all information about the structure of the input
- **kpointsdata_uuid** – node identifier or KpointsData node to be written into inp.xml
- **name** – str name to give the newly entered kpoint list (only MaX5 or later)
- **switch** – bool if True the entered kpoint list will be used directly (only Max5 or later)
• **kpoint_type** – str of the type of kpoint list given (mesh, path, etc.) only Max5 or later

**Returns** xmltree with entered kpoint list

### 7.1.6.3 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.clear_dict_empty_lists(to_clear_dict)

Removes entries from a nested dictionary which are empty lists.

**Parameters**

- **to_clear_dict** dict: python dictionary which should be ‘compressed’ return new dict dict: compressed python dict version of to_clear_dict

**Hints:** recursive

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

\[
\text{aiida_fleur.tools.merge_parameter.merge_parameter}(\text{Dict1, Dict2, overwrite=True, merge=True})
\]

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

- \text{Dict1} – AiiDA Dict Node
- \text{Dict2} – AiiDA Dict Node
- \text{overwrite} – bool, default True
- \text{merge} – bool, default True

returns: AiiDA Dict Node

#TODO be more careful how to merge ids in atom namelists, i.e species labels

\[
\text{aiida_fleur.tools.merge_parameter.merge_parameter_cf}(\text{Dict1, Dict2, overwrite=None})
\]

calcfunction of merge_parameters

\[
\text{aiida_fleur.tools.merge_parameter.merge_parameters}(\text{DictList, overwrite=True})
\]

Merge together all parameter nodes in the given list.

7.1.6.4 Corehole/level utility

Contains helper functions to create core-holes in Fleur input files from AiiDA data nodes.

\[
\text{aiida_fleur.tools.create_corehole.create_corehole_para}(\text{structure, kind, econfig, species_name='corehole', parameterdata=None})
\]

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

- \text{structure} – StructureData
- \text{kind} – a string with the kind_name (TODO: alternative the kind object)
- \text{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.

\[
\text{aiida_fleur.tools.extract_corelevels.clshifts_to_be}(\text{coreleveldict, reference_dict, warn=False})
\]

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:

\[
\text{reference_dict} = \{'W' : \{'4f7/2' : [124], '4f5/2' : [102]},
\text{'Be' : \{'ls' : [117]}}\}
\]
AiiDa-FLEUR Documentation, Release 2.0.0

Params coreleveldict  An example:

```python
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 conversion worked or not.

Parameters  
value_string – a string  

Returns  
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:

```plaintext
corestates = [{atomtype : '', corestates : list_of_corestates},
              {atomtype : '1', corestates : list_of_corestates},
              ...
              {atomtype : 'n', corestates : list_of_corestates}]
```

corestates numbers and their attributes:

- `j`: orbital angular momentum quantum number
- `l`: orbital angular momentum quantum number
- `n`: principal quantum number
- `weight`: weight

Example of output

```plaintext
[{{'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

7.1. Source code Documentation (API reference)
**Params** iteration node an etree element, iteration node

**Params** jspin integer 1 or 2

**Returns** a python dict of type:

```python
dict = {
    '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.

```python
daia_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.

```python
aiida_fleur.tools.element_econfig_list.econfigstr_hole(econfigstr, corelevel, highestunoccp, htype='valence')
```

# ‘1s2 | 2s2’ , ‘1s2’ , ‘2p0’ -> ‘1s1 | 2s2 2p1’

**Param** string

**Param** string

**Param** string

**Returns** string

```python
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...

```python
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

```python
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’

```python
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 string ‘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’

### 7.1.6.5 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 aida-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
- **nodes**: list of AiiDA nodes, pks, or uuids
- **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
- **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 uid of a nodes in a given group. Since 1.1.0, this function does !!! Now aida-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.
7.1.6.6 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.

```python
aiida_fleur.tools.read_cif_folder.read_cif_folder(path='~/home/docs/checkouts/readthedocs.org/user_builds/aiida-fleur/checkouts/develop/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)
- **recursive**: bool, If True: looks also in subfolders, if False: just given dir
- **store**: bool, if True: stores structures in database
- **log**: bool, if True, writes a logfile with information (pks, and co)
- **comments**: string: comment to add to the structures
- **extras**: dir/string/arb: extras added to the structures stored in the db

7.1.6.7 IO routines

Here we collect IO routines and their utility, for writing certain things to files, or post process files. For example collection of data or database evaluations, for other people.

7.1.6.8 Common utility 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.

```python
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

```python
aiida_fleur.tools.common_fleur_wf.calc_time_cost_function_total(natom, nkpt, kmax, niter, nspins=1)
```

Estimates the cost of simulating an all iteration of a system

```python
aiida_fleur.tools.common_fleur_wf.cost_ratio(total_costs, walltime_sec, ncores)
```

Estimates if simulation cost matches resources

```python
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.find_nested_process(wc_node, p_class)

This function finds all nested child processes of p_class

aiida_fleur.tools.common_fleur_wf.get_inputs_fleur(code, remote, fleurinp, options, label='', description='', settings=None, add_comp_para=None)

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
- **add_comp_para** – dict with extra keys controlling the behaviour of the parallelization of the FleurBaseWorkChain

Example of use:

```python
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.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,
        forbid_single_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 sacrif-
cice_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.
- **only_even_MPI** – if set to True, the function does not set MPI to an odd number (if possible)
- **forbid_single_mpi** – if set to True, the configuration 1 node 1 MPI per node will be
  forbidden

**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)

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

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.

**Parameters**

- **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])

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 package implementing the qhull algo for our purposes. For now only for 2D phase diagrams

**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. $E_{\text{form}} = E(A_xB_y) - x*E(A) - y*E(B)$

**Params**
- **struc_te_dict**: python dictionary in the form of {'formula': total_energy} for the compound(s)
- **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)
- **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.

**Params**
- **reaction**: string
- **formenergydict**: dictionary that contains the {compound: formation energy 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**
- **inpgendict**: python dict, used for inpgen parameter data node
- **mesh**: either as returned by kpointsdata or tuple of 3 integers

**Returns** python dict, used for inpgen parameter data 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])
8.1 Reference

8.1.1 Changelog

8.1.1.1 v.2.0.0

First release with official support for AiiDA version 2.0. Support for AiiDA 1.X is only available with releases from the 1.X series of aiida-fleur. Dropped python 3.7 support. Added support for python 3.11.

Breaking changes

- The entries last_calc_uuid from output dictionary of FleurSCFWorkChain and last_scf_wc_uuid from FleurRelaxWorkChain are removed. Reasoning for this is that having UUIDs in the output dictionary makes it impossible to take advantage of AiiDA's caching mechanism. Both workchains expose the relevant outputs of the under the namespaces last_calc and last_scf respectively

- Several input/output port changes:
  - FleurBandDOSworkChain: Removed last_calc_retrieved, replaced with namespace banddos_calc
  - FleurBaseWorkChain: Removed final_calc_uuid
  - Adjusted name of output dictionary to the naming schema output_<wc_abbrev>_wc_para: FleurDMIWorkChain, FleurMAEConvWorkChain, FleurSSDispWorkChain, FleurSSDicpConvWorkChain
  - FleurSCFWorkChain: Removed last_fleur_calc_output. Is available under last_calc.output_parameters
  - Ports for generic FleurinpData are renamed to consistently be fleurinp. Affects FleurCalculation, FleurinputgenCalculation, FleurBaseWorkChain
Expired Deprecations

- **FleurinpModifier**: Removed compatibility with old method names/behaviour before introducing masci-tools
- **FleurinpData**: Removed `get_tag`. Use `load_inpxml` and any evaluation routine afterwards instead
- Coordinate conversion functions `abs_to_rel`, etc., These are available in masci-tools
- Removed constants module. Now only available in masci-tools

Improvements

- New workchain `FleurRelaxTorqueWorkChain` for relaxing non-collinear magnetic configurations
- Fixes in DFT+U handling
  - `FleurCalculation`: added `fleurinp_nmmpmat_priority` key to `settings` to control from where to take the `n_mmp_mat` file if it’s in the fleurinp and parent_folder input
  - Fixed several errors in orbcontrol workchain when handling non-converged/failed calculation
  - Added inputs to orbcontrol to restart from intermediate charge densities
- Added `inpxml_changes` contextmanager for easier creation of the workflow parameters input of the same name

```python
from aida_fleur.data import inpxml_changes

wf_parameters = {}

with inpxml_changes(wf_parameters) as fm:
    fm.set_inpchanges({'kmax': 4, 'itmax': 100})
    fm.set_species('all', {'mtsphere': {'radius': 3}})

print(wf_parameters['inpxml_changes'])  #now contains the list like before
```

8.1.1.2 v.1.3.1

**release compatible with AiiDA-core 1.3.0+**

- Fix for masci-tools dependency constraint. The constraint would previously reject the next minor version of masci-tools (i.e 0.10.0)
- Small fixes in zenodo metadata
- Added `convert_inpxml` method to `FleurinpData` to convert to different file versions
8.1.1.3 v.1.3.0

release compatible with AiiDA-core 1.3.0+

- Guranteed support for Fleur versions up to Max6
- Dropped support for python 3.6
- Added CFcoef Workchain for calculating 4f crystal field coefficients
- General Improvements of Force theorem workchains, allow switching kpoints for force theorem calculations
- General Improvements of Orbcontrol workchain, allow starting from structure/charge density without SCF workchain
- Added support for inpgen profiles
- Refactored BaseFleurWorkChain; switched implementation of BaseRestartWorkChain from aiida-fleur to implementation provided by aiida-core
- Added support for starting SCF Workchain with first calculation using straight mixing either for the charge density or the DTF+U density matrix

8.1.1.4 v.1.2.1

release compatible with AiiDA-core 1.3.0+

- General improvements for CreateMagnetic workchain and related methods
- Added OrbControl workchain
- FleurBandDosWorkchain provides AiiDA BandsData for bandstructure calculations and XyData for DOS calculations as outputs
- General Improvements to plot_fleur function, e.g. can now visualize FleurBandDosWorkChain

8.1.1.5 v.1.2.0

release compatible with AiiDA-core 1.3.0+

possibly ready for aiida-core 2.0.0

- supports Fleur MaXR4 and MaXR5 versions with new inpgen MaXR4 requires providing versions in the code nodes
- Some features relying on the id in the inpgen files, may be broken by the new inpgen interface change when using MaXR5.1
- Added support for GW calculations with Spex, and the Strain workchain
- Major code refactoring, moving all xml tools to masci-tools (therefore requires masci-tools >=0.4.8)
- Also all file parsers are overworked and moved to masci-tools
- Work over of the BanddosWorkChain.
- FleurinpData now consistently supports more included xml files (kpts.xml, sym.xml, …)
- Added new modification functions to the FleurinpModifier for kpoint manipulation for Max5
8.1.1.6 v.1.1.4

release compatible with AiiDA-core 1.3.0

- still support of Fleur MaX4 version with inpgen
- Does not support yet Fleur MaX5 and new inpgen
- Fixed numpy dependency issue with aiida-common-workflows and quantum mobile

8.1.1.7 v1.1.3

release compatible with AiiDA-core 1.3.0

- still support of Fleur MaX4 version with inpgen
- Does not support yet for Fleur MaX5 and new inpgen
- Set_kpoints was moved from fleurinp to fleurinpmodifier
- Break_symmetry of a structure was refactored
- Implemented feature in fleurinputCalculation to set significant figures
- Implemented feature scf can now use default queues specified in code extras
- First implementation of relax type None, which cases the relax workchain to skip the relaxation, becoming a usual scf wc, which might make it easier to switch relaxation on and off in other workchains.
- Fleur parser parses now the total magnetic moment of the cell
- Introduced common constants, for bohr and htr, increased precision
- Command line interface (CLI) aiida-fleur with various functionalities exposed
- For devs: Increased test coverage, codecov is now added to CI and linked to badge removed some older outdated code

8.1.1.8 v1.1.2

release compatible with AiiDA-core 1.3.0

- still support of Fleur MaX4 version (release branch) with inpgen
- downgraded aiida-core dependency, do release does not depend on aiida-testing
- Added userfriendly LDA+U support
- SCF workchain can generate kpoints from a given density
- Base fleur has now time limit error handler
- Relax workchain can now run a final scf
- Update documentation for corehole, initial CLS and create magnetic workchains
- Various bug fixes and robustness improvements of magnetic workchains
- For devs: Enforced pre-commit, tests dir moved out of source
8.1.1.9 v1.1.1

release compatible with AiiDA-core 1.4.0

- still support of Fleur MaX4 version (release branch) with inpgen
- bugfixes and other general improvements
- new: BandDos workchain: workchain for Band and DOS calculation using the new Fleur BandDOS file
- basic workchains are now cachable, by moving cf out of workchains
- first calcjob and workchain regression tests for outside CI env
- provenance of the result nodes of magnetic workchains is fixed
- corehole and initial_cls workchain are fixed and working
- exit codes for inpgen parser

8.1.1.10 v1.1.0

release compatible with AiiDA-core 1.1.0

- support of Fleur Max4 version (release branch)
- make use of namespaces for nested workchains
- inputs for the workchains are checked more strictly
- exit codes are organised and consistent
- FleurRestart workchain: automatic parallelisation is able to make OMP threading
- new: BaseRelax workchain wrapping RelaxWorkChain and fixes its failures
- new: CreateMagnetic workchain that creates relaxed film structure
- increased unit test coverage for tools and utilities
- code clean-ups, pylint score increased to 7.49

8.1.1.11 v1.0.0a

release compatible with AiiDA-core 1.0.0b5

- added magnetic workchains
- added geometry optimisation (relax) workchain
- implemented the use of exit codes
- added FLEUR restart workchain
- integrated new Fleur input schema files
- other improvements of the workchains and calculations
- code clean-ups, documentation updates
8.1.1.12 v0.6.0

Release for MaX virtual machine, not so well tested, but used in production mode. Some things are currently half done

• added CI
• added basic tests, coverage still bad, but tests if plugin is installed right
• added MANIFEST
• fixed fleur_schema issued if installed as python package (with manifest)
• integrated the new Fleur schema files
• bunch of new utility
• advancements of workflows
• correction of AiiDA graphs of most workchains, Quick and dirt, still unclear what is the right way to do these things, aiida_core still changes
• increased pylint score from 0 to >5

8.1.1.13 v0.5.0

Merge with advanced workflow repo

• this included the corehole and initial corehole workflow as well calculation of formation energies. Therefore this is the first public released version of them with in MaX
• all the utility of the corelevel repo is now under aiida_fleur/tools

8.1.1.14 v0.4.0

• further improvment of scf, eos and other workchains
• a workchain delta form calculation a delta value, or performing calculation on the delta structures or a group of structures in a single shot
• lots of new utility methods for structure dealings, fleur parameters and so on
• lots of bugfixes
• new system for the schema files, user does not has to add aiida-fleur to pythonpath or hack the schema paths.
• added new tests, submission tests and standard fleur tests
• first documentation online, still very rusty still some issues there (stay with local one)
**8.1.1.15 v0.3.0**

**Merge with workflows repo**

- the second repository with basic workflows was merged into the plugin repository.
- Afterwards the repo was renamed from aiida_fleur_plugin to aiida-fleur

**Installation (new aiida plugin system):**

- everything is now pip installable (pip install -e .) (not yet on pypi) Therefore the files do not have to be copied anymore into the aiida_core source folder (make sure to add the aiida-fleur folder to your PYTHONPATH variable)
- all modules are now importet from the aiida_fleur folder (example ‘from aiida.tools.codespecific.fleur.convergence import fleur_convergence’ -> ‘from aiida_fleur.workflows.scf import fleur_scf_wc)

**Renaming**

- In the process (and due to the entry points some things have to be renamed) The plugin in aiida (fleur_inp.fleur -> fleur.fleur; fleur_inp.fleurinp -> fleur.fleurinp; fleur_inp.fleurinputgen -> fleur.inpgen)

**Workflows**

- some fine tuning of workflows. Naming scheme was introduced.
- Some first error catching and controlled shutdown (because of new AiiDA features).
- added consistent through all workflows the ‘serial’ key in wf_parameter nodes, which will turn of mpi.
- scf now uses minDistance and passes the walltime to fleur by default.

**Dokumentation**

- Due to the new plugin system of AiiDA the Dokumentation is now online on read-the-docs. (so far incomplete because of old AiiDA version on pypi) We still recommend to take a look at the docs in the repo itself (ggf build it)

**Utils**

- read fleur cif folder does not break the proverance any more
Further stuff

- 0.28 Fleur schema added to aiida-fleur

8.1.1.16 v0.2.0 tutorial version

Version for used at the MAX AiiDA-fleur tutorial in May 2017

Dokumentation

- added some basic explanations pages beyond the pure in code docs

Tests

- added basic tests of Fleur itself and tests for submission

Ploting

- There is a plot_methods repo on bitbucket which has methods to visualize common workflow output nodes.

Workflows

- first basic working workflows available
- some common workflow stuff is now in common_fleur_wf.py

Fleurmodifier

- Introduction of the Fleurinpmodifier class, to change fleurinp data

Fleurinp data

- restructureing of fleurinp data and Fleurinpmodifier, moved most xml methods into xml_util
- plus added further xml routines and rough tests.

8.1.1.17 v0.1 Base commit

Moved everything von bitbucket to github

Dokumentation

- Basic docs available locally
Installation

- provided copy_files script to copy the plugin files into AiiDA folder

Workflows

- Some basic sketches of basic workflows available (working AiiDa workflow system just released)
INDICES AND TABLES

• genindex
• modindex
• search
aiida_fleur.calculation.fleur, 148
aiida_fleur.calculation.fleurinputgen, 147
aiida_fleur.data.fleurinp, 149
aiida_fleur.data.fleurinpmod, 154
aiida_fleur.parsers.fleur, 148
aiida_fleur.parsers.fleur_inputgen, 148
aiida_fleur.tools.common_aiida, 219
aiida_fleur.tools.common_fleur_wf, 220
aiida_fleur.tools.common_fleur_wf_util, 223
aiida_fleur.tools.create_corehole, 216
aiida_fleur.tools.dict_util, 215
aiida_fleur.tools.element_econfig_list, 218
aiida_fleur.tools.extract_corelevels, 216
aiida_fleur.tools.io_routines,  220
aiida_fleur.tools.merge_parameter, 216
aiida_fleur.tools.read_cif_folder, 220
aiida_fleur.tools.StructureData_util, 206
aiida_fleur.tools.xml_aiida_modifiers, 214
aiida_fleur.workflows.banddos, 173
aiida_fleur.workflows.base_fleur, 171
aiida_fleur.workflows.cfcoeff, 187
aiida_fleur.workflows.corehole, 179
aiida_fleur.workflows.dmi, 184
aiida_fleur.workflows.dos, 174
aiida_fleur.workflows.eos, 175
aiida_fleur.workflows.initial_cls, 177
aiida_fleur.workflows.mae, 181
aiida_fleur.workflows.mae_conv, 182
aiida_fleur.workflows.orbcontrol, 185
aiida_fleur.workflows.relax, 176
aiida_fleur.workflows.scf, 171
aiida_fleur.workflows.ssdisp, 183
aiida_fleur.workflows.ssdisp_conv, 184
Symbols

__init__()  (aiida_fleur.data.fleurinp.FleurinpData
method), 149
-A
  aiida-fleur-data-fleurinp-list command
  line option, 190
-G
  aiida-fleur-data-fleurinp-list command
  line option, 190
-I
  aiida-fleur-launch-fleur command line
  option, 198
-M
  aiida-fleur-data-options-create command
  line option, 192
  aiida-fleur-launch-fleur command line
  option, 198
-N
  aiida-fleur-data-options-create command
  line option, 192
  aiida-fleur-launch-fleur command line
  option, 198
-P
  aiida-fleur-launch-banddos command line
  option, 194
  aiida-fleur-launch-fleur command line
  option, 198
  aiida-fleur-launch-mae command line
  option, 201
  aiida-fleur-launch-scf command line
  option, 203
-W
  aiida-fleur-data-options-create command
  line option, 192
  aiida-fleur-launch-fleur command line
  option, 198
--all-users
  aiida-fleur-data-fleurinp-list command
  line option, 190
--bokeh
  aiida-fleur-plot command line option, 204
--calc-parameters
aiida-fleur-launch-relax command line option, 202
aiida-fleur-launch-scf command line option, 203
aiida-fleur-launch-ssdisp command line option, 204
--dry-run
aiida-fleur-launch-create command line option, 192
aiida-fleur-data-parameter-import command line option, 193
aiida-fleur-data-structure-import command line option, 194
--eos-parameters
aiida-fleur-launch-create_magnetic command line option, 196
--extras
aiida-fleur-data-fleurinp-list command line option, 191
--filename
aiida-fleur-data-fleurinp-cat command line option, 189
aiida-fleur-data-fleurinp-open command line option, 191
--fleur
aiida-fleur-launch-banddos command line option, 194
aiida-fleur-launch-corehole command line option, 195
aiida-fleur-launch-create_magnetic command line option, 196
aiida-fleur-launch-dmi command line option, 196
aiida-fleur-launch-eos command line option, 197
aiida-fleur-launch-fleur command line option, 198
aiida-fleur-launch-init_cls command line option, 199
aiida-fleur-launch-mae command line option, 201
aiida-fleur-launch-relax command line option, 202
aiida-fleur-launch-scf command line option, 202
aiida-fleur-launch-ssdisp command line option, 203
--fleurinp
aiida-fleur-launch-corehole command line option, 195
aiida-fleur-launch-relax command line option, 202
aiida-fleur-launch-scf command line option, 203
--keys
aiida-fleur-launch-mae command line option, 201
aiida-fleur-launch-relax command line option, 202
aiida-fleur-launch-scf command line option, 202
aiida-fleur-launch-ssdisp command line option, 203
--format
aiida-fleur-workflow-inputdict command line option, 205
aiida-fleur-workflow-res command line option, 206
--groups
aiida-fleur-launch-mae command line option, 201
aiida-fleur-launch-relax command line option, 202
aiida-fleur-launch-scf command line option, 203
--info
aiida-fleur-workflow-inputdict command line option, 205
aiida-fleur-workflow-res command line option, 206
--inpgen
aiida-fleur-launch-create_magnetic command line option, 196
aiida-fleur-launch-dmi command line option, 196
aiida-fleur-launch-eos command line option, 197
aiida-fleur-launch-init_cls command line option, 199
aiida-fleur-launch-inpgen command line option, 200
aiida-fleur-launch-mae command line option, 201
aiida-fleur-launch-relax command line option, 202
aiida-fleur-launch-scf command line option, 202
aiida-fleur-launch-ssdisp command line option, 203
--keys
aiida-fleur-workflow-inputdict command line option, 205
aiida-fleur-workflow-res command line option, 206
--label
aiida-fleur-workflow-inputdict command line option, 205
aiida-fleur-workflow-res command line option, 206
--launch_base
    aiiad-fleurlaunch-fleur command line option, 199
--matplotlib
    aiiad-fleurlaunch-fleur command line option, 204
--max-num-machines
    aiiad-fleurlaunch-fleur command line option, 198
--max-wallclock-seconds
    aiiad-fleurlaunch-fleur command line option, 192
--num-mpiprocs-per-machine
    aiiad-fleurlaunch-fleur command line option, 199
--option-node
    aiiad-fleurlaunch-banddos command line option, 194
    aiiad-fleurlaunch-corehole command line option, 195
    aiiad-fleurlaunch-create_magnetic command line option, 196
    aiiad-fleurlaunch-dmi command line option, 197
    aiiad-fleurlaunch-eos command line option, 198
    aiiad-fleurlaunch-mae command line option, 201
    aiiad-fleurlaunch-relax command line option, 202
    aiiad-fleurlaunch-scf command line option, 203
    aiiad-fleurlaunch-ssdisp command line option, 204
--output-filename
    aiiad-fleurlaunch-banddos command line option, 194
    aiiad-fleurlaunch-corehole command line option, 195
    aiiad-fleurlaunch-create_magnetic command line option, 196
    aiiad-fleurlaunch-dmi command line option, 197
    aiiad-fleurlaunch-eos command line option, 198
    aiiad-fleurlaunch-mae command line option, 201
    aiiad-fleurlaunch-relax command line option, 202
    aiiad-fleurlaunch-scf command line option, 203
    aiiad-fleurlaunch-ssdisp command line option, 204
--parent-folder
    aiiad-fleurlaunch-banddos command line option, 194
    aiiad-fleurlaunch-corehole command line option, 195
    aiiad-fleurlaunch-create_magnetic command line option, 196
    aiiad-fleurlaunch-dmi command line option, 197
    aiiad-fleurlaunch-eos command line option, 198
    aiiad-fleurlaunch-mae command line option, 201
    aiiad-fleurlaunch-relax command line option, 202
    aiiad-fleurlaunch-scf command line option, 203
--profile
    aiiad-fleurlaunch-banddos command line option, 194
    aiiad-fleurlaunch-corehole command line option, 195
    aiiad-fleurlaunch-create_magnetic command line option, 196
    aiiad-fleurlaunch-dmi command line option, 197
    aiiad-fleurlaunch-eos command line option, 198
    aiiad-fleurlaunch-mae command line option, 201
    aiiad-fleurlaunch-relax command line option, 202
    aiiad-fleurlaunch-scf command line option, 203
--queue
    aiiad-fleurlaunch-banddos command line option, 194
    aiiad-fleurlaunch-corehole command line option, 195
    aiiad-fleurlaunch-create_magnetic command line option, 196
    aiiad-fleurlaunch-dmi command line option, 197
    aiiad-fleurlaunch-eos command line option, 198
    aiiad-fleurlaunch-mae command line option, 201
    aiiad-fleurlaunch-relax command line option, 202
    aiiad-fleurlaunch-scf command line option, 203
    aiiad-fleurlaunch-ssdisp command line option, 204
    aiiad-fleurlaunch-banddos command line option, 194
    aiiad-fleurlaunch-corehole command line option, 195
    aiiad-fleurlaunch-create_magnetic command line option, 196
    aiiad-fleurlaunch-dmi command line option, 197
    aiiad-fleurlaunch-eos command line option, 198
    aiiad-fleurlaunch-mae command line option, 201
    aiiad-fleurlaunch-relax command line option, 202
    aiiad-fleurlaunch-scf command line option, 203
    aiiad-fleurlaunch-ssdisp command line option, 204
    aiiad-fleurlaunch-relax command line option, 202
    aiiad-fleurlaunch-scf command line option, 203
    aiiad-fleurlaunch-ssdisp command line option, 204
    aiiad-fleurlaunch-corehole command line option, 195
    aiiad-fleurlaunch-create_magnetic command line option, 196
    aiiad-fleurlaunch-dmi command line option, 197
    aiiad-fleurlaunch-eos command line option, 198
    aiiad-fleurlaunch-mae command line option, 201
    aiiad-fleurlaunch-relax command line option, 202
    aiiad-fleurlaunch-scf command line option, 203
    aiiad-fleurlaunch-ssdisp command line option, 204
    aiiad-fleurlaunch-tests command line option, 205
    aiiad-fleurlaunch-workflow command line option, 206
--with-mpi
  aida-fleur-launch-fleur command line option, 198
-calc_p
  aida-fleur-launch-corehole command line option, 195
  aida-fleur-launch-create_magnetic command line option, 196
  aida-fleur-launch-dmi command line option, 196
  aida-fleur-launch-eos command line option, 197
  aida-fleur-launch-init_cls command line option, 199
  aida-fleur-launch-inpgen command line option, 200
  aida-fleur-launch-mae command line option, 201
  aida-fleur-launch-relax command line option, 202
  aida-fleur-launch-scf command line option, 202
  aida-fleur-launch-ssdisp command line option, 203
-d
  aida-fleur-launch-banddos command line option, 194
  aida-fleur-launch-corehole command line option, 195
  aida-fleur-launch-create_magnetic command line option, 196
  aida-fleur-launch-dmi command line option, 197
  aida-fleur-launch-eos command line option, 197
  aida-fleur-launch-fleur command line option, 198
  aida-fleur-launch-init_cls command line option, 199
  aida-fleur-launch-inpgen command line option, 200
  aida-fleur-launch-mae command line option, 201
  aida-fleur-launch-relax command line option, 202
  aida-fleur-launch-scf command line option, 203
  aida-fleur-launch-ssdisp command line option, 204
-eos
  aida-fleur-launch-create_magnetic command line option, 196
-f
  aida-fleur-data-fleurinp-cat command line option, 189
  aida-fleur-data-fleurinp-open command line option, 191
  aida-fleur-launch-banddos command line option, 194
  aida-fleur-launch-corehole command line option, 195
  aida-fleur-launch-create_magnetic command line option, 196
  aida-fleur-launch-dmi command line option, 196
  aida-fleur-launch-eos command line option, 197
  aida-fleur-launch-fleur command line option, 198
  aida-fleur-launch-init_cls command line option, 199
  aida-fleur-launch-inpgen command line option, 201
  aida-fleur-launch-mae command line option, 201
  aida-fleur-launch-relax command line option, 202
  aida-fleur-launch-scf command line option, 202
  aida-fleur-launch-ssdisp command line option, 203
-i
  aida-fleur-launch-corehole command line option, 195
  aida-fleur-launch-create_magnetic command line option, 196
  aida-fleur-launch-dmi command line option, 196
  aida-fleur-launch-eos command line option, 197
  aida-fleur-launch-fleur command line option, 198
  aida-fleur-launch-init_cls command line option, 199
  aida-fleur-launch-inpgen command line option, 200
  aida-fleur-launch-mae command line option, 201
  aida-fleur-launch-relax command line option, 202
  aida-fleur-launch-scf command line option, 203
  aida-fleur-launch-ssdisp command line option, 204
-inp
  aida-fleur-launch-banddos command line option, 194
aiida-fleur-launch-corehole command line option, 195
aiida-fleur-launch-fleur command line option, 198
aiida-fleur-launch-init_cls command line option, 199
aiida-fleur-launch-mae command line option, 201
aiida-fleur-launch-scf command line option, 203
-p
aiida-fleur command line option, 189
aiida-fleur-data-fleurinplist command line option, 190
-q
aiida-fleur-data-options-create command line option, 192
aiida-fleur-launch-fleur command line option, 198
aiida-fleur-launch-inpgen command line option, 200
-r
aiida-fleur-data-fleurinplist command line option, 190
aiida-fleur-launch-create_magnetic command line option, 196
-s
aiida-fleur-data-fleurinp-open command line option, 191
aiida-fleur-launch-corehole command line option, 195
aiida-fleur-launch-dmi command line option, 196
aiida-fleur-launch-eos command line option, 197
aiida-fleur-launch-init_cls command line option, 199
aiida-fleur-launch-inpgen command line option, 200
aiida-fleur-launch-mae command line option, 201
aiida-fleur-launch-relax command line option, 202
aiida-fleur-launch-scf command line option, 203
AiiDa-FLEUR Documentation, Release 2.0.0

-ab
aiida-fleur-launch-banddos command line option, 194
aiida-fleur-launch-corehole command line option, 195
aiida-fleur-launch-eos command line option, 197
aiida-fleur-launch-fleur command line option, 198
aiida-fleur-launch-init_cls command line option, 199
aiida-fleur-launch-inpgen command line option, 200
aiida-fleur-launch-mae command line option, 201
aiida-fleur-launch-relax command line option, 202
aiida-fleur-launch-scf command line option, 202, 203

-v
aiida-fleur command line option, 189

-wf
aiida-fleur-launch-banddos command line option, 194
aiida-fleur-launch-corehole command line option, 195
aiida-fleur-launch-create_magnetic command line option, 196
aiida-fleur-launch-dmi command line option, 197
aiida-fleur-launch-eos command line option, 197
aiida-fleur-launch-init_cls command line option, 199
aiida-fleur-launch-mae command line option, 201
aiida-fleur-launch-relax command line option, 202
aiida-fleur-launch-scf command line option, 202, 203
aiida-fleur-launch-ssdisp command line option, 203
adjust_calc_para_to_structure() (in module aiida_fleur.tools.StructureData_util), 206
adjust_film_relaxation() (in module aiida_fleur.tools.StructureData_util), 206
adjust_sym_film_relaxation() (in module aiida_fleur.tools.StructureData_util), 207
aiida_fleur.calculation.fleur module, 148
aiida_fleur.calculation.fleurinputgen module, 147
aiida_fleur.data.fleurinp module, 149
aiida_fleur.data.fleurinpmodifier module, 154
aiida_fleur.parsers.fleur module, 148
aiida_fleur.parsers.fleur_inputgen module, 148
aiida_fleur.tools.common_aiida module, 219
aiida_fleur.tools.common_fleur_wf module, 220
aiida_fleur.tools.common_fleur_wf_util module, 223
aiida_fleur.tools.create_corehole module, 216
aiida_fleur.tools.dict_util module, 215
aiida_fleur.tools.element_econfig_list module, 218
aiida_fleur.tools.extract_corelevels module, 216
aiida_fleur.tools.io_routines module, 220
aiida_fleur.tools.merge_parameter module, 216
aiida_fleur.tools.read_cif_folder module, 220
aiida_fleur.tools.StructureData_util module, 206
aiida_fleur.tools.xml_aiida_modifiers module, 214
aiida_fleur.workflows.banddos module, 173
aiida_fleur.workflows.base_fleur module, 171
aiida_fleur.workflows.cfcoeff module, 187
aiida_fleur.workflows.corehole module, 179
aiida_fleur.workflows.dmi module, 184
aiida_fleur.workflows.dos module, 174
aiida_fleur.workflows.eos module, 175

aiida_fleur.workflows.initial_cls module, 177

aiida_fleur.workflows.mae module, 181

aiida_fleur.workflows.mae_conv module, 182

aiida_fleur.workflows.orbcontrol module, 185

aiida_fleur.workflows.relax module, 176

aiida_fleur.workflows.scf module, 171

aiida_fleur.workflows.ssdisp module, 183

aiida_fleur.workflows.ssdisp_conv module, 184

aiida-fleur command line option
--profile, 189
--version, 189
-p, 189
-v, 189

aiida-fleur-data-fleurinp-cat command line option
--filename, 189
-f, 189

aiida-fleur-data-fleurinp-extract-inpgen command line option
--no-para, 190
--output-filename, 190
--para, 190
-o, 190

aiida-fleur-data-fleurinp-list command line option
-A, 190
-G, 190
--all-users, 190
--ctime, 191
--extras, 191
--groups, 190
--no-ctime, 191
--no-extras, 191
--no-strucinfo, 191
--no-uuid, 190
--past-days, 190
--raw, 190
--strucinfo, 191
--uuid, 190
-p, 190
-r, 190

aiida-fleur-data-fleurinp-open command line option
--filename, 191
--output-filename, 191
--save, 191
-f, 191
-o, 191
-s, 191

aiida-fleur-data-options-create command line option
-N, 192
-W, 192

aiida-fleur-data-parameter-import command line option
--dry-run, 193
--fleurinp, 193
--no-fleurinp, 193
--no-show, 193
--show, 193
-n, 193

aiida-fleur-data-structure-import command line option
--dry-run, 194
--fleurinp, 194
--no-fleurinp, 194
-n, 194

aiida-fleur-launch-banddos command line option
-P, 194
--daemon, 194
--fleur, 194
--fleurinp, 194
--option-node, 194
--parent-folder, 194
--settings, 194
--wf-parameters, 194
-d, 194
-f, 194
-inp, 194
-opt, 194
-set, 194
aiida-fleur-launch-corehole command line option
  --calc-parameters, 195
  --daemon, 195
  --fleur, 195
  --fleurinp, 195
  --inp_gen, 195
  --option-node, 195
  --settings, 195
  --structure, 195
  --wf-parameters, 195
  -calc_p, 195
  -d, 195
  -f, 195
  -i, 195
  -inp, 195
  -opt, 195
  -s, 195
  -set, 195
  -wf, 195

aiida-fleur-launch-create_magnetic command line option
  --calc-parameters, 196
  --daemon, 196
  --eos-parameters, 196
  --fleur, 196
  --fleurinp, 196
  --option-node, 196
  --relax-parameters, 196
  --scf-parameters, 196
  --wf-parameters, 196
  -calc_p, 196
  -d, 196
  -eos, 196
  -f, 196
  -i, 196
  -opt, 196
  -relax, 196
  -scf, 196
  -wf, 196

aiida-fleur-launch-dmi command line option
  --calc-parameters, 196
  --daemon, 197
  --fleur, 196
  --fleurinp, 196
  --launch_base, 199
  --max-num-machines, 198
  --max-wallclock-seconds, 198
  --no-launch_base, 199
  --num-mpiprocs-per-machine, 198
  --option-node, 196
  --parent-folder, 198
  --queue, 198
  --with-mpi, 198
  -calc_p, 196
  -d, 196
  -f, 196
  -i, 196
  -inp, 196
  -opt, 196
  -parent-folder, 198
  -q, 198
  -set, 198
  -wf, 196

aiida-fleur-launch-eos command line option
  --calc-parameters, 197
  --daemon, 197
  --fleur, 197
  --fleurinp, 197
  --option-node, 198
  --scf-parameters, 197
  --settings, 197
  --structure, 197
  --wf-parameters, 197
  -calc_p, 197
  -d, 197
  -f, 197
  -i, 197
  -opt, 198
  -s, 197
  -scf, 197
  -set, 197
  -wf, 197

aiida-fleur-launch-fleur command line option
  --daemon, 198
  --fleur, 198
  --fleurinp, 198
  --launch_base, 199
  --max-num-machines, 198
  --max-wallclock-seconds, 198
  --no-launch_base, 199
  --num-mpiprocs-per-machine, 198
  --option-node, 198
  --queue, 198
  --with-mpi, 198
  -calc_p, 198
  -d, 198
  -f, 198
  -i, 198
  -inp, 198
  -opt, 198
  -q, 198
  -set, 198
  -s, 198
  -fleur, 199

aiida-fleur-launch-init_cls command line option
  --calc-parameters, 199
  --daemon, 199
  --fleur, 199
--fleurinp, 199
--inpgen, 199
--option-node, 199
--settings, 199
--structure, 199
--wf-parameters, 199
-calcp, 199
-d, 199
-f, 199
-i, 199
-inp, 199
-opt, 199
-s, 199
-set, 199
-wf, 199

aiida-fleur-launch-inpgen command line
 option
--calc-parameters, 200
--daemon, 200
--inpgen, 200
--option-node, 200
--queue, 200
--settings, 200
--structure, 200
-calcp, 200
-d, 200
-i, 200
-opt, 200
-q, 200
-s, 200
-set, 200

aiida-fleur-launch-ssdisp command line
 option
--calc-parameters, 203
--daemon, 203
--fleur, 203
--fleurinp, 203
--inpgen, 203
--option-node, 203
--parent-folder, 203
--scf-parameters, 201
--settings, 201
--structure, 201
--wf-parameters, 201
-calcp, 201
-d, 201
-f, 201
-i, 201
-inp, 201
-opt, 201
-s, 201
-scfdisp, 201
-set, 201
-wf, 201

aiida-fleur-launch-relax command line
 option
--calc-parameters, 202
--daemon, 202
--fleur, 202
--inpgen, 202
--option-node, 202
--scf-parameters, 202
--settings, 202
--structure, 202
--wf-parameters, 202
-calcp, 202
-d, 202
-f, 202
-i, 202
-opt, 202
-s, 202
-scfdisp, 202
-set, 202
-wf, 202

aiida-fleur-launch-scf command line
 option
--calc-parameters, 202
--daemon, 203
--fleur, 203
--fleurinp, 203
--inpgen, 202
--option-node, 203
--parent-folder, 203
--settings, 202, 203
--structure, 202
--wf-parameters, 203
-calcp, 202
-d, 203
-f, 203
-i, 202
-inp, 203
-opt, 203
-s, 202
-set, 202, 203
-wf, 203

aiida-fleur-launch-ssdisp command line
 option
--calc-parameters, 203
--daemon, 204
--fleur, 203
--inpgen, 203
--option-node, 204
--scf-parameters, 203
--structure, 203
--wf-parameters, 203
-calcp, 203
-d, 204
-f, 203
aiida-fleur-plot command line option
--bokeh, 204
--matplotlib, 204
--no-show, 204
--no-show_dict, 204
--save, 204
--show, 204
--show_dict, 204
-f, 204

aiida-fleur-workflow-inputdict command line option
--format, 205
--info, 205
--keys, 205
--label, 205
--no-info, 205
--no-show, 205
--show, 205
-f, 205
-k, 205
-l, 205

aiida-fleur-workflow-res command line option
--format, 206
--info, 206
--keys, 206
--label, 206
--no-info, 206
--no-show, 206
--show, 206
-f, 206
-k, 206
-l, 206

align_nmmpmat_to_sqa() (aiida_fleur.data.fleurinpmodifier.FleurinpModifier class method), 155
analyse_relax() (aiida_fleur.workflows.relax.FleurRelaxWorkChain static method), 176
apply_fleurinp_modifications() (aiida_fleur.data.fleurinpmodifier.FleurinpModifier class method), 155
apply_modifications() (aiida_fleur.data.fleurinpmodifier.FleurinpModifier class method), 155

balance_equation() (in module aiida_fleur.tools.common_fleur_wf_util), 223
banddos_after_scf() (aiida_fleur.workflows.banddos.FleurBandDosWorkChain method), 173
banddos_wo_scf() (aiida_fleur.workflows.banddos.FleurBandDosWorkChain method), 173
birch_murnaghan() (in module aiida_fleur.workflows.eos), 175
birch_murnaghan_fit() (in module aiida_fleur.workflows.eos), 175
break_symmetry() (in module aiida_fleur.tools.StructureData_util), 207
break_symmetry_wf() (in module aiida_fleur.tools.StructureData_util), 208

calc_stoi() (in module aiida_fleur.tools.common_fleur_wf_util), 223
calc_time_cost_function() (in module aiida_fleur.tools.common_fleur_wf), 220
calc_time_cost_function_total() (in module aiida_fleur.tools.common_fleur_wf), 220
calculate_cf_coefficients() (in module aiida_fleur.workflows.cfcoeff), 188
center_film() (in module aiida_fleur.tools.StructureData_util), 208
center_film_wf() (in module aiida_fleur.tools.StructureData_util), 208
change_fleurinp() (aiida_fleur.workflows.banddos.FleurBandDosWorkChain method), 173
change_fleurinp() (aiida_fleur.workflows.dmi.FleurDMIWorkChain method), 184
change_fleurinp() (aiida_fleur.workflows.mae.FleurMaeWorkChain method), 181
change_fleurinp() (aiida_fleur.workflows.scf.FleurScfWorkChain method), 172
change_fleurinp() (aiida_fleur.workflows.ssdisp.FleurSSDispWorkChain method), 183
changes() (aiida_fleur.data.fleurinpmodifier.FleurinpModifier class method), 156
check_cf_calculation() (aiida_fleur.workflows.cfcoeff.FleurCFCoeffWorkChain method), 187
check_eos_energies() (in module aiida_fleur.tools.common_fleur_wf_util), 223
check_failure() (aiida_fleur.workflows.relax.FleurRelaxWorkChain method), 176
check_input() (aiida_fleur.workflows.corehole.FleurCoreholeWorkChain method), 180
clear_dict_empty_lists() (in module aiida_fleur.tools.dict_util), 215
clean_nones() (in module aiida_fleur.toolsSTRUCTURE_DATA_UTIL), 208
clshifts_to_be() (in module aiida_fleur.tools.extract_corelevels), 216
clshifts_to_be() (in module aiida_fleur.workflows.initial_cls), 178
clone_species() (aiida_fleur.data.fleurinpmodifier.FleurinpModifier method), 156
control_end_wc() (aiida_fleur.workflows.banddos.FleurBandDosWorkChain method), 173
control_end_wc() (aiida_fleur.workflows.dmi.FleurDMIWorkChain method), 184
control_end_wc() (aiida_fleur.workflows.fleurinp.FleurinpData method), 149
control_end_wc() (aiida_fleur.workflows.ess.FleurEssWorkChain method), 180
control_end_wc() (aiida_fleur.workflows.mae.FleurMaeWorkChain method), 181
control_end_wc() (aiida_fleur.workflows.mae_conv.FleurMaeConvWorkChain method), 182
control_end_wc() (aiida_fleur.workflows.orbcontrol.FleurOrbControlWorkChain method), 186
control_end_wc() (aiida_fleur.workflows.relax.FleurRelaxWorkChain method), 176
control_end_wc() (aiida_fleur.workflows.ssdisp.FleurSSDispWorkChain method), 183
control_end_wc() (aiida_fleur.workflows.ssdisp_conv.FleurSSDispConvWorkChain method), 184
control_end_wc() (aiida_fleur.workflows.mae_conv.FleurMaeConvWorkChain method), 182
control_end_wc() (aiida_fleur.workflows.orbcontrol.FleurOrbControlWorkChain method), 186
control_end_wc() (aiida_fleur.workflows.relax.FleurRelaxWorkChain method), 176
control_end_wc() (aiida_fleur.workflows.ssdisp.FleurSSDispWorkChain method), 183
condition() (aiida_fleur.workflows.relax.FleurRelaxWorkChain method), 176
condition() (aiida_fleur.workflows.mae.FleurMaeWorkChain method), 181
converge_scf() (aiida_fleur.workflows.banddos.FleurBandDosWorkChain method), 173
converge_scf() (aiida_fleur.workflows.dmi.FleurDMIWorkChain method), 184
converge_scf() (aiida_fleur.workflows.fleurinp.FleurinpData method), 152
converge_scf() (aiida_fleur.workflows.ess.FleurEssWorkChain method), 175
converge_scf() (aiida_fleur.workflows.mae.FleurMaeWorkChain method), 181
converge_scf() (aiida_fleur.workflows.mae_conv.FleurMaeConvWorkChain method), 182
converge_scf() (aiida_fleur.workflows.orbcontrol.FleurOrbControlWorkChain method), 186
converge_scf() (aiida_fleur.workflows.relax.FleurRelaxWorkChain method), 176
converge_scf() (aiida_fleur.workflows.ssdisp.FleurSSDispWorkChain method), 183
convert_fleur_config_to_econfig() (in module aiida_fleur.tools.ELEMENT_CONFIG_LIST), 218
convert_fleurdata_to_econfig() (in module aiida_fleur.data.fleurinp.FleurinpData method), 149
convert_inpxml() (in module aiida_fleur.data.fleurinp.FleurinpData method), 152
convert_inpxml() (in module aiida_fleur.tools.common_fleur_wf_util), 223
convert_to_float() (in module ida_fleur.tools.extract_corelevels), 217

cost_ratio() (in module ida_fleur.tools.common_fleur_wf), 220

create_aiida_bands_data() (in module ida_fleur.workflows.banddos), 173

create_aiida_dos_data() (in module ida_fleur.workflows.banddos), 174

create_all_slabs() (in module ida_fleur.tools.StructureData_util), 209

create_band_result_node() (in module ida_fleur.workflows.banddos), 174

create_cfccoeff_results_node() (in module ida_fleur.workflows.corehole), 181

create_configurations() (in module ida_fleur.workflows.orbcontrol), 186

create_corehole_para() (in module ida_fleur.tools.create_corehole), 216

create_corehole_result_node() (in module ida_fleur.workflows.corehole), 181

create_coreholes() (in module ida_fleur.workflows.corehole), 180

create_eos_result_node() (in module ida_fleur.workflows.eos), 176

create_group() (in module ida_fleur.tools.common_aiida), 219

create_initcls_result_node() (in module ida_fleur.workflows.initial_cls), 178

create_manual_slab_ase() (in module ida_fleur.tools.StructureData_util), 209

create_new_fleurinp() (in module ida_fleur.workflows.dos.fleur_dos_wc), 174

create_orbcontrol_result_node() (in module ida_fleur.workflows.orbcontrol), 187

create_relax_result_node() (in module ida_fleur.workflows.relax), 177

create_scf_result_node() (in module ida_fleur.workflows.scf), 172

create_slap() (in module ida_fleur.tools.StructureData_util), 209

create_supercell() (in module ida_fleur.workflows.corehole), 180

create_tag() (in module ida_fleur.data.fleurinpmodifier), 156

define() (in module ida_fleur.data.fleurinpmodifier), 149

define() (in module ida_fleur.calculation.fleurinputgen), 147

define() (in module ida_fleur.workflows.banddos), 173

define() (in module ida_fleur.workflows.base_fleur), 171

define() (in module ida_fleur.workflows.corehole), 180

define() (in module ida_fleur.workflows.dmi), 184

define() (in module ida_fleur.workflows.dos.fleur_dos_wc), 174

define() (in module ida_fleur.workflows.eos), 175

define() (in module ida_fleur.workflows.initial_cls), 175

define() (in module ida_fleur.workflows.mae), 181

define() (in module ida_fleur.workflows.mae_conv), 182

define() (in module ida_fleur.workflows.orbcontrol), 186

define() (in module ida_fleur.workflows.scf), 172

define() (in module ida_fleur.workflows.ssdisp), 183

define() (in module ida_fleur.workflows.ssdisp_conv), 184

define_AFM_structures() (in module ida_fleur.tools.StructureData_util), 210

del_file() (in module ida_fleur.data.fleurinp), 150

del_file() (in module ida_fleur.data.fleurinpmodifier), 156

delete_att() (in module ida_fleur.data.fleurinpmodifier), 157

delete_tag() (in module ida_fleur.data.fleurinpmodifier), 157

determine_convex_hull() (in module ida_fleur.tools.common_fleur_wf), 224

determine_favorable_reaction() (in module ida_fleur.tools.common_fleur_wf), 224

determine_reactions() (in module ida_fleur.tools.common_fleur_wf), 224

dict_merge() (in module ida_fleur.tools.dict_util), 215

D

define() (in module ida_fleur.calculation.fleur), 148
Index

E
ecnfigstr_hole() (in module ida_fleur.tools.element_ecnfig_list), 218
eos_structures() (in module ida_fleur.workflows.eos), 176
eos_structures_nocf() (in module ida_fleur.workflows.eos), 176
export_extras() (in module ida_fleur.workflows.eos), 176
extract_corelevels() (in module ida_fleur.tools.extract_corelevels), 217
extract_elementpara() (in module ida_fleur.tools.dict_util), 215
extract_nummp_file() (in module ida_fleur.workflows.orbcontrol), 187
extract_results() (in module ida_fleur.workflows.initial_cls), 179
extract_results_corehole() (in module ida_fleur.workflows.corehole), 181

F
FILENAME
aida_fleur-data-parameter-import
command line option, 193
aida_fleur-data-structure-import
command line option, 194
files (aida_fleur.data.fleurinp.FleurinpData property), 150
find_equi_atoms() (in module ida_fleur.tools.StructureData_util), 210
find_last_submitted_calcjob() (in module ida_fleur.tools.common_fleur_wf), 221
find_last_submitted_workchain() (in module ida_fleur.tools.common_fleur_wf), 221
find_nested_process() (in module ida_fleur.tools.common_fleur_wf), 221
find_parameters() (aida_fleur.workflows.initial_cls.FleurInitialCLSWorkChain
method), 178
find_primitive_cell() (in module ida_fleur.tools.StructureData_util), 210
find_primitive_cell_wf() (in module ida_fleur.tools.StructureData_util), 210
find_primitive_cells() (in module ida_fleur.tools.StructureData_util), 210
fleur_calc_get_structure() (in module ida_fleur.workflows.initial_cls), 179
fleur_dos_wc (class in ida_fleur.workflows.dos), 174
FleurInputgenParser (class in ida_fleur.parsers.fleur_inputgen), 148
FleurBandDosWorkChain (class in ida_fleur.workflows.banddos), 173
FleurBaseWorkChain (class in ida_fleur.workflows.base_fleur), 171
FleurCalculation (class in ida_fleur.workflows.fleur), 148
FleurFCoeffWorkChain (class in ida_fleur.workflows.ccoeff), 187
FleurCoreholeWorkChain (class in ida_fleur.workflows.corehole), 179
FleurDMIFWorkChain (class in ida_fleur.workflows.dmi), 184
FleurEosWorkChain (class in ida_fleur.workflows.eos), 175
FleurInitialCLSWorkChain (class in ida_fleur.workflows.initial_cls), 177
FleurinpData (class in aida_fleur.data.fleurinp), 149
fleuinpigen_needed() (aida_fleur.workflows.scf.FleurScfWorkChain
method), 172
FleurinpuinModifier (class in ida_fleur.data.fleurinpmodifier), 154
FleurinpuingetCalculation (class in ida_fleur.workflows.fleurinputget), 147
FleurMaeConvWorkChain (class in ida_fleur.workflows.mae_conv), 182
FleurMaeWorkChain (class in ida_fleur.workflows.mae), 181
FleurOrbControlWorkChain (class in ida_fleur.workflows.orbcontrol), 185
FleurParser (class in ida_fleur.parsers.fleur), 148
FleurRelaxWorkChain (class in ida_fleur.workflows.relax), 176
FleurScfWorkChain (class in ida_fleur.workflows.scf), 171
FleurSSDispConvWorkChain (class in ida_fleur.workflows.ssdisp_conv), 184
FleurSSDispWorkChain (class in ida_fleur.workflows.ssdisp), 183
force_after_scf() (aida_fleur.workflows.dmi.FleurDMIFWorkChain
method), 184
force_after_scf() (aida_fleur.workflows.mae.FleurMaeWorkChain
method), 181
force_after_scf() (aida_fleur.workflows.ssdisp.FleurSSDispWorkChain
method), 183
force_omega caching() (aida_fleur.workflows.dmi.FleurDMIFWorkChain
method), 185
force_omega caching() (aida_fleur.workflows.mae.FleurMaeWorkChain
method), 181
force_omega caching() (aida_fleur.workflows.ssdisp.FleurSSDispWorkChain
method), 183
freeze() (aida_fleur.data.fleurinpmodifier.FleurinpuinModifier
method), 157
fromList() (aida_fleur.data.fleurinpmodifier.FleurinpuinModifier
class method), 157
generate_density_matrix_configurations() (in module aiida_fleur.workflows.orbcontrol), 187
generate_new_fleurinp() (aiida_fleur.workflows.relax.FleurRelaxWorkChain method), 177
get_all_miller_indices() (in module aiida_fleur.tools.StructureData_util), 210
get_atomprocent() (in module aiida_fleur.tools.common_fleur_wf_util), 224
get_atomtype_site_symmetry() (in module aiida_fleur.tools.StructureData_util), 210
get_avail_actions() (aiida_fleur.data.fleurinpmodifier.FleurinpModifier method), 158
get_builder_continue_fixed() (aiida_fleur.workflows.orbcontrol.FleurOrbControlWorkChain class method), 186
get_builder_continue_relaxed() (aiida_fleur.workflows.orbcontrol.FleurOrbControlWorkChain class method), 186
get_content() (aiida_fleur.data.fleurinp.FleurinpData method), 150
get_coreconfig() (in module aiida_fleur.tools.element_econfig_list), 218
generate_modes() (aiida_fleur.data.fleurinp.FleurinpData method), 150
generate_fleurinp_from_folder_data() (in module aiida_fleur.data.fleurinp), 152
generate_fleurinp_from_folder_data_cf() (in module aiida_fleur.data.fleurinp), 152
generate_fleurinp_from_remote_data() (in module aiida_fleur.data.fleurinp), 152
generate_fleurinp_from_remote_data_cf() (in module aiida_fleur.data.fleurinp), 153
get_inputs_final_scf() (aiida_fleur.workflows.banddos.FleurBandDosWorkChain method), 173
get_inputs_scf() (aiida_fleur.workflows.dmi.FleurDMIWorkChain method), 185
get_inputs_scf() (aiida_fleur.workflows.eos.FleurEosWorkChain method), 175
get_inputs_scf() (aiida_fleur.workflows.mae.FleurMaeWorkChain method), 181
get_inputs_scf() (aiida_fleur.workflows.mae_conv.FleurMaeConvWorkChain method), 182
get_inputs_scf() (aiida_fleur.workflows.orbcontrol.FleurOrbControlWorkChain method), 186
get_inputs_scf() (aiida_fleur.workflows.relax.FleurRelaxWorkChain method), 177
get_inputs_scf() (aiida_fleur.workflows.ssdisp.FleurSSDispWorkChain method), 183
get_inputs_scf() (aiida_fleur.workflows.ssdisp_conv.FleurSSDispConvWorkChain method), 184
get_inputs_scf() (aiida_fleur.workflows.eos.FleurEosWorkChain method), 175
get_inputs_scf_no_ldau() (aiida_fleur.workflows.orbcontrol.FleurOrbControlWorkChain method), 186
generate_kpoints_mesh_from_kdensity() (in module aiida_fleur.tools.common_fleur_wf), 222
gen_kpointdata() (aiida_fleur.data.fleurinp.FleurinpData method), 150
gen_kpointdata() (in module aiida_fleur.data.fleurinp), 153
gen_kpointdata_ncf() (aiida_fleur.data.fleurinp.FleurinpData method), 150
get_layers() (in module aiida_fleur.tools.StructureData_util), 210
gen_linkname_outparams() (aiida_fleur.parsers.fleur.FleurParser method), 148
gen_linkname_outparams_complex() (aiida_fleur.parsers.fleur.FleurParser method), 148
get_mpi_proc() (in module aiida_fleur.tools.common_fleur_wf), 221
get_natoms_element() (in module aiida_fleur.tools.common_fleur_wf_util), 224
get_nkpts() (aiida_fleur.data.fleurinp.FleurinpData method), 150
get_nodes_from_group() (in module aiida_fleur.data.fleurinp.common_aiida), 219
get_para_from_group() (in module aiida_fleur.workflows.initial_cls), 179
get_parameterdata() (aiida_fleur.data.fleurinp.FleurinpData method), 150
get_parameterdata() (in module aiida_fleur.data.fleurinp), 153
get_parameterdata_ncf() (aiida_fleur.data.fleurinp.FleurinpData method), 150
get_ref_from_group() (in module aiida_fleur.workflows.initial_cls), 179
get_references() (aiida_fleur.workflows.initial_cls.FleurInitialCLSWorkChain method), 178
get_res() (aiida_fleur.workflows.scf.FleurScfWorkChain method), 172
get_results() (aiida_fleur.workflows.dmi.FleurDMIWorkChain method), 185
get_results() (aiida_fleur.workflows.mae.FleurMaeWorkChain method), 181
get_results() (aiida_fleur.workflows.mae_conv.FleurMaeConvWorkChain method), 182
get_results() (aiida_fleur.workflows.ssdisp.FleurSSDispWorkChain method), 183
get_results() (aiida_fleur.workflows.ssdisp_conv.FleurSSDispConvWorkChain method), 184
get_results_final_scf() (aiida_fleur.workflows.relax.FleurRelaxWorkChain method), 177
get_results_relax() (aiida_fleur.workflows.relax.FleurRelaxWorkChain method), 177
get_spacegroup() (in module aiida_fleur.data.fleurinp.common_aiida), 219
has_z_reflection() (in module aiida_fleur.tools.StructureData_util), 211
highest_unocc_valence() (in module aiida_fleur.tools.element_econfig_list), 218
import_extras() (in module aiida_fleur.data.fleurinpcommon_aiida), 219
inp_dict (aiida_fleur.data.fleurinp.FleurinpData property), 151
inp_version (aiida_fleur.data.fleurinp.FleurinpData property), 151
inp_gen_dict_set_mesh() (in module aiida_fleur.tools.common_fleur_wf_util), 224
inp_gen_needed() (aiida_fleur.workflows.orbcontrol.FleurOrbControlWorkChain method), 186
inpxml_changes() (in module aiida_fleur.data.fleurinpmodifier), 169
inspect_first() (aiida_fleur.workflows.eos.FleurEosWorkChain method), 175
inspect_fleur() (aiida_fleur.workflows.scf.FleurScfWorkChain method), 172
is_primitive() (in module aiida_fleur.data.fleurinpmodifier), 211
is_structure() (in module aiida_fleur.data.fleurinpmodifier), 211
load_inpxml() (aiida_fleur.data.fleurinp.FleurinpData method), 151
magnetic_slab_from_relaxed() (in module aiida_fleur.data.fleurinpcommon_aiida), 219
mark_atoms() (in module aiida_fleur.data.fleurinpcommon_aiida), 211
mark_fixed_atoms() (in module aiida_fleur.data.fleurinpcommon_aiida), 211
merge_parameter() (in module aiida_fleur.data.fleurinpmodifier), 170
module aiida_fleur.calculation.fleur, 148
aiida_fleur.calculation.fleurinputgen, 147
aiida_fleur.data.fleurinp, 149
aiida_fleur.data.fleurinpmodifier, 154
aiida_fleur.parsers.fleur, 148
aiida_fleur.parsers.fleur_inputgen, 148
aiida_fleur.tools.common_aiida, 219
aiida_fleur.tools.common_fleur_wf, 220
aiida_fleur.tools.common_fleur_wf_util, 223
aiida_fleur.tools.create_corehole, 216
aiida_fleur.tools.dict_util, 215
aiida_fleur.tools.element_econfig_list, 218
aiida_fleur.tools.extract_corelevels, 216
aiida_fleur.tools.io_routines, 220
aiida_fleur.tools.merge_parameter, 216
aiida_fleur.tools.read_cif_folder, 220
aiida_fleur.tools.StructureData_util, 206
aiida_fleur.tools.xml_aiida_modifiers, 214
aiida_fleur.workflows.banddos, 173
aiida_fleur.workflows.base_fleur, 171
aiida_fleur.workflows.cfcoefficient, 187
aiida_fleur.workflows.corehole, 179
aiida_fleur.workflows.dmi, 184
aiida_fleur.workflows.dos, 174
aiida_fleur.workflows.eos, 175
aiida_fleur.workflows.initial_cls, 177
aiida_fleur.workflows.mae, 181
aiida_fleur.workflows.mae_conv, 182
aiida_fleur.workflows.orbcontrol, 185
aiida_fleur.workflows.relax, 176
aiida_fleur.workflows.scf, 171
aiida_fleur.workflows.ssdisp, 183
aiida_fleur.workflows.ssdisp_conv, 184
move_atoms_incell() (in module aiida_fleur.tools.StructureData_util), 211
move_atoms_incell_wf() (in module aiida_fleur.tools.StructureData_util), 212
optimize_calc_options() (in module aiida_fleur.tools.common_fleur_wf), 222
P
parse() (aiida_fleur.parsers.fleur.FleurParser method), 149
parse() (aiida_fleur.parsers.fleur_inputgen.Fleur_inputgenParser method), 148
parse_relax_file() (in module aiida_fleur.parsers.fleur), 149
parse_state_card() (in module aiida_fleur.tools.extract_corelevels), 217
PARSER_INFO (aiida_fleur.data.fleurinp.FleurinpData property), 151
performance_extract_calcs() (in module aiida_fleur.tools.common_fleur_wf), 222
powerset() (in module aiida_fleur.tools.common_fleur_wf_util), 224
prepare_for_submission() (aiida_fleur.calculation.fleur.FleurCalculation method), 148
prepare_for_submission() (aiida_fleur.calculation.fleurinputgen.FleurinputgenCalculation method), 147
prepare_struc_corehole_wf() (in module aiida_fleur.workflows.corehole), 181
PROCESS
aiida_fleur-workflow-inputdict command line option, 205
aiida_fleur-workflow-res command line option, 206
Q
query_for_ref_structure() (in module aiida_fleur.workflows.initial_cls), 179
R
read_cif_folder() (in module aiida_fleur.tools.read_cif_folder), 220
reconstruct_cfcalculation() (in module aiida_fleur.workflows.cfcoefficient), 188
reconstruct_cfcoefficients() (in module aiida_fleur.workflows.cfcoefficient), 188
recursive_merge() (in module aiida_fleur.tools.dict_util), 215
rek_econ() (in module aiida_fleur.tools.element_econfig_list), 219
relax() (aiida_fleur.workflows.corehole.FleurCoreholeWorkChain method), 180
relax() (aiida_fleur.workflows.initial_cls.FleurInitialCLSWorkChain method), 178
relaxation_needed()

return_results()

replace_element()

return_results()

replace_elementf()

return_results()

replace_tag()

request_average_bond_length()

request_average_bond_length_store()

rescale()

rescale_nowf()

reset_straight_mixing()

return_results()

return_results()

return_results()

return_results()

return_results()

return_results()

return_results()

return_results()

return_results()

return_results()

return_results()

return_results()

return_results()

return_results()

return_results()

return_results()

return_results()

return_results()

return_results()

return_results()

save_mae_output_node()
save_output_node() (in module aiida_fleur.workflows.ssdisp_conv), 184

scf_needed() (aiida_fleur.workflows.banddos.FleurBandDosWorkChain method), 173

scf_needed() (aiida_fleur.workflows.dmi.FleurDMIWorkChain method), 185

scf_needed() (aiida_fleur.workflows.mae.FleurMaeWorkChain method), 182

scf_needed() (aiida_fleur.workflows.mae_conv.FleurMaeConvWorkChain method), 182

scf_no_ldau_needed() (aiida_fleur.workflows.orbcontrol.FleurOrbControlWorkChain method), 187

set_atomgroup() (aiida_fleur.data.fleurinpmodifier.FleurinpModifier method), 158

set_atomgroup_label() (aiida_fleur.data.fleurinpmodifier.FleurinpModifier method), 159

set_attrib_value() (aiida_fleur.data.fleurinpmodifier.FleurinpModifier method), 159

set_complex_tag() (aiida_fleur.data.fleurinpmodifier.FleurinpModifier method), 160

set_file() (aiida_fleur.data.fleurinp.FleurinpData method), 151

set_files() (aiida_fleur.data.fleurinp.FleurinpData method), 152

set_first_attrib_value() (aiida_fleur.data.fleurinpmodifier.FleurinpModifier method), 160

set_first_text() (aiida_fleur.data.fleurinp.FleurinpData method), 160

set_inpchanges() (aiida_fleur.data.fleurinpmodifier.FleurinpModifier method), 161

set_kpointsdata() (aiida_fleur.data.fleurinpmodifier.FleurinpModifier method), 163

set_kpointsdata_f() (in module aiida_fleur.tools.xml_aiida_modifiers), 214

set_kpointdata_f() (aiida_fleur.data.fleurinpmodifier.FleurinpModifier method), 164

set_kpointlist() (aiida_fleur.data.fleurinpmodifier.FleurinpModifier method), 162

set_kpointmesh() (aiida_fleur.data.fleurinpmodifier.FleurinpModifier method), 163

set_kpointpath() (aiida_fleur.data.fleurinpmodifier.FleurinpModifier method), 163

set_kpointsdata() (aiida_fleur.data.fleurinpmodifier.FleurinpModifier method), 163

set_nkpts() (aiida_fleur.data.fleurinpmodifier.FleurinpModifier method), 164

set_nmmpmat() (aiida_fleur.data.fleurinpmodifier.FleurinpModifier method), 164

set_simple_tag() (aiida_fleur.data.fleurinpmodifier.FleurinpModifier method), 165

set_species() (aiida_fleur.data.fleurinpmodifier.FleurinpModifier method), 165

set_species_label() (aiida_fleur.data.fleurinpmodifier.FleurinpModifier method), 165

set_text() (aiida_fleur.data.fleurinpmodifier.FleurinpModifier method), 165

set_xcfunctional() (aiida_fleur.data.fleurinpmodifier.FleurinpModifier method), 166

shift_value() (aiida_fleur.data.fleurinpmodifier.FleurinpModifier method), 166

shift_value_species_label() (aiida_fleur.data.fleurinpmodifier.FleurinpModifier method), 167

show() (aiida_fleur.data.fleurinpmodifier.FleurinpModifier method), 167

simplify_kind_name() (in module aiida_fleur.tools.StructureData_util), 213

sort_atoms_z_value() (in module aiida_fleur.tools.StructureData_util), 214

start() (aiida_fleur.workflows.banddos.FleurBandDosWorkChain method), 173

start() (aiida_fleur.workflows.banddos.FleurBandDosWorkChain method), 174

start() (aiida_fleur.workflows.banddos.FleurBandDosWorkChain method), 188

start() (aiida_fleur.workflows.dmi.FleurDMIWorkChain method), 177

start() (aiida_fleur.workflows.dos.fleur_dos_wc method), 174

start() (aiida_fleur.workflows.dos.fleur_dos_wc method), 175

start() (aiida_fleur.workflows.eos.FleurEosWorkChain method), 182

start() (aiida_fleur.workflows.mae.FleurMaeWorkChain method), 182

start() (aiida_fleur.workflows.mae_conv.FleurMaeConvWorkChain method), 182

start() (aiida_fleur.workflows.relax.FleurRelaxWorkChain method), 177

start() (aiida_fleur.workflows.relax.FleurRelaxWorkChain method), 177

start() (aiida_fleur.workflows.scf.FleurScfWorkChain method), 177

should_relax() (aiida_fleur.workflows.relax.FleurRelaxWorkChain method), 177

should_run_final_scf() (aiida_fleur.workflows.relax.FleurRelaxWorkChain method), 177

set_atomgroup() (aiida_fleur.data.fleurinpmodifier.FleurinpModifier method), 158

set_file() (aiida_fleur.data.fleurinp.FleurinpData method), 151

set_files() (aiida_fleur.data.fleurinp.FleurinpData method), 152

Index 259
AiiDa-FLEUR Documentation, Release 2.0.0

W

write_inpgen_file_aiida_struct() (in module aii-da_fleur.calculation.fleurinputgen), 147

X

xml_create_tag() (aiida_fleur.data.fleurinpmodifier.FleurinpModifier method), 168

xml_delete_att() (aiida_fleur.data.fleurinpmodifier.FleurinpModifier method), 168

xml_delete_tag() (aiida_fleur.data.fleurinpmodifier.FleurinpModifier method), 168

xml_replace_tag() (aiida_fleur.data.fleurinpmodifier.FleurinpModifier method), 169

xml_set_attrib_value_no_create() (aiida_fleur.data.fleurinpmodifier.FleurinpModifier method), 169

xml_set_text_no_create() (aiida_fleur.data.fleurinpmodifier.FleurinpModifier method), 169

U

ucell_to_atompr() (in module aiida_fleur.tools.common_fleur_wf), 225

undo() (aiida_fleur.data.fleurinpmodifier.FleurinpModifier method), 168

V

validate() (aiida_fleur.data.fleurinpmodifier.FleurinpModifier method), 168

validate_input() (aiida_fleur.workflows.cfcoeff.FleurCFCoeffWorkChain method), 188

validate_input() (aiida_fleur.workflows.orbcontrol.FleurOrbControlWorkChain method), 187

validate_input() (aiida_fleur.workflows.scf.FleurScfWorkChain method), 172

validate_inputs() (aiida_fleur.workflows.base_fleur.FleurBaseWorkChain method), 171

task_list (aiida_fleur.data.fleurinpmodifier.FleurinpModifier property), 168

test_and_get_codename() (in module aiida_fleur.tools.common_fleur_wf), 222

260 Index