AiiDA Siesta Plugin Documentation

Release 1.4.0

V.M. Garcia-Suarez, A. Garcia, E. Bosoni, V. Dikan, P. Febrer

Jul 16, 2022

Contents

The aiida-siesta python package interfaces the SIESTA DFT code (https://siesta-project.org/siesta/) with the AiiDA framework (http://www.aiida.net). The package contains: plugins for SIESTA itself and for other utility programs, new data structures, and basic workflows. It is distributed under the MIT license and available from (https://github.com/siesta-project/aiida_siesta_plugin). If you use this package, please cite J. Chem. Phys. **152**, 204108 (2020) (https://doi.org/10.1063/5.0005077).

CHAPTER 1

Acknowledgments:

The Siesta input plugin was originally developed by Victor M. Garcia-Suarez.

Alberto Garcia further improved the Siesta input plugin and wrote the parser for Siesta and the STM plugin.

Emanuele Bosoni contributed the band-structure support for the Siesta plugin.

Vladimir Dikan and Alberto Garcia developed the workflows and refined the architecture of the package.

Vladimir Dikan and Emanuele Bosoni ported the plugin and the base workflow to AiiDA 1.0. Alberto Garcia futher refined the 1.0-compatible functionality.

From November 2019 to May 2022, Emanuele Bosoni was in charge of the code's development and maintenance, under the supervision of Alberto Garcia.

Pol Febrer contributed the SiestaIterator and SiestaConverger workflows, including the underline abstract classes system.

This work is supported by the MaX European Centre of Excellence funded by the Horizon 2020 INFRAEDI-2018-1 program, Grant No. 824143, and by the INTERSECT (Interoperable material-to-device simulation box for disruptive electronics) project, funded by Horizon 2020 under grant agreement No 814487, as well as by the Spanish MINECO (projects FIS2012-37549-C05-05 and FIS2015-64886-C5-4-P)

We thank the AiiDA team, who are also supported by the MARVEL National Centre for Competency in Research funded by the Swiss National Science Foundation


INTERSECT


MINISTERIO
DE ECONOMÍA, INDUSTRIA
Y COMPETITIVIDAD
AGENCIA
ESTATAL DE
INVESTIGACIÓN


MARVEL
NATIONAL CENTRE OF COMPETENCE IN RESEARCH

CHAPTER 2

Contents:

2.1 Installation

2.1.1 Installation and dependences

It would be a good idea to create and switch to a new python virtual environment before the installation.

The latest release of the package can be obtained simply with:

```
pip install aiida-siesta
```

In this case, make sure to refer to the appropriate documentation part ("stable", not "latest").

Because the package is under development, in order to enjoy the most recent features one can clone the github repository (https://github.com/siesta-project/aiida_siesta_plugin) and install from the top level of the plugin directory with:

```
pip install -e .
```

As a pre-requisite, both commands above will install an appropriate version of the `aiida-core` python framework, if this is not already installed. In case of a fresh install of `aiida-core`, follow the AiiDA documentation in order to configure aiida.

Important: In any case, do not forget to run the following commands after the installation:

```
reentry scan -r aiida
verdi daemon restart
```

Since version 1.2.0, `aiida-siesta` also depends on `sisl` (https://github.com/zerothi/sisl). For the moment `sisl` is used only to facilitate the management of basis orbitals, but a closer integration among the two packages is foreseen in the future.

2.1.1.1 For developers

This plugin is open-source and contributions are welcomed. Before starting the development, the following steps are suggested:

- After cloning from github, install with `pip install .[dev]`. This will download all the tools for testing.
- Install pre-commit hooks. This will "force" to follow some python standards we require. In fact, the hooks will impede to commit unless the required standards are met.
- Make sure to run all the tests (simply `pytest test/` from the main folder of the package) to make sure the contribution is not breaking any part of the code. Ideally, write tests for the new part implemented.

2.2 Calculation plugins

2.2.1 Calculations

This section contains the documentation for the calculations plugins distributed in `aiida-siesta`. They are the fundamental blocks that enable to run through AiiDA some executable of the Siesta package, meaning the siesta code itself and some post-processing tools. For each calculation, we explain the inputs selection, the submission command and the returned outputs. From the AiiDA prospective, we describe here the functionalities of both the CalcJob class and the associated parser.

2.2.1.1 Siesta calculations

Description

A plugin for Siesta main code. It allows to prepare, submit and retrieve the results of a standard siesta calculation, including support for the parsing of the electronic bands and the output geometry of a relaxation. It is implemented in the class **SiestaCalculation**.

Supported Siesta versions

At least 4.0.1 of the 4.0 series, 4.1-b3 of the 4.1 series and the MaX-1.0 release, which can be found in the development platform (https://gitlab.com/siesta-project/siesta). For more up to date info on compatibility, please check the wiki.

Inputs

Some examples are referenced in the following list. They are located in the folder *aiida_siesta/examples/plugins/siesta*.

- **code**, class `Code`, *Mandatory*

  A code object linked to a Siesta executable. If you setup the code `Siesta4.0.1` on machine `kelvin` following the aiida guidelines, then the code is selected in this way:

```
codename = 'Siesta4.0.1@kelvin'
from aiida.orm import Code
code = Code.get_from_string(codename)
```

- **structure**, class `StructureData`, *Mandatory*

  A structure. Siesta employs "species labels" to implement special conditions (such as basis set characteristics) for specific atoms (e.g., surface atoms might have a richer basis set). This is implemented through the `name` attribute of the Site objects. For example:

```
from aiida.orm import StructureData

alat = 15. # angstrom
cell = [[alat, 0., 0.,],
  [0., alat, 0.,],
  [0., 0., alat,],
 ]

 # Benzene molecule with a special carbon atom
 s = StructureData(cell=cell)
 s.append_atom(position=(0.000,0.000,0.468),symbols=['H'])
 s.append_atom(position=(0.000,0.000,1.620),symbols=['C'])
 s.append_atom(position=(0.000,-2.233,1.754),symbols=['H'])
 s.append_atom(position=(0.000,2.233,1.754),symbols=['H'])
 s.append_atom(position=(0.000,-1.225,2.327),symbols='C',name="Cred")
 s.append_atom(position=(0.000,1.225,2.327),symbols=['C'])
 s.append_atom(position=(0.000,-1.225,3.737),symbols=['C'])
 s.append_atom(position=(0.000,1.225,3.737),symbols=['C'])
 s.append_atom(position=(0.000,-2.233,4.311),symbols=['H'])
 s.append_atom(position=(0.000,2.233,4.311),symbols=['H'])
 s.append_atom(position=(0.000,0.000,4.442),symbols=['C'])
 s.append_atom(position=(0.000,0.000,5.604),symbols=['H'])
```

  The class `StructureData` uses Angstrom as internal units, the cell and atom positions must be specified in Angstrom.

  The `StructureData` can also import ase structures or pymatgen structures. These two tools can be used to load structure from files. See example *example_cif_bands.py*.

Note: Siesta can handle *ghost atoms*, carrying only extra orbitals, to increase the variational freedom. In the AiiDA plugin, these ghost atoms are specified in the **basis** dictionary (see below), and they should **not** be part of the input StructureData object.

- **parameters**, class `Dict`, *Mandatory*

  A dictionary with scalar fdf variables and blocks, which are the basic elements of any Siesta input file. A given Siesta fdf file can be cast almost directly into this dictionary form, except that some items are blocked. The blocked keywords include the system information (`system-label`, `system-name`) and all the structure information as they will be automatically set by Aiida. Moreover, the keyword `dm-use-save-dm` is not allowed (the restart options are explained *here*) together with the keyword `geometry-must-converge` (set to True by default for each calculation with variable geometry). Also the `max-walltime` is blocked since it is set by the plugin to be equal to the `max_wallclock_seconds` passed in the *computational resources*. This should prevent the calculation to be terminated by the scheduler. In case a siesta max time smaller than the `max_wallclock_seconds` is required, it is suggested to increase the `max-walltime-slack` value. Finally, all the `pao` and `optical` options must be avoided here, because they belong to the **basis** and **optical** inputs respectively (see following of the list). Any units are specified for now as part of the value string. Blocks are entered by using an appropriate key and Python's multiline string constructor. For example:

```
from aiida.orm import Dict

```

(continues on next page)

(continued from previous page)

```
parameters = Dict(dict={
  "mesh-cutoff": "200 Ry",
  "dm-tolerance": "0.0001",
  "%block example-block":
    """
    first line
    second line
    %endblock example-block""",
})
```

Note that Siesta fdf keywords allow '.', '-', (or nothing) as internal separators. AiiDA does not allow the use of '.' in nodes to be inserted in the database, so it should not be used in the input script (or removed before assigning the dictionary to the Dict instance). For legibility, a single dash ('-') is suggested, as in the examples above. Moreover, because the parameters are passed through a python dictionary, if, by mistake, the user passes the same keyword two (or more) times, only the last specification will be considered. For instance:

```
parameters = Dict(dict={
  "mesh-cutoff": "200 Ry",
  "mesh-cutoff": "300 Ry",
  })
```

will set a `mesh-cutoff` of *300 Ry*. This is the opposite respect to what is done in the Siesta code, where the first assignment is the selected one. Please note that this applies also to keywords that correspond to the same fdf variable. For instance:

```
parameters = Dict(dict={
  "mesh-cutoff": "200 Ry",
  "Mesh-Cut-off": "300 Ry",
  })
```

will run a calculation with `mesh-cutoff` equal to *300 Ry*, whithout raising any error.

- **basis**, class `Dict`, *Optional*

  A dictionary specifically intended for basis set information. It follows the same structure as the **parameters** element, including the allowed use of fdf-block items. This raw interface allows a direct translation of the myriad basis-set options supported by the Siesta program. In future we might have a more structured input for basis-set information. An example:

```
from aiida.orm import Dict

basis_dict = {
'pao-basistype':'split',
'pao-splitnorm': 0.150,
'pao-energyshift': '0.020 Ry',
'%block pao-basis-sizes':
"""
C    SZP
Cred SZ
H    SZP
%endblock pao-basis-sizes""",
}

basis = Dict(dict=basis_dict)
```

  In case no basis is set (and no **ions** is passed in input), the Siesta calculation will not include any basis specification and it will run with the default basis: DZP plus (many) other defaults.

The **basis** dictionary also accepts a special key called `floating_sites` that can be used to specify the location and kind of *ghost atoms*, or sites carrying only floating orbitals. The associated value must be a list of dictionaries and each dictionary must include as keys at least the `name`, `symbols` and `position` of the floating site. An example is:

```
basis = Dict(dict={
        'floating_sites': [{"name":'Si_bond', "symbols":'Si', "position":(0.125,␣
↪0.125, 0.125)}],
        '%block pao-basis-sizes':
        """
        Si_bond SZ
        %endblock pao-basis-sizes""",
})
```

The "position" must be specified in Angstrom. A "name" that corresponds to an existing atomic site is forbidden. As shown in the example, in case a basis specification has to be added for one or more `floating_sites`, it must be included in the basis dictionary in the same way as those for any other atomic kinds. Please look at the examples *example_ghost.py* and *example_ghost_relax.py* for a practical example.

- **pseudos**, input namespace of class `PsfData` OR class `PsmlData`, *Optional*

  This input is mandatory except if the **ions** input is set (see below).

  This inputs exploits the functionalities of the *PsfData <aiida_pseudo.data.pseudo.psf.PsfData>* and *PsmlData <aiida_pseudo.data.pseudo.psml.PsmlData>* of the aiida-pseudo package.

  One pseudopotential file per atomic element is required. Several species (in the Siesta sense, which allows the same element to be treated differently according to its environment) can share the same pseudopotential. For the example above:

```
import os
from aiida_pseudo.data.pseudo.psf import PsfData

pseudo_file_to_species_map = [ ("C.psf", ['C', 'Cred']),("H.psf", ['H'])]
pseudos_dict = {}
for fname, kinds, in pseudo_file_to_species_map:
      absname = os.path.realpath(os.path.join("path/to/file",fname))
      pseudo = PsfData.get_or_create(absname)
      for j in kinds:
            pseudos_dict[j]=pseudo
```

  Alternatively, a pseudo for every atomic species can be set from a family of pseudopotentials:

```
from aiida.orm import Group
family = Group.get(label=FAM_NAME)
pseudos = family.get_pseudos(structure=s)
```

  where `s` is a *StructureData <aiida.orm.StructureData>* object and `FAM_NAME` is the name of the pseudopotentials family, that must be installed in the database.

  The simplest way to install a pseudo family is through the command:

```
aiida-pseudo install family /PATH/TO/FOLDER/ FAM_NAME -P pseudo.psf  #or pseudo.
↪psml
```

  where `/PATH/TO/FOLDER/` is a folder containing the pseudos. The aiida-pseudo package allows more sophisticated ways of creating pseudo family, for instance downloading the pseudos directly from a url or online repository (PseudoDojo for instance). Please refer to the corresponding documentation for more details.

  For a practical example, look at *example_psf_family.py*.

- **ions**, input namespace of class `IonData`, *Optional*

  The class *IonData <aiida_siesta.data.ion.IonData>* has been implemented along the lines of the *PsfData* class to carry information on the entity that in siesta terminology is called "ion", and that packages the set of basis orbitals and KB projectors for a given species. It contains also some extra metadata. The class *IonData* stores ".ion.xml" files and it also provides a method *get_content_ascii_format* that translates the content of an ".ion.xml" into an ".ion" file format, which is the only one currently accepted by Siesta.

  When this input is present, the plugin takes care of coping in the running folder the ".ion" files and set the "user_basis" siesta keyword to True. Moreover, when this input is present, **pseudos** and **basis** inputs are ignored (except possible *floating_orbitals* defined in the basis).

  One ion file per atomic element is required and must be passed to the calculation in a way similar to the pseudos. For instance:

```
import os
from aiida_siesta.data.ion import IonData

ion_file_to_species_map = [ ("C.ion", ['C']),("H.ion", ['H'])]
ions_dict = {}
for fname, kinds, in ion_file_to_species_map:
      absname = os.path.realpath(os.path.join("path/to/file",fname))
      ion = IonData.get_or_create(absname)
      for j in kinds:
              ions_dict[j]=ion
```

  The *example_ion.py* can be analyzed to better understand the use of **ions** inputs.

- **kpoints**, class `KpointsData`, *Optional*

  Reciprocal space points for the full sampling of the BZ during the self-consistent-field iteration. It must be given in mesh form. There is no support yet for Siesta's "kgrid-cutoff" keyword:

```
from aiida.orm import KpointsData
kpoints=KpointsData()
kp_mesh = 5
mesh_displ = 0.5 #optional
kpoints.set_kpoints_mesh([kp_mesh,kp_mesh,kp_mesh],[mesh_displ,mesh_displ,mesh_
↪displ])
```

  The class *KpointsData <aiida.orm.KpointsData>* also implements the methods `set_cell_from_structure` and `set_kpoints_mesh_from_density` that allow to obtain a uniform mesh automatically.

  If this node is not present, only the Gamma point is used for sampling.

- **bandskpoints**, class `KpointsData`, *Optional*

  Reciprocal space points for the calculation of bands. The **full list of kpoints must be passed** to `bandskpoints` and they must be in **units of the reciprocal lattice vectors**. There is no obligation to set the cell in `bandskpoints`, however this might be useful in order to exploit the functionality of the class `KpointsData`. If set, the cell must be the same of the input **structure**. Some examples on how to pass the kpoints are the following.

  One can manually listing a set of isolated kpoints:

```
from aiida.orm import KpointsData
bandskpoints=KpointsData()
kpp = [(0.1,  0.1, 0.1), (0.5,  0.5, 0.5), (0., 0., 0.)]
bandskpoints.set_kpoints(kpp)
```

In this case the Siesta input will use the "BandPoints" block.

Alternatively (recommended) the high-symmetry path associated to the structure under investigation can be automatically generated through the aiida tool `get_explicit_kpoints_path`. Here how to use it:

```
from aiida.orm import KpointsData
bandskpoints=KpointsData()
from aiida.tools import get_explicit_kpoints_path
symmpath_parameters = Dict(dict={'reference_distance': 0.02})
kpresult = get_explicit_kpoints_path(s, **symmpath_parameters.get_dict())
bandskpoints = kpresult['explicit_kpoints']
```

Where 's' in the input structure and `reference_distance` is the distance between two subsequent kpoints. In this case the block "BandLines" is set in the Siesta calculation.

Warning: "SeeK-path" might modify the structure to follow particular conventions and the generated kpoints might only apply on the internally generated 'primitive_structure' and not on the input structure that was provided. The correct way to use this tool is to use the generated 'primitive_structure' also for the Siesta calculation:

```
structure = kpresult['primitive_structure']
```

Warning: As we use the initial structure cell in order to obtain the kpoints path, it is very risky to apply this method when also a relaxation of the cell is performed! The cell might relax in a different symmetry resulting in a wrong path for the bands. Consider to use the *BandGapWorkChain* if a relaxation is needed before computing the bands.

Note: The `get_explicit_kpoints_path` make use of "SeeK-path". Please cite the HPKOT paper if you use this tool. "SeeK-path" is a external utility, not a requirement for aiida-core, therefore it is not available by default. It can be easily installed using `pip install seekpath`. "SeeK-path" allows to determine canonical unit cells and k-point information in an easy way. For more general information, refer to the SeeK-path documentation.

The final option covers the situation when one needs to calculate the bands on a specific path (and maybe needs to maintain a specific convention for the structure). The full list of kpoints must be passed and, very importantly, labels must be set for the high symmetry points! This is essential for the correct set up of the "BandLines" in Siesta. External tolls can be used to create equidistant points, whithin aiida the following (very involved) option is available:

```
from aiida.orm import KpointsData
bandskpoints=KpointsData()
from aiida.tools.data.array.kpoints.legacy import get_explicit_kpoints_path as␣
↪legacy_path
kpp = [('A',  (0.500,  0.250, 0.750), 'B', (0.500,  0.500, 0.500), 40),
('B', (0.500,  0.500, 0.500), 'C', (0., 0., 0.), 40)]
tmp=legacy_path(kpp)
bandskpoints.set_kpoints(tmp[3])
bandskpoints.labels=tmp[4]
```

The legacy `get_explicit_kpoints_path` shares only the name with the function in `aiida.tools`, but it is very different in scope.

The full list of cases can be explored looking at the example example_bands.py

> **Warning:** The implementation relies on the correct description of the labels in the class `KpointsData`. Refrain from improper use of `bandskpoints.labels` and follow the the instructions described above. An incorrect use of the labels might result in an incorrect parsing of the bands.

If the keyword node **bandskpoints** is not present, no band structure is computed.

- **optical**, class `Dict`, *Optional*

  This is the dedicated input to specify Siesta's keywords related to the calculation of optical properties. It is a simple dictionary and it follows the same concept of the **parameters** and **basis** inputs, including the requirements for the use of fdf-block items. It is mandatory to specify a "%block optical-mesh". All the other optical inputs are optional. If not already specified by the user, the "optical-calculation" keyword will automatically set to True by the plugin.

- **lua**, input namespace, *Optional*

  This input namespace allows control on the LUA interface to SIESTA. The user should remember that to enable the LUA interface, it is suggested to compile SIESTA with `flook` and to use the `flos` library (flos documentation). Follow the SIESTA manual for complete instructions. Since this option also requires the definition of the `LUA_PATH`, an *additional step* must be done before submission.

  This input namespace accepts the following elements:

```
spec.input('lua.script', valid_type=orm.SinglefileData, required=False)
spec.input('lua.parameters', valid_type=orm.Dict, required=False)
spec.input('lua.input_files', valid_type=orm.FolderData, required=False)
spec.input('lua.retrieve_list', valid_type=orm.List, required=False)
```

- **lua.script** is a Lua script implementing a specific functionality, and possibly being able to set its own operational parameters. For example, the LBFGS geometry relaxation algorithm, or the NEB path-optimization scheme, can be implemented in Lua. See the examples provided.
- **lua.parameters** is a dictionary containing the operational parameters for the script. For example, it can set the tolerance to be used in the script, or the value of the 'spring constant' in NEB simulations.
- **lua.input_files** is a set of auxiliary files packaged in a FolderData object. For example, the initial set of images for a NEB calculation.
- **lua.retrieve_list** contains a list of the files produced by the operation of the Lua script that need to be retrieved. They should be parsed by functionality-specific modules in client workchains.

- **settings**, class `Dict` , *Optional*

  An optional dictionary that activates non-default operations. For a list of possible values to pass, see the section on *advanced features*.

- **parent_calc_folder**, class `RemoteData` , *Optional*

  Optional port used to activate the *restart features*.

Submitting the calculation

Once all the inputs above are set, the subsequent step consists in passing them to the calculation class and run/submit it.

First, the Siesta calculation class is loaded:

```
from aiida_siesta.calculations.siesta import SiestaCalculation
builder = SiestaCalculation.get_builder()
```

The inputs (defined as in the previous section) are passed to the builder:

```
builder.code = code
builder.structure = structure
builder.parameters = parameters
builder.pseudos = pseudos_dict   #or builder.ions = ...
builder.basis = basis
builder.kpoints = kpoints
builder.bandskpoints = bandskpoints
```

Finally the resources for the calculation must be set, for instance:

```
builder.metadata.options.resources = {'num_machines': 1}
builder.metadata.options.max_wallclock_seconds = 1800
```

In case of LUA calculations, the `LUA_PATH` must be defined. To do so:

```
builder.metadata.options.environment_variables = {"LUA_PATH":"/flos_path/?.lua;/flos_
↪path/?/init.lua;;;"}
```

where `flos_path` is the path to the flos library repository (in the computer where SIESTA will run). Please note that the explicit path must be used due to a problem of aiida (https://github.com/aiidateam/aiida-core/issues/4836). This means that, for instance, the command `export LUA_PATH="$HOME/flos/?.lua;$HOME/flos/?/init.lua;$LUA_PATH;;"` suggested in the flos documentation must be substitute with explicit path of `$HOME`.

Optionally, label and description:

```
builder.metadata.label = 'My generic title'
builder.metadata.description 'My more detailed description'
```

To run the calculation in an interactive way:

```
from aiida.engine import run
results = run(builder)
```

Here the results variable will contain a dictionary containing all the nodes that were produced as output.

Another option is to submit it to the daemon:

```
from aiida.engine import submit
calc = submit(builder)
```

In this case, calc is the calculation node and not the results dictionary.

Note: In order to inspect the inputs created by AiiDA without actually running the calculation, we can perform a dry run of the submission process:

```
builder.metadata.dry_run = True
builder.metadata.store_provenance = False
```

This will create the input files, that are available for inspection.

Note: The use of the builder makes the process more intuitive, but it is not mandatory. The inputs can be provided as keywords argument when you launch the calculation, passing the calculation class as the first argument:

```
run(SiestaCalculation, structure=s, pseudos=pseudos, kpoints = kpoints, ...)
```

same syntax for the command `submit`.

A large set of examples covering some standard cases are in the folder *aiida_siesta/examples/plugins/siesta*. They can be run with:

```
runaiida example_name.py {--send, --dont-send} code@computer
```

The parameter `--dont-send` will activate the "dry run" option. In that case a test folder (*submit_test*) will be created, containing all the files that aiida generates automatically. The parameter `--send` will submit the example to the daemon. One of the two options needs to be present to run the script. The second argument contains the name of the code (`code@computer`) to use in the calculation. It must be a previously set up code, corresponding to a siesta executable.

Outputs

There are several output nodes that can be created by the plugin, according to the calculation details. All output nodes can be accessed with the `calculation.outputs` method.

- **output_parameters** `Dict`

  A dictionary with metadata, scalar result values, a warnings list, and possibly a timing section. Units are specified by means of an extra item with '_units' appended to the key:


```
{
  "siesta:Version": "siesta-4.0.2",
  "E_Fermi": -3.24,
  "E_Fermi_units": "eV",
  "FreeE": -6656.2343,
  "FreeE_units": "eV",
  "E_KS": -6656.2343,
  "E_KS_units": 'eV',
  "global_time": 55.213,
  "timing_decomposition": {
    "compute_DM": 33.208,
    "nlefsm-1": 0.582,
    "nlefsm-2": 0.045,
    "post-SCF": 2.556,
    "setup_H": 16.531,
    "setup_H0": 2.351,
    "siesta": 55.213,
    "state_init": 0.171
  },
  "warnings": [ "INFO: Job Completed"]
}
```


  The scalar quantities included are, currently, the Kohn-Sham (`E_KS`), Free (`FreeE`), Band (`Ebs`), and Fermi (`E_Fermi`) energies, and the total spin (`stot`). These are converted to `float`. The other quantities are or type `str`.

  The timing information (if present), includes the global walltime in seconds, and a decomposition by sections of the code. Most relevant are typically the `compute_DM` and `setup_H` sections.

The `warnings` list contains program messages, labeled as "INFO", "WARNING", or "FATAL", read directly from a *MESSAGES* file produced by Siesta, which include items from the execution of the program and also a possible 'out of time' condition. This is implemented by passing to the program the wallclock time specified in the script, and checking at each scf step for the walltime consumed. This `warnings` list can be examined by the parser itself to raise an exception in the "FATAL" case.

- **forces_and_stress** `ArrayData`

  Contains the final forces (*eV/Angstrom*) and stresses (*Ry/Angstrom^3*) in array form. To access their values:

```
forces_and_stress.get_array("forces")
forces_and_stress.get_array("stress")
```

- **output_structure** `StructureData`

  Present only if the calculation is moving the ions. Cell and ionic positions refer to the last configuration.

- **bands**, `BandsData`

  Present only if a band calculation is requested (signaled by the presence of a **bandskpoints** input node of class *KpointsData <aiida.orm.KpointsData>*). It contains an array with the list of electronic energies (in *eV*) for every kpoint. For spin-polarized calculations, there is an extra dimension for spin. In this class also the full list of kpoints is stored and they are in units of *1/Angstrom*. Therefore a direct comparison with the Siesta output SystLabel.bands is possible only after the conversion of *Angstrom* to *Bohr*. The bands are not rescaled by the Fermi energy. Tools for the generation of files that can be easly plot are available through `bands.export`.

- **optical_eps2** `ArrayData`

  Array containing the imaginary part of the dielectric function (epsilon_2) versus energy (*eV*). To access the values:

```
optical_eps2.get_array("e_eps2")
```

- **ions**, `IonData`

  Instances of *IonData* can be used as inputs of a `SiestaCalculation`, meaning `aiida_siesta` supports the use of pre-packaged information in ".ion" files. However, most of the time, pseudos and basis specifications are given separately for a siesta run, and the basis generation makes use of internal siesta algorithms that translate high-level definitions (basis-sizes, split-norm, ...) into the actual basis orbitals. In these cases siesta produces an ".ion.xml" file for each species in the structure. These files are parsed and stored into *IonData* instances that can be then easily reused in subsequent calculations. From *IonData* instances also the explicit orbitals of the basis can be obtained. One **ions** for each species is created and they will be output with the name `ions_El` where `El` is the label of the species.

- **remote_folder**, `RemoteData`

  The working remote folder for the last calculation executed.

- **retrieved**, `RemoteData`

  The local folder with the retrieved files.

No trajectories have been implemented yet.

Errors

Errors during the parsing stage are reported in the log of the calculation (accessible with the `verdi process report` command). Moreover, they are stored in the **output_parameters** node under the key `warnings`.

Restarts

A restarting capability is implemented through the optional input **parent_calc_folder**, `RemoteData`, which represents the remote scratch folder (**remote_folder** output) of a previous calculation.

The density-matrix file is copied from the old calculation scratch folder to the new calculation's one.

This approach enables continuation of runs which have failed due to lack of time or insufficient convergence in the allotted number of steps.

An informative example is *example_restart.py* in the folder *aiida_siesta/examples/plugins/siesta*.

Additional advanced features

While the input link with name **parameters** is used for the main Siesta options (as would be given in an fdf file), additional settings 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.

The keys of the settings dictionary are internally converted to uppercase by the plugin.

Adding command-line options

If you want to add command-line options to the executable (particularly relevant e.g. to tune the parallelization level), you can pass each option as a string in a list, as follows:

```
settings_dict = {
    'cmdline': ['-option1', '-option2'],
}
builder.settings = Dict(dict=settings_dict)
```

Note that very few user-level comand-line options (besides those already inserted by AiiDA for MPI operation) are currently implemented.

Retrieving more files

If you know that your calculation is producing additional files that you want to retrieve (and preserve in the AiiDA repository), you can add those files as a list as follows:

```
settings_dict = {
  'additional_retrieve_list': ['aiida.EIG', 'aiida.ORB_INDX'],
}
 builder.settings = Dict(dict=settings_dict)
```

See for example *example_ldos.py* in *aiida_siesta/examples/plugins/siesta*. The files can then be accesed through the output **retrieved** and its methods `get_object` and `get_object_content`.

2.2.1.2 STM calculations

Description

A plugin for *Util/plstm* of the Siesta distribution, a tool to simulate STM images. The code plstm is able to process the .LDOS file produced by Siesta. The .LDOS file contains informations on the local density of states (LDOS) in an

energy window. In the Tersoff-Hamann approximation, the LDOS can be used as a proxy for the simulation of STM experiments. This plugin requires in input the AiiDA folder where the .LDOS folder was generated and few other parameters (see Inputs section). It produces an array that can be plotted to obtain the STM images. The plugin is implemented in the class **STMCalculation**.

Supported Siesta versions

At least 4.0.1 of the 4.0 series, 4.1-b3 of the 4.1 series and the MaX-1.0 release, which can be found in the development platform (https://gitlab.com/siesta-project/siesta). For more up to date info on compatibility, please check the wiki.

Inputs

Some examples are referenced in the following list. They are located in the folder *aiida_siesta/examples/plugins/stm*.

- **code**, class `Code`, *Mandatory*

  A code object linked to a plstm executable. If you setup the code `plstm1` on machine `kelvin` following the aiida guidelines, then the code is selected in this way:

```
codename = 'plstm1@kelvin'
from aiida.orm import Code
code = Code.get_from_string(codename)
```

- **mode**, class `Str`, *Mandatory*

  Allowed values are `constant-height` or `constant-current`, corresponding to the two operation modes of the STM that are supported by the *plstm* code. Examples for both modes are presented in the example folder.

- **value**, class `Float`, *Mandatory*

  The value of height or current at which the user wants to simulate the STM. The height must be expressed in *Angstrom*, the current in *e/bohr**3*.

- **ldos_folder**, class `RemoteData`, *Mandatory*

  The parent folder of a previous Siesta calculation in which the .LDOS file was generated. To have more information on how to produce the .LDOS file, one can refer to the example *aiida_siesta/examples/plugins/siesta/example_ldos.py*. Please note that the **ldos_folder** must be on the same machine on which the STM analysis is performed. In other words, the input **code** must be installed on the same machine where the **ldos_folder** resides. This is a limitation of AiiDA that can not copy between different computers, but it is also required by *plstm* itself, as the .LDOS file is produced in an unformatted way.

- **spin_option**, class `Str`, *Optional*

  Input port that allows the selection of the spin options offered by *plstm*. It follows the same syntax of the code. The value "q" selects a total charge analysis. The value "s" selects the total spin magnitude analyisis (only available if the parent Siesta calculation is spin polarized). Finally, the values "x", "y" or "z" indicate a separate analysis of one the three spin components (only available if the parent Siesta calculation is performed with non-collinear options). If the port is not specified the default "q" option is activated.

- **settings**, class `Str`, *Optional*

  A port **settings** is available to activate some advanced features. For instance the modification of the command line instructions and the addition of files to retreave. For more info, the corresponding section of the Standard Siesta Plugin can be seen *here*.

Submitting the calculation

The submission of any CalcJob of AiiDA always follows the same schema. Therefore, to understand how to submit a STM calculation, it is sufficient to follow the explanation of the corresponding section of the Standard Siesta Plugin. The only change is to import the correct plugin:

```
from aiida_siesta.calculations.stm import STMCalculation
builder = STMCalculation.get_builder()
```

and, of course, to define the correct inputs allowed by **STMCalculation** (previous section).

Outputs

- **stm_array** `ArrayData`

  A collection of three 2D arrays (*grid_X*, *grid_Y*, *STM*) holding the section or topography information. They follow the *meshgrid* convention in Numpy. A heat-map plot can be generated with the *get_stm_image.py* script in the repository of examples.

- **output_parameters** `Dict`

  At this point, it constains only the parser information and the name of the retrieved file where the STM info were stored.

Errors

Errors during the parsing stage are reported in the log of the calculation (accessible with the `verdi process report` command).

2.3 Utilities

2.3.1 Utils

This section collects the documentation on tools that have been implemented in the package, but can not be classified as traditional AiiDA objects. The scope of these tools is a further improvement of the automatization of siesta calculations. Some of them are more for development purposes (FDFDict), others are for the benefit of any user (the protocol system).

2.3.1.1 The protocols system

Description

In order to submit **SiestaCalculations**, the user needs to manually select all the inputs, being careful to pass the correct specifications to perform the calculation (as explained in the *corresponding section*). The package `aiida_siesta` provides also a set of pre-selected inputs to run a **SiestaCalculations**, and the WorkChains distributed in the package, supporting the tasks of the relaxation of a structure and the calculations of bands. In other words, the user can obtain a `builder` of the **SiestaCalculation** that is ready to be submitted. This `builder`, in fact, is pre-filled with inputs selected according to the structure under investigation and very few options specified by the user. The lengthy inputs selection is substitute by:

```
inp_gen = SiestaCalculation.inputs_generator()
builder = inp_gen.get_filled_builder(structure, calc_engines, protocol)
```

The list of options to obtain the builder is presented *here*, however the main feature is the use of *protocols*. A *protocol* groups operational parameters for a Siesta calculation and it is meant to offer a set of inputs with the desired balance of accuracy and efficiency. At the moment only one protocol is shipped in the package, it is called *standard_psml*. More on it is presented in the next to next subsection. It is important to note that the implemented protocols are not, for the moment, input parameters that are guaranteed to perform in any situation. They are only based on reasonable assumptions and few tests. However, in the package it is also implemented a system that allows users to create their own protocols, as clarified *here*. Finally, it must be remembered that the `builder` produced according to a *protocol* and few other options is fully modifiable before submission, leaving full flexibility to the user. We expect in the future to have more and more "know how" and improve the reliability and richness of the available *protocols*.

We focus here on the description of the use of protocols for the **SiestaCalculation**, but the same system is available for all the WorkChains distributed in this package. A small paragraph in the documentation of each WorkChain will explain the details of the usage of protocols for that particular WorkChain.

Supported Siesta versions

The protocol system, at the moment, requires a version of siesta with support for psml pseudopotential. At least **the MaX-1.0 release of Siesta**, which can be found in the development platform (https://gitlab.com/siesta-project/siesta), meets this requirement. For more up to date info on compatibility, please check the wiki.

Available protocols

With the word *protocol* we mean a series of suggested inputs for AiiDA CalJobs/WorkChains that allow users to more easily automatize their workflows. These inputs reflects a certain set of operational parameters for a Siesta calculation. The choice of the inputs of a DFT simulation should be carefully tested for any new system. Therefore the use of protocols, in place of a careful and tested choice of inputs, it is always somehow limiting. It can be, however, considered a good starting point. This is the very beginning of the development and, for the moment, only one very basic protocol is implemented. A description of its variables is now reported. Each protocol contain a section with global variables and an *atomic_heuristics* dictionary, a dictionary intended to encode the peculiarities of particular elements.

- *standard_psml*

  The full list of variables for this protocol are collected in the *protocol_registry.yaml* file, located in `aiida_siesta/utils`.

  - *global variables*

    Pseudopotential ONCVPSPv0.4 (norm-conserving) of Pseudo Dojo in psml format, scalar relativistic, PBE and with *standard* accuracy (download available from the PseudoDojo web site). Basis set apply globally, with size DZP and `energy-shift` of 50 meV. The `mesh-cutoff` is 200 Ry, `electronic-temp` 25 meV, and a kpoint mesh with distance 0.1 are implemented. Concerning the trashold for convergence, we implement 1.e-4 tolerance for the density matrix, 0.04 ev/ang for forces and 0.1 GPa for stress. Few more global variables are related to mixing options: `scf-mixer-history` is set to 5, and `scf-mixer-weight` is 0.1. As only the Max-1.0 version of Siesta is supported, the default mixer is Pulay and the quantity mixed is the Hamiltonian.

  - *atomic_heuristics*

    The element "Ag" requires a bigger `mesh-cutoff` because `mesh-cutoff = 200 Ry` was leading to a "Failure to converge standard eigenproblem" error for the "Ag" elemental crystal. Custom basis for "Ca","Sr","Ba" are necessary because the automatic generation results in a too-large radius for the "s"

orbitals. The "Hg" custom basis introduces an increment of all radii of 5% compared to the automatic generated orbitals and adds a Z orbital for the "p" channel, while removing polarization. The elements "Li", "Be", "Mg", Na", "Fe", "Mn", "Sb" require a bigger `mesh-cutoff` because `mesh-cutoff = 200 Ry` resulted in a discontinuous equation of state.

This choice of parameters have been tested on crystal elements up to the element "Rn" and compared with the reference equation of state of the DeltaTest project, resulting on an average delta value of 7.1 meV. The parameters of this protocol for noble gasses do not result in an a minimum of the equation of state. Because Van der Waals forces are not included in the calculation, the result is not surprising. We warn users to use with care this protocol for noble gasses. It is important to stress that the present protocol has not been conceived to produce good results for the Delta test; the basis sets are mostly automatic and the choice of mesh-cutoff / kpoints-mesh is farely loose. The average value for the delta (7.1 meV) is just an indication that the parameters' choice gives reasonable results for elemental crystals. We are working on a more accurate (and expensive) protocol that will provide much better values of delta. New tests and checks on the *standard_psml* protocol will be added in the aiida-siesta wiki.

The management of the pseudos is, at the moment, very fragile. It imposes that the user loads a pseudo_family with the correct name that is hard-coded for the each protocol. This name is 'nc-sr-04_pbe_standard_psml' for the *standard_psml* protocol. Therefore a user, before using protocol, needs to download the correct pseudos and load them (see next section) with the correct name. —This last part will change soon, replaced with a proper setup-profile script —-

How to use protocols

In this section we explain how to obtain a pre-filled builder according to a protocol and an input structure, that is ready to be submitted (or modified and then submitted).

First of all, the scalar relativistic "standard" pseudo set from PseudoDojo must be installed as aiida family pseudo family:

```
aiida-pseudo install pseudo-dojo -v 0.4 -x PBE -r SR -p standard -f psml
```

Once this first step is done, the pre-filled builder can be accessed through the method `inputs_generator` of the **SiestaCalculation** (and of any other workchain). For example:

```
from aiida_siesta.calculations.siesta import SiestaCalculation
inp_gen = SiestaCalculation.inputs_generator()
builder = inp_gen.get_filled_builder(structure, calc_engines, protocol)
#here user can modify builder befor submission.
submit(builder)
```

The arguments of `get_filled_builder` of the input generator are explained here:

- **structure**, class `StructureData`, *Mandatory*

  A structure. See the *plugin documentation* for more details.

- **calc_engine**, python `dict`, *Mandatory*

  A dictionary containing the specifications of the code to run and the computational resources. An example:

```
calc_engines = {
    'siesta': {
        'code': codename,
        'options': {
                'resources': {'num_machines': 1, "num_mpiprocs_per_machine": 1},
                'max_wallclock_seconds': 360,
```

(continues on next page)

(continued from previous page)

```
                'queue_name': 'DevQ',
                'withmpi': True,
                'account': "tcphy113c"
            }
        }
}
```

The dictionary must present `siesta` as upper level key of the dictionary. This might seem unnecessary, but will become fundamental for the use of protocols in more complicated WorkChain, involving not only the siesta plugin, but also, for instance, the stm plugin. The structure of `calc_engines` for each WorkChain of the package will be specified in the WorkChain documentation.

- **protocol**, python `str`, *Mandatory*

  The protocol name, selected among the available ones, as explained in the previous section.

- **bands_path_generator**, python `str`, *Optional*

  The presence of this parameter triggers the calculation of bands. Two are the available value to pass as *bands_path_generator*: "seekpath" or "legacy". They set the way the path in k-space is produced. This path is used to display the bands. While "seekpath" modify the structure running the calculation on an equivalent "conventional" cell, "legacy" doesn't and preserves the input structure. However the "legacy" method is known to have bugs for certain structure cells.

- **relaxation_type**, python `str`, *Optional*

  The presence of this parameter triggers the possibility to relax the structure. The specifications of the relaxation_type are "atoms_only", "variable_cell" or "constant_volume", that should be self expalnatory. For the moment only the CG relaxation algorithm is implemented (in the future more will be added).

- **spin**, python `str`, *Optional*

  The presence of this parameter triggers the spin options. The specifications of the spin are the one of modern version of Siesta, they are "polarized", "non-collinear" and "spin-orbit". If no spin option is defined, the calculation will not be spin polarized.

An example of the use is in *aiida_siesta/examples/plugins/siesta/example_protocol.py*

The method `get_filled_builder` is definitely the most important tool offered by the `inputs_generator`, however through the `inputs_generator` other methods can be accessed to explore the various options of the protocol system. For instance, there is a method listing all the available protocols, the available relaxation types and so on.

How to create my protocols

The protocol system allows also to create customized protocol. To this end, a file similar to *aiida_siesta/utils/protocol_registry.yaml* must be created, listing the custom protocols. Then the path of this file must be added to the environment variable *AIIDA_SIESTA_PROTOCOLS*. This will be sufficient to let aiida-siesta recognize the protocols. The file containing the customized protocols must have the same structure of *protocol_registry.yaml*. An example:

```
my_protocol:
  description: 'My description'
  parameters:
    xc-functional: "GGA"
    xc-authors: "PBE"
    mesh-cutoff: '200 Ry'
```

(continues on next page)

(continued from previous page)

```
  ...
spin_additions:
  write-mulliken-pop: 1
relax_additions:
  scf-dm-tolerance: 1.e-4
  md-max-force-tol: '0.04 eV/ang'
  md-max-stress-tol: '0.1 GPa'
basis:
  pao-energy-shift: '50 meV'
  pao-basis-size: 'DZP'
pseudo_family: 'nc-sr-04_pbe_standard_psml'
kpoints:
  distance: 0.1
  offset: [0., 0., 0.]
atomic_heuristics:
  Li:
    parameters:
      mesh-cutoff: '250 Ry'
    basis:
      polarization: 'non-perturbative'
      pao-block: "Li 3 \n  ... "
      split-tail-norm: True
```

The protocol name should be the outer entry of the indentation. For each protocol, some keyword are mandatory. They are *description*, *parameters*, *basis* and *pseudo_family*. The *pseudo_family* must contain the name of a family (Psml or Psf family) that has been already uploaded in the database. The number of elements covered by your pseudo family will limit the materials you can simulate with your protocol. The *parameters* and *basis* entries are transformed into dictionaries and passed to AiiDA after possible modifications due to atom heuristics or spin/relax additions. For this reason, the syntax (lower case and '-' between words) must be respected in full.

Two optional keywords are *relax_additions* and *spin_additions*. This two entries are not meant to host the siesta keywords that activate the relaxation or spin options, but possible additions/modifications to the *parameters* entry, to apply in case of relaxation or spin. When the use of protocols is called and the relax/spin options are requested (see here), the system will automatically take care of introducing the correct siesta keyword (*MD.TypeOfRun*, *MD.VariableCell*, *spin* etc.) that are indispensable to run the task. However, it might happen that a user desires a more loose *scf-dm-tolerance* for the task of the relaxation or a different *scf-mixer-weight* when the spin is active. The *relax_additions* and *spin_additions* keywords have been created exactly for this purpose. Please be carefull that (except for the *mesh-cutoff*) if a keyword in *spin_additions* or *relax_additions* is already present in *parameters*, its value in *parameters* will overriden. In other words, values in *spin_additions* or *relax_additions* have priority compared to the one in *parameters*. Moreover *relax_additions* has priority respect to *spin_additions*. For the *mesh-cutoff* the situation is different, because the biggest value will always be considered, no metter where it is specified. Another optional entry is *kpoints*, where a *distance* and an *offset* only can be specified. The system will take care to create a uniform mesh for the structure under investigation with a density that correspond to a distance (in 1/Angstrom) between adjacent kpoints equal to *dinstance*.

The final allowed (optional) keyword is *atomic_heuristics*. In it, two only sub-keys are allowed: *parameters* and *basis*. In *parameters*, only a 'mesh-cutoff' can be specified. This *mesh-cutoff* applies globally and only if it is the biggest one among the all *mesh-cutoff* that apply. This system is meant to signal elements that requires a bigger 'mesh-cutoff' than normal. For *basis*, we allow 'split-tail-norm', 'polarization', 'size' and 'pao-block'. The 'size' and' polarization' introduce a block reporting the change of pao size and polarization schema only for the element under definition. The 'pao-block' allows to specify an explicit "block Pao-basis" for the element. The 'split-tail-norm' instead activate in siesta the key 'pao-split-tail-norm', that applies globally.

We conclude this subsection with few more notes to keep in mind. First, the units mut be specified for each siesta keyword that require units and they must be consisten throughout the protocol. This means that it is not possible to define 'mesh-cutoff' in Ry in *parameters*, but in eV in the *atomic_heuristics*. Second, it is up to the creator to

remember to introcude the correct 'xc-functional' and 'xc-authors' keywords in the protocol that matches the same exchange-correlation functional of the pseudos in the pseudo family. This also means that we do not support pseudos presenting different exchange-correlation functionals in the same family. Third, we impose a description for each protocol because in the description the creator must underline the limitations of the protocol. For instance, the case when a certain protocol do not support spin-orbit as the pseudos are not relativistics. The schema we presented here is certanly not perfect and it is far to cover all the possible situations, however it must be remembered that any user has always the chance to modify the inputs (builder) before submission.

2.3.1.2 FDF dictionary

Description

The FDFDict class represents data from a .fdf-file (the standard input of the siesta code). It behaves like a normal python dictionary, but with translation rules that follow the standards of the Flexible Data Format (FDF). The FDF format was developed inside the siesta package in order to facilitate the creation of the input file of siesta. Among other features, it substitute strings in favour of default values. In particular it drops dashes/dots/colons and imposes lowercase. The FDFDict class accepts in input a python dictionary and applies the same rules to the "keys" of the dictionary. An example:

```
from aiida_siesta.calculations.tkdict import FDFDict
inp_dict = {"ThisKey": 3,"a-no-ther": 4,"t.h.i.r.d" : 5}
f = FDFDict(inp_dict)
print(f.keys())
```

returns `dict_keys(['thiskey', 'another', 'third'])`.

When two keys in the same dictionary will become the same string after translation, the last definition will remain:

```
from aiida_siesta.calculations.tkdict import FDFDict
inp_dict = {"w":3,"e":4,"w--":5}
f = FDFDict(inp_dict)
print(f.get_dict())
```

returns `{'w': 5, 'e': 4}`.

The method `get_dict` returns the translated dictionary, but the class keeps record also of the last unstraslated key for each key. This can be seen just printing `f`. The method `get_untranslated_dict` returns the dictionary with the last unstranslated keys as keys. Therefore in our example, the `get_untranslated_dict` returns `{'w--': 5, 'e': 4}`.

Getter and setter are implemented to get and set the value automatically for each equivalent key. `f["w"]`, `f["w---"]` will return the same value. The call `f["w---"] = 3` will reset the value of key `"w"`, also changing the "last untranslated key" to `"w---"`.

Many more methods are available in the FDFDict class. They can be explored from the source code (`aiida_siesta.calculations.tkdict`). It is a useful tool for the development of new CalcJobs and WorkChains.

2.3.1.3 PAO manager

EXPERIMENTAL FEATURE!

Description

Class to help modifications of PAO basis block. Also translates orbitals info contained in ion files into a PAO block. For the moment can only treat one single site at the time and can be initialized only from an IonData instance:

```
ion=load_node(pk) # pk of an IonData instance
pao_manager = ion.get_pao_modifier()
```

It offers several methods to manipulate the basis specifications, for instance adding and removing orbitals (also polarized), increase or decrease all the radii of a percentage, manually modify single radii of orbitals. It the returns a string that can be directly insered into the basis input of a **SiestaCalculation** (or workchains of the package) under the *%pao-basis block* key:

```
pao_manager.get_pao_block()
```

2.4 Workflows

2.4.1 Workflows

In this section we document the AiiDA WorkChains distributed in `aiida-siesta`. They are tools that automatize some simple tasks that are commonly faced during the the research process. The WorkChains are constructed using exclusively the calculations plugin described in the section "Calculations".

2.4.1.1 Base workflow

Description

The SIESTA program is able to perform, in a single run, the computation of the electronic structure, the optional relaxation of the input structure, and a final analysis step in which a variety of magnitudes can be computed: band structures, projected densities of states, etc. The operations to be carried out are specified in a very flexible input format. Accordingly, the **SiestaBaseWorkChain** has been designed to be able to run the most general SIESTA calculation, with support for most of the available options (limited only by corresponding support in the parser plugin). The option specifications of the **SiestaBaseWorkChain** follow the conventions already presented in the *Siesta plugin*. Therefore, for instance, the addition of the input keyword **bandskpoints** triggers the calculation of the band structure of a system, while it is sufficient to add the SIESTA MD keywords to the **parameters** input in order to perform the relaxation of a structure. In contarst to the **SiestaCalculation** plugin, however, the workchain is able to automatically restart a calculation in case of failure (lack of electronic-structure or geometry relaxation convergence, termination due to walltime restrictions, etc). Therefore, the **SiestaBaseWorkChain** is the suggested tool to run Siesta calculations in the AiiDA framework. In fact, it retains the same level of flexibility of the most general Siesta calculation, but it adds robusness thanks to its ability to automatically respond to erros. Examples on the use of the **SiestaBaseWorkChain** are presented in the folder */aiida_siesta/examples/workflows*.

Supported Siesta versions

At least 4.0.1 of the 4.0 series, 4.1-b3 of the 4.1 series and the MaX-1.0 release, which can be found in the development platform (https://gitlab.com/siesta-project/siesta). For more up to date info on compatibility, please check the wiki.

Inputs

All the siesta plugin inputs are also inputs of the **SiestaBaseWorkChain**. Therefore, detailed information on them can be found *here*. The only difference is regarding the way the computational resources are passed. The siesta plugin makes use of `metadada.options` for this task, here, instead, we have a dedicated input node.

- **options**, class `Dict`, *Mandatory*

  Execution options for the siesta calculation. In this dictionary the computational resources and scheduler specifications (queue, account, etc ..) must be specified. An example is:

```
options = Dict(
     dict={
        'max_wallclock_seconds': 360,
        'withmpi': True,
        'account': 'tcphy113c',
        'queue_name': 'DevQ',
        'resources': {'num_machines': 1,'num_mpiprocs_per_machine': 2},
        }
     )
```

  The *resources* and *max_wallclock_seconds* are required by AiiDA, the rest of the options depend on the scheduler of the machine one is submitting to.

The **SiestaBaseWorkChain** also has some additional inputs that allow to control additional features.

- **pseudo_family**, class `Str`, *Optional*

  String representing the name of a pseudopotential family stored in the database. Pseudofamilies can be installed in the database via the `aiida-pseudo install family` CLI interface. As already specified in the description of the **pseudos** input *here*.

- **clean_workdir**, class `Bool`, *Optional*

  If true, work directories of all called calculations will be cleaned out. Default is false.

- **max_iterations**, class `Int`, *Optional*

  The maximum number of iterations allowed in the restart cycle for calculations. The **SiestaBaseWorkChain** tries to deal with some common siesta errors (see *here*) and restart the calculation with appropriate modifications. The integer **max_iterations** is the maximum number of times the restart is performed no matter what error is recorded. The input is optional, if not specified, the default *Int(5)* is used.

Relaxation and bands

As already mentioned in the introduction, in addition to simple scf calculations, the **SiestaBaseWorkChain** can be used to perform the relaxation of a structure and the electronic bands calculations. For the electronic bands, however, we suggest the use of the **BandgapWorkChain** distributed in this package, because it adds the feature to automatically calculate the band gap. Concerning the relaxation of a structure, the **SiestaBaseWorkChain** simply exploits the internal relaxation implemented in Siesta in order to complete the task. The full set of a Siesta relaxation options can be accessed just adding the corresponding keyword and value in the **parameters** input dictionary. The only additional feature that the **SiestaBaseWorkChain** adds is that it requires to reach the target forces and stress to consider completed the task. If this does not happen in a single Siesta run, the workchain restarts automatically the relaxation. The maximum number of restarts is specified with the keyword **max_iterations**, as explained in the previous subsection.

Submitting the WorkChain

WorkChains are submitted in AiiDA exacly like any other calculation. Therefore:

```
from aiida_siesta.workflows.base import SiestaBaseWorkChain
from aiida.engine import
builder = SiestaBaseWorkChain.get_builder()
builder.options = options
... All the inputs here ...
submit(builder) #or run
```

There is no need to set the computational resources with the metadata as they are already defined in the input **options**, however `builder.metadata.label` and `builder.metadata.description` could be used to label and describe the WorkChain. Again, the use of the `builder` is not mandatory, the inputs can be passed as arguments of `sumbit/run` as explained in the siesta plugin section.

Outputs

The outputs of the **SiestaBaseWorkChain** mirror exactly the one of the siesta plugin. Therefore all the information can be obtained in the *corresponding section*.

Error handling

We list here the errors that are handled by the **SiestaBaseWorkChain** and the corresponding action taken. The error are actually detected by the siesta parser, in the WorkChain, the handling is performed.

- **SCF_NOT_CONV**

  When the convergence of the self-consistent cycle is not reached in `max-scf-iterations` or in the allocated `max_walltime`, siesta raises the **SCF_NOT_CONV** error. The **SiestaBaseWorkChain** is able to detect this error and restart the calculation with no modifications on the input parameters.

- **GEOM_NOT_CONV**

  When the convergence of the geometry (during a relaxation) is not reached in the allocated `max_walltime`, siesta raises the **GEOM_NOT_CONV** error. The **SiestaBaseWorkChain** is able to detect this error and restart the calculation with no modifications on the input parameters.

- **SPLIT_NORM**

  The **SiestaBaseWorkChain** deals with problems connected to the basis set creation. If a "too small split-norm" error is detected, the WorkChains reacts in two ways. If a global split-norm was defined in input through `pao-split-norm`, its value is reset to the minimum acceptable. If no global split-norm was defined the option `pao-split-tail-norm = True` is set.

Two more errors are detected by the WorkChain, but not handled at the moment, only a specific error code is returned as output without attempting a restart.

- **BASIS_POLARIZ**

  If an error on the polarization of one orbital is detected, the error code 403 is returned. The solution to this problem is to set the "non-perturbative" polarization scheme for the element that presents an error, however this possibility is available only in recent versions of AiiDA, making inconvenient to treat automatically the resolution of this error.

- **ERROR_BANDS**

If a calculation of the electronic bands is requested, but an error in the parsing of the bands file is detected, the error code 404 is returned. In this case, the WorkChain will anyway return all the other outputs because the checks on the bands file are always performed at the very end of the calculation.

The **SiestaBaseWorkChain** also inherits the error codes of the **BaseRestartWorkChain** of the aiida-core distribution. For instance, if an unexpected error is raised twice, the workchain finishes with exit code 402, if the maximum number of iterations is reached, error 401 is returned. More in the section BaseRestartWorkChain of the aiida-core package.

Protocol system

The protocol system is available for this WorkChain. The `SiestaBaseWorkChain.inputs_generator()` makes available all the methods explained in the *protocols documentation*. For example:

```
from aiida_siesta.workflows.base import SiestaBaseWorkChain
inp_gen = SiestaBaseWorkChain.inputs_generator()
builder = inp_gen.get_filled_builder(structure, calc_engines, protocol)
#here user can modify builder befor submission.
submit(builder)
```

is sufficient to submit a **SiestaBaseWorkChain** on `structure` following the specifications of `protocols` and computational resources collected in `calc_engines`. The structure of `calc_engines` is the same as for the **SiestaCalculation** input generator (again see *protocols documentation*).

2.4.1.2 Bandgap workflow

Description

The **BandgapWorkChain** is an extension of the **SietaBaseWorkChain** that introduces some logic to automatically obtain the bands and applyes a simple post-process with the scope to return the metallic or insulating nature of the material and, possibly, the band gap.

To calculate the gap, this workchain makes use of a tool distributed in aiida-core, the method `find_bandgap` hosted in `aiida.orm.nodes.data.array.bands`.

The optional automatic generation of the kpoints path for the bands is done using SeeK-path.

Supported Siesta versions

At least 4.0.1 of the 4.0 series, 4.1-b3 of the 4.1 series and the MaX-1.0 release, which can be found in the development platform (https://gitlab.com/siesta-project/siesta). For more up to date info on compatibility, please check the wiki.

Inputs

All the **SiestaBaseWorkChain** inputs are as well inputs of the **BangapWorkChain**, therefore the system and DFT specifications (structure, parameters, etc.) are inputted in the WorkChain using the same syntax explained in the **SiestaBaseWorkChain** *documentation*. There is however the addition of an importan feature. If **bandskpoints** are not set in inputs, the **BandgapWorkChain** will anyway calculate the bands following these rules:

- If a single-point calculation is requested, the kpoints path for bands is set automatically using SeeK-path. Please note that this choice might change the structure, as explained in the SeeK-path documentation.

- If a relaxation was asked, first a siesta calculation without bands is performed to take care of the relaxation, then a separate single-point calculation is set up and the bands are calculated for a symmetry path in k-space decided by SeeK-path using the output structure of the relaxation. This overcomes the problem of the compatibility between bands and variable-cell relaxations. In fact, the final cell obtained from a relaxation, can not be known in advance, and to set the kpoint path without knowing the cell is generally a poor choice. Again note that SeeK-path might change the structure. In this second case, only the structure of the final single-point calculation will be changed. The changed structure is returned as **output_structure** port of the workchain.

If the **bandskpoints** is set by the user in inputs, no action is taken and the behaviour follow what explained for the **SiestaBaseWorkChain** *documentation*.

An additional input is present:

- **seekpath_dict** class `Dict`, *Optional*

  Dictionary hosting the parametrs to pass to the `get_explicit_kpoints_path` method of SeeK-path. The default sets `{'reference_distance': 0.02, 'symprec': 0.0001}`, meaning target distance between neighboring k-points of 0.02 1/ang and symmetry precision parameter of 0.0001. Full list of the possible options and their explanation can be found here.

Outputs

- All the outputs of **SiestaBaseWorkChain** are also outputs of this WorkChain, they can be explored in the relative section of the **SiestaBaseWorkChain**.
- **band_gap_info** `Dict`

  A dictionary containing a bool (*is_insulator*) set to True if the material has a band gap, to False otherwise. Moreover the dictionary contains the value of the gap in *eV*.

Protocol system

The protocol system is available for this WorkChain. The `BandgapWorkChain.inputs_generator()` makes available all the methods explained in the *protocols documentation*. The bands options are still valid and they will set a *bandskpoints* input to the workchain. To avail of the automatic generation of bands path, do not pass any `bands_path_generator` to `get_filled_builder`.

2.4.1.3 Equation Of State workflow

Description

The **EqOfStateFixedCellShape** WorkChain is a tool for the calculation of the equation of state of a solid. Density Functional Theory (DFT) calculations with the SIESTA code are performed at 7 equidistant volumes around a starting volume in order to obtain the energy (E) versus volume (V) data. The starting volume is an optional input of the WorkChain, called **volume_per_atom**. If the latter is not specified, the input structure volume is use as starting volume. The WorchChain ensure robustness in the convergence of each SIESTA calculation thanks to the fact that each DFT run is submitted through the **SiestaBaseWorkChain**, that automatically manages some common failures (lack of electronic-structure or geometry relaxation convergence, termination due to walltime restrictions, etc). All the **SiestaBaseWorkChain** inputs are as well inputs of the **EqOfStateFixedCellShape**, therefore the system and DFT specifications (structure, parameters, etc.) are inputted in the WorkChain using the same syntax explained in the **SiestaBaseWorkChain** *documentation*. As the name of the class suggest, the **EqOfStateFixedCellShape** is designed to obtain the E(V) curve under the restriction of fixed cell shape. This means that no algorithm for stress minimization is implemented in the WorkChain. However the option relaxation `MD.ConstantVolume` (see SIESTA manual) might be added into the parameters dictionary to let SIESTA to relax the structure at fixed volume. There is no point,

for obvious reasons, to run this WorkChain with the relaxation option `MD.VariableCell`. This WorkChain also tries to perform a Birch_Murnaghan fit on the calculated E(V) data, following the DeltaProject implementation. If the fit fails, a warning is stored in the report of the WorkChain (accessible through `verdi process report <PK>`), but the E(V) data for the 7 volumes are always returned, leading to a succesfull termination of the process.

Supported Siesta versions

At least 4.0.1 of the 4.0 series, 4.1-b3 of the 4.1 series and the MaX-1.0 release, which can be found in the development platform (https://gitlab.com/siesta-project/siesta). For more up to date info on compatibility, please check the wiki.

Inputs

- All the inputs of the **SiestaBaseWorkChain**, as explained *here*.
- **volume_per_atom**, class `Float`, *Optional*

  A decimal number corresponding to the volume per atom around which to perform the equation of state.
- **batch_size**, class `Int`, *Optional*

  Number of volumes to run at the same time. By default, it is set to one, therefore one volume at the time is submitted

Outputs

- **results_dict** `Dict`

  A dictionary containing a key *eos_data* that collects the computed E(V) values and relative units of measure. If the Birch-Murnaghan fit is succesfull, also the key *fit_res* will be present in this disctionary. It reports the following values extracted from the fit: the equilibrium volume (Vo, in *Angstom^3/atom*), the minimum energy (Eo, in *eV/atom*), the Bulk Modulus (Bo, in *ev/Angstrom^3*) and its derivative respect to the presure B1.
- **equilibrium_structure** `StructureData`

  Present only if the Birch-Murnaghan fit is succesfull, it is the AiiDA structure at the equilibrium volume Vo.

Protocol system

The protocol system is available for this WorkChain. The `EqOfStateFixedCellShape.inputs_generator()` makes available all the methods explained in the *protocols documentation*, the only difference is that the relaxation type "variable-cell" is not available.

2.4.1.4 STM workflow

Description

The **SiestaSTMWorkchain** workflow consists in 3 steps:

- Performing of a siesta calculation on an input structure (including relaxation if needed) through the **SiestaBaseWorkChain**.
- Performing of a further siesta calculation aimed to produce a .LDOS file.

- A call to the *plstm* code to post process the .LDOS file and create simulated STM images. The call is made via the **STMCalculation** plugin, which is also included in the `aiida_siesta` distribution.

The .LDOS file contains informations on the local density of states (LDOS) in an energy window. The LDOS can be seen as a "partial charge density" to which only those wavefunctions with eigenvalues in a given energy interval contribute. In the Tersoff-Hamann approximation, the LDOS can be used as a proxy for the simulation of STM experiments. The 3D LDOS file is then processed by the specialized program *plstm* to produce a 2D section in "constant-height" or "constant-current" mode, optionally projected on spin components (see the header/manual for plstm, and note that non-collinear and spin-orbit modes are supported). The "constant-height" mode corresponds to the creation of a plot of the LDOS in a 2D section at a given height in the unit cell (simulating the height of a STM tip). The "constant-current" mode simulates the topography map by recording the z coordinates with a given value of the LDOS.

The inputs to the STM workchain include all the inputs of the **SiestaBaseWorkChain** to give full flexibility on the choice of the siesta calculation parameters. The energy window for the LDOS is specified respect to the Fermi energy. In fact, a range of energies around the Fermi Level (or regions near to the HOMO and/or LUMO) are the meaninful energies for the STM images production. The tip height ("constant-height" mode) or the LDOS iso-value ("constant-current" mode) must be specified by the user in input. The workchain returns an AiiDA ArrayData object whose contents can be displayed by standard tools within AiiDA and the wider Python ecosystem.

Supported Siesta versions

At least 4.0.1 of the 4.0 series, 4.1-b3 of the 4.1 series and the MaX-1.0 release, which can be found in the development platform (https://gitlab.com/siesta-project/siesta). For more up to date info on compatibility, please check the wiki.

Inputs

- All the inputs of the **SiestaBaseWorkChain**, as explained *here*.
- **stm_code**, class `Code`, *Mandatory*

  A code associated to the STM (plstm) plugin (siesta.stm). See plugin documantation for more details.
- **stm_mode**, class `Str`, *Mandatory*

  Allowed values are `constant-height` or `constant-current`, corresponding to the two operation modes of the STM that are supported by the plstm code.
- **stm_value**, class `Float`, *Mandatory*

  The value of height or current at which the user wants to simulate the STM. This value represents the tip height in "constant-height" mode or the LDOS iso-value in "constant-current" mode. The height must be expressed in *Angstrom*, the current in *e/bohr**3*.
- **emin**, class `Float`, *Mandatory*

  The lower limit of the energy window for which the LDOS is to be computed (in eV and respect to the Fermi level).
- **emax**, class `Float`, *Mandatory*

  The upper limit of the energy window for which the LDOS is to be computed (in *eV* and respect to the Fermi level).
- **stm_spin**, class `Str`, *Mandatory*

  Allowed values are `none`, `collinear` or `non-collinear`. Please note that this keyword only influences the STM post process! It does not change the parameters of the siesta calculation, that must be specified in the **parameters** input port. In fact, this keyword will be automatically reset if a *stm_spin* option incompatible

with the parent siesta spin option is chosen. A warning will be issued in case this happens. This keyword also influences the structure of the output port **stm_array**. If fact, if the `non-collinear` value is chosen, the workflow automatically performs the STM analysis in the three spin components and for the total charge option, resulting in a richer **stm_array** (see description in the Outputs section).

- **stm_options**, class `Dict`, *Optional*

  This dictionary can be used to specify the computational resources to be used for the STM calculation (the *plstm* code). It is optional because, if not specified, the same resources of the siesta calculations are used, except that the parallel options are stripped off. In other words, by default, the *plstm* code runs on a single processor.

Outputs

- **stm_array** `ArrayData`

  In case the **stm_spin** is `none` or `collinear` this output port is a collection of three 2D arrays (*grid_X*, *grid_Y*, *STM*) holding the section or topography information. Exactly like the output of the STM plugin. In case the **stm_spin** is `non-collinear`, this output port is a collection of six 2D arrays (*grid_X*, *grid_Y*, *STM_q*, *STM_sx*, *STM_sy*, *STM_sz*) holding the section or topography information for the total charge STM analysis and the three spin components. Both cases follow the *meshgrid* convention in Numpy. A contour plot can be generated with the *get_stm_image.py* script in the repository of examples. The *get_stm_image.py* script automatically detects how many arrays are in **stm_array**, therefore it is completely general.

- **output_structure** `StructureData`

  Present only if the siesta calculation is moving the ions. Cell and ionic positions refer to the last configuration, on which the STM analysis is performed.

Protocol system

The protocol system is available for this WorkChain. The `SiestaSTMWorkchain.inputs_generator()` makes available all the methods explained in the *protocols documentation*, but `get_filled_builder` now requires in inputs also the `stm_mode` (a python *str* <*str*>, accepted values are "constant-height" and "constant-current") and `stm_value` (a python *float* <*float*> indicating the value of height in Ang or current in e/bohr**3). Moreover in the `calc_engines` dictionary, also indications on the resources for the stm calculation must specified, following the syntax of this example:

```
calc_engines = {
  'siesta': {
      'code': codename,
      'options': {'resources': {'num_machines': 1, "num_mpiprocs_per_machine": 1},
↪"max_wallclock_seconds": 3600 }
      },
  'stm': {
      'code': stmcodename,
      'options': {'resources': {'num_machines': 1, "num_mpiprocs_per_machine": 1},
↪"max_wallclock_seconds": 1360 }
      }
  }
```

The STM spin mode is chosen accordingly to the `spin` input passed to `get_filled_builder`, setting "collinear" stm_spin in case of polarized calculation, "non-collinear" in case of "spin-orbit" or "non-collinear" calculations and no spin in case of an unpolarized calculation. Therefore, if, for instance, the user wants to post-process a spin calculation with "no-spin" STM mode, he/she needs to manually modify the builder before submission. Also the **emin** and **emax** inputs of **SiestaSTMWorkchain** are internally chosen by the inputs generator: they select an energy window of *6*

eV below the Fermi energy. If the choice doesn't suit the purpose, the user can manually modify the builder before submission.

2.4.1.5 Iterator workflow

Description

The **SiestaIterator** is a tool to facilitate the submission of several Siesta Calculations in an automatic way. It allows the iteration over Siesta parameters and, more in general, over inputs of a **SiestaBaseWorkChain**. An example on the use of the **SiestaConverger** is */aiida_siesta/examples/workflows/example_iterate.py*.

Supported Siesta versions

At least 4.0.1 of the 4.0 series, 4.1-b3 of the 4.1 series and the MaX-1.0 release, which can be found in the development platform (https://gitlab.com/siesta-project/siesta). For more up to date info on compatibility, please check the wiki.

Inputs

All the **SiestaBaseWorkChain** inputs are as well inputs of the **SiestaIterator**, therefore the system and DFT specifications (structure, parameters, etc.) are inputted in the WorkChain using the same syntax explained in the **SiestaBaseWorkChain** *documentation*. The additional inputs are:

- **iterate_over**, class `Dict`, *Mandatory*

  A dictionary where each key is the name of a parameter we want to iterate over (`str`) and each value is a `list` with all the values to iterate over for the corresponding key. Accepted keys are:

  - Name of the input ports of the **SiestaBaseWorkChain**. Meaning all the names listed *here*. In this case, the corresponding values list must contains the list of `Data` nodes (stored or unstored) accepted by the key. Examples are:

```
code1 = load_code("SiestaHere@localhost")
code2 = load_code("SiestaThere@remotemachine")
iterate_over = {"code" : [code1,code2]}

struct1 = StructureData(ase=ase_struct_1)
struct2 = StructureData(ase=ase_struct_2)
iterate_over = {"structure" : [struct1,struct2]}
```

  - Name of accepted Siesta input keywords (for instance `mesh-cutoff`, `pao-energy-shift`, etc ...). In this case, the corresponding values list must contains the list of values directly, meaning `str`, `float`, `int` or `bool` python types. Examples are:

```
iterate_over = {"spin" : ["polarized", "spin-orbit"]}
```

> **Warning:** In order to guarantee full flexibility, no check on the Siesta parameters is performed. If you pass as key something not recognized by Siesta, the SiestaIterator will include it in the *parameters* input and run the calculation with no warning issued. Because Siesta will not understand the keyword, it will ignore it, resulting in a series of identical calculations.

  The *iterate_over* is a dictionary because it is possible to iterate over several keywords at the same time. Something of this kind:

```
struct1 = StructureData(ase=ase_struct_1)
struct2 = StructureData(ase=ase_struct_2)
iterate_over = {"structure" : [struct1,struct2], "spin" : ["polarized", "spin-
↪orbit"]}
```

is perfectly acceptable and the way the algorithm handle with these multiple iterations is decided by the **SiestaIterator** input explained next in this list.

- **iterate_mode**, class `Str`, *Optional*

  Indicates the way the parameters should be iterated. Currently allowed values are 'zip' (zips all the parameters together, this imposes that all keys should have the same number of values in the list!) and 'product' (performs a cartesian product of the parameters, meaning that all possible combinations of parameters and values are explored).

  The option 'zip' is the default one.

- **batch_size**, class `Int`, *Optional*

  The maximum number of simulations that should run at the same time. You can set this to a very large number if you want that all simulations run in one single batch. As default, only one single calculation at the time is submitted.

Outputs

This WorkChain does not generate any output! It is, in fact, a tool to help the submission of multiple calculations and keep them all connected and easy accessible through the main workchain node, but it does not have any precise scope. AiiDA provides a powerful querying system to explore all the results of the submitted calculations and a tool to organize the data.

Protocol system

The protocol system is not directly available for this WorkChain. However inputs of the **SiestaBaseWorkChain** can be obtained in a dictionary in this way:

```
inp_gen = SiestaBaseWorkChain.inputs_generator()
inputs = inp_gen.get_inputs_dict(structure, calc_engines, protocols)
```

The inputs of `get_inputs_dict` are explained in the *protocols documentation*. Then the user must define at least the input **iterate_over** in order to be able to submit the **SiestaIterator** WorkChain.

2.4.1.6 Converger workflow

Description

The **SiestaConverger** is a tool to facilitate convergence tests with Siesta. It extends the **SiestaIterator** to accept a target quantity that is checked after each step to evaluate whether convergence has been reached or not. The convergence check just consists in calculating the difference in the target quantity between the present step and the step before and comparing it with a threshold value passed by the user in input. An example on the use of the **SiestaConverger** is */aiida_siesta/examples/workflows/example_convergence.py*.

Supported Siesta versions

At least 4.0.1 of the 4.0 series, 4.1-b3 of the 4.1 series and the MaX-1.0 release, which can be found in the development platform (https://gitlab.com/siesta-project/siesta). For more up to date info on compatibility, please check the wiki.

Inputs

All the **SiestaIterator** inputs are as well inputs of the **SiestaConvereger**, they are described in the corresponding *documentation*. Additional inputs are:

- **target**, class `Str`, *Optional*

  The parameter the user wants to track in order to check if convergence has been reached. All the quantities returned in the **output_parameters** dictionary of the **SiestaBaseWorkChain** are accepted for this scope, excluding keys that don't have a *float* or *int* as a value. Typical values are the Kohn-Sham (`E_KS`), Free (`FreeE`), Band (`Ebs`), and Fermi (`E_Fermi`) energies, and the total spin (`stot`); however the user might also think to converge calculations-time related quantities.

  The *E_KS* is the default value.

- **threshold**, class `Float`, *Optional*

  The maximum difference between two consecutive steps to consider that convergence is reached. Default is `Float(0.01)`.

Outputs

The following outputs are returned:

- **converged** `Bool`

  Returning *True* or *False*, whether the target has converged or not.

- **converged_target_value** `Float`

  The value of the target when the convergence has been reached. Returned only if the convergence is succesfull.

- **converged_parameters** `Dict`

  The values for the parameters that was enough to achieve convergence. If converged is not achieved, it won't be returned.

Protocol system

The protocol system is not directly available for this WorkChain. However inputs of the **SiestaBaseWorkChain** can be obtained in a dictionary in this way:

```
inp_gen = SiestaBaseWorkChain.inputs_generator()
inputs = inp_gen.get_inputs_dict(structure, calc_engines, protocols)
```

The inputs of `get_inputs_dict` are explained in the *protocols documentation*. Then the user must define at least the input **iterate_over** in order to be able to submit the **SiestaConverger** WorkChain (if no **target** is specified, the *E_KS* is used).

2.4.1.7 Sequential Converger workflow

Description

The **SiestaSequentialConverger** is an iterator that sequentially runs **SiestaConvergers**. Once the convergence over a parameter is reached, the converged value is used for the following convergence test (on a new parameter). An example on the use of the **SiestaConverger** is */aiida_siesta/examples/workflows/example_seq_converger.py*

Supported Siesta versions

At least 4.0.1 of the 4.0 series, 4.1-b3 of the 4.1 series and the MaX-1.0 release, which can be found in the development platform (https://gitlab.com/siesta-project/siesta). For more up to date info on compatibility, please check the wiki.

Inputs

Two are the required inputs:

- **converger_inputs**, class `dict`, *Mandatory*

  A dictionary containing all the inputs required by the **SiestaConverger**, except the **iterate_over** port. The explanations of the converger inputs can be examined *here <siesta-converger-inputs>*. Please note that the normal inputs of a **SiestaBaseWorkChain** process (structure, parameters, basis, code, ...) must be included as well in this dictionary.

  The same default values as **SiestaConverger** apply if some ports are not specified here.

- **iterate_over**, class `list`, *Mandatory*

  There is a specific port for the quantities to iterate over and now the accepted value for this port is a *list*, not a dictionary like it was for the **SiestaConverger** or **SiestaIterator**. In fact, now the user should indicate a list of parameters that he/she wants to converge sequentially. A practical example:

```
iterate_over=[
    {
     'kpoints_0': [4,10,12,14,16,18,20],
     'kpoints_1': [4,10,12,14,16,18,20],
     'kpoints_2': [4,10,12,14,16,18,20],
    },
    {
     'meshcutoff': ["500 Ry", "600 Ry", "700 Ry", "800 Ry", "900 Ry"],
    },
    {
     'pao-energyshift': ["0.02 Ry", "0.015 Ry", "0.01 Ry", "0.005 Ry", "0.001 Ry"]
    }
]
```

  With this specification, we signal that we want to converge first the kpoints by increasing all components at the same time (assuming "zip" is selected as 'iterate_mode' in the **converger_inputs** dictionary), then the 'meshcutoff' and finally the 'energy shift'. The converged kpoints will be used for the convergence of 'meshcutoff', the converged kpoints and 'meshcutoff' will be used for the convergence process of 'energy shift'.

  Note that one can converge the same parameters again if wanted, for instance set up different rounds for kpoints convergence.

Warning: If one of the parameters does not converge, no action is taken and the following convergence step is performed using the inputs specified in **converger_inputs**, not using the last attempted value in the previous convergence. For instance, in the example above, if the `meshcutoff` does not converged at `900 Ry`, the `pao-energyshift` convergence will be done using the inputs parameters specified in the `parameters` of `converger_inputs`, not including `meshcutoff = "900 Ry"`.

Outputs

The following outputs are returned:

- **converged_target_value** `Dict`

  The value of the target when the convergence has been reached. Returned only if at least one of the sequential convergences has been completed succesfull.

- **converged_parameters** `Dict`

  The values for the parameters that was enough to achieve convergence. If converged is not achieved, it will be an empty dictionary.

- **unconverged_parameters** `List`

  If one or more parameters fail to converge, we list them in this output.

Protocol system

The protocol system is not directly available for this WorkChain. However inputs of the **SiestaBaseWorkChain** can be obtained in a dictionary in this way:

```
inp_gen = SiestaBaseWorkChain.inputs_generator()
inputs = inp_gen.get_inputs_dict(structure, calc_engines, protocols)
```

The inputs of `get_inputs_dict` are explained in the *protocols documentation*. Then the user can place these `inputs` in the **converger_inputs** dictionary (together with the other **SiestaConverger** inputs specifications). The input **iterate_over** is also required in order to be able to submit the **SiestaSequentialConverger** WorkChain and it must be set manually.

2.4.1.8 Basis optimization

Description

The AiiDA-siesta package offers three workchains to help users in selecting the optimal basis set for a given system:

1) The **SimplexBasisOptimization** that finds the minimum of a quantity (typically the basis enthalpy) varying a set of input variables (typically cutoff radii of orbitals) using the Nelder–Mead (simplex / amoeba) method. 2) The **TwoStepsBasisOpt** that performs a two level optimization, running simplex iterations followed by periodic restarts with new simplex hyper-tetrahedra of progressively smaller sizes. This replicates more closely the optimization util of the siesta distribution. 3) The **BasisOptimizationWorkChain** that performs a full optimization testing first basis cardinality and then applying the **SimplexBasisOptimization** to optimize all the radius of the orbitals.

The simplex optimization is performed taking advantage of the *aiida-optimize package <https://aiida-optimize.readthedocs.io/en/latest/index.html>*_, in particular the Nelder–Mead engine implemented in that package.

Supported Siesta versions

At least 4.0.1 of the 4.0 series, 4.1-b3 of the 4.1 series and the MaX-1.0 release, which can be found in the development platform (https://gitlab.com/siesta-project/siesta). For more up to date info on compatibility, please check the wiki.

SimplexBasisOptimization

The Nelder–Mead method (commonly known as simplex or amoeba method) is a numerical method to find the minimum of a function with N variables. A simplex is a special polytope of N+1 vertices in N dimensions (for instance a triangle in 2D, a tetrahedron in 3D and so forth). The Nelder–Mead methods in N dimensions starts from a set of N+1 test points arranged as a simplex. The value of the function is calculated at each test point and these values are then used in order to find a new test point and to replace one of the old test points with the new one in case it returns a smaller value for the function under investigation. Repeating the procedure, the technique progresses until all the N+1 test points produce values that are all within a threshold. When this happens the minimum has been reached. In the context of the basis optimization the function is usually the basis enthalpy (but also other quantities are supported) and the variable are the parameters defining the basis (typically cutoff radii of the basis orbitals).

An example of the use of **SimplexBasisOptimization** is in */aiida_siesta/examples/workflows/example_simplex.py*

Inputs

Inputs are organized in two namespaces and are described in the following:

- **siesta_base**, input namespace, *Mandatory*

  Accepts all the inputs of a **SiestaBaseWorkChain** (listed *here <siesta-base-wc-inputs>*) with the only mandatory modification to include in the "basis" input some variables to optimize. The variable must be defined using a dollar and a string. An example:

```
basis = Dict(dict={
 '%block pao-basis': "\nSi    2\n n=3    0    2\n 4.99376       $sz2 \n n=3    1    2 P␣
↪1\n 6.2538       $pz2 \n%endblock pao-basis"
 })
```

  An upper and lower value must be set for each variable and optionally one or more starting points (see next point in this list). Please note that variables are typically defined for orbitals radii in the `pao-basis` block, but one can also create variables for "higher level" keywords like the `energy-shift` or `split-norm`.

- **simplex**, input namespace, *Mandatory*

  Here all the inputs for the simplex method can be defined. They are listed in the next lines.

- **simplex.variables_dict** class `Dict`, *Mandatory*

  A dictionary containing all the info about the variables that are modified in order to find the minimum basis enthalpy. An example related to the basis block above:

```
variables_dict = Dict(dict={
    "sz2":[2.0,4.8,3.0],
    "pz2":[2.0,6.0,3.0]
    })
```

  The variables names must be defined here as keys of the dictionary and must correspond to the strings defined in the `basis` input, but removing the dollar symbol. The list associated to each string defines in this

order: 1) The lower limit for the variable, 2) The upper limit, 3) the starting value to construct the simplex hyper-tetrahedron. The up and down limit of the variables are used in such way: if the algorithm attempts the calculation of the function out of range, a huge value for the function is returned. The starting value is going to be the point from which the simplex hyper-tetrahedron is constructed. In particular, the first test point is directly formed by the specified starting points (in the example above is [3.0,3.0]). The other N test points are obtained substituing one component with `num + range * simplex_inps.initial_step_fraction`, where `num` is the defined starting point, `range` is the upper - lower limit and `simplex_inps.initial_step_fraction` is a number between 0 and 1 defined in the next point of this list. Supposing `simplex_inps.initial_step_fraction = 0.2`, in out example, the other two test points are [3.0,3.8] and [3.56,3.0].

When 3) is not defined, it is chosen randomly between the boundaries, but it is always suggested to set it since it will be used to construct the Alternatively to 3), N+1 values can be entered and this would correspond to define explicitly all the components of all the simplex initial points.

- **simplex.initial_step_fraction** class `Float`, *Optional*

  A fractional increment to be used in the construction of the starting simplex hyper-tetrahedron. See point above for more details. Default at `Float(0.4)`. It is ignored if all the components af all the test points are set in the point above.

- **simplex.max_iters** class `Int`, *Optional*

  The maximum iterations for the Nelder–Mead algorithm. Please note that an iteration step usually involves more then one new test point. So the points tested at the end will be way more than the `max_iters`. Once the `simplex.max_iters` is reached, the workchain stops returning the best simplex so far, even if the threshold convergence has not been reached. Default is `Int(40)`.

- **simplex.output_name** class `Str`, *Optional*

  The name of the output that needs to be minimized. In principle all the numerical values returned in the "output_parameters" of a **SiestaBaseWorkChain** are accepted, but typically the "basis_entalpy" or the "harris_energy" are of interest. Defalut is `Str("basis_entalpy")`

- **simplex.tolerance_function** class `Float`, *Optional*

  The tolerance accepted to define the optimization converged. If the values of the functions for all points in the simplex are all within the `simplex.tolerance_function`, the optimization is considered concluded. The default is `Float(0.01)`. Please note that the choice of this parameter must be related to the variance of the output function, therefore the default might be unreasonable for your application. In the future an extension implementing a fractional tolerance will be provided.

Outputs

The following outputs are returned:

- **last_simplex** class `List`

  The output containing the values of the last simplex. Always returned, even if the optimization does not reached the required tolerance. It is a list of lists. The first element of the list is always the best choice of the parameters obtained by the optimization so far.

- **optimal_process_input** class `List`

  This output contains the optimal set of parameters obtained after optimization. This corresponds to the first entry of the list return by the **last_simplex**, however it is returned only if the optimization succeed.

- **optimal_process_output** class `Float`

The value of the function for the optimal set of parameters obtained with the optimization. Returned only if the optimization succeed.

- **optimal_process_uuid** class `List`

  The uuid of the **SiestaBaseWorkChain** that has the **optimal_process_input** as variables and that returned the **optimal_process_output**. Returned only if the optimization succeed.

It is important to note that the optimization is entirely an AiiDA process, therefore the provenance of all calculation called is preserved. We can have a look at the attempted variables values and the obtained basis entalpy in this simple way. In the verdi shell:

```
node=load_node(<PK>)  #PK of your SimplexBasisOptimization
for wc in node.called[0].called:
     print(wc.inputs.the_values.get_list(),wc.outputs.ene.value)
```

And many more info can be extracted from the inputs and outputs of each run `wc`. These `wc` are **SiestaBaseWorkChain** wrapped into a thin layer that attach to each calculation the information needed by the optimizer.

TwoStepsBasisOpt

This workchain uses the **SimplexBasisOptimization**, but it adds a step in the optimization, which consists in restarting the simplex with a subsequently smaller **simplex.initial_step_fraction**. This is implemented in the original simplex optimization code that can be found in the Util of the SIESTA package. There the fractional step is called "lambda" and we will follow the same notation here.

Inputs

All the inputs of **SimplexBasisOptimization** are inputs of this workchain except the **simplex.initial_step_fraction**. This include the way to specify the optimization variables in the `siesta_base.basis` input. This workchain adds a further called **macrostep**. This allows:

- **macrostep.initial_lambda** class `Float`

  The value of lambda to be used as **simplex.initial_step_fraction** in the first iteration. Default `Float(0.4)`,

- **macrostep.lambda_scaling_factor** class `Float`

  The rate at which lambda decreases between from a macrostep to the other. Default `Float(0.5)`

- **macrostep.minimum_lambda** class `Float`

  When this value for lambda is reached, the macrostep iteration stops. Default `Float(0.01)`.

Outputs

Same outputs of **SimplexBasisOptimization**.

BasisOptimizationWorkChain

This workchain manages entirely the optimization of the basis sets for a SIESTA calculation. It first run calculations with different basis sizes (using the "PAO-BasisSize" option of SIESTA) and gets the size that gives minimum of the monitored quantity (e.g. basis enthalpy).

NOTE: This does not include yet the possibility to test different basis sizes for different species.

It then allow to add extra orbitals to the calculation manually and see if this leads to a further decrease in the monitored quantity.

Then automatically sets up a **SimplexBasisOptimization** according to an optimization schema defined by the user.

Inputs

All the inputs of **SimplexBasisOptimization** are inputs of this workchain except the **simplex.variables_dict**. Please note that whathever is specified in **siesta_base.basis** will be copied in every calculation. So we prevwnt in this keyword to set the basis bloch or the basis sizes since the alghoritm will take care of it. In **siesta_base.basis** can put keywords like the "pao-non-perturbative-polarization-schema" or choices on the pseudopotential grid.

Few more inputs are allowed:

- **basis_sizes** class `List` *Optional*

  The list of basis sizes to try out. Default `List(list=["DZ", "DZP", "TZ"])`.

- **add_orbital** class `List` *Optional*

  A dict of lists, the key of the dict must be the name of an element of the periodic table, the list must list the orbitals to add at that atom, for instance:

  ```
  add_orbital = Dict(dict={
    "Ca":["3d1","4f1"],
    "O" :["4f2"]
    })
  ```

  This would add a f orbital with two zetas for O and a d and f orbital to Ca (one zeta each). As already specified, the presence of this input implies an extra step between the check of basis cardinality and the actual **SimplexBasisOptimization**.

- **sizes_monitored_quantity** `List` *Optional*

  The quantity to monitor in the check of the cardinality. If not specified is going to be the same specified in **simplex.output_name**.

- **optimization_schema.global_energy_shift** `List`

  If set to true, the energy shift and the pao-split-norm are used as optimization variables, not the explicit radius of the basis block. Default is False

- **optimization_schema.global_split_norm** `List`

  If set to true, the pao-split-norm is optimized as a global variable. Please note that this can be used in conjunction with **global_energy_shift** in order to optimize only global variables and not the pao block, but it can be also used alone to set that the first zeta radii of the orbitals are optimized, but the second zetas no! If **optimization_schema.global_split_norm** is True and **optimization_schema.global_energy_shift** is False the basis block is created putting all the second and further zetas to zero and the globas pao-split-norm is a variable for optimization. Default False.

- **optimization_schema.charge_confinement** `List`

  If set to true, the empty orbitals will receive a charge confinement and the charge of the confinement is a variable for optimization. Default False

To conclude, the inputs allow to do various type of optimizations. As default all the radia are optimized, but this can be modified using the **optimization_schema** keywords

Outputs

Only one output is produced:

- **optimal_basis_block** class `Dict`

  Returning the optimal pao block, meaning the one that gives the minimum of the monitored quantity.

Protocol system

The protocol system is not directly available for this WorkChain. However inputs of the **SiestaBaseWorkChain** can be obtained in a dictionary in this way:

```
inp_gen = SiestaBaseWorkChain.inputs_generator()
inputs = inp_gen.get_inputs_dict(structure, calc_engines, protocols)
```

The inputs of `get_inputs_dict` are explained in the *protocols documentation*. Then the user can place these `inputs` in the **siesta_base** namespace.

2.4.1.9 Epsilon workflow

Description

The **EpsilonWorkChain** is a simple extension of the **SietaBaseWorkChain** that introduces a post-processing step to obtain the electronic contribution to the static dielectric constant from the epsilon_2(omega) data. For developers, this workchain can be taken as an example to understand how easy is to include simple post-processes on top of the **SietaBaseWorkChain**. An example on the use of the **EpsilonWorkChain** is in */aiida_siesta/examples/workflows/example_epsilon.py*.

Supported Siesta versions

At least 4.0.1 of the 4.0 series, 4.1-b3 of the 4.1 series and the MaX-1.0 release, which can be found in the development platform (https://gitlab.com/siesta-project/siesta). For more up to date info on compatibility, please check the wiki.

Inputs

All the **SiestaBaseWorkChain** inputs are as well inputs of the **EpsilonWorkChain**, therefore the system and DFT specifications (structure, parameters, etc.) can be defined as input in the WorkChain using the same syntax explained in the **SiestaBaseWorkChain** *documentation*. Here we only impose a mandatory definition of the **optical** input port.

Outputs

- All the outputs of **SiestaBaseWorkChain** are also outputs of this WorkChain, they can be explored in the relative section of the **SiestaBaseWorkChain**.
- **epsilon** `Float`

  The low frequency (static) dielectric constant (electronic contribution) computed from the eps2(omega) data using Kramers-Kronig relations.

Protocol system

The protocol system is available for this WorkChain. The `EpsilonWorkChain.inputs_generator()` makes available all the methods explained in the *protocols documentation*. In addition, the **optical** input is populated, setting the optical mesh equal to the kpoints mesh of the calculation, the "optical-broaden" to 0.5 eV and the "optical-polarization-type" to "polarized" with optical vector of [1.0 0.0 0.0].

2.4.1.10 NEB Base workflow

Description

The **SiestaBaseNEBWorkChain** is the core building block for the creation of workflows that enable the search of the Minimum Energy Pathway (MEP) connecting two local minima of the potential energy surface through the Nudge Elastic Band (NEB) method. In particular, this workchain performs NEB MEP optimizations starting from a guessed path and exploiting the LUA functionality of SIESTA. This workchain is very useful for the investigation of reaction paths and energy barriers. For instance, it can be used to study the energetic barrier for interstitial diffusion of an impurity in an host structure. For some concrete examples, look at the *aiida-siesta-barrier* project.

An example on the use of the **SiestaBaseNEBWorkChain** is in */aiida_siesta/examples/workflows/example_neb_ghost.py*.

Supported Siesta versions

At least 4.0.1 of the 4.0 series, 4.1-b3 of the 4.1 series and the MaX-1.0 release, which can be found in the development platform (https://gitlab.com/siesta-project/siesta). For more up to date info on compatibility, please check the wiki.

Inputs

Many of the **SiestaBaseWorkChain** inputs are as well inputs of the **SiestaBaseNEBWorkChain**, therefore the system and DFT specifications (structure, parameters, etc.) can be defined as input in the WorkChain using the same syntax explained in the **SiestaBaseWorkChain** *documentation*. The only exceptions are the **structure** and the **lua** namespace that are not explicit inputs for this workchain. In fact, more than one single structure is required by NEB method and they are passed through the dedicated input **starting_path**. The **lua** inputs are mostly defined internally except the lua script that is now named **neb_script**. A more detailed description of the two new inputs follows:

- **starting_path**, class `TrajectoryData`, *Mandatory*

  A set of structures collected in a `TrajectoryData` object. Each structure correspond to an image for the NEB method. The object must have the kinds of the structure as attributes.

- **neb_script**, class `SingleFileData`, *Mandatory*

  A lua script that controls the NEB calculation. An example can be seen in */aiida_siesta/examples/fixtures/lua_scripts/neb.lua*.

Note: The use of LUA scripts also requires the user to pass to aiida the environmental variable that indicates where the flos library is. More info *here*.

Outputs

- **neb_output_package**, class `TrajectoryData`, *Mandatory*

  A `TrajectoryData` object with the final structures after the NEB optimization and the energy of each one of them. Moreover the reaction barrier and

  other useful info are reported as attributes of the node.

Protocol system

No protocol system is in place for this workchain.

2.5 Tutorials

2.5.1 Tutorials

Tutorials to help user in the understanding of the use of AiiDA and the tools delivered within the `aiida-siesta` package.

2.5.1.1 2020, ICN2, Barcelona, Spain

Related resources	
Virtual Machine	Quantum Mobile 20.06.1
python packages	aiida-core 1.4.2, aiida-siesta 1.1.0,
codes	Siesta v4.1-rc1

These are the notes of the tutorial delivered to the "Theory and Simulation group" at ICN2 (Barcelona) the 19th of October 2020. The tutorial was carried on using the Quantum Mobile Virtual Machine, however the steps described below can be replicated (with some small modifications pointed out along the way) on a local machine to obtain a working AiiDA (and aiida-siesta) installation. Tutors: Emanuele Bosoni, Pol Febrer.

Installation

Installation is through `pip` after moving to a new virual environment (we use `virtualenvwrapper`, but any alternative is valid, only make sure to select a python version 3.6 or above). We call the virtual environment `tutorial`.

```
mkvirtualenv tutorial
workon tutorial
pip install aiida-siesta==1.1.0
```

This will install an appropriate version of `aiida-core` (last release at the time of the tutorial is 1.4.2).

Note: If you are not on the Quantum Mobile Virtual Machine, a preliminary step is required to install PostgreSQL and RabbitMQ, as described here.

Because we want to isolate the current AiiDA installation from other installations that might be present in the machine, we specify the aiida configuration directory in the virtual environment activation file:

```
echo 'export AIIDA_PATH=$VIRTUAL_ENV' >> $VIRTUAL_ENV/bin/postactivate
workon tutorial
```

The configuration directory is now `~/.virtualenvs/tutorial`.

Moreover we can add in the same file a line to activate tab-completion:

```
echo 'eval "$(_VERDI_COMPLETE=source verdi)"' >> $VIRTUAL_ENV/bin/postactivate
workon tutorial
```

Check the status of the installation:

```
verdi status
```

This should show that the configuration directory is set, but no profile has been created yet.

Setting up the AiiDA profile

The aiida profile is set up with one single command:

```
verdi quicksetup
```

An interactive shell will ask some data and after that the creation of the profile starts. It concludes with the message "Success: database migration completed". Now is time to scan for plugins and start the daemon:

```
reentry scan
verdi daemon start
```

If all the steps have been successful, you should be able to see all green ticks when typing

```
verdi status
```

and also be able to see "siesta.siesta" among the available calculations plugins:

```
verdi plugin list aiida.calculations
```

We are ready to set up a code and computer.

Computer and code setup

The setup of a computer is done through:

```
verdi computer setup
```

and filling in the interactive shell requirements. For the Quantum Mobile they are:

```
Computer label: localhost
Hostname: localhost
Description []: This machine
Transport plugin: local
Scheduler plugin: slurm
Shebang line (first line of each script, starting with #!) [#!/bin/bash]:
Work directory on the computer [/scratch/{username}/aiida/]: /home/max/aiidarun
Mpirun command [mpirun -np {tot_num_mpiprocs}]:
Default number of CPUs per machine: 2
```

Then a file is automatically opened with "vi" editor. It allows to insert text to prepend/append to any submission script. We don't require it. Therefore just press `Esc` and type `:wq`.

Any remote computer can be set up in the exact same way, just making sure to have a password-less access to it (ssh key) and selecting "ssh" as "Transport plugin".

The computer must be configured, this allows to select some advanced features:

```
verdi computer configure local localhost
```

The default values are ok for the sake of this tutorial, therefore just press enter. The computer setup is over and the success of this action can be checked with:

```
verdi computer test localhost
```

The next step is the setup of a code.

Note: This section covers the set up of the Siesta code already installed in the Quantum Mobile virtual machine. In case of local installation, make sure to include the right specifications for your Siesta code (that might be on a remote cluster).

The command is:

```
verdi code setup
```

and the interactive shell will facilitate the setting up of the code. For Quantum Mobile we insert:

```
Label: siesta-v4.1
Description []: siesta-v4.1-rc1
Default calculation input plugin: siesta.siesta
Installed on target computer? [True]:
Computer: localhost
Remote absolute path: /usr/local/bin/siesta
```

Again a file is opened, asking to specify an optional text to prepend/append to the submission script. Typically here is where we include the calls to modules that are needed to run the code. In our case we insert "ulimit -s unlimited" as prepend text. The writing mode of "vi" is activated pressing `i`, after the insertion, `Esc` and `:wq` to save the file.

The code is set up.

Note: It is also possible to set up computer and codes from a configuration file. See section *Setting up the hpcq* for an example.

Creating a pseudo family

Before starting to play with *aiida-siesta*, it can be useful to learn how to set up of a pseudopotential family. We download a set of pseudopotentials from PseudoDojo:

```
wget http://www.pseudo-dojo.org/pseudos/nc-sr-04_pbe_standard_psml.tgz
mkdir nc-sr-04_pbe_standard_psml
tar -xf nc-sr-04_pbe_standard_psml.tgz -C nc-sr-04_pbe_standard_psml
```

and store tham in the database under the name "nc-sr-04_pbe_standard_psml":

```
verdi data psml uploadfamily nc-sr-04_pbe_standard_psml nc-sr-04_pbe_standard_psml
↪"Scalar-relativistic psml standard"
```

Same can be done for psf pseudopotentials, for instance:

```
wget https://icmab.es/leem/SIESTA_MATERIAL/tmp_PseudoDojo/nc-sr-04_pbe_standard-psf.
↪tgz
tar -xf nc-sr-04_pbe_standard-psf.tgz
verdi data psf uploadfamily nc-sr-04_pbe_standard-psf nc-sr-04_pbe_standard-psf
↪"Scalar Relativistic psf"
```

Note: The presudopotentials sets used in this tutorial come with no guarantee!! Use with care!

Submit a single siesta calculation

Open the file `example_bands.py` and explore the setting up of the various inputs. Run the script with:

```
runaiida example_bands.py --dont-send
```

The option `--dont-send` has been added in order to activate the "dry_run" option that every aiida process has. This option allows to create all the inputs of the calculation, but do not submit it. You can explore in the folder `submit_test` how AiiDA prepared all the inputs of a siesta calculation for you.

Now run:

```
runaiida example_bands.py --send
```

AiiDA took charge of your script, created the inputs and submitted the calculation. Look at the state of the process with the command `verdi process show <pk>` as suggested in the shell. The `<pk>` number uniquely identify your calculation and it will be used later on.

In few seconds the calculation is finished. You will relized that when `verdi process show <pk>` shows "Finished" status and reports the outuds. We explore the outputs. This can be done from command line, for instance:

```
verdi data array show <PK_forces_and_stress>
```

however it is worth exploring the shell provided by AiiDA:

```
verdi shell
```

Inside the shell:

```
l=load_node(<PK_calculation>)
```

and explore all the methods making use of tab completion. For instance:

```
l.outputs.bands.export(path="Si_bands", fileformat="gnuplot", y_max_lim=10)
```

The command above creates a file that can be plot with gnuplot in order to visualize the bands. Open a new shell and type:

```
gnuplot --persist Si_bands
```

Take the chance to explore in the `verdi shell` some methods and attributes of data types associated to the inputs and outputs of a `SiestaCalculation`. Use tab complition of `l.inputs`, `l.outputs`, `l.attributes`, ..

The submission script can be modified very easily in order to run a `SiestaBaseWorkChain` instead of a `SiestaCalculation`. Look at the commented part of the `example_bands.py` script in order to understand the differences. The "dry_run" option is not available for the `SiestaBaseWorkChain`. A `SiestaBaseWorkChain` automatically takes care of fixing some common errors of a siesta calculation, therefore it adds robustness in running siesta calculations.

Protocols

Go back to the `verdi shell` and look at the following:

```
from aiida_siesta.workflows.base import SiestaBaseWorkChain
inp_gen=SiestaBaseWorkChain.inputs_generator()
```

You just imported the inputs generator for the `SiestaBaseWorkChain`. We can explore its functionality:

```
inp_gen.get_protocol_names()
inp_gen.get_spins()
```

And many more... Use tab complition to explore them. These methods allows you to understand which options you can pass to `get_filled_builder`, as will be explained in a second.

The main feature of the input generator is the possibility to obtain a `builder` (a tool that helps you build the inputs for the specific process) that is ready to be submitted:

```
struct = l.inputs.structure
calc_engines = {
    'siesta': {
        'code': "siesta-v4.1@localhost",
        'options': {'resources': {'num_machines': 1, "num_mpiprocs_per_machine": 1},
↪"max_wallclock_seconds": 3600}
        }
    }
builder = inp_gen.get_filled_builder(struct,calc_engines,"standard_psml")
```

The `calc_engines` is a dictionary with fixed keys, whose aim is to pass the computational resourses for the calculation.

Explore the `builder`:

```
builder.parameters.attributes
builder.basis.attributes
...
```

We can add spin polarization to the calculation with:

```
builder = inp_gen.get_filled_builder(struct,calc_engines,"standard_psml",spin=
↪"polarized")
```

Try again `builder.parameters.attributes`, what are the differences compared to before?

We could run the builder straight away, however:

```
inp_gen.get_protocol_info("standard_psml")
```

remind us that the protocol we are using does not support siesta-4.1 because it uses psml pseudopotentials.

Close the shell and look at the file `my_protocols_registry.yaml`. It contains a new set of inputs and the psf pseudos. This file can be modified at will and its content will become a new protocol. Simply do:

```
export AIIDA_SIESTA_PROTOCOLS="/home/max/abs_path_to/my_protocols_registry.yaml"
```

taking care of passing the correct absolute path where you have `my_protocols_registry.yaml`.

Now open the shell and:

```
from aiida_siesta.workflows.base import SiestaBaseWorkChain
inp_gen=SiestaBaseWorkChain.inputs_generator()
inp_gen.get_protocol_names()
```

The new protocol is on the list and we can use it to run a calculation:

```
l=load_node(<PK_calculation>)
struct = l.inputs.structure
calc_engines = {
    'siesta': {
        'code': "siesta-v4.1@localhost",
        'options': {'resources': {'num_machines': 1, "num_mpiprocs_per_machine": 1},
↪"max_wallclock_seconds": 3600}
        }
    }
builder = inp_gen.get_filled_builder(struct,calc_engines,"my_protocol")

from aiida.engine import run
run(builder)
```

The command `run` send the calcualation in the shell in interactive mode (does not submit to the builder as `submit` would do). Our set up will occupy the shell for a minute or so and at the end it will return the outputs of the calculation.

Using jupyter in the Quantum Mobile VM

For the next sections, we are going to use jupyter notebooks to make it more interactive. Installing jupyter in Quantum Mobile is quite easy. Since **jupyter has some incompatibilities with aiida** (to be solved with https://github.com/aiidateam/aiida-core/pull/4317), we are going to install it in the base python, which will make it accessible globally. So, if you are inside a virtual environment, just leave:

```
deactivate
```

And proceed to install jupyter:

```
pip3 install jupyter
```

Now, we just need to tell jupyter that our environment exists. For this, you need to activate the environment:

```
workon tutorial
```

And then use *ipykernel* to inform jupyter:

```
pip install ipykernel
ipython kernel install --user --name=tutorial
```

That's about it. **Let's move on!**

Run a convergence workflow

It's quite easy to run a convergence workflow using *aiida-siesta*. You can find detailed information about it in `this notebook`

However, as a quick summary you can do:

```
from aiida_siesta.workflows.converge import SiestaSequentialConverger
from aiida.engine import run

calc_node=load_node(<PK_calculation>)

run(SiestaSequentialConverger,

        iterate_over=[
                {"kpoints_0": [4,5,6,7,8,9,10,11,12,13,14,15]},
                {"kpoints_1": [4,5,6,7,8,9,10,11,12,13,14,15]}
        ],

        converger_inputs={
                'code':load_code('siesta-v4.1@localhost'),
                'pseudo_family': Str('nc-sr-04_pbe_standard-psf')
                'structure': calc_node.inputs.structure,
                'parameters': Dict(),
                'options': {'resources': {'num_machines': 1, "num_mpiprocs_per_machine
↪": 1},"max_wallclock_seconds": 3600}
                'batch_size': Int(3)
        }

)
```

to converge your structure's kpoints (first and second components), running three simulations at a time.

Create a WorkChain

In this section, we will guide you through your first steps at creating workchains.

Please download `this zip file` where you will find all the contents for the section. Then unzip it and enter the directory:

```
unzip first_workchain.zip
cd first_workchain
```

Once you are inside, launch jupyter:

```
jupyter notebook
```

and open the *First workchain.ipynb* notebook. From here, just follow the instructions on the notebook :)

Setting up the hpcq

We already set up a computer and code in the *Computer and code setup* section. Remote computers, and therefore HPCQ, are no different. To set them up, you need to follow the same steps. There's just one difference, you need to **generate the ssh keys** so that aiida can login to the remote computer in your behalf without needing the password.

You can generate them with:

```
ssh-keygen -t rsa -b 4096 -m PEM
```

And then register them to the hpcq so that it allows you to access:

```
ssh-copy-id <username>@10.100.2.51
```

Just with that, you would be able to access like *ssh <username>@10.100.2.51*, but aiida wants to access without knowing your username (*ssh 10.100.2.51*). For this, you need to:

```
vi ~/.ssh/config
```

And include the following lines to the file (with one empty line before and after):

```
Host 10.100.2.51
  User <username>
```

With this, you are all good to go! We just need to setup the computer and the code.

There's a **small gotcha** though with the hpcq: *Each farm needs to be setup as a different computer, as it has a different architecture and it runs a different compilation of the code.*

Therefore for each farm we will need to setup a computer with this configuration:

```
label: "hpcq-farm<farm name>"
hostname: "10.100.2.51"
description: "hpcq farm <farm name>"
shebang: "#!/bin/bash"
transport: "ssh"
scheduler: "slurm"
work_dir: "/home/ICN2/{username}/.aiida"
mpirun_command: "mpirun -np {tot_num_mpiprocs}"
mpiprocs_per_machine: <num cores per node of the farm>
```

where *<farm name>* and *<num cores per node of the farm>* need to be replaced by the appropiate values for each farm.

Correspondingly, we need to setup a code with this configuration:

```
label: "siesta_farm<farm name>"
description: "Siesta compilation to run in hpcq-farm<farm name>"
input_plugin: "siesta.siesta"
on_computer: true
remote_abs_path: <path_to_siesta>
computer: "hpcq-farm<farm name>"
prepend_text: |
        <load all modules that you need here>

        ulimit -l unlimited
        ulimit -s 51200
        ulimit -n 51200
custom_scheduler_commands:  "#SBATCH -p <farm name>"
```

We know this is a cumbersome process, therefore you can `download all the config files from here`.

Unzip the downloaded zip and enter the directory to check what you have there:

```
unzip aiida-hpcq-config.zip
cd aiida-hpcq-config
```

You still need to go through each of them manually. So, enter the *Computers* directory and setup the ones you want by running the following command:

```
verdi computer setup --config hpcq-farm<farm name>.yml
```

and then configure it:

```
verdi computer configure ssh hpcq-farm<farm name>
```

You can (should) test it to check that everything is ok:

```
verdi computer test hpcq-farm<farm name>
```

Then, for each farm that you set up, we need to set up its code. With the downloaded zip, you are provided some binaries for each farm in <siesta-binaries>. For a quick test, you can copy the *siesta-binaries* folder to your hpcq home, and then use the config files in the *Codes* directory:

```
verdi code setup --config siesta-farm<farm name>.yml
```

Now you will be able to submit calculations to the hpcq by setting the code input to the *siesta@hpcq-farm<farm name>* :)

2.5.1.2 2021, Virtual event

Related resources	
Virtual Machine	Siesta Mobile 0.2.0
python packages	aiida-core 1.6.1, aiida-siesta 1.2.0,
codes	Siesta Max-1.3.0-1

These are the notes of the tutorial delivered to the CECAM school "First-principles simulations of materials with SIESTA" running virtually from 28th of June to 2nd of July 2021. The tutorial was carried on using the Siesta Mobile Virtual Machine, however reference to the relevant aiida documentation is reported at the beginning. If you are running on Siesta Mobile, jump *here*.

Tutor: Emanuele Bosoni

Installation and setup (ONLY IF NOT IN SIESTA MOBILE)

Installation is through `pip` after moving to a new virual environment (we use `virtualenvwrapper`, but any alternative is valid, only make sure to select a python version 3.6 or above). We call the virtual environment `tutorial`.

```
mkvirtualenv tutorial
workon tutorial
pip install aiida==1.6.1
pip install aiida-siesta==1.2.0
```

Follow the instructions in the AiiDA documentation. to set up aiida.

Computer and code setup (ONLY IF NOT IN SIESTA MOBILE)

Follow the instructions to set up your computer and a siesta executable.

Creating a pseudo family

In the Siesta Mobile, activate the virtual environment `workon siesta_school`. Otherwise activate the environment you created before.

Check the status of aiida typing `verdi status`. Check the codes installed with `verdi code list`.

Before starting to play with *aiida-siesta*, it can be useful to learn how to set up a pseudopotential family containing a collection of pseudos from PseudoDojo. Just do:

```
aiida-pseudo install pseudo-dojo -v 0.4 -x PBE -r SR -p standard -f psml
```

This will install version 0.4 PBE scalar relativistic and standard accuracy in psml form under the name "PseudoDojo/0.4/PBE/SR/standard/psml".

If you have a FOLDER containing psf pseudopotentials, you can create a family with:

```
aiida-pseudo install family /PATH/TO/FOLDER/ FAM_NAME -P pseudo.psf
```

Submit a single siesta calculation

Open the file `example_bands.py` and explore the setting up of the various inputs. Focus in particular in the understanding of the structure definition and also notice how easy is in AiiDA to request the generation of an automatic k-point path for the bands. We use the pseudos family we created before. Run the script with (if not in Siesta Mobile, change the code name inside the script):

```
runaiida example_bands.py --dont-send
```

The option `--dont-send` has been added in order to activate the "dry_run" option that every aiida process has. This option allows to create all the inputs of the calculation, but do not submit it. You can explore in the folder `submit_test` how AiiDA prepared all the inputs of a siesta calculation for you.

Now run:

```
runaiida example_bands.py --send
```

AiiDA took charge of your script, created the inputs and submitted the calculation. Look at the state of the process with the command `verdi process show <pk>` as suggested in the shell. The `<pk>` number uniquely identify your calculation and it will be used later on.

In few seconds the calculation is finished. You will relized that when `verdi process show <pk>` shows "Finished" status and reports the outuds. We explore the outputs. This can be done from command line, for instance:

```
verdi data array show <PK_forces_and_stress>
```

however it is worth exploring the shell provided by AiiDA:

```
verdi shell
```

Inside the shell:

```
l=load_node(<PK_calculation>)
```

and explore all the methods making use of tab completion. For instance:

```
l.outputs.bands.export(path="Si_bands", fileformat="gnuplot", y_max_lim=10)
```

The command above creates a file that can be plot with gnuplot in order to visualize the bands. Open a new shell and type:

```
gnuplot --persist Si_bands
```

Take the chance to explore in the `verdi shell` some methods and attributes of data types associated to the inputs and outputs of a `SiestaCalculation`. Use tab complition of `l.inputs`, `l.outputs`, `l.attributes`, ..

The submission script can be modified very easily in order to run a `SiestaBaseWorkChain` instead of a `SiestaCalculation`. Look at the commented part of the `example_bands.py` script in order to understand the differences. If you want to try to run a SiestaBaseWorkChain, just uncomment the `inputs["option"] = Dict ...` part and comment the line above (`inputs['metadata']['options'] = ..` was just for the SiestaCalculation), change the definition of `process` and run the script. The "dry_run" option is not available for the `SiestaBaseWorkChain`. A `SiestaBaseWorkChain` automatically takes care of fixing some common errors of a siesta calculation, therefore it adds robustness in running siesta calculations.

Protocols

Go back to the `verdi shell` and look at the following:

```
from aiida_siesta.workflows.base import SiestaBaseWorkChain
inp_gen=SiestaBaseWorkChain.inputs_generator()
```

You just imported the inputs generator for the `SiestaBaseWorkChain`. We can explore its functionality:

```
inp_gen.get_protocol_names()
inp_gen.get_spins()
```

And many more... Use tab complition to explore them. These methods allows you to understand which options you can pass to `get_filled_builder`, as will be explained in a second.

The main feature of the input generator is the possibility to obtain a `builder` (a tool that helps you build the inputs for the specific process) that is ready to be submitted:

```
l=load_node(<PK_calculation>)  #The PK loaded before
struct = l.inputs.structure
calc_engines = {
    'siesta': {
        'code': "siesta-school--MaX-1.3.0-1@localhost",
        'options': {'resources': {'num_machines': 1, "num_mpiprocs_per_machine": 1},
↪"max_wallclock_seconds": 3600}
        }
    }
builder = inp_gen.get_filled_builder(struct,calc_engines,"standard_psml")
```

The `calc_engines` is a dictionary with fixed keys, whose aim is to pass the computational resourses for the calculation.

Explore the `builder`:

```
builder.parameters.attributes
builder.basis.attributes
...
```

We can add spin polarization to the calculation with:

```
builder = inp_gen.get_filled_builder(struct,calc_engines,"standard_psml",spin=
↪"polarized")
```

Try again `builder.parameters.attributes`, what are the differences compared to before?

We can run the builder straight away:

```
from aiida.engine import run
run(builder)
```

The calculation will take about 10 minutes, therefore let it run and go on with the tutorial. At the end of this section you can come back on this terminal and explore the results if you wish.

We are now going to create our own protocol. Look at the file `my_protocols_registry.yaml`. This is the way you specify a protocol in aiida-siesta, using YAML syntax. You can recognize the same pseudos family used before and other familiar siesta keywords. The `spin_additions` are added just for spin pilarized calculations. The `relax_additions` only for the times a relaxation is requested. The `atom_heuristics` are added just if in the structure there is the indicated element. Look at the corresponding docs, for more info.

This file can be modified at will and its content will become a new protocol. Simply look at the folder where you are `pwd` and attach the file to the correct environment variable, like that:

```
export AIIDA_SIESTA_PROTOCOLS="path_discovered_with_pwd/my_protocols_registry.yaml"
```

taking care of passing the correct absolute path where you have `my_protocols_registry.yaml`.

Now open the shell and:

```
from aiida_siesta.workflows.base import SiestaBaseWorkChain
inp_gen=SiestaBaseWorkChain.inputs_generator()
inp_gen.get_protocol_names()
```

The new protocol is on the list and we can use it to run a calculation:

```
l=load_node(<PK_calculation>)
struct = l.inputs.structure
calc_engines = {
    'siesta': {
        'code': "siesta-school--Max-1.3.0-1@localhost",
        'options': {'resources': {'num_machines': 1, "num_mpiprocs_per_machine": 1},
↪"max_wallclock_seconds": 3600}
        }
    }
builder = inp_gen.get_filled_builder(struct,calc_engines,"my_protocol")

from aiida.engine import run
run(builder)
```

The command `run` send the calcualation in the shell in interactive mode (does not submit to the builder as `submit` would do). Our set up will occupy the shell for a minute or so and at the end it will return the outputs of the calculation.

Run a convergence workflow

It's quite easy to run a convergence workflow using *aiida-siesta*.

For instance, in a verdi shell you can do (taking care again to integrate the correct PK):

```
from aiida_siesta.workflows.converge import SiestaSequentialConverger
from aiida.engine import run

calc_node=load_node(<PK_calculation>)

run(SiestaSequentialConverger,

        iterate_over=[
                {
                 "kpoints_0": [4,6,8,10,12,14,16],
                 "kpoints_1": [4,6,8,10,12,14,16],
                 "kpoints_2": [4,6,8,10,12,14,16],
                },
                {
                 'meshcutoff': ["500 Ry", "600 Ry", "700 Ry", "800 Ry", "900 Ry"],
                }
        ],

        converger_inputs={
                'code':load_code('siesta-school--MaX-1.3.0-1@localhost'),
                'pseudo_family': Str('PseudoDojo/0.4/PBE/SR/standard/psml'),
                'structure': calc_node.inputs.structure,
                'parameters': Dict(),
                'options': Dict(dict={'resources': {'num_machines': 1, "num_mpiprocs_
↪per_machine": 1},"max_wallclock_seconds": 3600}),
                'batch_size': Int(3)
        }

)
```

This code will converge your structure's kpoints (increasing all the components at the same time) and subsequently the meshcutoff using the converged kpoints. Three simulations at a time will be performed as specified by the `batch_size` input.

More info in the documentation.

Want to know more??

In general about Aiida (create your workflos and so on)? Aiida tutorials

On aiida siesta? docs

Ask me: ebosoni@icmab.es