AiiDA Siesta Plugin Documentation

Release 0.11

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

Mar 12, 2020

Contents

The aiida-siesta python package interfaces the SIESTA DFT code (http://www.icmab.es/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/albgar/aiida_siesta_plugin).

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.

We acknowledge partial support from the Spanish Research Agency (projects FIS2012-37549-C05-05, FIS2015-64886-C5-4-P and PGC2018-096955-B-C44) and by the MaX European Centre of Excellence funded by the Horizon 2020 EINFRA-5 program, Grant No. 676598.

We thank the AiiDA team, who are also supported by the [MARVEL National Centre for Competency in Research](<http://nccr-marvel.ch>) funded by the Swiss National Science Foundation


MARVEL
NATIONAL CENTRE OF COMPETENCE IN RESEARCH

CHAPTER 2

Contents:

2.1 Installation

2.1.1 Installation

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

Install the plugin by executing, from the top level of the plugin directory:

```
pip install -e .
```

As a pre-requisite, this will install an appropriate version of the aiida_core python framework.

Important: Next, do not forget to run the following commands

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

2.2 SIESTA plugins

2.2.1 Standard Siesta plugin

Description

A plugin for Siesta's basic functionality.

Supported Siesta versions

At least 4.0.1 of the 4.0 series, and 4.1-b3 of the 4.1 series, which can be found in the development platform (https://gitlab.com/siesta-project/siesta).

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

- **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 (for structure data) are blocked. 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

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.

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

  The PsfData and PsmlData classes have been implemented along the lines of the Upf class for QE.

  One pseudopotential file per atomic element. 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_siesta.data.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, created = PsfData.get_or_create(absname, use_first=True)
      for j in kinds:
              pseudos_dict[j]=pseudo
```

  Alternatively, a pseudo for every atomic species can be set with the **use_pseudos_from_family** method, if a family of pseudopotentials has been installed. For an example see example_psf_family.py

---

Note: The verdi command-line interface now supports entry points defined by external packages. We have implemented *verdi data psf* and *verdi data psml* suites of commands: *uploadfamily*, *exportfamily*, and *listfamilies*.

  It can be argued that a single *SiestaPseudo* class, with psf and psml subclasses, might have been implemented. But the *PsmlData* class aims to transcend Siesta and to be used by other codes.

- **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,
```

(continues on next page)

(continued from previous page)

```
'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, the Siesta calculation will not include any basis specification and it will run with the default Basis: DZP plus (many) other defaults.

- **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])
```

  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. This keyword is meant to facilitate the management of kpoints exploiting the functionality of the class `KpointsData`. The full list of kpoints must be passed to the class and they must be in units of the reciprocal lattice vectors. Moreover the cell must be set in the `KpointsData` class.

  This can be achieved manually listing a set of kpoints:

```
from aiida.orm import KpointsData
bandskpoints=KpointsData()
bandskpoints.set_cell(structure.cell, structure.pbc)
kpp = [(0.500,  0.250, 0.750), (0.500,  0.500, 0.500), (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.

Note: '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.

Warning: as explained in the aiida documentation, "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']
```

The final option (unrecommended) covers the situation when one really needs to maintain a specific convention for the structure or one needs to calculate the bands on a specific path that is not a high-symmetry direction, 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_cell(structure.cell, structure.pbc)
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 the use of 'bandskpoints.labels' in any other situation apart from the one 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.

- **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*.

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
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
```

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",
  "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 to include are specified in a global-variable in the parser. Currently they are the Kohn-Sham, Free, Band, and Fermi energies, and the total spin. These are converted to 'float'. As this dictionary is sorted, keys for program values and metadata are intermixed.

  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 (GPa) in array form.

- **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). 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`.

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 for 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'],
}
```

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'],
}
```

See for example example_ldos.py in aiida_siesta/examples/plugins/siesta.

2.2.2 STM plugin

Description

A plugin for Util/plstm

Supported Siesta versions

At least 4.0.1 of the 4.0 series, and 4.1-b3 of the 4.1 series, which can be found in the development platform (https://gitlab.com/siesta-project/siesta)

Inputs

- **parameters**, class `Dict`

A dictionary with a few parameters to specify the mode of calculation and the height or isovalue at which to process the LDOS:

```
{
  "z": "5.8"      # In Ang
}
```

(The *mode of calculation* is hard-wired to *constant-height* for now)

- **ldos_folder**, class `RemoteData`

The parent folder of a previous Siesta calculation in which the LDOS file was generated.

Outputs

- **stm_array** `ArrayData`

A collection of three 2D arrays (X, Y, Z) holding the section or topography information. They follow the *meshgrid* convention in Numpy. A contour plot can be generated with the *get_stm_image.py* script in the repository of examples.

- **output_parameters** `Dict`

At this point only parser information is returned.

Errors

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

2.3 SIESTA Workflows

2.3.1 SIESTA 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). In addition, the workchain is able to restart a calculation in case of failure (lack of electronic-structure or geometry relaxation convergence, termination due to walltime restrictions, etc).

Supported Siesta versions

At least 4.0.1 of the 4.0 series, and 4.1-b3 of the 4.1 series, which can be found in the development platform (https://gitlab.com/siesta-project/siesta).

Inputs

- **code**, class `Code`

A database object representing a Siesta executable.

- **structure**, class `StructureData`

A structure. See the plugin documentation for more details.

- **parameters**, class `Dict`

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 (e.g. for structure data) are blocked. 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:

```
{
  "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.

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

(Optional) A dictionary of PsfData or PsmlData objects representing the pseudopotentials for the calculation. If it is not input, a **pseudo_family** specification must be used (see below).

The PsfData and PsmlData classes have been implemented along the lines of the Upf class for QE.

- **pseudo_family**, class `Str`

(Optional) String representing the name of the pseudopotential family (that can be uploaded via the *verdi data psf* CLI interface) to be used.

- **basis**, class `Dict`

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.

- **kpoints**, class `KpointsData`

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.

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

- **bandskpoints**, class `KpointsData`

Reciprocal space points for the calculation of bands. They can be given as a simple list of k-points, as segments with start and end point and number of points, or as a complete automatic path, using the functionality of modern versions of the class.

If this node is not present, no band structure is computed.

- **settings**, class `Dict`

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

- **options**, class `Dict`

Execution options

- **clean_workdir**, class `Bool`

(Optional) If true, work directories of all called calculations will be cleaned out.

- **max_iterations**, class `Int`

(Optional) The maximum number of iterations allowed in the restart cycle for calculations.

Outputs

- **output_parameters** `Dict`

A dictionary with metadata and scalar result values from the last calculation executed.

- **output_structure** `StructureData`

Present only if the workchain is modifying the geometry of the system.

- **bands**, `BandsData`

Present only if a band calculation is requested (signaled by the presence of a **bandskpoints** input node of class KpointsData) Contains an array with the list of electronic energies for every kpoint. For spin-polarized calculations, there is an extra dimension for spin.

- **forces_and_stress** `ArrayData`

Contains the final forces (eV/Angstrom) and stresses (GPa) in array form.

- **remote_folder**, `RemoteData`

The working remote folder for the last calculation executed.

2.3.2 SIESTA Bands workflow

Description

The **SiestaBandsWorkchain** workflow can be used to visualize the electronic band structure of the system of interest. Its inputs are a structure object and a specification of the quality and cost level of the calculation. The latter is implemented internally, as in Quantum Espresso, as a set of *protocols*, which group operational parameters to offer the desired balance of accuracy and efficiency. Optionally, the workflow will relax the geometry of the system before computing the band structure. As discussed in the context of the Base workflow, the computation could be implemented as a single (restartable) SIESTA calculation, but it is instead segmented into different steps (optional relaxation followed by a final electronic-structure plus band calculation) to provide future support for different levels of accuracy in the two stages. Support for the *fat-bands* feature that tags energy levels with orbital projections will be added soon.

Supported Siesta versions

At least 4.0.1 of the 4.0 series, and 4.1-b3 of the 4.1 series, which can be found in the development platform (https://gitlab.com/siesta-project/siesta).

Inputs

- **code**, class `Code`

A database object representing a Siesta executable.

- **structure**, class `StructureData`

A structure. See the plugin documentation for more details.

- **protocol**, class `Str`

Either "standard" or "fast" at this point. Each has its own set of associated parameters.

- standard:

```
{
   'kpoints_mesh_offset': [0., 0., 0.],
   'kpoints_mesh_density': 0.2,
   'dm_convergence_threshold': 1.0e-4,
   'forces_convergence_threshold': "0.02 eV/Ang",
   'min_meshcutoff': 100, # In Rydberg (!)
   'electronic_temperature': "25.0 meV",
   'md-type-of-run': "cg",
   'md-num-cg-steps': 10,
   'pseudo_familyname': 'lda-ag',
   'atomic_heuristics': {
       'H': { 'cutoff': 100 },
       'Si': { 'cutoff': 100 }
   },
   'basis': {
       'pao-energy-shift': '100 meV',
       'pao-basis-size': 'DZP'
   }
}
```

- fast:

```
{
   'kpoints_mesh_offset': [0., 0., 0.],
   'kpoints_mesh_density': 0.25,
   'dm_convergence_threshold': 1.0e-3,
   'forces_convergence_threshold': "0.2 eV/Ang",
   'min_meshcutoff': 80, # In Rydberg (!)
   'electronic_temperature': "25.0 meV",
   'md-type-of-run': "cg",
   'md-num-cg-steps': 8,
   'pseudo_familyname': 'lda-ag',
   'atomic_heuristics': {
       'H': { 'cutoff': 50 },
       'Si': { 'cutoff': 50 }
   },
   'basis': {
       'pao-energy-shift': '100 meV',
       'pao-basis-size': 'SZP'
   }
 }
```

The *atomic_heuristics* dictionary is intended to encode the peculiarities of particular elements. It is work in progress.

The *basis* section applies globally for now.

Outputs

- **output_parameters** `Dict`

A dictionary with metadata and scalar result values from the final *scf+bands* calculation executed.

- **bands**, `BandsData`

Contains an array with the list of electronic energies for every kpoint. For spin-polarized calculations, there is an extra dimension for spin.

- **output_structure** `StructureData`

Present only if the workchain is modifying the geometry of the system.

2.3.3 SIESTA STM workflow

Description

The **SiestaSTMWorkchain** workflow is functionally very similar to the **SiestaBandsWorkchain** workflow, but instead of a band structure, the analysis stage produces a file with 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 a specialized program **plstm** to produce a plot of the LDOS in a 2D section at a given height in the unit cell (simulating the height of a STM tip), or a simulated topography map by recording the z coordinates with a given value of the LDOS.

The inputs to the STM workchain include a (possibly already relaxed) structure and the protocol specification. The energy window for the LDOS and the tip height or the LDOS iso-value can be in principle specified by the user if full control is desired (probably after evaluation of the results of the **SiestaBandsWorkchain** workflow), but for the purposes of a turn-key solution, a range of energies around the Fermi Level (or regions near to the HOMO and/or LUMO), and a range of heights should automatically be selected by the workflow and the results presented to the user for further consideration. The workflow executes the **plstm** program via an AiiDA plugin, which is also included in

the **aiida-siesta** distribution. Its parser stage 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, and 4.1-b3 of the 4.1 series, which can be found in the development platform (https://gitlab.com/siesta-project/siesta/).

Inputs

- **code**, class `Code`

A database object representing a Siesta executable.

- **stm_code**, class `Code`

A code associated to the STM (plstm) plugin (siesta.stm)

- **structure**, class `StructureData`

A structure. See the plugin documentation for more details.

- **height**, class `Float`

The height of the plane at which the image is desired (in Ang).

- **e1**, class `Float`

The lower limit of the energy window for which the LDOS is to be computed (in eV).

- **e2**, class `Float`

The upper limit of the energy window for which the LDOS is to be computed (in eV).

- **protocol**, class `Str`

Either "standard" or "fast" at this point. Each has its own set of associated parameters.

- standard:

```
{
   'kpoints_mesh_offset': [0., 0., 0.],
   'kpoints_mesh_density': 0.2,
   'dm_convergence_threshold': 1.0e-4,
   'forces_convergence_threshold': "0.02 eV/Ang",
   'min_meshcutoff': 100, # In Rydberg (!)
   'electronic_temperature': "25.0 meV",
   'md-type-of-run': "cg",
   'md-num-cg-steps': 10,
   'pseudo_familyname': 'lda-ag',
   'atomic_heuristics': {
       'H': { 'cutoff': 100 },
       'Si': { 'cutoff': 100 }
   },
   'basis': {
       'pao-energy-shift': '100 meV',
       'pao-basis-size': 'DZP'
   }
}
```

- fast:

```
{
   'kpoints_mesh_offset': [0., 0., 0.],
   'kpoints_mesh_density': 0.25,
   'dm_convergence_threshold': 1.0e-3,
   'forces_convergence_threshold': "0.2 eV/Ang",
   'min_meshcutoff': 80, # In Rydberg (!)
   'electronic_temperature': "25.0 meV",
   'md-type-of-run': "cg",
   'md-num-cg-steps': 8,
   'pseudo_familyname': 'lda-ag',
   'atomic_heuristics': {
       'H': { 'cutoff': 50 },
       'Si': { 'cutoff': 50 }
   },
   'basis': {
       'pao-energy-shift': '100 meV',
       'pao-basis-size': 'SZP'
   }
 }
```

The *atomic_heuristics* dictionary is intended to encode the peculiarities of particular elements. It is work in progress.

The *basis* section applies globally for now.

Outputs

- **stm_array** `ArrayData`

A collection of three 2D arrays (X, Y, Z) holding the section or topography information. They follow the *meshgrid* convention in Numpy. A contour plot can be generated with the *get_stm_image.py* script in the repository of examples.

2.4 Indices and tables

- genindex
- modindex
- search