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 MINECO (projects FIS2012-37549-C05-05 and FIS2015-64886-C5-4-P) 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 command

```
reentry scan -r aiida
```

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 (http://launchpad.net/siesta/).

Inputs

- **structure**, class `StructureData`

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:

```
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'])
```

- **parameters**, class `ParameterData`

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:

```
{
  "mesh-cutoff": "200 Ry",
  "dm-tolerance": "0.0001",
      "%block example-block": """
      first line
      second line              """,
}
```

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 ParameterData instance).

- **pseudo**, class `PsfData`

The PsfData class has been implemented along the lines of the Upf class for QE.

One pseudopotential file per atomic element. Several species in the Siesta sense can share the same pseudopotential. For the example above:

```
pseudo_file_to_species_map = [ ("C.psf", ['C', 'Cred']),
                               ("H.psf", 'H')
                               ]
```

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. (But the family approach does not yet support multiple species sharing the same pseudopotential.)

Note: The verdi command-line interface has recently been upgraded to support entry points defined by external packages. We have implemented a *verdi data psf* family of commands: *uploadfamily*, *exportfamily*, and *listfamilies*.

- **basis**, class `ParameterData`

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 `ParameterData`

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

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.out` method.

The output parser takes advantage of the structured output available in Siesta as a Chemical Markup Language (CML) file. The CML-writer functionality should be compiled in and active in the run!

- **output_parameters** `ParameterData` (accessed by `calculation.res`)

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",
```

(continues on next page)

(continued from previous page)

```
  "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.

- **output_array** `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_array**, `BandsData`

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

No trajectories have been implemented yet.

Errors

Errors during the parsing stage are reported in the log of the calculation (accessible with the `verdi calculation logshow` command). Moreover, they are stored in the ParameterData under the key `warnings`, and are accessible with `Calculation.res.warnings`.

Restarts

A restarting capability is implemented following the basic idiom:

```
c = load_node(Failed_Calc_PK)
c2 = c.create_restart(force_restart=True)
c2.store_all()
c2.submit()
```

The density-matrix file is copied from the old calculation scratch folder to the new calculation's one. If an **ouput_structure** node is available, it is used as the structure for restarting.

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

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 ParameterData.

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

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

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'],
}
```

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 (http://launchpad.net/siesta/).

Inputs

- **parameters**, class `ParameterData`

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)

- **parent_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** `ParameterData` (accessed by `calculation.res`)

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 calculation logshow` 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 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 (http://launchpad.net/siesta/).

Inputs

- **code**, a code
- **structure**, class `StructureData`

A structure. See the plugin documentation for more details.

- **parameters**, class `ParameterData`

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             """,
}
```

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 ParameterData instance).

- **pseudos**

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

The PsfData class has been implemented along the lines of the Upf class for QE.

- **pseudo_family**, a Str

(Optional)

- **basis**, class `ParameterData`

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 `ParameterData`

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

- **options**, class `ParameterData`

Execution options

- **clean_workdir**, Bool
- **max_iterations**, Int

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

Outputs

- **output_parameters** `ParameterData` (accessed by `calculation.res`)

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_array**, `BandsData`

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

- **remote_folder**

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 (http://launchpad.net/siesta/).

Inputs

- **code**, a code
- **structure**, class `StructureData`

A structure. See the plugin documentation for more details.

- **protocol**, 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

- **scf_plus_band_parameters** `ParameterData`

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

- **bandstructure**, `BandsData`

Contains the list of electronic energies for every kpoint. For spin-polarized calculations, the 'bands' array has an extra dimension for spin.

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 (http://launchpad.net/siesta/).

Inputs

- **code**, a code associated to the Siesta plugin
- **stm_code**, a code associated to the STM (plstm) plugin
- **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**, 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",
```

(continues on next page)

(continued from previous page)

```
    '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_structure** `ParameterData`

The final relaxed structure (if applicable)

- **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