sisl Documentation

Release 0.8.2

Nick R. Papior

Mar 31, 2017

Contents

sisl has a number of features that makes it easy to jump right into and perform a large variation of tasks.

1. Easy creation of geometries. Similar to ASE sisl provides an easy scripting engine to create and manipulate geometries. The goal of sisl is not specifically DFT-related software which typically only works with a limited number of atoms. One of the main features of sisl is the enourmously fast creation and manipulation of geometries such as attaching two geometries together, rotating atoms, removing atoms, changing bond-lengths etc. Everything is optimized for extremely large scale systems `>1,000,000` atoms such that creating geometries for tight-binding models becomes a breeze.
2. Easy creation of tight-binding Hamiltonians via intrinsic and very fast algorithms for creation sparse matrices. One of the key-points is that the Hamiltonian is treated as a matrix. I.e. one may easily specify couplings without using routine calls. For large systems, `>100,000`, it also becomes advantegeous to iterate on sub-grids of atoms to speed up the creation by orders of magnitudes. sisl intrinsically implements such algorithms.
3. Post-processing of data from DFT software. One may easily add additional post-processing tools to use sisl on non-implemented data-files.

CHAPTER 1

Installation

sisl is very easy to install using any of your preferred methods.

pip

Installing sisl using PyPi can be done using

```
pip install sisl
```

conda

Installing sisl using conda can be done using

    conda install -c zerothi sisl

On conda sisl is also shipped in a developer installation for more up-to-date releases, this may be installed using:

```
conda install -c zerothi sisl-dev
```

Manual installation

sisl may also be installed using the regular `setup.py` script. To do this the following packages are required to be in `PYTHONPATH`:

- six
- setuptools
- numpy
- scipy
- netCDF4
- A fortran compiler

If the above listed items are installed, sisl can be installed by first downloading the latest release on this page. Subsequently install sisl by

```
python setup.py install --prefix=<prefix>
```

CHAPTER 2

Scripts

`sisl` implements a set of command-line utitilies that enables easy interaction with *all* the data files compatible with `sisl`.

`sdata`

The `sdata` executable is a tool for reading and performing actions on *all* `sisl` file formats applicable (all `Sile` 's).

Essentially it performs operations dependent on the file that is being processed. If for instance the file contains any kind of `Geometry` it allows the same operations as `sgeom`.

For a short help description of the possible uses do:

```
sdata <in> --help
```

which shows a help dependent on which kind of file `<in>` is.

As the options for this utility depends on the input *file*, it is not completely documented.

Siles with `Geometry`

If the `Sile` specified contains a `Geometry` one gets *all* the options like `sgeom`. I.e. `sdata` is a generic form of the `sgeom` script.

Siles with `Grid`

If the `Sile` specified contains a `Grid` one gets *all* the options like `sgrid`. I.e. `sdata` is a generic form of the `sgrid` script.

`sgeom`

The `sgeom` executable is a tool for reading and transforming general coordinate formats to other formats, or alter them.

For a short help description of the possible uses do:

```
sgeom --help
```

Here we list a few of the most frequent used commands.

Conversion

The simplest usage is transforming from one format to another format. `sgeom` takes at least two mandatory arguments, the first being the input file format, and the second (and any third + argumets) the output file formats

```
sgeom <in> <out> [<out2>] [[<out3>] ...]
```

Hence to convert from an **fdf** SIESTA input file to an **xyz** file for plotting in a GUI program one can do this:

```
sgeom RUN.fdf RUN.xyz
```

and the `RUN.xyz` file will be created.

Remark that the input file *must* be the first argument of `sgeom`.

Available formats

The currently available formats are:

- **xyz**, standard coordinate format Note that the the *xyz* file format does not *per see* contain the cell size. The `XYZSile` writes the cell information in the `xyz` file comment section (2nd line). Hence if the file was written with sisl you retain the cell information.
- **gout**, reads geometries from GULP output
- **nc**, reads/writes NetCDF4 files created by SIESTA
- **TBT.nc/PHT.nc**, reads NetCDF4 files created by TBtrans/PHtrans
- **tb**, intrinsic file format for geometry/tight-binding models
- **fdf**, SIESTA native format
- **XV**, SIESTA coordinate format with velocities
- **POSCAR/CONTCAR**, VASP coordinate format, does *not* contain species, i.e. returns Hydrogen geometry.
- **ASCII**, BigDFT coordinate format
- **win**, Wannier90 input file
- **xsf**, XCrySDen coordinate format

Advanced Features

More advanced features are represented here.

The `sgeom` utility enables highly advanced creation of several geometry structures by invocing the arguments *in order*.

I.e. if one performs:

```
sgeom <in> --repeat x 3 repx3.xyz --repeat y 3 repx3_repy3.xyz
```

will read `<in>`, repeat the geometry 3 times along the first unit-cell vector, store the resulting geometry in `repx3.xyz`. Subsequently it will repeat the already repeated structure 3 times along the second unit-cell vector and store the now `3x3` repeated structure as `repx3_repy3.xyz`.

Repeating/Tiling structures

One may use periodicity to create larger structures from a simpler structure. This is useful for creating larger bulk structures. To repeat a structure do

```
sgeom <in> --repeat [ax|yb|zc] <int> <out>
```

which repeats the structure one atom at a time, `<int>` times, in the corresponding direction. Note that `x` and `a` correspond to the same cell direction (the first).

To repeat the structure in *chunks* one can use the `--tile` option:

```
sgeom <in> --tile [ax|yb|zc] <int> <out>
```

which results in the same structure as `--repeat` however with different atomic ordering.

Both tiling and repeating have the shorter variants:

```
sgeom <in> -t[xyz] <int> -r[xyz] <int>
```

to ease the commands.

To repeat a structure 4 times along the *x* cell direction:

```
sgeom RUN.fdf --repeat x 4 RUN4x.fdf
sgeom RUN.fdf --repeat-x 4 RUN4x.fdf
sgeom RUN.fdf --tile x 4 RUN4x.fdf
sgeom RUN.fdf --tile-x 4 RUN4x.fdf
```

where all the above yields the same structure, albeit with different orderings.

Rotating structure

To rotate the structure around certain cell directions one can do:

```
sgeom <in> --rotate [ax|yb|zc] <angle> <out>
```

which rotates the structure around the origo with a normal vector along the specified cell direction. The input angle is in degrees and *not* in radians. If one wish to use radians append an `r` in the angle specification.

Again there are shorthand commands:

```
sgeom <in> -R[xyz] <angle>
```

Combining command line arguments

All command line options may be used together. However, one should be aware that the order of the command lines determine the order of operations.

If one starts by repeating the structure, then rotate it, then shift the structure, it will be different from, shift the structure, then rotate, then repeat.

Be also aware that outputting structures are done *at the time in the command line order*. This means one can store the intermediate steps while performing the entire operation.

sgrid

The `sgrid` executable is a tool for manipulating a simulation grid and transforming it into CUBE format for plotting 3D data in, *e.g.* VMD or XCrySDen.

Currently this is primarily intended for usage with SIESTA.

For a short help description of the possible uses do:

```
sgrid --help
```

Here we list a few of the most frequent used commands. Note that all commands are available via Python scripts and the `Grid` class.

Creating CUBE files

The simplest usage is converting a grid file to CUBE file using

```
sgrid Rho.grid.nc Rho.cube
```

which converts a SIESTA grid file of the electron density into a corresponding CUBE file. The CUBE file writeout is implemented in `Cube`.

Conveniently CUBE files can accomodate geometries and species for inclusion in the 3D plot and this can be added to the file via the `--geometry` flag, any geometry format implemented in `sisl` are also compatible with `sgrid`.

```
sgrid Rho.grid.nc --geometry RUN.fdf Rho.cube
```

the shorthand is `-g`.

Grid differences

Often differences between two grids are needed. For this one can use the `--diff` flag which takes one additional grid file for the difference. I.e.

```
sgrid Rho.grid.nc[0] -g RUN.fdf --diff Rho.grid.nc[1] diff_up-down.cube
```

which takes the difference between the spin up and spin down in the same `Rho.grid.nc` file.

Reducing grid sizes

Often grids are far too large in that only a small part of the full cell is needed to be studied. One can remove certain parts of the grid after reading, before writing. This will greatly decrease the output file *and* greatly speed-up the process as writing huge ASCII files is *extremely* time consuming. There are two methods for reducing grids:

```
sgrid <file> --sub x <pos|<frac>f>
sgrid <file> --remove x [+-]<pos|<frac>f>
```

This needs an example, say the unit cell is an orthogonal unit-cell with side lengths 10x10x20 Angstrom. To reduce the cell to a middle square of 5x5x5 Angstrom you can do:

```
sgrid Rho.grid.nc --sub x 2.5:7.5 --sub y 2.5:7.5 --sub z 7.5:12.5 5x5x5.cube
```

note that the order of the reductions are made in the order of appearence. So *two* subsequent sub/remove commands with the same direction will not yield the same final grid. The individual commands can be understood via

- `--sub x 2.5:7.5`: keep the grid along the first cell direction above 2.5 Å and below 5 Å.
- `--sub y 2.5:7.5`: keep the grid along the second cell direction above 2.5 Å and below 5 Å.
- `--sub z 7.5:12.5`: keep the grid along the third cell direction above 7.5 Å and below 12.5 Å.

When one is dealing with fractional coordinates is can be convenient to use fractional grid operations. The length unit for the position is *always* in Ångstrøm, unless an optional **f** is appended which forces the unit to be in fractional position (must be between 0 and 1).

Averaging and summing

Sometimes it is convenient to average or sum grids along cell directions:

```
sgrid Rho.grid.nc --average x meanx.cube
sgrid Rho.grid.nc --sum x sumx.cube
```

which takes the average or the sum along the first cell direction, respectively. Note that this results in the number of partitions along that direction to be 1 (not all 3D software is capable of reading such a CUBE file).

Advanced features

The above operations are not the limited use of the `sisl` library. However, to accomblish more complex things you need to manually script the actions using the `Grid` class and the methods available for that method. For inspiration you can check the `sgrid` executable to see how the commands are used in the script.

CHAPTER 3

File formats

`sisl` implements a generic interface for interacting with many different file formats. When using the command line utilities all these files are accepted as input for, especially *sdata* while only those which contains geometries (`Geometry`) may be used with *sgeom*.

In `sisl` any file is named a `Sile` to destinguish it from `File`.

Here is a list of the currently supported file-formats with the file-endings defining the file format:

xyz `XYZSile` file format, generic file format for many geometry viewers.

cube `CUBESile` file format, real-space grid file format (also contains geometry)

xsf `XSFSile` file format, **XCrySDen_** file format

ham `HamiltonianSile` file format, native file format for `sisl`

dat `TableSile` for tabular data

Below there is a list of file formats especially targetting a variety of DFT codes.

- **BigDFT_** File formats inherent to **BigDFT_**:

  ascii `ASCIISileBigDFT` input file for BigDFT, currently only implements geometry

- **SIESTA_** File formats inherent to **SIESTA_**:

  fdf `fdfSileSiesta` input file for SIESTA

  bands `bandsSileSiesta` contains the band-structure output of SIESTA, with *sdata* one may plot this file using the command-line.

  out `outSileSiesta` output file of SIESTA, currently this may be used to query certain elements from the output, such as the final geometry, etc.

  grid.nc `gridncSileSiesta` real-space grid files of SIESTA. This `Sile` allows reading the **NetCDF_** output of SIESTA for the real-space quantities, such as, electrostatic potential, charge density, etc.

  nc `ncSileSiesta` generic output file of SIESTA (only `>=4.1`). This `Sile` may contain *all* real-space grids, Hamiltonians, density matrices, etc.

  TSHS `TSHSSileSiesta` contains the Hamiltonian (read to get a `Hamiltonian` instance) and overlap matrix from a **TranSIESTA_** run.

TBT.nc `tbtncSileSiesta` is the output file of **TBtrans_** which contains all transport related quantities.

TBT.AV.nc `tbtavncSileSiesta` is the **k**-averaged equivalent of `tbtncSileSiesta`, this may be generated using *sdata siesta.TBT.nc –tbt-av*.

XV `XVSileSiesta` is the currently runned geometry in SIESTA.

- **VASP_** File formats inherent to VASP:

  POSCAR `POSCARSileVASP` contains the geometry of the VASP run.

  CONTCAR `CONTCARSileVASP` is the continuation geometries from VASP.

- **Wannier90_** File formats inherent to Wannier90:

  win `winSileW90` is the seed file for Wannier90. From this one may read the `Geometry` or the `Hamiltonian` if it has been output by Wannier90.

CHAPTER 4

Welcome to sisl documentation!

`sisl` is a tool to manipulate an increasing amount of density functional theory code input and/or output. It is also a tight-binding code which implements extremely fast and scalable tight-binding creation algorithms (`>1,000,000` orbitals). In particular is `sisl` developed with TBtrans in mind to act as a tight-binding Hamiltonian input engine for *N*-electrode transport calculations.

Features

`sisl` consists of several distinct features:

- Geometries; create, extend, combine, manipulate different geometries readed from a large variety of DFT-codes and/or from generically used file formats.
- Hamiltonian; easily create tight-binding Hamiltonians with user chosen number of orbitals per atom. Or read in Hamiltonians from DFT software such as SIESTA, Wannier90, etc. Secondly, there is intrinsic capability of orthogonal *and* non-orthogonal Hamiltonians.
- Generic output files from DFT-software. A generic set of output files are implemented which provides easy examination of output files.
- Command line utilities for processing of data files for a wide variety of file formats:
  - *sdata* Read and transform *any* `sisl` data file. This script is capable of handling geometries, grids, special data files such as binary files etc.
  - *sgeom* a geometry conversion tool which reads and writes many commonly encounted files for geometries, such as XYZ files etc. as well as DFT related input and output files.
  - *sgrid* a real-space grid conversion tool which reads and writes many commonly encounted files for real-space grids. *Mainly targetted SIESTA_*.

Introduction

Installation

The easiest way to install `sisl` is via the **`pypi`** interface. Install via:

```
pip install sisl
```

In case you are using `conda` simply do:

```
conda install -c zerothi sisl
```

Alternatively you can download the releases on the release page. And install via the regular **`setup.py`** interface:

```
python setup.py install
```

which will install `sisl` in your default location, use **`--prefix <path>`** for manual control of the placement.

Requirements

To succesfully use `sisl` these Python packages must be installed:

- six
- setuptools
- numpy (`>=1.9`)
- scipy
- netCDF4

CHAPTER 5

API links

sisl	sisl package
sisl.atom	Atomic information in different object containers.
sisl.geometry	Geometry class to retain the atomic structure.
sisl.grid	Define a grid
sisl.supercell	Define a supercell

Indices

- genindex
- modindex
- search