sisl Documentation

Release 0.8.5

Nick R. Papior

Jul 21, 2017

Contents

sisl has a number of features which 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 targets a limited number of atoms. One of the main features of sisl is the enourmously fast creation and manipulation of very large 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 creating 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, `>10,000` atoms, it 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

Package

sisl is mainly a Python package with many intrinsic capabilities.

DFT

Many intrinsic DFT program files are handled by sisl and extraction of the necessary physical quantities are easily performed.

Its main focus has been SIESTA which thus has the largest amount of implemented output files.

Geometry manipulation

Geometries can easily be generated from basic routines and enables easy repetitions, additions, removal etc. of different atoms/geometries, for instance to generate a graphene flake one can use this small snippet:

```
>>> import sisl
>>> graphene = sisl.geom.graphene(1.42).repeat(100, 0).repeat(100, 1)
```

which generates a graphene flake of $2 * 100 * 100 = 20000$ atoms.

CHAPTER 2

Command line usage

The functionality of sisl is also extended to command line utilities for easy manipulation of data from DFT programs. There are a variety of commands to manipulate generic data (`sdata`), geometries (`sgeom`) or grid-related quantities (`sgrid`).

CHAPTER 3

Installation

sisl is 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 by

```
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 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-python
- 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 4

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 5

Tutorials

sisl is shipped with these tutorials which introduces the basics.

All examples are assumed to have this in the header:

```
import numpy as np
from sisl import *
```

to enable numpy and sisl.

Below is a list of the current tutorials:

Geometry creation – part 1

To create a `Geometry` one needs to define a set of attributes. The *only* required information is the atomic coordinates:

```
>>> single_hydrogen = Geometry([[0., 0., 0.]])
>>> print(single_hydrogen)
{na: 1, no: 1, species:
 {Atoms(1):
    (1) == [H, Z: 1, orbs: 1, mass(au): 1.00794, maxR: -1.00000],
 },
 nsc: [1, 1, 1], maxR: -1.0
}
```

this will create a `Geometry` object with 1 Hydrogen atom with a single orbital (default if not specified), and a supercell of 10 A in each Cartesian direction. When printing a `Geometry` object a list of information is printed in an XML-like fashion. `na` corresponds to the total number of atoms in the geometry, while `no` refers to the total number of orbitals. The species are printed in a sub-tree and `Atoms(1)` means that there is one distinct atomic specie in the geometry. That atom is a Hydrogen, with mass listed in atomic-units. `maxR` refers to the maximum range of all the orbitals associated with that atom. A negative number means that there is no specified range. Lastly `nsc` refers to the number of neighbouring super-cells that is represented by the object. In this case `[1, 1, 1]` means that it is a molecule and there are no super-cells (only the unit-cell).

To specify the atomic specie one may do:

```
>>> single_carbon = Geometry([[0., 0., 0.]], Atom('C'))
```

which changes the Hydrogen to a Carbon atom. See <link to atom_01.rst> on how to create different atoms.

To create a geometry with two different atomic species, for instance a chain of alternating Natrium an Chloride atoms, separated by 1.6 A one may do:

```
>>> chain = Geometry([[0. , 0., 0.],
                      [1.6, 0., 0.]], [Atom('Na'), Atom('Cl')],
                      [3.2, 10., 10.])
```

note the last argument which specifies the Cartesian lattice vectors. sisl is clever enough to repeat atomic species if the number of atomic coordinates is a multiple of the number of passed atoms, i.e.:

```
>>> chainx2 = Geometry([[0. , 0., 0.],
                        [1.6, 0., 0.],
                        [3.2, 0., 0.],
                        [4.8, 0., 0.]]], [Atom('Na'), Atom('Cl')],
                        [6.4, 10., 10.])
```

which is twice the length of the first chain with alternating Natrium and Chloride atoms, but otherwise identical.

This is the most basic form of creating geometries in sisl and is the starting point of almost anything related to sisl.

Geometry creation – part 2

Many geometries are intrinsically enabled via the `sisl.geom` submodule.

Here we list the currently default geometries:

- `honeycomb` (graphene unit-cell):

```
hBN = geom.honeycomb(1.5, [Atom('B'), Atom('N')])
```

- `graphene` (equivalent to `honeycomb` with Carbon atoms):

```
graphene = geom.graphene(1.42)
```

- Simple-, body- and face-centered cubic as well as HCP All have the same interface:

```
sc = geom.sc(2.5)
bcc = geom.bcc(2.5)
fcc = geom.fcc(2.5)
hcp = geom.hcp(2.5)
```

- Nanotubes with different chirality:

```
ntb = geom.nanotube(1.54, chirality=(n, m))
```

- Diamond:

```
d = geom.diamond(3.57)
```

Specifying super-cell information

An important thing when dealing with geometries in how the *super-cell* is used. First, recall that the number of supercells can be retrieved by:

```
>>> geometry = Geometry([[0, 0, 0]])
>>> print(geometry)
{na: 1, no: 1, species:
```

```
 {Atoms(1):
    (1) == [H, Z: 1, orbs: 1, mass(au): 1.00794, maxR: -1.00000],
 },
 nsc: [1, 1, 1], maxR: -1.0
}
>>> geometry.nsc # or geometry.sc.nsc
array([1, 1, 1], dtype=int32)
```

where `nsc` is the specific super-cell information. In the default case only the unit-cell is taken into consideration (`nsc: [1, 1, 1]`). However when using the `Geometry.close` or `Geometry.within` functions one may retrieve neighbouring atoms depending on the size of the supercell.

Specifying the number of super-cells may be done when creating the geometry, or after it has been created:

```
>>> geometry = Geometry([[0, 0, 0]], sc=SuperCell(5, [3, 3, 3]))
>>> geometry.nsc
array([3, 3, 3], dtype=int32)
>>> geometry.set_nsc([3, 1, 5])
>>> geometry.nsc
array([3, 1, 5], dtype=int32)
```

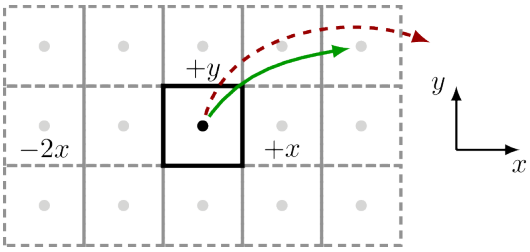
The final geometry enables intrinsic routines to interact with the 2 closest neighbouring cells along the first lattice vector (`1 + 2 == 3`), and the 4 closest neighbouring cells along the third lattice vector (`1 + 2 + 2 == 5`). Note that the number of neighbouring supercells is *always* an uneven number because if it connects in the positive direction it also connects in the negative, hence the primary unit-cell plus 2 per neighbouring cell.

Example – square

Here we show a square 2D lattice with one atom in the unit-cell and a supercell which extends 2 cells along the Cartesian x lattice vector (5 in total) and 1 cell along the Cartesian y lattice vector (3 in total):

```
>>> square = Geometry([[0.5,0.5,0]], sc=SuperCell([1,1,10], [5, 3, 1]))
```

which results in this underlying geometry:



With this setup, sisl, can handle couplings that are within the defined supercell structure, see green, full arrow. Any other couplings that reach farther than the specified supercell cannot be defined (and will thus *always* be zero), see the red, dashed arrow.

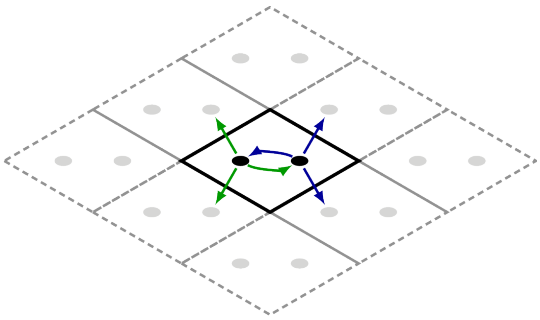
Note that even though the geometry is purely 2D, sisl **requires** the last non-used dimension. For 2D cases the non-used direction should *always* have a supercell of 1.

Example – graphene

A commonly encountered example is the graphene unit-cell. In a tight-binding picture one may suffice with a nearest-neighbour coupling.

Here we create the simple graphene 2D lattice with 2 atoms per unit-cell and a supercell of `[3, 3, 1]` to account for nearest neighbour couplings.

```
>>> graphene = geom.graphene()
```



which results in this underlying geometry:

The couplings from each unit-cell atom is highlighted by green (first atom) and blue (second atom) arrows. When dealing with Hamiltonians the supercell is extremely important to obtain the correct electronic structure. If one wishes to use the 3rd nearest neighbour couplings one is forced to use a supercell of `[5, 5, 1]` (please try and convince yourself of this).

Electronic structure setup – part 1

A `Hamiltonian` is an extension of a `Geometry`. From the `Geometry` it reads the number of orbitals, the supercell information.

Hamiltonians are matrices, and in sisl all Hamiltonians are treated as sparse matrices, i.e. matrices where there are an overweight of zeroes in the full matrix. As the Hamiltonian is treated as a matrix one can do regular assignments of the matrix elements, and basic math operations as well.

Here we create a square lattice and from this a `Hamiltonian`:

```
>>> geometry = Geometry([[0, 0, 0]])
>>> H = Hamiltonian(geometry)
>>> print(H)
{spin: 1, non-zero: 0
 {na: 1, no: 1, species:
  {Atoms(1):
    (1) == [H, Z: 1, orbs: 1, mass(au): 1.00794, maxR: -1.00000],
  },
 nsc: [1, 1, 1], maxR: -1.0
 }
}
```

which informs that the Hamiltonian currently only has 1 spin-component, is a matrix with complete zeroes (*non-zero* is 0). The geometry is a basic geometry with only one orbital per atom as $na = no$.

This geometry and Hamiltonian represents a lone atom with one orbital with zero on-site energy, a rather uninteresting case.

The examples here will be re-performed in *Electronic structure setup – part 2* by highlighting how the Hamiltonian can be setup in a more easy way.

Example – square

Let us try and continue from *Geometry creation – part 1* and create a square 2D lattice with one atom in the unit-cell and a supercell which couples only to nearest neighbour atoms.

```
>>> square = Geometry([[0.5,0.5,0]], sc=SuperCell([1, 1, 10], [3, 3, 1]))
>>> H = Hamiltonian(square)
```

Now we have a periodic structure with couplings allowed only to nearest neighbour atoms. Note, that it still only has 1 orbital. In the following we setup the on-site and the 4 nearest neighbour couplings to, -4 and 1, respectively:

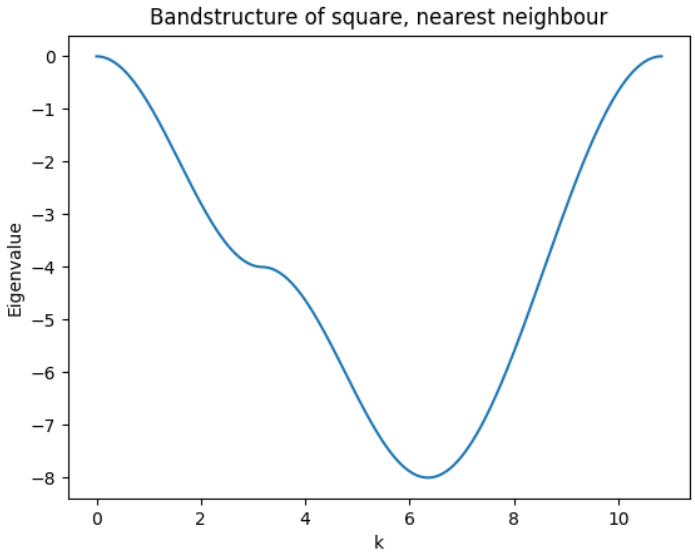
```
>>> H[0, 0] = -4
>>> H[0, 0, (1, 0)] = 1
>>> H[0, 0, (-1, 0)] = 1
>>> H[0, 0, (0, 1)] = 1
>>> H[0, 0, (0, -1)] = 1
>>> print(H)
{spin: 1, non-zero: 5
 {na: 1, no: 1, species:
  {Atoms(1):
    (1) == [H, Z: 1, orbs: 1, mass(au): 1.00794, maxR: -1.00000],
  },
  nsc: [3, 3, 1], maxR: -1.0
 }
}
```

There are a couple of things going on here (the items corresponds to lines in the above snippet):

1. Specifies the on-site energy of the orbital. Note that we assign as would do in a normal matrix.
2. Sets the coupling element from the first orbital in the primary unit-cell to the first orbital in the unit-cell neighbouring in the x direction, hence `(1, 0)`.
3. Sets the coupling element from the first orbital in the primary unit-cell to the first orbital in the unit-cell neighbouring in the $-x$ direction, hence `(-1, 0)`.
4. Sets the coupling element from the first orbital in the primary unit-cell to the first orbital in the unit-cell neighbouring in the y direction, hence `(0, 1)`.
5. Sets the coupling element from the first orbital in the primary unit-cell to the first orbital in the unit-cell neighbouring in the $-y$ direction, hence `(0, -1)`.

sisl does not intrinsically enforce symmetry, *that is the responsibility of the user*. This completes the Hamiltonian for nearest neighbour interaction and enables the calculation of the band-structure of the system.

In the below figure we plot the band-structure going from the Γ point to the band-edge along x, to the corner and back.



The complete code for this example (plus the band-structure) can be found `here`.

Example – graphene

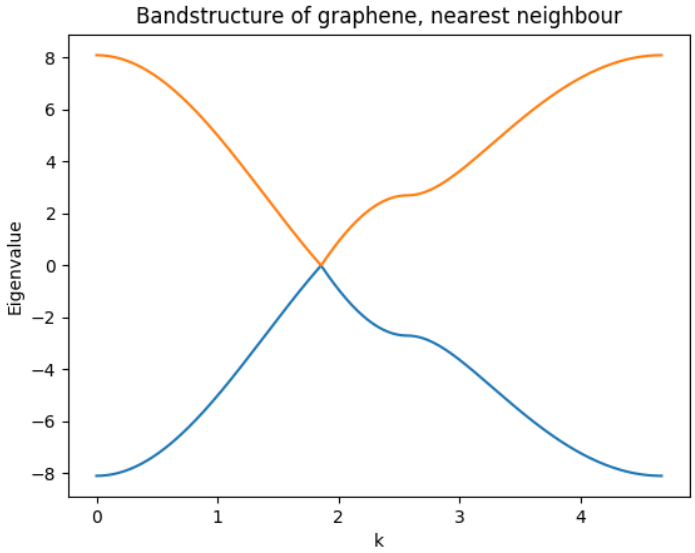
A commonly encountered example is the graphene unit-cell. In a tight-binding picture one may suffice with a nearest-neighbour coupling.

Here we create the simple graphene 2D lattice with 2 atoms per unit-cell and a supercell of `[3, 3, 1]` to account for nearest neighbour couplings.

```
>>> graphene = geom.graphene()
>>> H = Hamiltonian(graphene)
```

The nearest neighbour tight-binding model for graphene uses 0 onsite energy and 2.7 as the hopping parameter. These are specified as this:

```
>>> H[0, 1] = 2.7
>>> H[0, 1, (-1, 0)] = 2.7
>>> H[0, 1, (0, -1)] = 2.7
>>> H[1, 0] = 2.7
>>> H[1, 0, (1, 0)] = 2.7
>>> H[1, 0, (0, 1)] = 2.7
```



The complete code for this example (plus the band-structure) can be found `here`.

Electronic structure setup – part 2

Following *part 1* we focus on how to generalize the specification of the hopping parameters in a more generic way.

First, we re-create the square geometry (with one orbital per atom). However, to generalize the specification of the hopping parameters it is essential that we specify how long range the orbitals interact. In the following we set the atomic specie to be a Hydrogen atom with a single orbital with a range of 1

```
>>> Hydrogen = Atom(1, R=1.)
>>> square = Geometry([[0.5, 0.5, 0]], Hydrogen,
                      sc=SuperCell([1, 1, 10], [3, 3, 1]))
>>> H = Hamiltonian(square)
>>> print(H)
{spin: 1, non-zero: 0
 {na: 1, no: 1, species:
  {Atoms(1):
```

```
    (1) == [H, Z: 1, orbs: 1, mass(au): 1.00794, maxR: 1.00000],
  },
  nsc: [3, 3, 1], maxR: 1.0
 }
}
```

Note how the `maxR` variable has changed from `-1.0` to `1.0`. This corresponds to the maximal orbital range in the geometry. Here there is only one type of orbital, but for geometries with several different orbitals, there may be different orbital ranges.

Now one can assign the generalized parameters:

```
>>> for ia in square: # loop atomic indices (here equivalent to the orbital␣
↪indices)
...     idx_a = square.close(ia, R=[0.1, 1.1])
...     H[ia, idx_a[0]] = -4.
...     H[ia, idx_a[1]] = 1.
```

The `Geometry.close` function is a convenience function to return atomic indices of atoms within a certain radius. For instance `close(0, R=1.)` returns all atomic indices within a spherical radius of `1` from the first atom in the geometry, including it-self. `close([0., 0., 1.], R=1.)` will return all atomic indices within `1` of the coordinate `[0., 0., 1.]`. If one specifies a list of `R` it will return the atomic indices in the sphere within the first element; and for the later values it will return the atomic indices in the spherical shell between the corresponding radii and the previous radii.

The above code is the preferred method of creating a Hamiltonian. It is safe because it ensures that all parameters are set, and symmetrized.

For very large geometries (larger than 50,000 atoms) the above code will be *extremely* slow. Hence, the preferred method to setup the Hamiltonian for these large geometries is:

```
>>> for ias, idxs in square.iter_block():
...     for ia in ias:
...         idx_a = square.close(ia, R=[0.1, 1.1], idx=idxs)
...         H[ia, idx_a[0]] = -4.
...         H[ia, idx_a[1]] = 1.
```

The code above is the preferred method of specifying the Hamiltonian parameters.

The complete code for this example (plus the band-structure) can be found `here`.

CHAPTER 6

Examples

sisl is shipped with these examples which describes a large variation of use cases.

All examples are assumed to have this in the header:

```
import numpy as np
from sisl import *
```

to enable numpy and sisl.

Graphene tight-binding model

This example creates a minimal graphene unit-cell of two atoms. The Carbon atoms are described with a single orbital per atom and with a cutoff radius of 1.42 Å.

The `Hamiltonian` `H` is an object which may be treated as a sparse matrix. The `for` loop below loops over all atoms (`ia`) in the graphene unit-cell. The `close` function returns a list of length `len(R)` with elements where all neighbouring atoms within the radius defined in `R` are listed. Comments in the below example clarifies each of the steps carefully.

```
# This example creates the tight-binding Hamiltonian
# for graphene with on-site energy 0, and hopping energy
# -2.7 eV.

import sisl

bond = 1.42
# Construct the atom with the appropriate orbital range
# Note the 0.01 which is for numerical accuracy.
C = sisl.Atom(6, R = bond + 0.01)
# Create graphene unit-cell
gr = sisl.geom.graphene(bond, C)

# Create the tight-binding Hamiltonian
H = sisl.Hamiltonian(gr)
R = [0.1 * bond, bond + 0.01]

for ia in gr:
    idx_a = gr.close(ia, R)
```

```
    # On-site
    H[ia, idx_a[0]] = 0.
    # Nearest neighbour hopping
    H[ia, idx_a[1]] = -2.7

# Calculate eigenvalues at K-point
print(H.eigh([2./3, 1./3, 0.]))
```

CHAPTER 7

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.

- **'ScaleUp'_** File formats inherent to ScaleUp

  REF `REFSileScaleUp` is the geometry file for ScaleUp.

  restart `restartSileScaleUp` is the displacement geometry file for ScaleUp.

CHAPTER 8

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 implementing extremely fast and scalable tight-binding creation algorithms (`>1,000,000` orbitals). sisl is developed in particular 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 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.

Installation

Follow *these steps* to install sisl.

CHAPTER 9

API links

A selected list of links to the API documentation of the most commonly used objects:

`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
`sisl.physics`	Module containing a variety of different physical quantities.

Indices

- genindex
- modindex
- search