Nanonet Documentation

Release 1.0

M. V. Klymenko, J. A. Vaitkus, J. S. Smith, J. H. Cole

Dec 19, 2019

CONTENTS:

CHAPTER

ONE

NANONET PROJECT

1.1 What is Nanonet?

The project represents an extendable Python framework for the electronic structure computations based on the tight-binding method. The code can deal with both finite and periodic systems translated in one, two or three dimensions.

All computations can be governed by means of the python application programming interface (pyAPI) or the command line interface (CLI).

1.2 Installation

1.2.1 Prerequisites

The source distribution can be obtained from GitHub:

```
git clone https://github.com/freude/NanoNet.git
cd NanoNet
```

All dependencies can be installed at once by invoking the following command from within the source directory:

```
pip install -r requirements.txt
```

1.2.2 Installing

In order to install the package `tb` just invoke the following line in the bash from within the source directory:

```
pip install .
```

1.2.3 Running the tests

All tests may be run by invoking the command:

```
nosetests --with-doctest
```

1.3 Usage and interfaces

1.3.1 Python application programming interface

Below is a short example demonstrating usage of the `tb` package. More illustrative examples can be found in the ipython notebooks in the directory `jupyter_notebooks` inside the source directory.

If the package is properly installed, the work starts with the import of all necessary modules:

```
import tb
```

Below we demonstrate band structure computation for bulk silicon using empirical tight-binding method.

1. First, one needs to specify atomic species and corresponding basis sets. It is possible to use custom basis set as is shown in examples in the ipython notebooks. Here we use predefined basis sets.

   ```
   tb.Orbitals.orbital_sets = {'Si': 'SiliconSP3D5S'}
   ```

2. Specify geometry of the system - determine position of atoms and specify periodic boundary conditions if any. This is done by creating an object of the class Hamiltonian with proper arguments.

   ```
   xyz_file = """2
   Si cell
   Si1       0.0000000000    0.0000000000    0.0000000000
   Si2       1.3750000000    1.3750000000    1.3750000000
   """

   h = tb.Hamiltonian(xyz=xyz_file, nn_distance=2.0)
   ```

3. Initialize the Hamiltonian - compute Hamiltonian matrix elements

   For isolated system:

   ```
   h.initialize()
   ```

4. Specify periodic boundary conditions:

   ```
   a_si = 5.50
   PRIMITIVE_CELL = [[0, 0.5 * a_si, 0.5 * a_si],
                     [0.5 * a_si, 0, 0.5 * a_si],
                     [0.5 * a_si, 0.5 * a_si, 0]]
   h.set_periodic_bc(PRIMITIVE_CELL)
   ```

5. Specify wave vectors:

   ```
   sym_points = ['L', 'GAMMA', 'X', 'W', 'K', 'L', 'W', 'X', 'K', 'GAMMA']
   num_points = [15, 20, 15, 10, 15, 15, 15, 15, 20]
   k = tb.get_k_coords(sym_points, num_points)
   ```

6. Find the eigenvalues and eigenstates of the Hamiltonian for each wave vector.

   ```
   vals = np.zeros((sum(num_points), h.h_matrix.shape[0]), dtype=np.complex)

   for jj, i in enumerate(k):
       vals[jj, :], _ = h.diagonalize_periodic_bc(list(i))

   import matplotlib.pyplot as plt
   ```

(continues on next page)

(continued from previous page)

```
plt.plot(np.sort(np.real(vals)))
plt.show()
```

7. Done.

1.3.2 Command line interface

The package is equipped with the command line tool `tb` the usage of which reads:

```
tb [-h] [--k_points_file K_POINTS_FILE] [--xyz XYZ]
   [--show SHOW] [--save SAVE]
   [--code_name CODE_NAME] param_file

    positional arguments:
      param_file            Path to the file in the yaml-format containing all
                            parameters needed to run computations.

    optional arguments:
      -h, --help            show this help message and exit
      --k_points_file K_POINTS_FILE
                            Path to the txt file containing coordinates of wave
                            vectors for the band structure computations. If not
                            specified, default values will be used.
      --xyz XYZ             Path to the file containing atomic coordinates. If
                            specified, it overrides the coordinates specified in
                            the param_files.
      --show SHOW, -S SHOW  Show figures, 0/1/2. 0 shows nothing, 1 outputs
                            figures on screen, 2 saves figures on disk without
                            showing.
      --save SAVE, -s SAVE  Save results of computations on disk, 0/1.
      --code_name CODE_NAME
                            Code name is added to the names of all saved data
                            files.
```

The results of computations will be stored in `band_structure.pkl` file in the current directory. This file name can be modified by specifying the parameter `--code_name`.

On the computers with `mpi` functions installed, instead of `tb` one has to use its mpi-version `tbmpi`. The script `tbmpi` parallelises the loop running over the wave vectors. This script can be used together with the command `mpirun` (below is an example generating 8 parallel processes):

```
mpirun -n 8 tbmpi --show=2 --save=1 --xyz=si.xyz --k_points=k_points.txt input.yaml
```

1.4 License

This project is licensed under the MIT License.

MIT License

Copyright (c) 2019 RMIT

Permission is hereby granted, free of charge, to any person obtaining a copy of this software and associated documentation files (the “Software”), to deal in the Software without restriction, including without limitation the rights to use,

1.5 Acknowledgments

We acknowledge support of the RMIT University, Australian Research Council through grant CE170100026, and National Computational Infrastructure, which is supported by the Australian Government.

CHAPTER

TWO

PACKAGES AND MODULES

2.1 Softrware architecture

A generic control flow for applications developed with *Nanonet* is shown schematically in Fig. 1. The input parameters are the list of atomic coordinates and a table of two-center integrals. The framework contains two packages **tb** and **negf**. The package **tb** is the core responsible for composing and diagonalizing Hamiltonian matrices. The **negf** package processes TB matrices; it contains subroutines for computing Green's functions, namely implementing the Recursive Green's Function (RGF) algorithm.

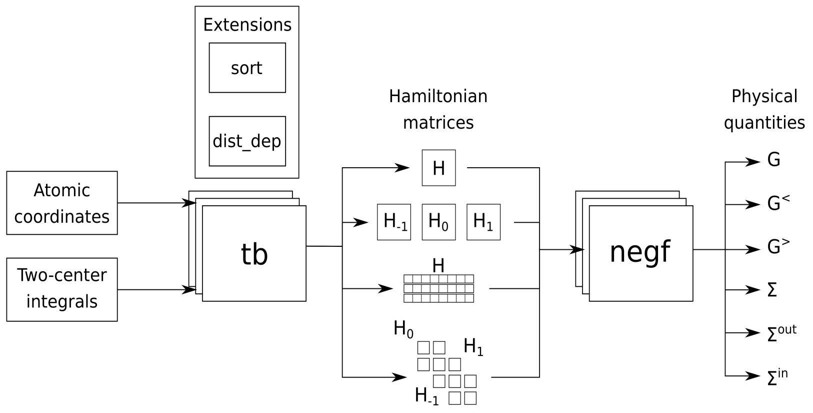


Fig. 1: A generic control flow for applications developed with Nanonet.

tb represents a library of Python classes facilitating building Hamiltonian matrices, imposing periodic boundary conditions and computing electronic structure for molecules and solids using the TB method. The Hamiltonian matrices are built from an XYZ file containing atomic coordinates and a list of TB parameters.

The software architecture relies on the object-oriented paradigms — the framework represents a library of classes whose UML diagram is shown in Appendix. The central class of the framework is called **Hamiltonian** and contains all necessary information and functionality to construct Hamiltonian matrices that represents its main goal. This class inherits properties and member functions from classes **BasisTB** and **StructureDesignerXYZ** — abstractions for basis sets and geometrical configuration of atoms correspondingly. The first one stores a look-up table that allows one to associate a set of orbitals with a label denoting a chemical element. The second one stores $kd$-tree built from the list of atomic coordinates.

The class **CyclicTopology** is used when periodic boundary conditions are applied. It translates atomic coordinates according to translation vectors and creates a kd-tree for atoms outside the primitive cell, needed to compute the Hamiltonian matrix responsible for coupling between neighbouring primitive cells.

The orbital sets are created using facilities of the class **Orbitals**. This class is the parent class for all basis sets. The current version of the software package contains predefined basis sets: sp3d5s* model for silicon, *SiliconSP3D5S*, single s-orbital for hydrogen, and sp3 model for bismuth.

The version of the class **Hamiltonian** that uses sparse matrix representations is implemented in sub-class **HamiltonianSp** having same interface with some redefined member-functions.

The package **negf** is written in the procedural programming paradigm and contains functions that computes complex band structure, self-energies of leads and non-equilibrium Green's functions.

2.2 Package tb

2.2.1 Module hamiltonian

The module contains a library of classes facilitating computations of Hamiltonian matrices.

class tb.hamiltonian.**BasisTB**(***kwargs*)

The class contains information about sets of quantum numbers and dimensionality of the Hilbert space. It is also equipped with the member functions translating quantum numbers into a matrix index and vise versa using a set of index offsets.

Examples

```
>>> from verbosity import set_verbosity
>>> import tb
>>> set_verbosity(0)
>>> orb = tb.Orbitals('A')
>>> orb.add_orbital(title='s', energy=-1)
>>> orb.add_orbital(title='1s', energy=0)
>>> tb.Orbitals('B').add_orbital(title='s', energy=0)
>>> xyz = '''2
... Two atoms
... A1 0 0 0
... B1 0 0 1'''
>>> basis = tb.hamiltonian.BasisTB(xyz=xyz)
>>> print(basis.basis_size)
3
>>> print(basis.atom_list['B1'])
[0. 0. 1.]
>>> print(basis.qn2ind({'atoms': 0, 'l': 0}))
0
>>> print(basis.qn2ind({'atoms': 0, 'l': 1}))
1
>>> print(basis.qn2ind({'atoms': 1, 'l': 0}))
2
>>> print(type(basis.orbitals_dict['A']))
<class 'tb.orbitals.Orbitals'>
```

orbitals_dict

Returns the dictionary data structure of orbitals. In the dictionary

qn2ind(*qn*)

Computes a matrix index of an element from the index of atom and the index of atomic orbital.

Parameters **qn** (*dict*) – A dictionary with two keys *atoms* and *l*, where the fist one is the atom index and the later is the orbital index.

Returns **ans** – index of an element in the TB matrix

Return type int

class tb.hamiltonian.**Hamiltonian**(***kwargs*)

Class defines a Hamiltonian matrix as well as a set of member-functions allowing to build, diagonalize and visualize the matrix.

Examples

```
>>> from verbosity import set_verbosity
>>> import tb
>>> set_verbosity(0)
>>> tb.Orbitals('A').add_orbital(title='s', energy=-1)
>>> tb.Orbitals('B').add_orbital(title='s', energy=-2)
>>> xyz_file = '''2
... Two atoms
... A1 0 0 0
... B1 0 0 1.5'''
>>> tb.set_tb_params(PARAMS_A_B={'ss_sigma': 0.1})
>>> h = tb.Hamiltonian(xyz=xyz_file, nn_distance=2.0).initialize()
>>> h.h_matrix
array([[-1. +0.j,  0.1+0.j],
       [ 0.1+0.j, -2. +0.j]])
```

_compute_h_matrix_bc_add(*split_the_leads=False*)

Compute additive Bloch exponentials needed to specify pbc

_compute_h_matrix_bc_factor()

Compute the exponential Bloch factors needed when the periodic boundary conditions are applied.

_get_me(*atom1*, *atom2*, *l1*, *l2*, *coords=None*)

Compute the matrix element <atom1, l1|H|l2, atom2>. The function is called in the member function initialize() and invokes the function me() from the module diatomic_matrix_element.

Parameters

- **atom1** – atom index
- **atom2** – atom index
- **l1** – index of a localized basis function
- **l2** – index of a localized basis function
- **coords** – coordinates of radius vector pointing from one atom to another it may differ from the actual coordinates of atoms

Returns matrix element

Return type float

_reset_periodic_bc()

Reset the matrices determining periodic boundary conditions to their default state :return:

diagonalize()

Diagonalize the Hamiltonian matrix for the finite isolated system :return:

diagonalize_periodic_bc(*k_vector*)

Diagonalize the Hamiltonian matrix with the periodic boundary conditions for a certain value of the wave vector k_vector

Parameters **k_vector** – wave vector

Returns

get_hamiltonians()

Return a list of Hamiltonian matrices. For 1D systems, the list is [Hl, Hc, Hr], where Hc is the Hamiltonian describing interactions between atoms within a unit cell, Hl and Hr are Hamiltonians describing couplings between atoms in the unit cell and atoms in the left and right adjacent unit cells.

Returns list of Hamiltonians

Return type list

get_site_coordinates()

Return coordinates of atoms.

Returns atomic coordinates

Return type numpy.ndarray

initialize(*int_radial_dep=None*, *radial_dep=None*)

Compute matrix elements of the Hamiltonian.

set_periodic_bc(*primitive_cell*)

Set periodic boundary conditions. The function creates an object of the class CyclicTopology.

Parameters **primitive_cell** – list of vectors defining a primitive cell

2.2.2 Module hamiltonian_sparse

The module contains all necessary classes needed to compute the Hamiltonian matrix

class tb.hamiltonian_sparse.**HamiltonianSp**(***kwargs*)

Class defines a Hamiltonian matrix as well as a set of member-functions allowing to build, diagonalize and visualize the matrix.

_reset_periodic_bc()

Resets the matrices determining periodic boundary conditions to their default state :return:

diagonalize()

Diagonalize the Hamiltonian matrix for the finite isolated system :return:

diagonalize_periodic_bc(*k_vector*)

Diagonalize the Hamiltonian matrix with the periodic boundary conditions for a certain value of the wave vector k_vector

Parameters **k_vector** – wave vector

Returns

get_hamiltonians()

Return a list of Hamiltonian matrices. For 1D systems, the list is [Hl, Hc, Hr], where Hc is the Hamiltonian describing interactions between atoms within a unit cell, Hl and Hr are Hamiltonians describing couplings between atoms in the unit cell and atoms in the left and right adjacent unit cells.

Returns list of Hamiltonians

Return type list

initialize()

The function computes matrix elements of the Hamiltonian.

2.2.3 Module hamiltonian_initializer

The module contains functions facilitating setting tight-binding parameters and initializing Hamiltonian objects from a Python dictionary.

`tb.hamiltonian_initializer.`**`initializer`**(***kwargs*)

Creates a Hamiltonian object from a set of parameters stored in a Python dictionary.

This functions is used by CLI scripts to create Hamiltonian objects from a configuration file (normally in a yaml format) which is previously parsed into a Python dictionary data structure.

Parameters **`kwargs`** (*`dict`*) – Dictionary of parameters needed to make a Hamiltonian object.

Returns **h** – instance of the class Hamiltonian

Return type tb.Hamiltonian

`tb.hamiltonian_initializer.`**`set_tb_params`**(***kwargs*)

Initialize a set of the user-defined tight-binding parameters.

Parameters **`kwargs`** (*`dict of dict`*) – Dictionary of the tight-binding parameters. The dictionary follows a certain name convention - each new entry should conform with following format: for the entry names, PARAMS_<el1>_<el2>_<order>, where <el1> and <el2> are chemical elements of a pair of atoms, and <order> is a number specifying the order of nearest neighbours; for the dictionary values, <orb1><orb2>_<mol>, where <orb1> and <orb2> are the orbital quantum numbers (s, p, d etc.) and <mol> is the symmetry of a molecular orbital (sigma, pi etc).

2.2.4 Module structure_designer

The module contains classes defining geometrical structure and boundary conditions for the tight-binding model.

`class` `tb.structure_designer.`**`CyclicTopology`**(*primitive_cell_vectors*, *labels*, *coords*, *nn_distance*)

The class provides functionality for determining the periodic boundary conditions for a crystal cell. The object of the class is instantiated by a set of the primitive cell vectors.

`class` `tb.structure_designer.`**`StructDesignerXYZ`**(***kwargs*)

The class builds an atomic structure from either the filename of a xyz-file or xyz data itself represented as a Python string. The class arrange atomic coordinates in kd-tree and sorts them if needed according to a specified sorting procedure.

`nn_distance`

nearest neighbour search radius (default 0)

Type float

`num_of_species`

number of chemical elements corresponding to the number of distinct basis sets.

Type int

`atom_list`

list of atomic species and their coordinates

Type OrderedDict

`kd_tree`

kd-tree for fast nearest-neighbour search

Type scipy.spatial.ckdtree.cKDTree

left_lead
list of atomic indices connected to the left lead, needed for sorting atomic coordinates (default [])

Type list

right_lead
list of atomic indices connected to the right lead, needed for sorting atomic coordinates (default [])

Type list

sort_func
function for sorting atomic coordinates (default None)

Type func

2.2.5 Module orbitals

Module contains the class *Orbitals* that allows to generate any user defined basis set based on the linear combination of atomic orbitals (LCAO). Also, the module contains a set of predefined basis sets *SiliconSP3D5S*, *HydrogenS*, *Bismuth*.

class `tb.orbitals.`**Bismuth**
Class defines the *sp3* basis set for the bismuth atom

class `tb.orbitals.`**HydrogenS**
Class defines the simplest basis set for the hydrogen atom, consisting of a single s-orbital

class `tb.orbitals.`**Orbitals**(*title*)
This is the parent class for all basis sets for all atoms. It also contains a factory function, which generates objects of the class Orbitals from a list of labels and the dictionary *orbital_sets* making a correspondence between an atom and its basis set

add_orbital(*title*, *energy=0.0*, *principal=0*, *orbital=0*, *magnetic=0*, *spin=0*)
Adds an orbital to the set of orbitals

Parameters

- **title** – a string representing an orbital label, it usually specifies its symmetry, e.g. *s*, *px*, *py* etc.
- **energy** – energy of the orbital
- **principal** – principal quantum number *n-1*
- **orbital** – orbital quantum number *l*
- **magnetic** – magnetic quantum number *m*
- **spin** – spin quantum number *s*

static atoms_factory(*labels*)
Taking a list of labels creates a dictionary of *Orbitals* objects from those labels. The set of orbitals for each atom and corresponding class is specified in the class variable *orbital_sets*

Parameters **labels** (*list (str)*) – list of labels

Returns dictionary of *Orbitals* objects

class `tb.orbitals.`**SiliconSP3D5S**
Class defines the *sp3d5s** basis set for the silicon atom

2.2.6 Module diatomic_matrix_element

The module contains functions computing hopping parameters with arbitrary rotations of atomic orbitals based on the table of empirical diatomic couplings defined in the module params. Computations are based mostly on analytical equations derived in [A.V. Podolskiy and P. Vogl, Phys. Rev. B. 69, 233101 (2004)].

`tb.diatomic_matrix_element.`**`d_me`**(*N*, *l*, *m1*, *m2*)

Computes rotational matrix elements according to A.V. Podolskiy and P. Vogl, Phys. Rev. B. 69, 233101 (2004)

Parameters

- **`N`** – directional cosine relative to z-axis
- **`l`** – orbital quantum number
- **`m1`** – magnetic quantum number
- **`m2`** – magnetic quantum number

Returns rotational matrix element

`tb.diatomic_matrix_element.`**`me`**(*atom1*, *ll1*, *atom2*, *ll2*, *coords*, *which_neighbour=0*)

Computes the non-diagonal matrix element of the tight-binding Hamiltonian - coupling between two sites, both are described by LCAO basis sets. This function is evoked in the member function _get_me() of the Hamiltonian object.

Parameters

- **`atom1`** (*`tb.Orbitals`*) – basis set associated with the first site
- **`ll1`** (*`int`*) – index specifying a particular orbital in the basis set for the first site
- **`atom2`** (*`tb.Orbitals`*) – basis set associated with the first site
- **`ll2`** (*`int`*) – index specifying a particular orbital in the basis set for the second site
- **`coords`** (*`array`*) – coordinates of radius vector pointing from one site to another
- **`which_neighbour`** (*`int`*) – Order of a nearest neighbour (first-, second-, third- etc)

Returns **ans** – Inter-atomic matrix element

Return type float

`tb.diatomic_matrix_element.`**`me_diatomic`**(*bond*, *n*, *l_min*, *l_max*, *m*, *which_neighbour*)

The function looks up into the table of parameters making a query parametrized by:

Parameters

- **`bond`** – a bond type represented by a list of atom labels
- **`n`** – combination of the principal quantum numbers of atoms
- **`l_min`** – min(l1, l2), where l1 and l2 are orbital quantum numbers of atoms
- **`l_max`** – max(l1, l2), where l1 and l2 are orbital quantum numbers of atoms
- **`m`** – symmetry of the electron wave function in the diatomic molecule takes values “sigma”, “pi” and “delta”

Returns numerical value of the corresponding tabular parameter

Return type float

2.2.7 Module block_tridiagonalization

This module contains a set of functions facilitating computations of the block-tridiagonal structure of a band matrix.

`tb.block_tridiagonalization.`**`accum`**(*accmap*, *input*, *func=None*, *size=None*, *fill_value=0*, *dtype=None*)

An accumulation function similar to Matlab's *accumarray* function.

Parameters

- **`accmap`** (*`ndarray`*) – This is the "accumulation map". It maps input (i.e. indices into *a*) to their destination in the output array. The first *a.ndim* dimensions of *accmap* must be the same as *a.shape*. That is, *accmap.shape[:a.ndim]* must equal *a.shape*. For example, if *a* has shape (15,4), then *accmap.shape[:2]* must equal (15,4). In this case *accmap[i,j]* gives the index into the output array where element (i,j) of *a* is to be accumulated. If the output is, say, a 2D, then *accmap* must have shape (15,4,2). The value in the last dimension give indices into the output array. If the output is 1D, then the shape of *accmap* can be either (15,4) or (15,4,1)
- **`input`** (*`ndarray`*) – The input data to be accumulated.
- **`func`** (*`callable or None`*) – The accumulation function. The function will be passed a list of values from *a* to be accumulated. If None, numpy.sum is assumed.
- **`size`** (*`ndarray or None`*) – The size of the output array. If None, the size will be determined from *accmap*.
- **`fill_value`** (*`scalar`*) – The default value for elements of the output array.
- **`dtype`** (*`numpy data type, or None`*) – The data type of the output array. If None, the data type of *a* is used.

Returns

out – The accumulated results.

The shape of *out* is *size* if *size* is given. Otherwise the shape is determined by the (lexicographically) largest indices of the output found in *accmap*.

Return type ndarray

`tb.block_tridiagonalization.`**`compute_blocks`**(*left_block*, *right_block*, *edge*, *edge1*)

This is an implementation of the greedy algorithm for computing block-tridiagonal representation of a matrix. The information regarding the input matrix is represented by the sparsity patters edges, *edge* and *edge1*.

Parameters

- **`left_block`** (*`int`*) – a predefined size of the leftmost block
- **`right_block`** (*`int`*) – a predefined size of the rightmost block
- **`edge`** (*`ndarray`*) – edge of sparsity pattern
- **`edge1`** (*`ndarray`*) – conjugate edge of sparsity pattern

Returns **ans** – list of diagonal block sizes

Return type list

Examples

```
>>> import numpy as np
>>> from tb.block_tridiagonalization import compute_edge
>>> input_matrix = np.array([[1, 1, 0, 0], [1, 1, 1, 0], [0, 1, 1, 1], [0, 0, 1,␣
↪1]])
>>> input_matrix
array([[1, 1, 0, 0],
       [1, 1, 1, 0],
       [0, 1, 1, 1],
       [0, 0, 1, 1]])
>>> e1, e2 = compute_edge(input_matrix)
>>> compute_blocks(1, 1, e1, e2)
[1, 1, 1, 1]
>>> input_matrix = np.array([[1, 1, 1, 0], [1, 1, 1, 0], [1, 1, 1, 1], [0, 0, 1,␣
↪1]])
>>> input_matrix
array([[1, 1, 1, 0],
       [1, 1, 1, 0],
       [1, 1, 1, 1],
       [0, 0, 1, 1]])
>>> e1, e2 = compute_edge(input_matrix)
>>> compute_blocks(1, 1, e1, e2)
[1, 2, 1]
>>> e1, e2 = compute_edge(input_matrix)
>>> compute_blocks(2, 2, e1, e2)
[2, 2]
```

`tb.block_tridiagonalization.`**`compute_blocks_optimized`**(*edge*, *edge1*, *left=1*, *right=1*)

Computes optimal sizes of diagonal blocks of a matrix whose sparsity pattern is defined by the sparsity pattern profiles edge and edge1. This function is based on the algorithm which uses defined above function find_optimal_cut() to subdivide the problem into sub-problems in a optimal way according to some cost function.

Parameters

- **`edge`** (*`ndarray`*) – sparsity pattern profile of the matrix
- **`edge1`** (*`ndarray`*) – conjugated sparsity pattern profile of the matrix
- **`left`** (*`int`*) – size of the leftmost diagonal block (constrained)
- **`right`** (*`int`*) – size of the rightmost diagonal block (constrained)

Returns **blocks** – list of optimal sizes of diagonal blocks

Return type list

`tb.block_tridiagonalization.`**`compute_edge`**(*mat*)

Computes edges of the sparsity pattern of a matrix.

Parameters **`mat`** (*`ndarray`*) – Input matrix

Returns

- **edge** (*ndarray*) – edge of the sparsity pattern
- **edge1** (*ndarray*) – conjugate edge of the sparsity pattern

Examples

```
>>> import numpy as np
>>> from tb.block_tridiagonalization import compute_edge
>>> input_matrix = np.array([[1, 1, 0, 0], [1, 1, 1, 0], [0, 1, 1, 1], [0, 0, 1,␣
↪1]])
>>> input_matrix
array([[1, 1, 0, 0],
       [1, 1, 1, 0],
       [0, 1, 1, 1],
       [0, 0, 1, 1]])
>>> e1, e2 = compute_edge(input_matrix)
>>> e1
array([2, 3, 4, 4])
>>> e2
array([2, 3, 4, 4])
>>> input_matrix = np.array([[1, 0, 0, 0], [0, 1, 1, 0], [0, 1, 1, 1], [0, 0, 1,␣
↪1]])
>>> input_matrix
array([[1, 0, 0, 0],
       [0, 1, 1, 0],
       [0, 1, 1, 1],
       [0, 0, 1, 1]])
>>> e1, e2 = compute_edge(input_matrix)
>>> e1
array([1, 3, 4, 4])
>>> e2
array([2, 3, 3, 4])
```

`tb.block_tridiagonalization.`**`cut_in_blocks`**(*h_0*, *blocks*)

Cut a matrix into diagonal, upper-diagonal and lower-diagonal blocks if sizes of the diagonal blocks are specified.

Parameters

- **h_0** (*ndarray*) – Input matrix
- **blocks** (*ndarray (dtype=int)*) – Sizes of diagonal blocks

Returns

h_0_s, h_l_s, h_r_s – List of diagonal matrices, list of lower-diagonal matrices and list of upper-diagonal matrices.

Note that if the size of the list h_0_s is N, the sizes of h_l_s, h_r_s are N-1.

Return type ndarray

Examples

```
>>> import numpy as np
>>> from tb.block_tridiagonalization import cut_in_blocks
>>> a = np.array([[1, 1, 0, 0], [1, 1, 1, 0], [0, 1, 1, 1], [0, 0, 1, 1]])
>>> a
array([[1, 1, 0, 0],
       [1, 1, 1, 0],
       [0, 1, 1, 1],
       [0, 0, 1, 1]])
```

(continues on next page)

(continued from previous page)

```
>>> # Sum the diagonals.
>>> blocks = [2, 2]
>>> blocks
[2, 2]
>>> h0, h1, h2 = cut_in_blocks(a, blocks)
>>> h0
[array([[1, 1],
       [1, 1]]), array([[1, 1],
       [1, 1]])]
>>> h1
[array([[0, 1],
       [0, 0]])]
>>> h2
[array([[0, 0],
       [1, 0]])]
```

`tb.block_tridiagonalization.`**`find_optimal_cut`**(*edge*, *edge1*, *left*, *right*)

Computes the index corresponding to the optimal cut such that applying the function compute_blocks() to the sub-blocks defined by the cut reduces the cost function comparing to the case when the function compute_blocks() is applied to the whole matrix. If cutting point can not be find, the algorithm returns the result from the function compute_blocks().

Parameters

- **`edge`** (*`ndarray`*) – sparsity pattern profile of the matrix
- **`edge1`** (*`ndarray`*) – conjugated sparsity pattern profile of the matrix
- **`left`** (*`int`*) – size of the leftmost diagonal block
- **`right`** (*`int`*) – size of the rightmost diagonal block

Returns

- **blocks** (*list*) – list of diagonal block sizes
- **sep** (*int*) – the index of the optimal cut
- **right_block** (*int*) – size of the rightmost sub-block of the left block (relative to the cutting point)
- **left_block** (*int*) – size of the leftmost sub-block of the right block (relative to the cutting point)

`tb.block_tridiagonalization.`**`show_blocks`**(*subblocks*, *input_mat*)

This is a script for visualizing the sparsity pattern and a block-tridiagonal structure of a matrix.

Parameters

- **`subblocks`** (*`list`*) – list of sizes of the diagonal blocks
- **`input_mat`** (*`ndarray`*) – Hamiltonian matrix

`tb.block_tridiagonalization.`**`split_into_subblocks`**(*h_0*, *h_l*, *h_r*)

Split Hamiltonian matrix and coupling matrices into subblocks

Parameters

- **`h_0`** – Hamiltonian matrix
- **`h_l`** – left inter-cell coupling matrices

- **h_r** – right inter-cell coupling matrices

Return h_0_s, h_l_s, h_r_s lists of subblocks

tb.block_tridiagonalization.**split_into_subblocks_optimized**(*h_0*, *left=1*, *right=1*)

Parameters

- **h_0** –
- **left** –
- **right** –

Returns

2.2.8 Module sorting_algorithms

This module contains three sorting function: lexicographic sort of atomic coordinates, sort that uses projections on a vector pointing from one electrode to another as the sorting keys and sort that uses a potential function over atomic coordinates as the sorting keys. A user can define his own sorting procedure - the user-defined sorting function should contain ***kwargs* in the list of arguments and it can uses in its body one of the arguments with following name convention: *coords* is the list of atomic coordinates, *left_lead* is the list of the indices of the atoms contacting the left lead, *right_lead* is the list of the indices of the atoms contacting the right lead, and *mat* is the adjacency matrix of the tight-binding model. All functions return the list of sorted atomic indices.

tb.sorting_algorithms.**sort_capacitance**(*coords*, *mat*, *left_lead*, *right_lead*, ***kwargs*)

Sorting procedure that uses a potential function defined over atomic coordinates as the sorting keys.

Parameters

- **coords** (*array*) – list of atomic coordinates
- **mat** (*2D array*) – adjacency matrix of the tight-binding model
- **left_lead** (*array*) – list of the atom indices contacting the left lead
- **right_lead** (*array*) – list of the atom indices contacting the right lead

Returns ans – list of reordered indices

Return type array

tb.sorting_algorithms.**sort_lexico**(*coords=None*, ***kwargs*)

Lexicographic sort

Parameters coords (*array*) – list of atomic coordinates

Returns ans – list of reordered indices

Return type array

tb.sorting_algorithms.**sort_projection**(*coords=None*, *left_lead=None*, *right_lead=None*, ***kwargs*)

Sorting procedure that uses projections on a vector pointing from one electrode to another as the sorting keys.

Parameters

- **coords** (*array*) – list of atomic coordinates
- **left_lead** (*array*) – list of the atom indices contacting the left lead
- **right_lead** (*array*) – list of the atom indices contacting the right lead

Returns ans – list of reordered indices

Return type array

2.3 Package negf

2.3.1 Module greens_functions

The module contains functions that computes Green's functions and their poles

`negf.greens_functions.`**`group_velocity`**(*eigenvector*, *eigenvalue*, *h_r*)

Computes the group velocity of wave packets

Parameters

- **`eigenvector`** (*`numpy.complex`*) – eigenvector
- **`eigenvalue`** – eigenvalue
- **`h_r`** (*`numpy.matrix`*) – coupling Hamiltonian

Returns group velocity for a pair consisting of an eigenvector and an eigenvalue

`negf.greens_functions.`**`surface_greens_function`**(*E*, *h_l*, *h_0*, *h_r*, *iterate=True*, *damp=0.0001j*)

Computes surface self-energies using the eigenvalue decomposition. The procedure is described in [M. Wimmer, Quantum transport in nanostructures: From computational concepts to spintronics in graphene and magnetic tunnel junctions, 2009, ISBN-9783868450255].

Parameters

- **`E`** – energy array
- **`h_l`** – left-side coupling Hamiltonian
- **`h_0`** – channel Hamiltonian
- **`h_r`** – right-side coupling Hamiltonian
- **`iterate`** – iterate to stabilize TB matrix
- **`damp`** – damping

Returns left- and right-side self-energies

`negf.greens_functions.`**`surface_greens_function_poles`**(*h_list*)

Computes eigenvalues and eigenvectors for the complex band structure problem. The eigenvalues correspond to the wave vectors as *exp(ik)*.

Parameters **`h_list`** – list of the Hamiltonian blocks - blocks describes coupling with left-side neighbours, Hamiltonian of the side and coupling with right-side neighbours

Returns eigenvalues, k, and eigenvectors, U,

Return type numpy.matrix, numpy.matrix

2.3.2 Module recursive_greens_functions

`negf.recursive_greens_functions.`**`recursive_gf`**(*energy*, *mat_l_list*, *mat_d_list*, *mat_u_list*, *s_in=0*, *s_out=0*, *damp=1e-06j*)

The recursive Green's function algorithm is taken from M. P. Anantram, M. S. Lundstrom and D. E. Nikonov, Proceedings of the IEEE, 96, 1511 - 1550 (2008) DOI: 10.1109/JPROC.2008.927355

In order to get the electron correlation function output, the parameters s_in has to be set. For the hole correlation function, the parameter s_out has to be set.

Parameters

- **energy** (*numpy array*) – energy
- **mat_d_list** (*list of numpy arrays*) – list of diagonal blocks
- **mat_u_list** (*list of numpy arrays*) – list of upper-diagonal blocks
- **mat_l_list** (*list of numpy arrays*) – list of lower-diagonal blocks

Return grd, grl, gru, gr_left retarded Green's function: block-diagonal,

lower block-diagonal, upper block-diagonal, left-connected

Rtype grd, grl, gru, gr_left list of numpy arrays

CHAPTER

THREE

LEARNING BY EXAMPLES

3.1 Build Hückel model with custom parameters and user-defined basis set

3.1.1 Prerequisites

To set up jupyter-notebook, following packages has to be imported:

```
[1]: %matplotlib inline
import matplotlib.pyplot as plt
import numpy as np
from IPython.display import display, Math, Latex
```

Here we compute energy spectra of a group of atoms using tight-binding method - no periodic boundary conditions applied. As an example, we compute the compute energy spectrum for the benzene molecule in the framework of the Hückel model. In order to start working with the package, one has to import it first:

```
[2]: import tb
```

3.1.2 Define a basis set

First, one needs to specify a basis sets for the each atom species used in computations. Although, there is an embeded library of basis functions for several atoms defined in the class `Atom()`, we show example of a custom user-defined basis set and custom empirical parameters without usage of the library. First we create an `Atom()` object and add required number of orbitals to it. In our case, each atom has a single orbital with the orbital symmetry p_z.

```
[3]: a = tb.Orbitals('C')
a.add_orbital('pz', energy=-6.1, orbital=1, magnetic = 0)
tb.Orbitals.orbital_sets = {'C': a}
```

3.1.3 Add nearest-neighbours coupling parameters

When the basis set is determined, we have to specify the nearest-neighbour coupling:

```
[4]: tb.set_tb_params(PARAMS_C_C={'pp_pi': -2.9})
```

3.1.4 Specify geometry of the problem

The geomery of the problem may be described in several possible ways. One of them is passing the xyz-file determining positions if nodes/atoms into the constructor of the class `Hamiltonian()`.

```
[5]: xyz_file="""6
Benzene cell for the Huckel model


C1        0.00000        1.40272        0.00000
C2       -1.21479        0.70136        0.00000
C3       -1.21479       -0.70136        0.00000
C4        0.00000       -1.40272        0.00000
C5        1.21479       -0.70136        0.00000
C6        1.21479        0.70136        0.00000
"""

h = tb.Hamiltonian(xyz=xyz_file, nn_distance=1.41)
```

```
The verbosity level is 2
The radius of the neighbourhood is 1.41 Ang

----------------------

The xyz-file:
 6
Benzene cell for the Huckel model


C1        0.00000        1.40272        0.00000
C2       -1.21479        0.70136        0.00000
C3       -1.21479       -0.70136        0.00000
C4        0.00000       -1.40272        0.00000
C5        1.21479       -0.70136        0.00000
C6        1.21479        0.70136        0.00000

----------------------

Basis set
 Num of species {'C': 6}


 C
title | energy | n | l | m | s
----+-----+--+--+--+-
pz    | -6.1   | 0 | 1 | 0 | 0
----+-----+--+--+--+-

----------------------
```

Note, that along with xyz-file, we have specified the parameter `nn_distance`. This parameter stands for the maximal possible distance between nearest neighbours. It is very imporant parameter since it determines the topology on the set of atoms. Making it larger may lead to including second-nearest neighbours etc.

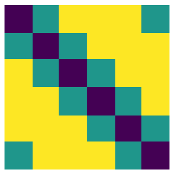
3.1.5 Compute Hamiltonian matrix elements and show Hamiltonian matrix

The Hamiltonian matrix can be computed by the member function `initialize()` of the object `h`. The matrix is stored in the attribute `h_matrix` of the corresponding object.

[6]:
```
h.initialize()

plt.figure(num=None, figsize=(1.5, 1.5))
plt.axis('off')
plt.imshow(np.real(h.h_matrix))
```

```
Radial dependence function: None

----------------------

Unique distances:
    1.4027 Ang between atoms C and C
----------------------

/home/mk/TB_project/tb_env3/lib/python3.6/site-packages/tb/diatomic_matrix_element.
↪py:99: UserWarning: There is no parameter PARAMS_C_C[pp_sigma] in the dictionary
  bond + '[' + label + ']' + ' in the dictionary', UserWarning)
```

[6]:
```
<matplotlib.image.AxesImage at 0x7fb984b2f080>
```



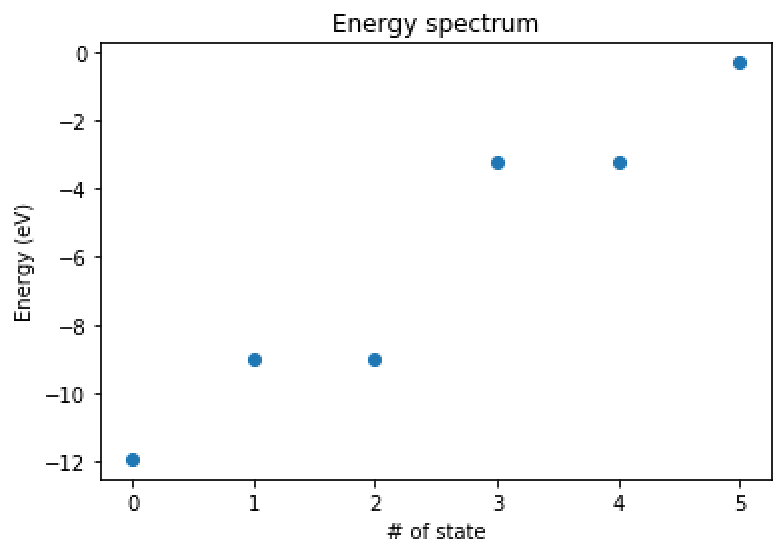
The warning message says that the programm tries to compute interatomic hoping matrix element for the σ-type diatomic symmetry between p-orbitals as well. Since we have not specify this parameter explicitelly it has been taken as zero by default. It has been made for purpose since we are interested only in π orbitals.

Now everything is ready to compute energies and eigenvectors

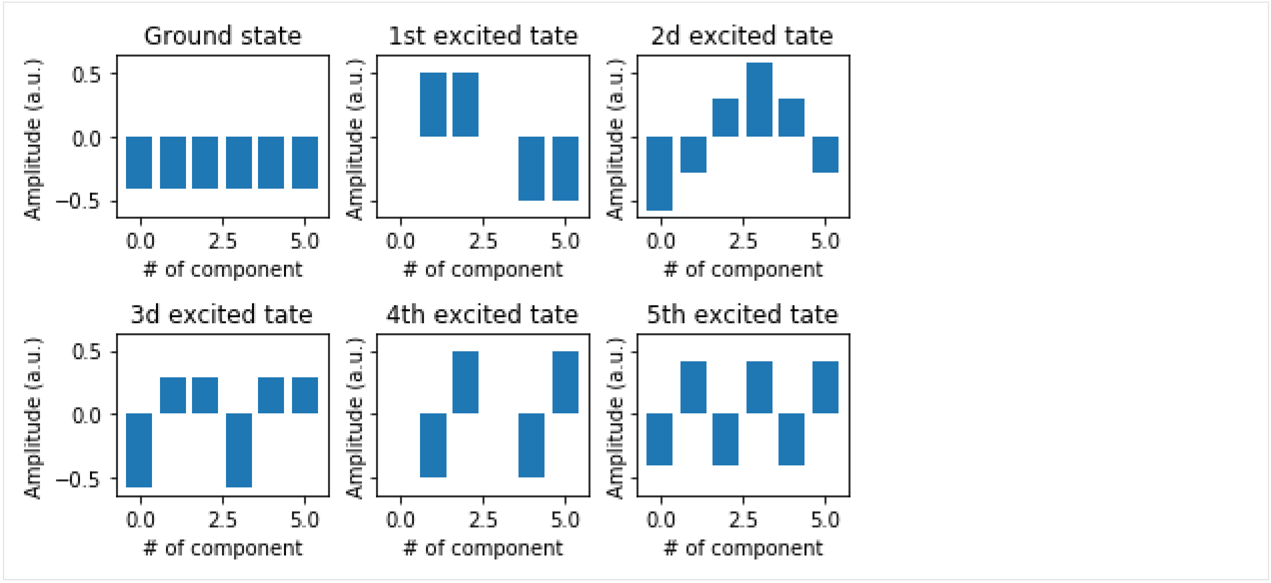
3.1.6 Hamiltonian matrix diagonalization and visualization

[7]:
```
E, V = h.diagonalize()
E = np.real(E)
ind = np.argsort(E)
E = E[ind]
V = V[:, ind]
```

[8]:
```
ax = plt.axes()
ax.set_title('Energy spectrum')
ax.set_xlabel('# of state')
ax.set_ylabel('Energy (eV)')
plt.scatter(range(len(E)), sorted(np.real(E)))
plt.show()
```



```
[9]: fig, ax = plt.subplots(2, 3, sharey='all')

ax[0, 0].set_title('Ground state')
ax[0, 0].set_xlabel('# of component')
ax[0, 0].set_ylabel('Amplitude (a.u.)')
ax[0, 0].bar(range(len(E)), (np.real(V[:, 0])))

ax[0, 1].set_title('1st excited tate')
ax[0, 1].set_xlabel('# of component')
ax[0, 1].set_ylabel('Amplitude (a.u.)')
ax[0, 1].bar(range(len(E)), (np.real(V[:, 1])))

ax[0, 2].set_title('2d excited tate')
ax[0, 2].set_xlabel('# of component')
ax[0, 2].set_ylabel('Amplitude (a.u.)')
ax[0, 2].bar(range(len(E)), (np.real(V[:, 2])))

ax[1, 0].set_title('3d excited tate')
ax[1, 0].set_xlabel('# of component')
ax[1, 0].set_ylabel('Amplitude (a.u.)')
ax[1, 0].bar(range(len(E)), (np.real(V[:, 3])))

ax[1, 1].set_title('4th excited tate')
ax[1, 1].set_xlabel('# of component')
ax[1, 1].set_ylabel('Amplitude (a.u.)')
ax[1, 1].bar(range(len(E)), (np.real(V[:, 4])))

ax[1, 2].set_title('5th excited tate')
ax[1, 2].set_xlabel('# of component')
ax[1, 2].set_ylabel('Amplitude (a.u.)')
ax[1, 2].bar(range(len(E)), (np.real(V[:, 5])))
fig.tight_layout()
```



3.2 Chain of coupled Hydrogen-like atoms

3.2.1 Prerequisites

```
[1]: %matplotlib inline
from IPython.display import display, Math, Latex
import numpy as np
np.warnings.filterwarnings('ignore')
import matplotlib.pyplot as plt

import tb
```

```
 _   _                   _   _      _
| \ | | __ _ _ __   ___ | \ | | ___| |_
|  \| |/ _` | ’_ \ / _ \|  \| |/ _ \ __|
| |\  | (_| | | | | (_) | |\  |  __/ |_
|_| \_|\__,_|_| |_|\___/|_| \_|\___|\__|


Vesion 1.0
```

3.2.2 Adding new species

Let us define two atoms, called A and B, each has a single s orbital and diffrrent energies -1 eV and -0.7 eV.

```
[2]: a = tb.Orbitals('A')
a.add_orbital(title='s', energy=-1, )
b = tb.Orbitals('B')
b.add_orbital(title='s', energy=-0.7, )
```

The geometrical parameters of the system are specified by a xyz-file:

[3]:
```
xyz_file="""2
H cell
A       0.0000000000    0.0000000000    0.0000000000
B       0.0000000000    0.0000000000    1.0000000000
"""
```

Now one needs to specify the coupling parameters between pairs of atoms with a given diatomic symmetry:

[4]:
```
tb.set_tb_params(PARAMS_A_B={'ss_sigma': 0.3})
```

3.2.3 Computing Hamiltonian matrix elements

At this point the hamiltonian matrix can be computed and visualized:

[5]:
```
h = tb.Hamiltonian(xyz=xyz_file, nn_distance=1.1).initialize()
```

```
The verbosity level is 2
The radius of the neighbourhood is 1.1 Ang

----------------------

The xyz-file:
 2
H cell
A       0.0000000000    0.0000000000    0.0000000000
B       0.0000000000    0.0000000000    1.0000000000

----------------------

Basis set
 Num of species {'A': 1, 'B': 1}


 A
title | energy | n | l | m | s
----+-----+--+--+--+-
s     | -1     | 0 | 0 | 0 | 0
----+-----+--+--+--+-


 B
title | energy | n | l | m | s
----+-----+--+--+--+-
s     | -0.7   | 0 | 0 | 0 | 0
----+-----+--+--+--+-

----------------------

Radial dependence function: None

----------------------

Discrete radial dependence function: None

----------------------

```

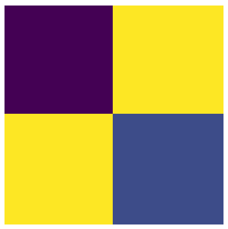
(continues on next page)

(continued from previous page)

```
Unique distances:
    1. Ang between atoms B and A
    1. Ang between atoms A and B
----------------------
```

[6]:
```
plt.figure(figsize=(2,2))
plt.axis('off')
plt.imshow(np.real(h.h_matrix))
print(h.h_matrix)
```

```
[[-1. +0.j  0.3+0.j]
 [ 0.3+0.j -0.7+0.j]]
```



Let us specify periodic bondary conditions. Below, assume that system is translated in along axis z:

[7]:
```
PRIMITIVE_CELL = [[0, 0, 2.0]]
h.set_periodic_bc(PRIMITIVE_CELL)
```

```
Primitive_cell_vectors:
 [[0, 0, 2.0]]

----------------------
```

3.2.4 Band structure computation and visualization

Now we determine a set of k-points where we want to compute band structure.

[8]:
```
num_points = 20
kk = np.linspace(0, 3.14/2, num_points, endpoint=True)
```

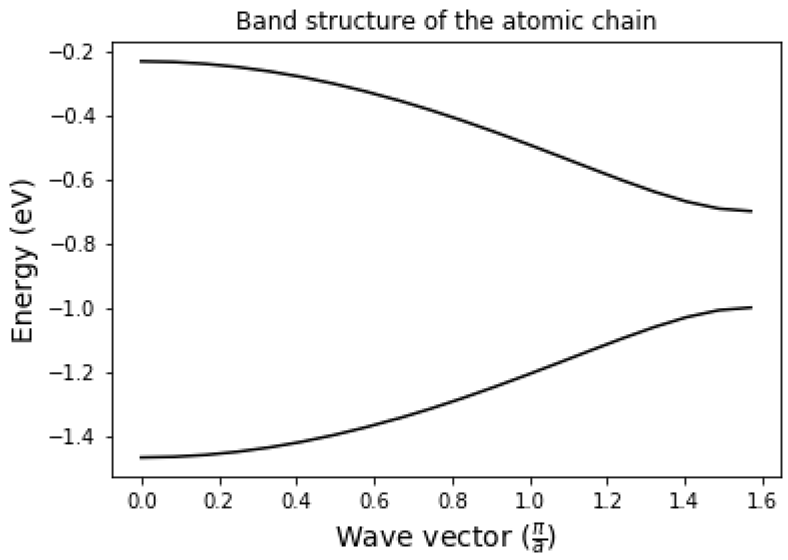
The band structure computations are performed below for each k-point:

[9]:
```
band_sructure = []

for jj in range(num_points):
    vals, _ = h.diagonalize_periodic_bc([0.0, 0.0, kk[jj]])
    band_sructure.append(vals)

band_sructure = np.array(band_sructure)

ax = plt.axes()
ax.set_title('Band structure of the atomic chain')
ax.set_xlabel(r'Wave vector ($\frac{\pi}{a}$)', fontsize=14)
ax.set_ylabel(r'Energy (eV)', fontsize=14)
```

(continues on next page)

(continued from previous page)

```
ax.plot(kk, np.sort(np.real(band_sructure)), 'k')
plt.show()
```



[ ]:

```
```

3.3 Bulk silicon

3.3.1 Prerequisites

[1]:

```
%matplotlib inline
from IPython.display import display, Math, Latex
```

3.3.2 Initialize Hamiltonian matrix

Here we compute energy spectra of a group of atoms using tight-binding method with no periodic boundary conditions applied.

As an example, we are going to compute energy spectrum for two coupled silicon atoms.

First one needs to specify basis sets for each atom kind used in the poject. There is an embeded library of basis functions for several atoms defined in the class `Orbitals()`: a basis set for Si atom is called `SiliocnSP3D5S`, and basis set `HydrogenS` for H atom. The library may be extended in future.

[2]:

```
import tb
tb.Orbitals.orbital_sets = {'Si': 'SiliconSP3D5S'}
```

```
 _   _                   _   _      _
| \ | | __ _ _ __   ___ | \ | | ___| |_
|  \| |/ _` | ’_ \ / _ \|  \| |/ _ \ __|
| |\  | (_| | | | | (_) | |\  |  __/ |_
|_| \_|\__,_|_| |_|\___/|_| \_|\___|\__|
```

(continues on next page)

(continued from previous page)

```
Vesion 1.0
```

Now the Hamiltonian matrix must be determined. The Hamiltonian matrix may be initialized in several ways. One of them is through the xyz-file determining positions if nodes/atoms.

```
[3]: xyz_file="""2
     Si2 cell
     Si1       0.0000000000    0.0000000000    0.0000000000
     Si2       1.3750000000    1.3750000000    1.3750000000
     """

     h = tb.Hamiltonian(xyz=xyz_file, nn_distance=2.5)
```

```
The verbosity level is 2
The radius of the neighbourhood is 2.5 Ang

----------------------

The xyz-file:
 2
Si2 cell
Si1       0.0000000000    0.0000000000    0.0000000000
Si2       1.3750000000    1.3750000000    1.3750000000

----------------------

Basis set
 Num of species {'Si': 2}


 Si
title   | energy  | n | l | m  | s
-----+-------+--+--+---+-
s       | -2.0196 | 0 | 0 | 0  | 0
c       | 19.6748 | 1 | 0 | 0  | 0
px      | 4.5448  | 0 | 1 | -1 | 0
py      | 4.5448  | 0 | 1 | 1  | 0
pz      | 4.5448  | 0 | 1 | 0  | 0
dz2     | 14.1836 | 0 | 2 | -1 | 0
dxz     | 14.1836 | 0 | 2 | -2 | 0
dyz     | 14.1836 | 0 | 2 | 2  | 0
dxy     | 14.1836 | 0 | 2 | 1  | 0
dx2my2  | 14.1836 | 0 | 2 | 0  | 0
-----+-------+--+--+---+-

----------------------
```

The object `h` contains all information needed to build the tight-binding Hamiltonian. In order to actually compute the tight-binding Hamiltonian an additional command has to be invoked. The `Hamiltonian()` can also accept a path to the xyz-file instead of a string containing formated data as has been done above.

Now we are ready to compute the Hamiltonian matrix:

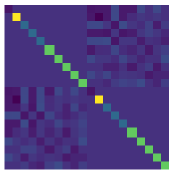
```
[4]: h.initialize()
```

```
Radial dependence function: None

-----------------------

Discrete radial dependence function: None

-----------------------

Unique distances:
    2.3816 Ang between atoms Si and Si
-----------------------
```

```
[4]: <tb.hamiltonian.Hamiltonian at 0x7ff9855deb70>
```

One may access the resuted Hamiltonian matrix and visualize it.

```
[5]: import numpy as np
import matplotlib.pyplot as plt
plt.figure(num=None, figsize=(1.5, 1.5))
plt.axis('off')
plt.imshow(np.real(h.h_matrix))
plt.show()
```



3.3.3 Set periodic boundary conditions

The previous computations can be extended adding periodic boundary conditions which are specified by the basis vectors of the primitive cell. If the translation symmetry is in all three dimentions, three primitive lattice basis vectors must be specified.

```
[6]: a_si = 5.50
PRIMITIVE_CELL = [[0, 0.5 * a_si, 0.5 * a_si],
                  [0.5 * a_si, 0, 0.5 * a_si],
                  [0.5 * a_si, 0.5 * a_si, 0]]
```

Having the primitive cell basis vectors, the periodic boundary conditions can be added to the problem by the help of the member function of the previously crated object `h` of the class `Hamiltonian()`

```
[7]: h.set_periodic_bc(PRIMITIVE_CELL)
```

```
Primitive_cell_vectors:
 [[0, 2.75, 2.75], [2.75, 0, 2.75], [2.75, 2.75, 0]]

-----------------------
```

3.3.4 Generate a set of wave vector coordinates

In order to diagonalize the Hamiltonian matrix, one has to define a set of wave vectors for which the matrix diagoalizaion will be performed. There are several ways to genegate the array of k-points. Here we show the one realized be specifying a path in the Brillouine zone through a number of high-symmentry points.

[8]:
```
from tb import get_k_coords

sym_points = ['L', 'GAMMA', 'X', 'W', 'K', 'L', 'W', 'X', 'K', 'GAMMA']
num_points = [15, 20, 15, 10, 15, 15, 15, 15, 20]
k = get_k_coords(sym_points, num_points, 'Si')
```

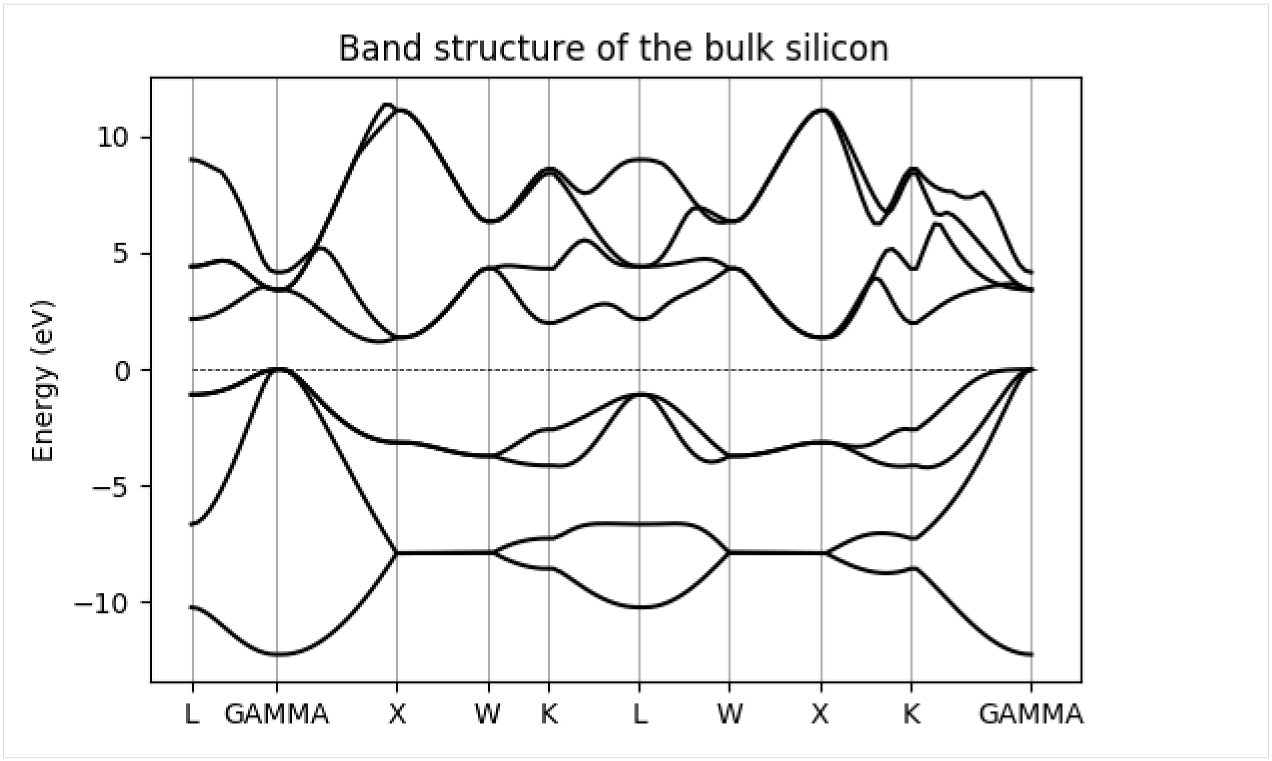
3.3.5 Computing band structure of bulk silicon

The matrix diagonalization is performed in a loop for each k-point.

[9]:
```
vals = np.zeros((sum(num_points), h.h_matrix.shape[0]), dtype=np.complex)

for jj, item in enumerate(k):
    vals[jj, :], _ = h.diagonalize_periodic_bc(item)
```

3.3.6 Visualize

[10]:
```
plt.figure(dpi=100)
ax = plt.axes()
ax.set_title('Band structure of the bulk silicon')
ax.set_ylabel('Energy (eV)')
ax.plot(np.sort(np.real(vals))[:, :8], 'k')
ax.plot([0, vals.shape[0]], [0, 0], '--', color='k', linewidth=0.5)
plt.xticks(np.insert(np.cumsum(num_points)-1,0,0), labels=sym_points)
ax.xaxis.grid()
plt.show()
```



```
[ ]:
```

3.4 Silicon nanowire

Here we compute band structure of a silicon hydrogen-passivated infinie nanowire. The nanowire width equals two crystall latices of bulk silicon. The nanowire is translated along [001] crystalographic axis of silicon.

3.4.1 Prerequisites

```
[1]: %matplotlib inline
import matplotlib.pyplot as plt
import numpy as np
import tb
```

```
 _   _                   _   _      _
| \ | | __ _ _ __   ___ | \ | | ___| |_
|  \| |/ _` | '_ \ / _ \|  \| |/ _ \ __|
| |\  | (_| | | | | (_) | |\  |  __/ |_
|_| \_|\__,_|_| |_|\___/|_| \_|\___|\__|


Vesion 1.0
```

3.4.2 Specify basis sets

Here we use two predefined basis sets, called ‘SiliconSP3D5S’ and ‘HydrogenS’, stored in the progam.

```
[2]: a_si = 5.50
PRIMITIVE_CELL = [[0, 0, a_si]]
tb.Orbitals.orbital_sets = {'Si': 'SiliconSP3D5S', 'H': 'HydrogenS'}

h = tb.Hamiltonian(xyz='../examples/input_samples/SiNW2.xyz', nn_distance=2.4)
h.initialize()
h.set_periodic_bc(PRIMITIVE_CELL)
```

```
The verbosity level is 1
The radius of the neighbourhood is 2.4 Ang

----------------------

The xyz-file:
 77
H62Si82 cell written by cluster.py
Si1    0.000000    0.000000    0.000000
Si2    2.750000    2.750000    0.000000
Si3    2.750000    0.000000    2.750000
Si4    0.000000    2.750000    2.750000
Si5    1.375000    1.375000    1.375000
Si6    4.125000    4.125000    1.375000
Si7    4.125000    1.375000    4.125000
Si8    1.375000    4.125000    4.125000
Si9    0.000000    5.500000    0.000000
                  .
                  .
                  .
There are 69 more coordinates
----------------------

Basis set
 Num of species {'Si': 41, 'H': 36}


 Si
title  | energy  | n | l | m  | s
-----+------+--+--+---+-
s      | -2.0196 | 0 | 0 | 0  | 0
c      | 19.6748 | 1 | 0 | 0  | 0
px     | 4.5448  | 0 | 1 | -1 | 0
py     | 4.5448  | 0 | 1 | 1  | 0
pz     | 4.5448  | 0 | 1 | 0  | 0
dz2    | 14.1836 | 0 | 2 | -1 | 0
dxz    | 14.1836 | 0 | 2 | -2 | 0
dyz    | 14.1836 | 0 | 2 | 2  | 0
dxy    | 14.1836 | 0 | 2 | 1  | 0
dx2my2 | 14.1836 | 0 | 2 | 0  | 0
-----+------+--+--+---+-


 H
title | energy | n | l | m | s
----+-----+--+--+--+-
s     | 0.9998 | 0 | 0 | 0 | 0
----+-----+--+--+--+-

----------------------
```

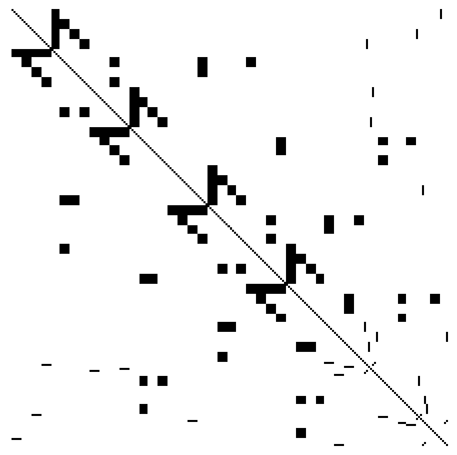
(continues on next page)

(continued from previous page)

```
Radial dependence function: None

----------------------

Discrete radial dependence function: None

----------------------

Unique distances:
    2.3816 Ang between atoms Si and Si
    1.4885 Ang between atoms Si and H
    1.4885 Ang between atoms H and Si
    1.4584 Ang between atoms H and H
----------------------

Primitive_cell_vectors:
 [[0, 0, 5.5]]

----------------------
```

[3]:
```
plt.axis('off')
plt.spy(np.abs(h.h_matrix))
```

[3]:
```
<matplotlib.image.AxesImage at 0x7f5db7bf2c88>
```

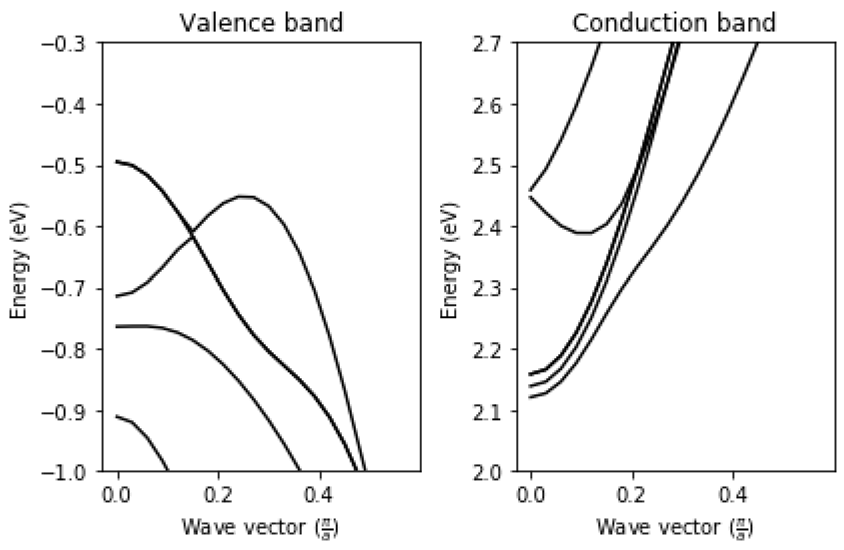


3.4.3 Band structure computation

[4]:
```
num_points = 20
kk = np.linspace(0, 3.14 / a_si, num_points, endpoint=True)
band_sructure = []

for jj in range(num_points):
    vals, _ = h.diagonalize_periodic_bc([0.0, 0.0, kk[jj]])
    band_sructure.append(vals)

band_sructure = np.array(band_sructure)
```

3.4.4 Visualization

[5]:
```
split = 100
fig, ax = plt.subplots(1, 2)
ax[0].set_ylim(-1.0, -0.3)
ax[0].plot(kk, np.sort(np.real(band_sructure))[:, :split], 'k')
ax[0].set_xlabel(r'Wave vector ($\frac{\pi}{a}$)')
ax[0].set_ylabel(r'Energy (eV)')
ax[0].set_title('Valence band')

ax[1].set_ylim(2.0, 2.7)
ax[1].plot(kk, np.sort(np.real(band_sructure))[:, split:], 'k')
ax[1].set_xlabel(r'Wave vector ($\frac{\pi}{a}$)')
ax[1].set_ylabel(r'Energy (eV)')
ax[1].set_title('Conduction band')
fig.tight_layout()
plt.savefig('nanowire_bs.pdf')
plt.show()
```



[ ]:

3.5 Band structure of bulk bismuth

[1]:
```
from tb import Hamiltonian, Orbitals
from tb import set_tb_params, get_k_coords
```

```
 _   _                   _   _      _
| \ | | __ _ _ __   ___ | \ | | ___| |_
|  \| |/ _` | '_ \ / _ \|  \| |/ _ \ __|
| |\  | (_| | | | | (_) | |\  |  __/ |_
|_| \_|\__,_|_| |_|\___/|_| \_|\___|\__|



Vesion 1.0
```

Below we set a LCAO sp3 basis set for Bi atoms.

[2]:
```
bi_orb = Orbitals('Bi')
bi_orb.add_orbital("s", energy=-10.906,
                   principal=0, orbital=0, magnetic=0, spin=0)
bi_orb.add_orbital("px", energy=-0.486,
                   principal=0, orbital=1, magnetic=-1, spin=0)
bi_orb.add_orbital("py", energy=-0.486,
                   principal=0, orbital=1, magnetic=1, spin=0)
bi_orb.add_orbital("pz", energy=-0.486,
                   principal=0, orbital=1, magnetic=0, spin=0)
bi_orb.add_orbital("s", energy=-10.906,
                   principal=0, orbital=0, magnetic=0, spin=1)
bi_orb.add_orbital("px", energy=-0.486,
                   principal=0, orbital=1, magnetic=-1, spin=1)
bi_orb.add_orbital("py", energy=-0.486,
                   principal=0, orbital=1, magnetic=1, spin=1)
bi_orb.add_orbital("pz", energy=-0.486,
                   principal=0, orbital=1, magnetic=0, spin=1)
```

The primitive cell of crystalline bismuth has two atoms:

[3]:
```
xyz_coords = """2
Bi2 cell
Bi1       0.0     0.0     0.0
Bi2       0.0     0.0     5.52321494
"""
```

[4]:
```
h = Hamiltonian(xyz=xyz_coords, nn_distance=4.6, so_coupling=1.5)
```

```
The verbosity level is 2
The radius of the neighbourhood is 4.6 Ang

----------------------

The xyz-file:
 2
Bi2 cell
Bi1       0.0     0.0     0.0
Bi2       0.0     0.0     5.52321494

----------------------

Basis set
 Num of species {'Bi': 2}


 Bi
title | energy  | n | l | m  | s
----+-------+--+--+---+-
s     | -10.906 | 0 | 0 | 0  | 0
px    | -0.486  | 0 | 1 | -1 | 0
py    | -0.486  | 0 | 1 | 1  | 0
pz    | -0.486  | 0 | 1 | 0  | 0
s     | -10.906 | 0 | 0 | 0  | 1
px    | -0.486  | 0 | 1 | -1 | 1
py    | -0.486  | 0 | 1 | 1  | 1
pz    | -0.486  | 0 | 1 | 0  | 1
----+-------+--+--+---+-
```

(continues on next page)

(continued from previous page)

```
----------------------
```

[5]:
```
import numpy as np
def radial_dep(coords):

    norm_of_coords = np.linalg.norm(coords)
    if norm_of_coords < 3.3:
        return 1
    elif 3.7 > norm_of_coords > 3.3:
        return 2
    elif 5.0 > norm_of_coords > 3.7:
        return 3
    else:
        return 100
```

[6]:
```
# 1NN - Bi-Bi
PAR1 = {'ss_sigma': -0.608,
        'sp_sigma': 1.320,
        'pp_sigma': 1.854,
        'pp_pi': -0.600}

# 2NN - Bi-Bi
PAR2 = {'ss_sigma': -0.384,
        'sp_sigma': 0.433,
        'pp_sigma': 1.396,
        'pp_pi': -0.344}

# 3NN - Bi-Bi
PAR3 = {'ss_sigma': 0,
        'sp_sigma': 0,
        'pp_sigma': 0.156,
        'pp_pi': 0}
```

[7]:
```
set_tb_params(PARAMS_BI_BI1=PAR1, PARAMS_BI_BI2=PAR2, PARAMS_BI_BI3=PAR3)
```

[8]:
```
h.initialize(radial_dep)
```

```
Radial dependence function: None

----------------------

Discrete radial dependence function:

def radial_dep(coords):

    norm_of_coords = np.linalg.norm(coords)
    if norm_of_coords < 3.3:
        return 1
    elif 3.7 > norm_of_coords > 3.3:
        return 2
    elif 5.0 > norm_of_coords > 3.7:
        return 3
    else:
        return 100
```

(continues on next page)

(continued from previous page)

```
-----------------------

Unique distances:

-----------------------
```

```
[8]: <tb.hamiltonian.Hamiltonian at 0x7f17b588ce48>
```

```
[9]: primitive_cell = [[-2.2666    , -1.30862212,  3.93223333],
                       [ 2.2666    , -1.30862212,  3.93223333],
                       [ 0.        ,  2.61724424,  3.93223333]]
```

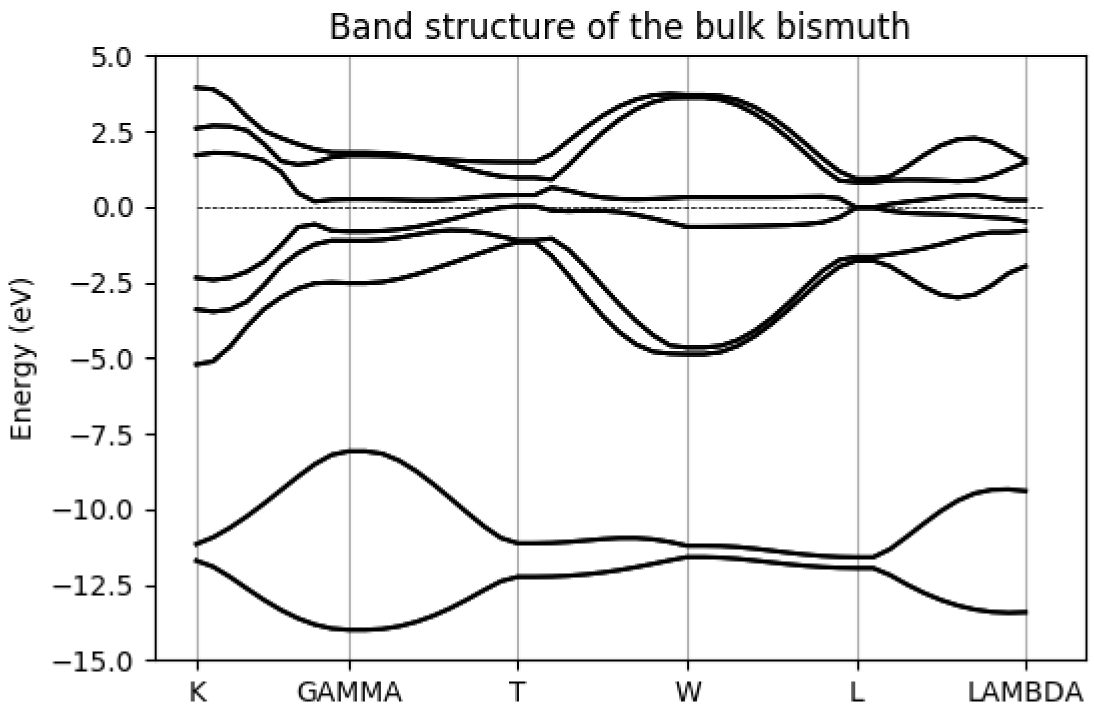
```
[10]: h.set_periodic_bc(primitive_cell)
```

```
Primitive_cell_vectors:
 [[-2.2666, -1.30862212, 3.93223333], [2.2666, -1.30862212, 3.93223333], [0.0, 2.
↪61724424, 3.93223333]]

-----------------------
```

```
[11]: sym_points = ['K', 'GAMMA', 'T', 'W', 'L', 'LAMBDA']
num_points = [10, 10, 10, 10, 10]
special_k_points = {'GAMMA': [0.0, 0.0, 0.0],
 'K': [0.35985144675492087, -0.8002652081237402, 0.5326462926072546],
 'L': [0.69305, -0.4001326040618701, 0.2663231463036273],
 'LAMBDA': [0.0, 0.0, 0.39948471945544095],
 'T': [0.0, 0.0, 0.7989694389108819],
 'U': [0.5397771701323816, -0.31164049447834485, 0.7989694389108819],
 'W': [0.3598514467549211, -0.6232809889566897, 0.7989694389108819],
 'X': [0.0, -0.8002652081237402, 0.5326462926072546]}
k_points = get_k_coords(sym_points, num_points, special_k_points)
```

```
[12]: band_structure = []
for jj, item in enumerate(k_points):
    [eigenvalues, _] =\
        h.diagonalize_periodic_bc(k_points[jj])
    band_structure.append(eigenvalues)
```

```
[13]: import matplotlib.pyplot as plt
plt.figure(dpi=100)
ax = plt.axes()
plt.ylim((-15, 5))
ax.set_title('Band structure of the bulk bismuth')
ax.set_ylabel('Energy (eV)')
ax.plot(band_structure, 'k')
ax.plot([0, len(band_structure)], [0, 0], '--', color='k', linewidth=0.5)
plt.xticks(np.insert(np.cumsum(num_points)-1,0,0), labels=sym_points)
ax.xaxis.grid()
plt.show()
```



```
[ ]:
```

3.6 Green’s function of an atomic chain

This example demonstrates a computation of the retarded Green’s function function of an atomic chain. The chain consists of identical atoms of an abstract chemical element, say an element “A”.

```
[1]:
import sys
import numpy as np
import matplotlib.pyplot as plt
import tb
import negf

xyz_file = """1
H cell
A1       0.0000000000    0.0000000000    0.0000000000
"""

a = tb.Orbitals('A')
```

```
 _   _                   _   _      _
| \ | | __ _ _ __   ___ | \ | | ___| |_
|  \| |/ _` | ’_ \ / _ \|  \| |/ _ \ __|
| |\  | (_| | | | | (_) | |\  |  __/ |_
|_| \_|\__,_|_| |_|\___/|_| \_|\___|\__|
```

Let us assume each atomic site has one s-type orbital and the energy level of -0.7 eV. The coupling matrix element equals -0.5 eV.

```
[2]:    a.add_orbital('s', -0.7)
        tb.Orbitals.orbital_sets = {'A': a}
        tb.set_tb_params(PARAMS_A_A={'ss_sigma': -0.5})
```

With all these parameters we can create an instance of the class Hamiltonian. The distance between nearest neighbours is set to 1.1 A.

```
[3]:    h = tb.Hamiltonian(xyz=xyz_file, nn_distance=1.1).initialize()
```

```
The verbosity level is 2
The radius of the neighbourhood is 1.1 Ang

----------------------

The xyz-file:
 1
H cell
A1       0.0000000000    0.0000000000    0.0000000000

----------------------

Basis set
 Num of species {'A': 1}


 A
title | energy | n | l | m | s
----+-----+--+--+--+-
s     | -0.7   | 0 | 0 | 0 | 0
----+-----+--+--+--+-

----------------------

Radial dependence function: None

----------------------

Discrete radial dependence function: None

----------------------

Unique distances:

----------------------
```

Now we need to set periodic boundary conditions with a one-dimensional unit cell and lattice constant of 1 A.

```
[4]:    h.set_periodic_bc([[0, 0, 1.0]])
```

```
Primitive_cell_vectors:
 [[0, 0, 1.0]]

----------------------
```

[ ]:

```

```

[5]:

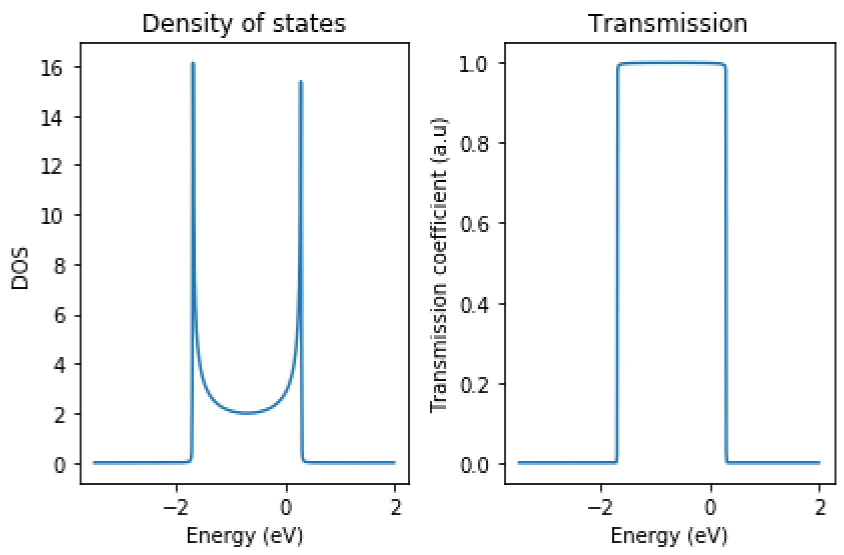
```
    h_l, h_0, h_r = h.get_hamiltonians()

    energy = np.linspace(-3.5, 2.0, 500)

    sgf_l = []
    sgf_r = []

    for E in energy:
        L, R = negf.surface_greens_function(E, h_l, h_0, h_r)
        sgf_l.append(L)
        sgf_r.append(R)

    sgf_l = np.array(sgf_l)
    sgf_r = np.array(sgf_r)

    num_sites = h_0.shape[0]
    gf = np.linalg.pinv(np.multiply.outer(energy+0.001j, np.identity(num_sites)) - h_
↪0 - sgf_l - sgf_r)

    dos = -np.trace(np.imag(gf), axis1=1, axis2=2)

    tr = np.zeros((energy.shape[0]), dtype=np.complex)

    for j, E in enumerate(energy):
        gf0 = np.matrix(gf[j, :, :])
        gamma_l = 1j * (np.matrix(sgf_l[j, :, :]) - np.matrix(sgf_l[j, :, :]).H)
        gamma_r = 1j * (np.matrix(sgf_r[j, :, :]) - np.matrix(sgf_r[j, :, :]).H)
        tr[j] = np.real(np.trace(gamma_l * gf0 * gamma_r * gf0.H))
        dos[j] = np.real(np.trace(1j * (gf0 - gf0.H)))
```

```
/home/mk/TB_project/tb_env3/lib/python3.6/site-packages/tb/diatomic_matrix_element.
↪py:115: RuntimeWarning: divide by zero encountered in double_scalars
  prefactor = ((0.5 * (1 + N)) ** l) * (((1 - N) / (1 + N)) ** (m1 * 0.5 - m2 * 0.5))␣
↪* \
/home/mk/TB_project/tb_env3/lib/python3.6/site-packages/tb/diatomic_matrix_element.
↪py:123: RuntimeWarning: divide by zero encountered in double_scalars
  ans += ((-1) ** t) * (((1 - N) / (1 + N)) ** t) / \
```

[6]:

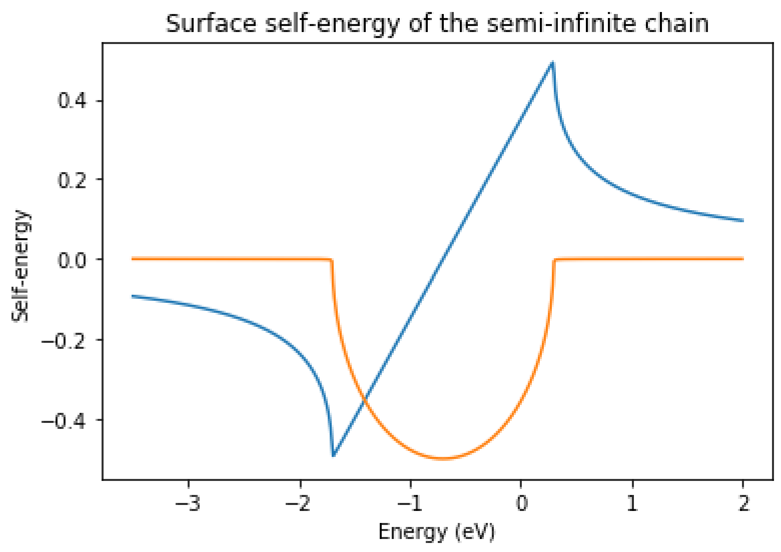
```
fig, ax = plt.subplots(1, 2)
ax[0].plot(energy, dos)
ax[0].set_xlabel('Energy (eV)')
ax[0].set_ylabel('DOS')
ax[0].set_title('Density of states')

ax[1].plot(energy, tr)
ax[1].set_xlabel('Energy (eV)')
ax[1].set_ylabel('Transmission coefficient (a.u)')
ax[1].set_title('Transmission')
fig.tight_layout()
plt.show()
```

```
/home/mk/TB_project/tb_env3/lib/python3.6/site-packages/numpy/core/numeric.py:492:␣
↪ComplexWarning: Casting complex values to real discards the imaginary part
  return array(a, dtype, copy=False, order=order)
```



```
[ ]:
```

```
[7]: ax = plt.axes()
ax.set_title('Surface self-energy of the semi-infinite chain')
ax.plot(energy, np.real(np.squeeze(sgf_l)))
ax.plot(energy, np.imag(np.squeeze(sgf_l)))
ax.set_xlabel('Energy (eV)')
ax.set_ylabel('Self-energy')
plt.show()
```



```
[ ]:
```

3.7 Reducing matrix bandwidth by sorting and compute block-tridiagonal representation of matrix

In this tutorial we will demostrate application of various sorting procedures to a list of atomic coodinates resultin in a reduced matrix bandwidth. Also, we will show how to to apply algorithms for computing the block-tridiagonal representation of a band matrix.

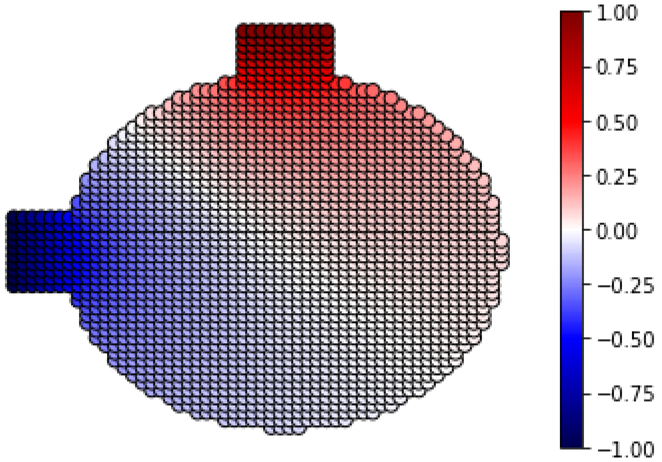
[1]:
```
%matplotlib inline
import matplotlib.pyplot as plt
import numpy as np
import tb
from tb.sorting_algorithms import sort_capacitance, sort_lexico, sort_projection


a = tb.Orbitals('A')
a.add_orbital(title='s', energy=-1, )
tb.set_tb_params(PARAMS_A_A={'ss_sigma': 0.3})

left_lead = np.array([ 644,  697,  750,  803,  857,  911,  965, 1019, 1072, 1125])
upper_lead = np.array([1878, 1879, 1880, 1881, 1882, 1883, 1884, 1885, 1886, 1887])

h = tb.HamiltonianSp(xyz='../examples/input_samples/QB1888.xyz',
                     nn_distance=1.12,
                     sort_func=sort_capacitance,
                     left_lead=left_lead,
                     right_lead=upper_lead).initialize()
```

```
 _   _                   _   _      _
| \ | | __ _ _ __   ___ | \ | | ___| |_
|  \| |/ _` | ’_ \ / _ \|  \| |/ _ \ __|
| |\  | (_| | | | | (_) | |\  |  __/ |_
|_| \_|\__,_|_| |_|\___/|_| \_|\___|\__|


Vesion 1.0
The verbosity level is 1
The radius of the neighbourhood is 1.12 Ang

---------------------

The xyz-file:
 1888
Quantum billiard
A0 -23.898305084745765 -1.5254237288135606 -0.0
A1 -23.898305084745765 -0.5084745762711869 -0.0
A2 -23.898305084745765 0.5084745762711833 -0.0
A3 -23.898305084745765 1.525423728813557 -0.0
A4 -22.88135593220339 -6.610169491525426 -0.0
A5 -22.88135593220339 -5.593220338983052 -0.0
A6 -22.88135593220339 -4.576271186440678 -0.0
A7 -22.88135593220339 -3.5593220338983045 -0.0
A8 -22.88135593220339 -2.5423728813559343 -0.0
                  .
                  .
                  .
```

(continues on next page)

(continued from previous page)

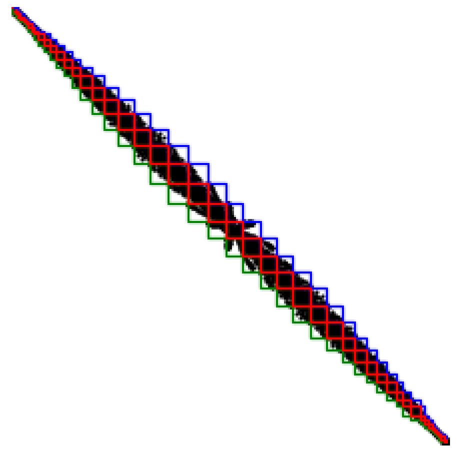
```
There are 1879 more coordinates
----------------------

/home/mk/TB_project/tb_env3/lib/python3.6/site-packages/numpy/core/numeric.py:544:␣
↪ComplexWarning: Casting complex values to real discards the imaginary part
  return array(a, dtype, copy=False, order=order, subok=True)
```



```
Basis set
 Num of species {'A': 1888}


 A
title | energy | n | l | m | s
----+-----+--+--+--+-
s     | -1     | 0 | 0 | 0 | 0
----+-----+--+--+--+-

----------------------
```

[2]:
```
hl1, h01, hr1, subblocks = h.get_hamiltonians_block_tridiagonal()
```

[3]:
```
from tb.block_tridiagonalization import show_blocks

show_blocks(subblocks, h.h_matrix)
```



```
[ ]:
```

CHAPTER

FOUR

INDICES AND TABLES

- genindex
- modindex
- search

PYTHON MODULE INDEX

INDEX