supercell_core Documentation

Release 0.0.2

Tomasz Necio

Jan 17, 2021

CONTENTS:

CHAPTER

ONE

SUPERCELL_CORE

1.1 supercell_core package

1.1.1 Submodules

1.1.2 supercell_core.atom module

class Atom(*element: str, pos: Sequence[Union[int, float]], pos_unit: supercell_core.physics.Unit = <Unit.Angstrom: 1>, velocity: Optional[Sequence[Union[int, float]]] = None, spin: Sequence[Union[int, float]] = (0, 0, 0), selective_dynamics=(True, True, True)*)

Bases: object

Class representing an atom in a crystal lattice. This is a container class, and doesn't implement any methods except for basis change. Access its properties directly.

element [string] Symbol of the chemical element

pos [(float, float, float)] Position in a cell

pos_unit [Unit] Unit of *pos* vector (Crystal or Angstrom)

velocity [(float, float, float), optional] Velocity in angstrom/fs. When read from / written to VASP POSCAR file this is the atoms' intitial velocity

spin: (int, int, int) Magnetic moment of the atom. Usually you are interested in the z-component, i.e. spins are of the form (0, 0, int). Default: (0, 0, 0)

selective_dynamics: (bool, bool, bool) Specifies whether the respective coordinates of the atom were / will be allowed to change during the ionic relaxation in VASP calculations

basis_change(*M: numpy.ndarray, new_unit:* supercell_core.physics.Unit) → *supercell_core.atom.Atom*

Creates new atom with values in different basis. Note: currently, this has no effect on the value of spin parameter!

Parameters

- **M** (*Matrix2x2*) – basis change matrix
- **new_unit** (Unit) – unit of *pos* of the new atom

Returns

Return type *Atom*

element: str

pos: Sequence[Union[int, float]]

pos_unit: *supercell_core.physics.Unit*

selective_dynamics: Tuple[bool, bool, bool]

spin: Sequence[Union[int, float]]

velocity: Optional[Sequence[Union[int, float]]]

1.1.3 supercell_core.calc module

Small utilities to help with various numeric calculations, especially with operations on arrays of matrices

inv(*m: numpy.ndarray*) → numpy.ndarray

Inverts 2x2 matrices. Why not use np.linalg.inv? To avoid LinAlgError when just one of the matrices is singular, and fill it with nans instead These are all 2x2 matrices so no harm in inverting them

Parameters **m** (*np.ndarray, shape (.., 2, 2)*) – Array of matrices

Returns

Return type np.ndarray

matmul(*m1: numpy.ndarray*, *m2: numpy.ndarray*) → numpy.ndarray

Calculates matrix multiplication of 2D matrices stored in arrays *m1* and *m2*

Parameters

- **m1** (*np.ndarray, shape(.., 2, 2)*) –
- **m2** (*np.ndarray, shape(.., 2, 2)*) –

Returns

Return type np.ndarray, shape(.., 2, 2)

matnorm(*m: numpy.ndarray*, *p: float*, *q: float*) → numpy.ndarray

Calculates L_{p, q} matrix norm for every 2D matrix in array *m*. This norm is defined as $(\sum_j(\sum_i |m_{ij}|^p)^{q/p})^{1/q}$ [1]

Parameters

- **m** (*np.ndarray, shape (.., 2, 2)*) –
- **p** (*float >= 1*) –
- **q** (*float >= 1*) –

Returns

Return type np.ndarray

Notes

Useful norms for strain calculations: Frobenius norm: ord_p = ord_q = 2 Absolute sum of elements: ord_p = ord_q = 1

References

[1] https://en.wikipedia.org/wiki/Matrix_norm#L2,1_and_Lp,q_norms

matvecmul(*m: numpy.ndarray*, *v: numpy.ndarray*) → numpy.ndarray
Acts with 2D matrix m on 2D vectors stored in array *v*

Parameters

- **m** (*np.ndarray, shape (.., 2, 2)*) –
- **v** (*np.ndarray, shape (.., 2)*) –

Returns

Return type np.ndarray, shape = v.shape

rotate(*a: numpy.ndarray*, *theta: float*) → numpy.ndarray
Rotates vectors stored in 2D matrices columns in array *a* by angle *theta* counterclockwise

Parameters

- **a** (*np.ndarray*) –
- **theta** (*float*) –

Returns

Return type np.ndarray

vecnorm(*v: numpy.ndarray*, *q: float*) → numpy.ndarray
Calculates norm of every 2D vector in *v*

Parameters

- **m** (*np.ndarray, shape (.., 2)*) –
- **q** (*float >= 1*) –

Returns

Return type np.ndarray

1.1.4 supercell_core.errors module

Contains custom Exceptions, Warnings, and their descriptions

exception BadUnitError
Bases: Exception

An exception thrown when specified unit can not be used in a given context

exception LinearDependenceError
Bases: Exception

An exception thrown when a set of linearly independent vectors is expected, but supplied values have a pair of linearly dependent vectors in them.

exception ParseError
Bases: Exception

An exception thrown when incorrect data are passed to parsing or reading functions

exception UndefinedBehaviourError
Bases: Exception

An exception thrown when it's not clear what the user's intention was

class WarningText(*value*)
Bases: enum.Enum

Enumeration that contains warning messages text

Warnings are logged when some function in the package was used in a way that suggests something is incorrect, but it's not clear, and calculation can be done anyway

AtomOutsideElementaryCell = 'Atom position specified outside the elementary cell'

ReassigningPartOfVector = 'You reassign only a part of a vector, other elements of whi

UnknownChemicalElement = 'Chemical element symbol not found in the periodic table'

ZComponentWhenNoZVector = 'A z-component of a vector was specified, but only 2 vectors

1.1.5 supercell_core.heterostructure module

class Heterostructure
Bases: object

Class describing a system of a number of 2D crystals deposited on a substrate.

Elementary cell vectors of the substrate are hereafter described as a_1 through a_3, of the 2D layers above as b_i_1 through b_i_3 (where i is layer number, counting from the substrate up, starting at 1), elementary cell of the whole system (where 2D layers have been changed due to strain) is called heterostructure _supercell_ and its vectors are referred to as c_1, c_2, and c_3.

Do not confuse *Heterostructure* with a class representing heterostructure lattice. *Heterostructure* describes a collection of crystals that build up the structure; to obtain the crystal lattice resulting from joining these crystals you must first define a way in which these crystals come together (angles, etc. – use *opt* or *calc* methods to do this), and the resulting lattice will be available as *supercell* attribute of the *Result* class (see documentation of the relevant methods)

add_layer(*layer: supercell_core.lattice.Lattice*, *pos: Optional[int] = None*, *theta: Union[int, float, Tuple[Union[int, float], Union[int, float], Union[int, float]]] = (0, 3.141592653589793, 0.0017453292519943296)*) → *supercell_core.heterostructure.Heterostructure*
Adds a 2D crystal to the system

Parameters

- **layer** (Lattice) – Lattice object describing elementary cell of the crystal to add to the system
- **pos** (*int, optional*) – Position of the layer in the stack, counting from the substrate up (first position is 0). If not specified, layer will be added at the top of the stack
- **theta** (*Angle or AngleRange, optional*) – If specified, supplied value of *theta* will be used in place of the default values in calculations such as *opt*. If a single angle is passed then the *theta* angle of that layer is fixed to that value. If an AngleRange (three floats) is passed then it is treated as a range of possible values for the *theta* value (start, stop, step). Unit: radians Default: (0, pi, 0.1 * DEGREE)

Returns for chaining

Return type *Heterostructure*

add_layers(*layers: List[Union[supercell_core.lattice.Lattice, Tuple[supercell_core.lattice.Lattice, Union[int, float, Tuple[Union[int, float], Union[int, float], Union[int, float]]]]]]*) → *supercell_core.heterostructure.Heterostructure*

Adds a lot of layers to the heterostructure at once

Parameters **layers** (*List[Lattice,res.set_vectors() or (Lattice, float), or (Lattice, (float, float, float))]*) – List of layers to add to the structure. If list element is a tuple, the second element serves the same way as *theta* parameter in *add_layer*

Returns for chaining

Return type *Heterostructure*

calc(*M=((1, 0), (0, 1))*, *thetas=None*, *calc_atoms=True*) → *supercell_core.result.Result*

Calculates strain tensor and other properties of the system in given circumstances

!! See Notes for the definition of strain tensor used here.

Parameters

- **M** (*2x2 matrix*) – Base change matrix from the base of the supercell lattice elementary cell vectors to the base of the substrate elementary cell vectors (<c_1, c_2> -> <a_1, a_2>), or, in other words, c_1 = M_11 * a_1 + M_21 * a_2, and c_2 = M_12 * a_1 + M_22 * a_2.
- **thetas** (*List[Angle|None], optional*) – Required if *theta* parameter is not fixed for all the layers in the system. If specified, it will be zipped with the layers list (starting from the layer closest to the substrate), and then, if the value corresponding to a layer is not None then that layer's *theta* angle will be set to that value. Example:

```
>>> from supercell_core import *
>>> h = heterostructure()
>>> h.set_substrate(lattice())
>>> lay1, lay2, lay3 = lattice(), lattice(), lattice()
>>> h.add_layers([(lay1, 180 * DEGREE), lay2, lay3])
>>> h.calc(M=((1, 2), (3, 4)),                    thetas = [None,
↪ 45 * DEGREE, 90 * DEGREE)                  # will be set to␣
↪(180, 45, 90) degrees for layers                  # (lay1,␣
↪lay2, lay3) repsectively
```

Returns Result object containing the results of the calculation. For more information see documentation of *Result*

Return type *Result*

Notes

Let ei be strain tensor of layer i. Let ai_1 and ai_2 be lattice vectors of layer i when not under strain. Then: $\sum_{k=1}^{2}(ei + I)_j^k ai_k = a'i_j$, where a'i_j is j-th lattice vector of a when embedded in the heterostructure.

This definition is different than the one given in [1]. To calculate strain tensor as defined in [1], use *wiki_definition=True* in relevant *Result* methods.

References

[1] https://en.wikipedia.org/wiki/Infinitesimal_strain_theory

get_layer(*pos: int*) → Tuple[*supercell_core.lattice.Lattice*, Tuple[Union[int, float], Union[int, float], Union[int, float]]]

Get Lattice object describing layer at position *pos* and information on preferred theta angles for that layer

Parameters **pos** (*int*) – Position of the layer in the stack, counting from the substrate up (first position is 0).

Returns

- *Lattice* – Lattice object describing the layer
- *(float, float, float)* – (start, stop, step) (radians) – angles used in *calc* and *opt* If a specific angle is set, returned tuple is (angle, angle, 1.0)

Raises **IndexError** – If there's less layers in the stack than *pos*

layers() → List[*supercell_core.lattice.Lattice*]

Returns layers in the heterostructure

Returns List of layers on the substrate as Lattice objects

Return type List[*Lattice*]

opt(*max_el: int = 6*, *thetas: Optional[List[Optional[List[float]]]] = None*, *algorithm: str = 'fast'*, *log: bool = False*) → *supercell_core.result.Result*

Minimises strain, and calculates its value. Note: definition of strain tensor used here is the same as in documentation of *Heterostructure.calc*

Parameters

- **max_el** (*int, optional*) – Defines maximum absolute value of the strain tensor element The opt calculation is O(`max_el`^2). Default: 6
- **thetas** (*List[List[float]|None] | List[float], optional*) – Allows to override thetas specified for the layers. If specified, it must be equal in length to the number of layers. For a given layer, the list represents values of theta to check. If None is is passed instead of one of the inner lists, then default is not overriden. If there are only two layers, the argument can be passed just as a list of floats, without the need for nesting. All angles are in radians.
- **algorithm** (*str, optional*) – Default: "fast" Accepted values: "fast", "direct"
- **log** (*bool, optional*) – Default: False If True, the Result will contain pandas.DataFrame in *log* attribute The resulting DataFrame will contain information on supercell for every combination of interlayer angles considered. Requires *pandas* extra dependency

Returns Result object containing the results of the optimisation. For more information see documentation of *Result*

Return type *Result*

remove_layer(*pos: int*) → *supercell_core.heterostructure.Heterostructure*

Removes layer in position *pos*

Parameters **pos** (*int*) – Position of the layer in the stack, counting from the substrate up (first position is 0).

Returns

Return type *Heterostructure*

Raises **IndexError** – If there's less layers in the stack than *pos*

set_substrate(*substrate:* supercell_core.lattice.Lattice) → *supercell_core.heterostructure.Heterostructure*

Defines substrate layer of the heterostructure

Parameters **substrate** (Lattice) – Lattice object describing elementary cell of the substrate on which 2D crystals will be laid down

Returns for chaining

Return type *Heterostructure*

substrate() → *supercell_core.lattice.Lattice*

Getter for the substrate of the heterostructure

Returns

Return type *Lattice*

Raises **AttributeError** – if substrate wasn't set yet

heterostructure() → *supercell_core.heterostructure.Heterostructure*

Creates a Heterostructure object

Returns a new Heterostructure object

Return type *Heterostructure*

1.1.6 supercell_core.input_parsers module

eat_line(*s: List[str]*) → List[str]

get_line(*s: List[str]*) → str

iter_magmom(*magmom: str*)

parse_POSCAR(*poscar: str*, *atomic_species: Optional[List[str]] = None*, *magmom: Optional[str] = None*, *normalise_positions: Optional[bool] = False*) → *supercell_core.lattice.Lattice*

Reads lattice data from a string, treating it as VASP POSCAR file Format documentation: https://cms.mpi.univie.ac.at/wiki/index.php/POSCAR

Parameters

- **poscar** (*str*) – Contents of the POSCAR file
- **atomic_species** (*List[str]*) – Contains symbols of chemical elements in the same order as in POTCAR One symbol per atomic species Required if the POSCAR file does not specify the atomic species
- **magmom** (*str, optional*) – Contents of the MAGMOM line from INCAR file. Default: all spins set to zero
- **normalise_positions** (*bool, optional*) – If True, atomic positions are moved to be within the elementary cell (preserving location of atoms in the whole crystal) Default: False

Returns object representing the same lattice as the VASP would-be input

Return type *Lattice*

Raises

- **IOError** – If something is wrong with I/O on the file

- **ParseError** – If supplied file is not a valid POSCAR, or if the arguments supplied cannot be reasonably paired with the input file in a way that makes a proper Lattice

read_POSCAR(*filename: str*, ***kwargs*) → *supercell_core.lattice.Lattice*

Reads VASP input file "POSCAR"

Parameters

- **filename** (*str*) –
- **kwargs** – Passed on to *parse_POSCAR*

Returns object representing the same lattice as the VASP input files

Return type *Lattice*

Raises

- **IOError** – If something is wrong with I/O on the file
- **ParseError** – If supplied file is not a valid POSCAR, or if the arguments supplied cannot be reasonably paired with the input file in a way that makes a proper Lattice

read_supercell_in(*filename: str*) → *supercell_core.heterostructure.Heterostructure*

1.1.7 supercell_core.lattice module

class Lattice

Bases: object

Object representing a 2D crystal lattice (or rather its unit cell)

Initially set with default values: no atoms, unit cell vectors equal to unit vectors (of length 1 angstrom each) along x, y, z axes.

Elementary cell vectors are here and elsewhere referred to as b_1, b_2, b_3.

add_atom(*element: str, pos: Sequence[Union[int, float]], spin: Sequence[Union[int, float]] = (0, 0, 0), unit: supercell_core.physics.Unit = <Unit.Angstrom: 1>, normalise_positions: bool = False*) → *supercell_core.lattice.Lattice*

Adds a single atom to the unit cell of the lattice

Parameters

- **element** (*str*) – Symbol of the chemical element (does NOT accept full names) If unknown symbol is passed, warns the user and defaults to hydrogen
- **pos** (*2D or 3D vector*) – Position of the atom in the unit cell. If atomic position is not within the parallelepiped described by unit cell vectors, position is accepted but a warning is issued.
- **spin** (*3D vector, optional*) – Describes spin of the atom (s_x, s_y, s_z), default: (0, 0, 0)
- **unit** (Unit) – Gives unit in which *pos* was given (must be either *Unit.Angstrom* or *Unit.Crystal*)
- **normalise_positions** (*bool*) – If True, atomic positions are moved to be within the elementary cell (preserving location of atoms in the whole crystal) Default: False

Returns for chaining

Return type *Lattice*

Warns **UserWarning** – If an unknown chemical element symbol is passed as *element* If the atomic position is outside the elementary cell defined by lattice vectors, and *normalise_positions* is False

Raises **TypeError** – If supplied arguments are of incorrect type

add_atoms(*atoms: List[*supercell_core.atom.Atom*]*) → *supercell_core.lattice.Lattice*
Adds atoms listed in *atoms* to the unit cell

Parameters **atoms** (*List[Atom]*) – List of atoms to add to the lattice unit cell

Returns for chaining

Return type *Lattice*

atoms(*unit: supercell_core.physics.Unit = <Unit.Angstrom: 1>*) → List[*supercell_core.atom.Atom*]
Lists atoms in the unit cell

Parameters **unit** (*Unit*) – Unit in which to return atomic positions

Returns List of atoms in an unit cell of the lattice

Return type List[*Atom*]

basis_change_matrix() → numpy.ndarray
Returns basis change matrix from basis of lattice vectors (Direct) to Cartesian basis.

Returns

Return type np.ndarray, shape (3,3)

draw(*ax=None*, *cell_coords=None*, *scatter_kwargs=None*, *border_kwargs=None*)
Requires matplotlib. Creates an image of the lattice elementary cell as a matplotlib plot.

Parameters

- **cell_coords** (*List[Vector2D], optional*) – Each point in *cell_coords* is an offset in Lattice vector basis. For each of those offsets, all atoms in the cell will be drawn with their positions moved by this offset. Use integer values to get consistent drawings. Note: You can use things like [(-0.5, -0.5),

  (-0.5, 0.5), (0.5, -0.5), (0.5, 0.5)] to get a picture of a translated elementary cell.

  Default: [(0, 0)] (draws just one elementary cell)

- **ax** (*matplotlib axes.Axes object, optional*) – If given, will draw the elementary cell to ax

Returns

Return type Figure, Axes

Notes

Resulting axes has points labeled; use *ax.legend(loc='best')* to turn on the legend.

rotate(*theta: float*)
Rotates the lattice around the origin.

Parameters **theta** (*float*) – Angle in radians

Returns for chaining

Return type *Lattice*

save_POSCAR(*filename: Optional[str] = None*, *silent: bool = False*) → *supercell_core.lattice.Lattice*
Saves lattice structure in VASP POSCAR file. Order of the atomic species is the same as order of their first occurence in the list generated by *atoms* method of this object. This order is printed to stdout. If atoms have non-zero z-spins, the MAGMOM flag is also printed to stdout.

Parameters

- **filename** (*str, optional*) – if not provided, writes to stdout
- **silent** (*bool, default: False*) – if True, the atomic order and MAGMOM flag are not printed to stdout

Returns for chaining

Return type *Lattice*

save_xsf(*filename: Optional[str] = None*) → *supercell_core.lattice.Lattice*
Saves lattice structure in XCrysDen XSF file

Parameters **filename** (*str, optional*) – if not provided, writes to stdout

Returns for chaining

Return type *Lattice*

set_vectors(**args: Sequence[Union[int, float]]*, *atoms_behaviour: Optional[*supercell_core.physics.Unit*] = None*) → *supercell_core.lattice.Lattice*
Sets (or changes) lattice vectors to given values

Parameters

- **args** (*2 2D vectors, or 3 3D vectors*) – Two or three vectors describing unit cell. Since this program is used to calculate values regarding 2D lattices, it is not necessary to specify z-component of the vectors. In that case, the z-component defaults to 0 for the first two vectors (unless it was set by previous invocation of *set_vectors*; this issues a warning but works). All vectors must be the same length. Unit : angstrom (1e-10 m)
- **atoms_behaviour** (*Unit, optional*) – Described how atomic positions of the atoms already added to the lattice should behave. They are treated as if they were specified with the unit *atomic_behaviour*. This argument is optional if the lattice does not contain any atoms, otherwise necessary.

Returns for chaining

Return type *Lattice*

Raises

- **TypeError** – If supplied values don't match expected types
- ***LinearDependenceError*** – If supplied vectors are not linearly independent
- ***UndefinedBehaviourError*** – If trying to change unit cell vectors of a lattice that contains atoms, without specifying what to do with the atomic positions in the unit cell

Warns **UserWarning** – When unit cell vectors are reset in a way that disregards previous change of the values of z-component or the third vector (suggesting that user might not be aware that their values are not default)

translate_atoms(*vec: Sequence[Union[int, float]]*, *unit: supercell_core.physics.Unit = <Unit.Angstrom: 1>*)
Moves all atoms in the unit cell by some vector *vec*

Parameters

- **vec** (*VectorLike*) – Translation vector. If len(vec) == 2, third argument is set to zero.

- **unit** (Unit, *optional*) – Unit of *vec*, by default: angstroms

Returns

Return type None

vectors() → List[numpy.ndarray]
Lists lattice vectors

Returns List of unit cell vectors (in angstrom)

Return type List[Vec3D]

lattice()
Creates a Lattice object

Returns a new Lattice object

Return type *Lattice*

1.1.8 supercell_core.physics module

This file contains names of various physical qties, objects, units etc. and their mapping to values used in the calculations

class Unit(*value*)
Bases: enum.Enum

Enumeration representing different ways in which lattice vectors and atomic positions can be specified

Angstrom
angstrom (1e-10 m)

Crystal
representation of vector in the base of elementary cell vectors of a given lattice

Angstrom = 1

Crystal = 2

atomic_number(*element_symbol: str*) → int
Inverse function of *element_symbol* :param element_symbol: must be a valid symbol of chemical element, can not be its full name :type element_symbol: str

Returns atomic number of the element

Return type int

Raises **ValueError or TypeError** – If there is no chemical element with such symbol

element_symbol(*atomic_no: int*) → str
Returns symbol of a chemical element given its atomic number

Parameters **atomic_no** (*int*) – atomic number of an element

Returns symbol of that element

Return type str

Raises **IndexError or TypeError** – If there is no chemical element with such atomic number

1.1.9 supercell_core.result module

class Result(*heterostructure: Heterostructure*, *superlattice:* supercell_core.lattice.Lattice, *thetas: List[Union[int, float]]*, *strain_tensors: List[numpy.ndarray]*, *strain_tensors_wiki: List[numpy.ndarray]*, *ADt: numpy.ndarray*, *ABtrs: List[numpy.ndarray]*, *atom_count=None*)

Bases: object

A class containing results of supercell calculations.

Notes

Terminology:

MN – basis change matrix from N to M

(the order of the letters makes it easy to combine these: MM' @ M'N == MN)

v_M – vector v (unit vector of basis V) in basis M v_Ms – an array of vectors v in basis M stg_lay – list of "stg" for each of the layers

(len(stg_lay) == len(self.layers()))

A, a – basis of lattice vectors of the substrate B, b – basis of lattice vectors of a given layer Br, br – basis B rotated by theta angle Btr, btr – basis of lattice vectors of a given layer when embedded

in the heterostructure (rotated – r, and stretched due to strain – t)

D, d – basis of vectors composed of integer linear combinations of the B basis vectors

Dt, dt – basis of vectors composed of the integer linear combinations of the A basis vectors, represents possible supercell lattice vectors

X, x – cartesian basis (unit vectors: (1 angstrom, 0), (0, 1 angstrom))

Note that the vector space of all the mentioned objects is R^2

M() → numpy.ndarray
Returns 2D matrix M: M @ (v in supercell basis) = (v in substrate lattice basis). In other words, matrix composed of superlattice vectors in substrate lattice basis. All matrix components are integers.

Returns

Return type Matrix2x2

atom_count() → int
Returns number of atoms in the superlattice elementary cell

Returns

Return type int

layer_Ms() → List[numpy.ndarray]
Returns list of matrices Mi: Mi @ (v in supercell basis) = (v in basis of heterostructure layer no. i when it is stretched due to strain). In other words, list of matrices composed of superlattice vectors in stretched layers' lattice basis. All matrix components are integers.

Returns

Return type Matrix2x2

max_strain()
Returns absolute maximum value of sum of strains on all layers.

Returns

Return type float

Notes

The returned value is minimised in *Heterostructure.opt* calculations.

strain_tensors(*wiki_definition=False*) → List[numpy.ndarray]
Returns list of strain tensors for each of the heterostructure's layers.

Parameters **wiki_definition** (*optional, bool*) – Default: False. If True, definition from [1] will be used instead of the default (see docs of *Heterostructure.calc* for details)

Returns

Return type List[Matrix2x2]

References

[1] https://en.wikipedia.org/wiki/Infinitesimal_strain_theory

superlattice() → *supercell_core.lattice.Lattice*
Returns a Lattice object representing supercell (elementary cell of the heterostructure)

Returns

Return type *Lattice*

thetas() → List[Union[int, float]]
Returns list of theta values corresponding to the layers in the heterostructure. (in radians)

Returns

Return type List[float]

1.1.10 Module contents

Top-level exported functions and classes:

```
>>> from .input_parsers import read_POSCAR, parse_POSCAR
>>> from .lattice import Atom, lattice, Lattice
>>> from .heterostructure import heterostructure, Heterostructure
>>> from .physics import VectorNumpy, VectorLike, Unit, DEGREE, PERIODIC_TABLE, \
>>>     element_symbol, Z_SPIN_DOWN, Z_SPIN_UP
```

CHAPTER

TWO

README

CHAPTER

THREE

INDICES AND TABLES

- genindex
- modindex
- search

PYTHON MODULE INDEX

S

INDEX

R

S

T

U

V

W

Z