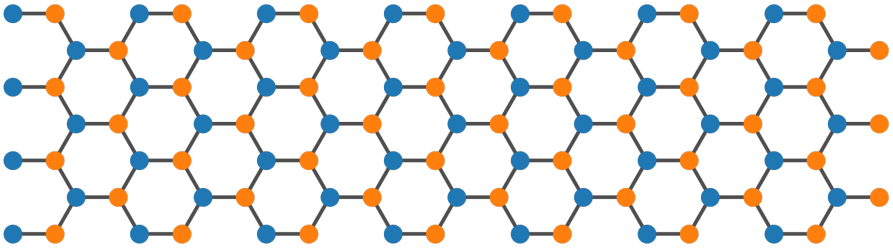
lattpy

Dylan L. Jones

Nov 02, 2022

USER GUIDE



"Any dimension and shape you like."

LattPy is a simple and efficient Python package for modeling Bravais lattices and constructing (finite) lattice structures in any dimension. It provides an easy interface for constructing lattice structures by simplifying the configuration of the unit cell and the neighbor connections - making it possible to construct complex models in just a few lines of code and without the headache of adding neighbor connections manually. You will save time and mental energy for more important matters.

Master			
Dev			

Warning: This project is still in development and might change significantly in the future!

CHAPTER

ONE

INSTALLATION

LattPy is available on PyPI:

```
$ pip install lattpy
```

Alternatively, it can be installed via GitHub:

```
$ pip install git+https://github.com/dylanljones/lattpy.git@VERSION
```

where *VERSION* is a release or tag (e.g. *0.6.4*). The project can also be cloned/forked and installed via

```
$ python setup.py install
```

CHAPTER

TWO

QUICK-START

A new instance of a lattice model is initialized using the unit-vectors of the Bravais lattice. After the initialization the atoms of the unit-cell need to be added. To finish the configuration the connections between the atoms in the lattice have to be set. This can either be done for each atom-pair individually by calling `add_connection` or for all possible pairs at once by callling `add_connections`. The argument is the number of unique distances of neighbors. Setting a value of 1 will compute only the nearest neighbors of the atom.

```
>>> import numpy as np
>>> from lattpy import Lattice
>>>
>>> latt = Lattice(np.eye(2))                  # Construct a Bravais lattice with square␣
↪unit-vectors
>>> latt.add_atom(pos=[0.0, 0.0])              # Add an Atom to the unit cell of the␣
↪lattice
>>> latt.add_connections(1)                    # Set the maximum number of distances␣
↪between all atoms
>>>
>>> latt = Lattice(np.eye(2))                  # Construct a Bravais lattice with square␣
↪unit-vectors
>>> latt.add_atom(pos=[0.0, 0.0], atom="A")    # Add an Atom to the unit cell of the␣
↪lattice
>>> latt.add_atom(pos=[0.5, 0.5], atom="B")    # Add an Atom to the unit cell of the␣
↪lattice
>>> latt.add_connection("A", "A", 1)           # Set the max number of distances between␣
↪A and A
>>> latt.add_connection("A", "B", 1)           # Set the max number of distances between␣
↪A and B
>>> latt.add_connection("B", "B", 1)           # Set the max number of distances between␣
↪B and B
>>> latt.analyze()
```

Configuring all connections using the `add_connections`-method will call the `analyze`-method directly. Otherwise this has to be called at the end of the lattice setup or by using `analyze=True` in the last call of `add_connection`. This will compute the number of neighbors, their distances and their positions for each atom in the unitcell.

To speed up the configuration prefabs of common lattices are included. The previous lattice can also be created with

```
>>> from lattpy import simple_square
>>> latt = simple_square(a=1.0, neighbors=1)
```

lattpy.simple_chain([a, atom, neighbors])	Creates a 1D lattice with one atom at the origin of the unit cell.
lattpy.alternating_chain([a, atom1, atom2, ...])	Creates a 1D lattice with two atoms in the unit cell.
lattpy.simple_square([a, atom, neighbors])	Creates a square lattice with one atom at the origin of the unit cell.
lattpy.simple_rectangular([a1, a2, atom, ...])	Creates a rectangular lattice with one atom at the origin of the unit cell.
lattpy.graphene([a])	Creates a hexagonal lattice with two atoms in the unit cell.
lattpy.simple_cubic([a, atom, neighbors])	Creates a cubic lattice with one atom at the origin of the unit cell.
lattpy.nacl_structure([a, atom1, atom2, ...])	Creates a NaCl lattice structure.

So far only the lattice structure has been configured. To actually construct a (finite) model of the lattice the model has to be built:

```
>>> latt.build(shape=(5, 3))
```

To view the built lattice the `plot`-method can be used:

```
>>> import matplotlib.pyplot as plt
>>> from lattpy import simple_square
>>> latt = simple_square()
>>> latt.build(shape=(5, 3))
>>> latt.plot()
>>> plt.show()
```

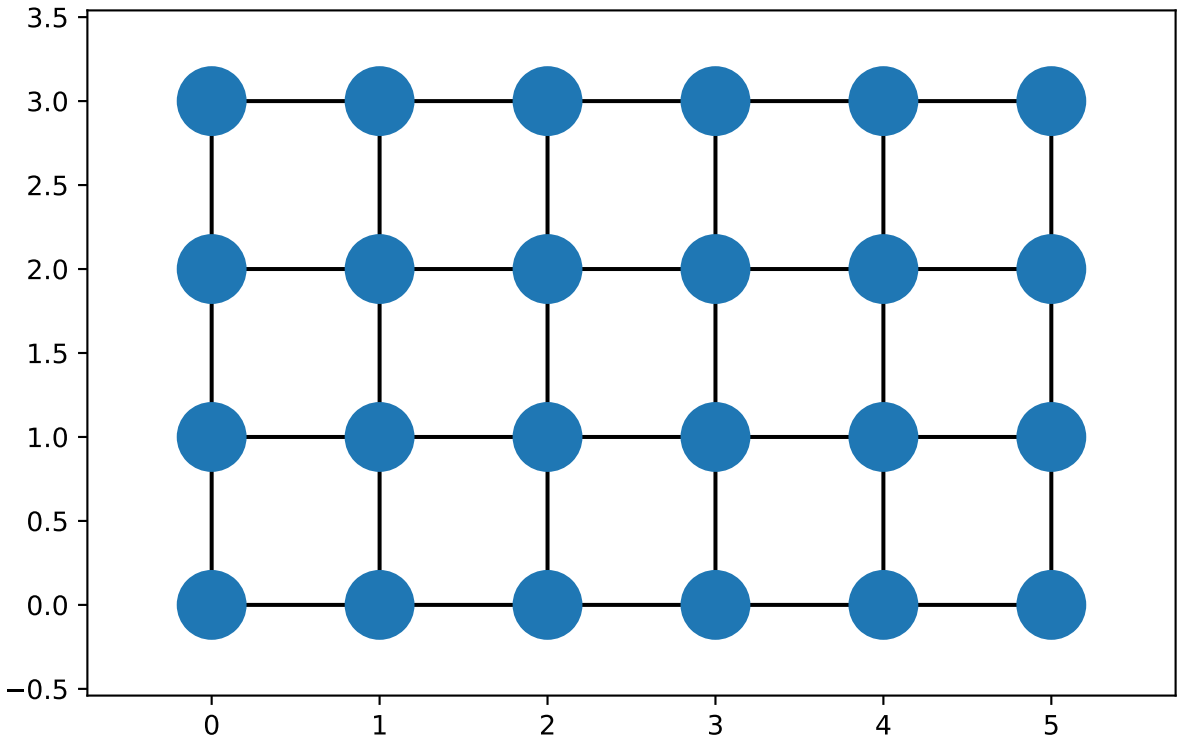
After configuring the lattice the attributes are available. Even without building a (finite) lattice structure all attributes can be computed on the fly for a given lattice vector, consisting of the translation vector `n` and the atom index `alpha`. For computing the (translated) atom positions the `get_position` method is used. Also, the neighbors and the vectors to these neighbors can be calculated. The `dist_idx`-parameter specifies the distance of the neighbors (0 for nearest neighbors, 1 for next nearest neighbors, ...):

```
>>> latt.get_position(n=[0, 0], alpha=0)
[0. 0.]
>>> latt.get_neighbors([0, 0], alpha=0, distidx=0)
[[ 1  0  0]
 [ 0 -1  0]
 [-1  0  0]
 [ 0  1  0]]
>>> latt.get_neighbor_vectors(alpha=0, distidx=0)
[[ 1.  0.]
 [ 0. -1.]
 [-1.  0.]
 [ 0.  1.]]
```

Also, the reciprocal lattice vectors can be computed

```
>>> latt.reciprocal_vectors()
[[6.28318531 0.        ]
 [0.         6.28318531]]
```

or used to construct the reciprocal lattice:


3.5
3.0
2.5
2.0
1.5
1.0
0.5
0.0
−0.5
0
1
2
3
4
5

```
>>> rlatt = latt.reciprocal_lattice()
```

The 1. Brillouin zone is the Wigner-Seitz cell of the reciprocal lattice:

```
>>> bz = rlatt.wigner_seitz_cell()
```

The 1.BZ can also be obtained by calling the explicit method of the direct lattice:

```
>>> bz = latt.brillouin_zone()
```

If the lattice has been built the necessary data is cached. The lattice sites of the structure then can be accessed by a simple index `i`. The syntax is the same as before, just without the `get_` prefix:

```
>>> i = 2
>>>
>>> # Get position of the atom with index i=2
>>> positions = latt.position(i)
>>> # Get the atom indices of the nearest neighbors of the atom with index i=2
>>> neighbor_indices = latt.neighbors(i, distidx=0)
>>> # the nearest neighbors can also be found by calling (equivalent to dist_idx=0)
>>> neighbor_indices = latt.nearest_neighbors(i)
```

CHAPTER

THREE

TUTORIAL

In this tutorial the main features and usecases of LattPy are discussed and explained. Throughout the tutorial the packages `numpy` and `matplotlib` are used. *LattPy* is imported as `lp` - using a similar alias as the other scientific computing libaries:

```
>>> import numpy as np
>>> import matplotlib.pyplot as plt
>>> import lattpy as lp
```

3.1 Configuration

The `Lattice` object of LattPy can be configured in a few steps. There are three fundamental steps to defining a new structure:

1. Defining basis vectors of the lattice
2. Adding atoms to the unit cell
3. Adding connections to neighbors

3.1.1 Basis vectors

The core of a Bravais lattice are the basis vectors $\boldsymbol{A} = \boldsymbol{a}_i$ with $i = 1, \ldots, d$, which define the unit cell of the lattice. Each lattice point is defined by a translation vector $\boldsymbol{n} = (n_1, \ldots, n_d)$:

$$\boldsymbol{R}_{\boldsymbol{n}} = \sum_{i=1}^{d} n_i \boldsymbol{a}_i.$$

A new `Lattice` instance can be created by simply passing the basis vectors of the system. A one-dimensional lattice can be initialized by passing a scalar or an 1×1 array to the `Lattice` constructor:

```
>>> latt = lp.Lattice(1.0)
>>> latt.vectors
[[1.0]]
```

For higher dimensional lattices an $d \times d$ array with the basis vectors as rows,

$$\boldsymbol{A} = \begin{pmatrix} a_{11} & \ldots & a_{1d} \\ \vdots & \ddots & \vdots \\ a_{d1} & \ldots & a_{dd} \end{pmatrix},$$

is expected. A square lattice, for example, can be initialized by a 2D identity matrix:

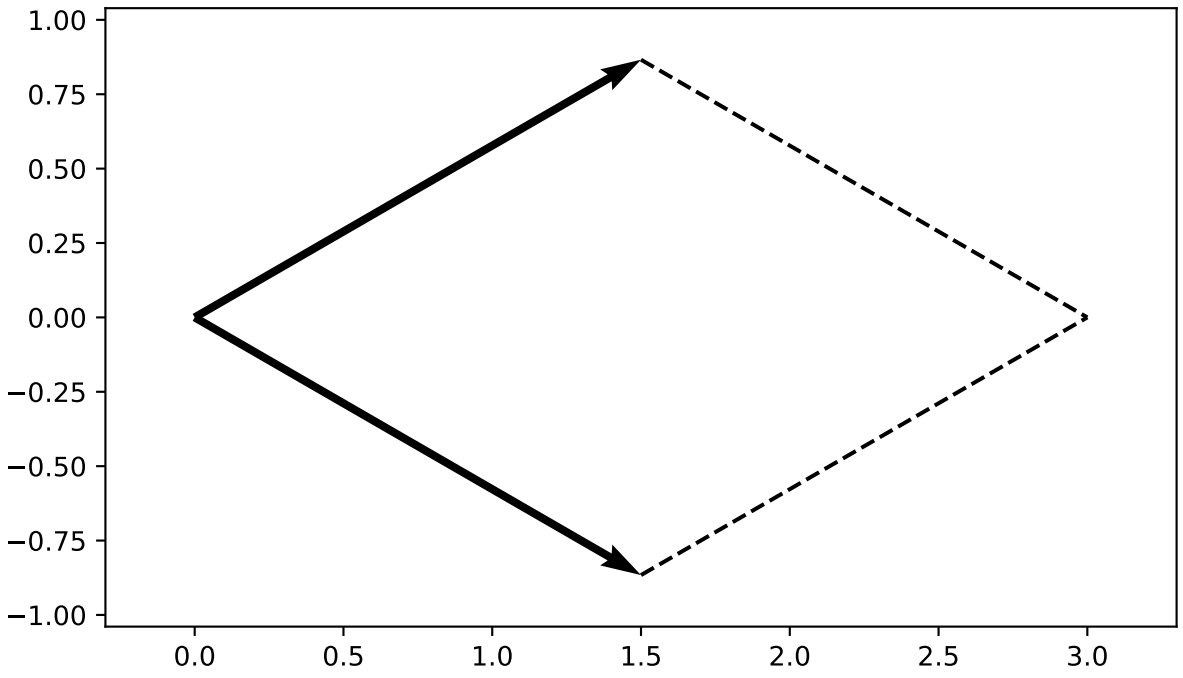
```
>>> latt = lp.Lattice(np.eye(2))
>>> latt.vectors
[[1. 0.]
 [0. 1.]]
```

The basis vectors of frequently used lattices can be intialized via the class-methods of the `Lattice` object, for example:

`lattpy.Lattice.chain`([a])	Initializes a one-dimensional lattice.
`lattpy.Lattice.square`([a])	Initializes a 2D lattice with square basis vectors.
`lattpy.Lattice.rectangular`([a1, a2])	Initializes a 2D lattice with rectangular basis vectors.
`lattpy.Lattice.hexagonal`([a])	Initializes a 2D lattice with hexagonal basis vectors.
`lattpy.Lattice.oblique`(alpha[, a1, a2])	Initializes a 2D lattice with oblique basis vectors.
`lattpy.Lattice.hexagonal3d`([a, az])	Initializes a 3D lattice with hexagonal basis vectors.
`lattpy.Lattice.sc`([a])	Initializes a 3D simple cubic lattice.
`lattpy.Lattice.fcc`([a])	Initializes a 3D face centered cubic lattice.
`lattpy.Lattice.bcc`([a])	Initializes a 3D body centered cubic lattice.

The resulting unit cell of the lattice can be visualized via the `plot_cell()` method:

```
>>> latt = lp.Lattice.hexagonal(a=1)
>>> latt.plot_cell()
>>> plt.show()
```



3.1.2 Adding atoms

Until now only the lattice type has been defined via the basis vectors. To define a lattice structure we also have to specify the basis of the lattice by adding atoms to the unit cell. The positions of the atoms in the lattice then are given by

$$\boldsymbol{R}_{n\alpha} = \boldsymbol{R}_{\boldsymbol{n}} + \boldsymbol{r}_{\boldsymbol{\alpha}},$$

where $\boldsymbol{r}_{\boldsymbol{\mu}}$ is the position of the atom α relative to the origin of the unit cell.

In LattPy, atoms can be added to the `Lattice` object by calling `add_atom()` and supplying the position and type of the atom:

```
>>> latt = lp.Lattice.square()
>>> latt.add_atom([0.0, 0.0], "A")
```

If the position is omitted the atom is placed at the origin of the unit cell. The type of the atom can either be the name or an `Atom` instance:

```
>>> latt = lp.Lattice.square()
>>> latt.add_atom([0.0, 0.0], "A")
>>> latt.add_atom([0.5, 0.5], lp.Atom("B"))
>>> latt.atoms[0]
Atom(A, size=10, 0)
>>> latt.atoms[1]
Atom(B, size=10, 1)
```

If a name is passed, a new `Atom` instance is created. We again can view the current state of the unit cell:

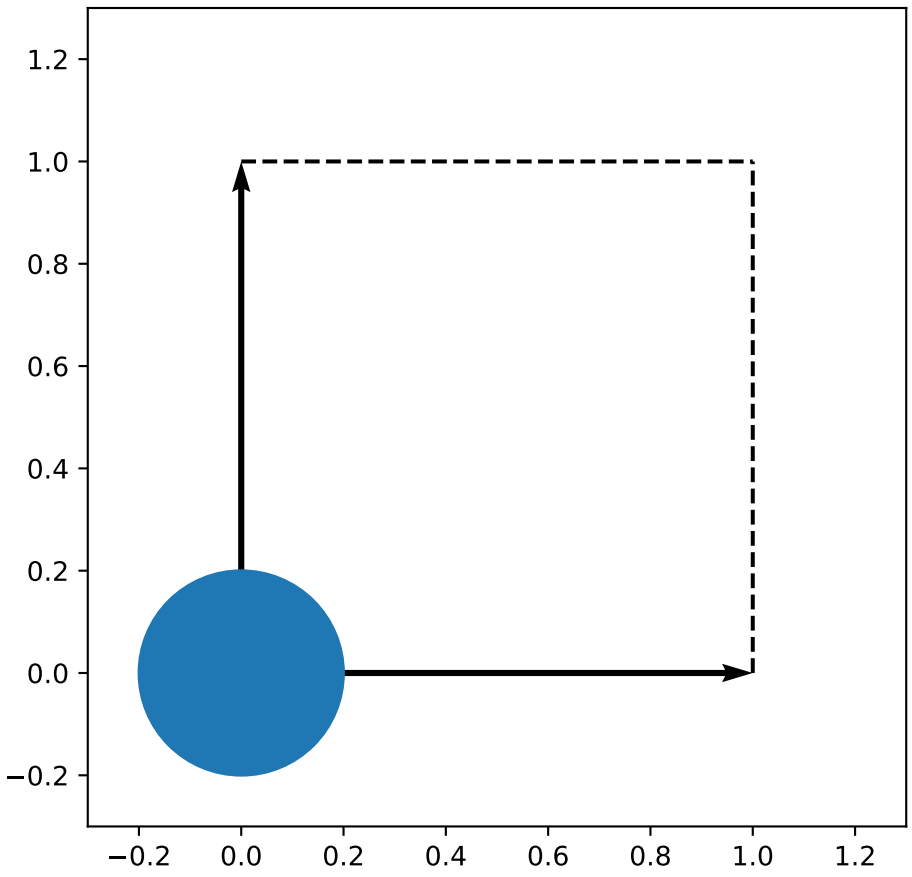
```
>>> latt = lp.Lattice.square()
>>> latt.add_atom([0.0, 0.0], "A")
>>> ax = latt.plot_cell()
>>> ax.set_xlim(-0.3, 1.3)
>>> ax.set_ylim(-0.3, 1.3)
>>> plt.show()
```

3.1.3 Adding connections

Finally, the connections of the atoms to theirs neighbors have to be set up. LattPy automatically connects the neighbors of sites up to a specified level of neighbor distances, i.e. nearest neighbors, next nearest neighbors and so on. The maximal neighbor distance can be configured for each pair of atoms independently. Assuming a square lattice with two atoms A and B in the unit cell, the connections between the A atoms can be set to next nearest neighbors, while the connections between A and B can be set to nearest neighbors only:

```
>>> latt = lp.Lattice.square()
>>> latt.add_atom([0.0, 0.0], "A")
>>> latt.add_atom([0.5, 0.5], "B")
>>> latt.add_connection("A", "A", 2)
>>> latt.add_connection("A", "B", 1)
>>> latt.analyze()
```

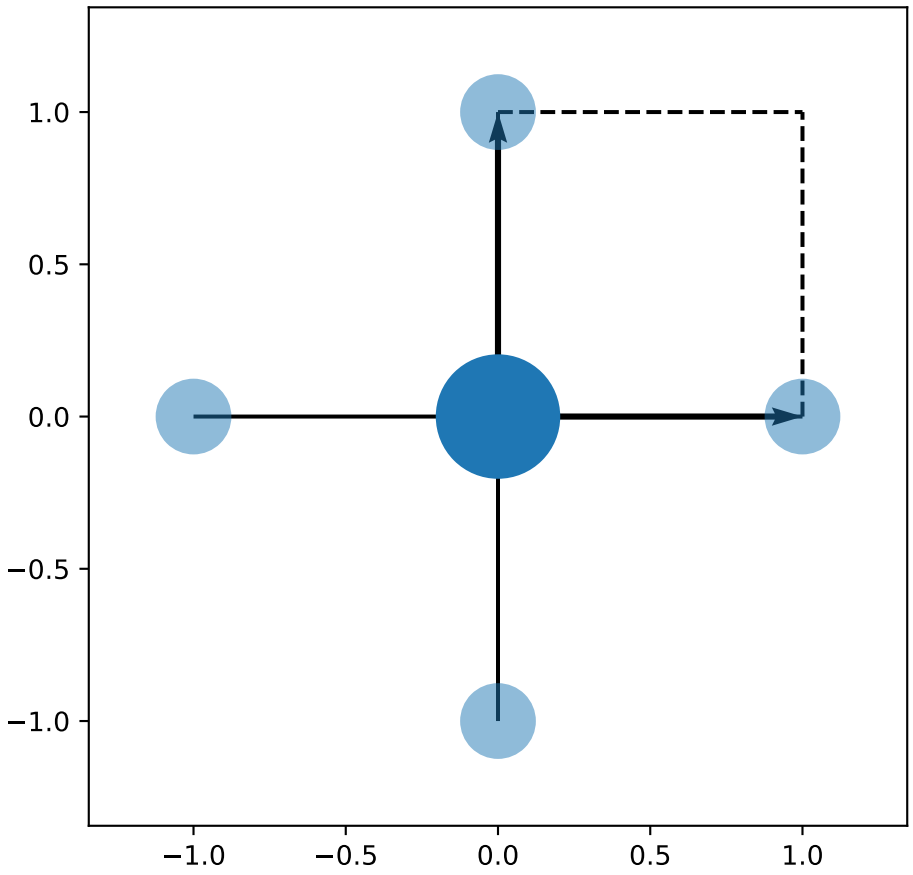
After setting up all the desired connections in the lattice the `analyze` method has to be called. This computes the actual neighbors for all configured distances of the atoms in the unit cell. Alternatively, the distances for all pairs of


1.2
1.0
0.8
0.6
0.4
0.2
0.0
−0.2
−0.2
0.0
0.2
0.4
0.6
0.8
1.0
1.2

the sites in the unit cell can be configured at once by calling the `add_connections` method, which internally calls the `analyze` method. This speeds up the configuration of simple lattices.

The final unit cell of the lattice, including the atoms and the neighbor information, can again be visualized:

```
>>> latt = lp.Lattice.square()
>>> latt.add_atom()
>>> latt.add_connections(1)
>>> latt.plot_cell()
>>> plt.show()
```



3.2 General lattice attributes

After configuring the lattice the general attributes and methods are available. Even without building a (finite) lattice structure all properties can be computed on the fly for a given lattice vector, consisting of the translation vector `n` and the index `alpha` of the atom in the unit cell.

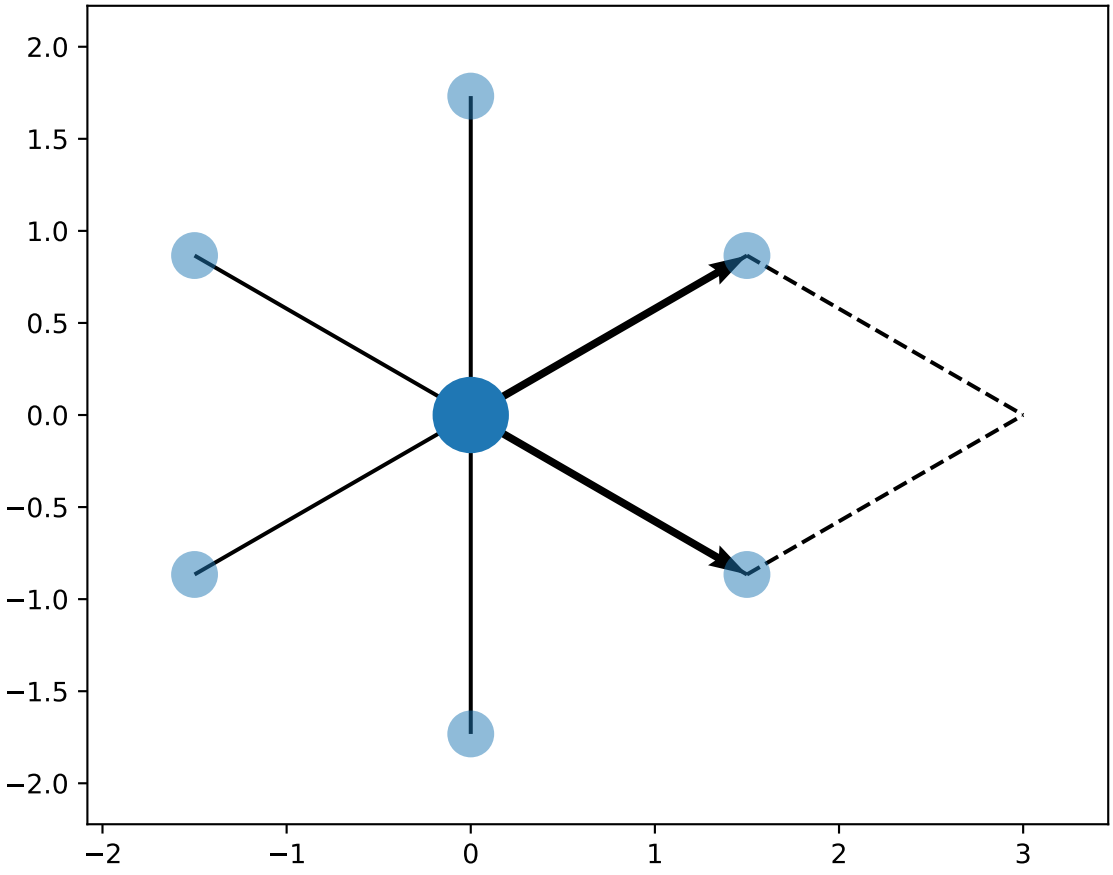
We will discuss all properties with a simple hexagonal lattice as example:

```
>>> latt = lp.Lattice.hexagonal()
>>> latt.add_atom()
>>> latt.add_connections()
```

(continues on next page)

(continued from previous page)

```
>>> latt.plot_cell()
>>> plt.show()
```



3.2.1 Unit cell properties

The basis vectors of the lattice can be accessed via the `vectors` property:

```
>>> latt.vectors
[[ 1.5        0.8660254]
 [ 1.5       -0.8660254]]
```

The size and volume of the unit cell defined by the basis vectors are also available:

```
>>> latt.cell_size
[1.5        1.73205081]
>>> latt.cell_volume
2.598076211353316
```

The results are all computed in cartesian corrdinates.

3.2.2 Transformations and atom positions

Coordinates in *cartesian* coordinates (also referred to as *world* coordinates) can be tranformed to the *lattice* or *basis* coordinate system and vice versa. Consider the point $\boldsymbol{n} = (n_1, \ldots, n_d)$ in the basis coordinate system, which can be understood as a translation vector. The point $\boldsymbol{x} = (x_1, \ldots, x_d)$ in cartesian coordinates then is given by

$$\boldsymbol{x} = \sum_{i=1}^{d} n_i \boldsymbol{a}_i.$$

```
>>> n = [1, 0]
>>> x = latt.transform(n)
>>> x
[1.5       0.8660254]
>>> latt.itransform(x)
[1. 0.]
```

The points in the world coordinate system do not have to match the lattice points defined by the basis vectors:

```
>>> latt.itransform([1.5, 0.0])
[0.5 0.5]
```

latt = lp.Lattice.hexagonal() ax = latt.plot_cell()

ax.plot([1.5], [0.0], marker="x", color="r", ms=10) lp.plotting.draw_arrows(ax, 0.5 * latt.vectors[0], color="r", width=0.005) lp.plotting.draw_arrows(ax, [0.5 * latt.vectors[1]], pos=[0.5 * latt.vectors[0]], color="r", width=0.005) plt.show()

Both methods are vectorized and support multiple points as inputs:

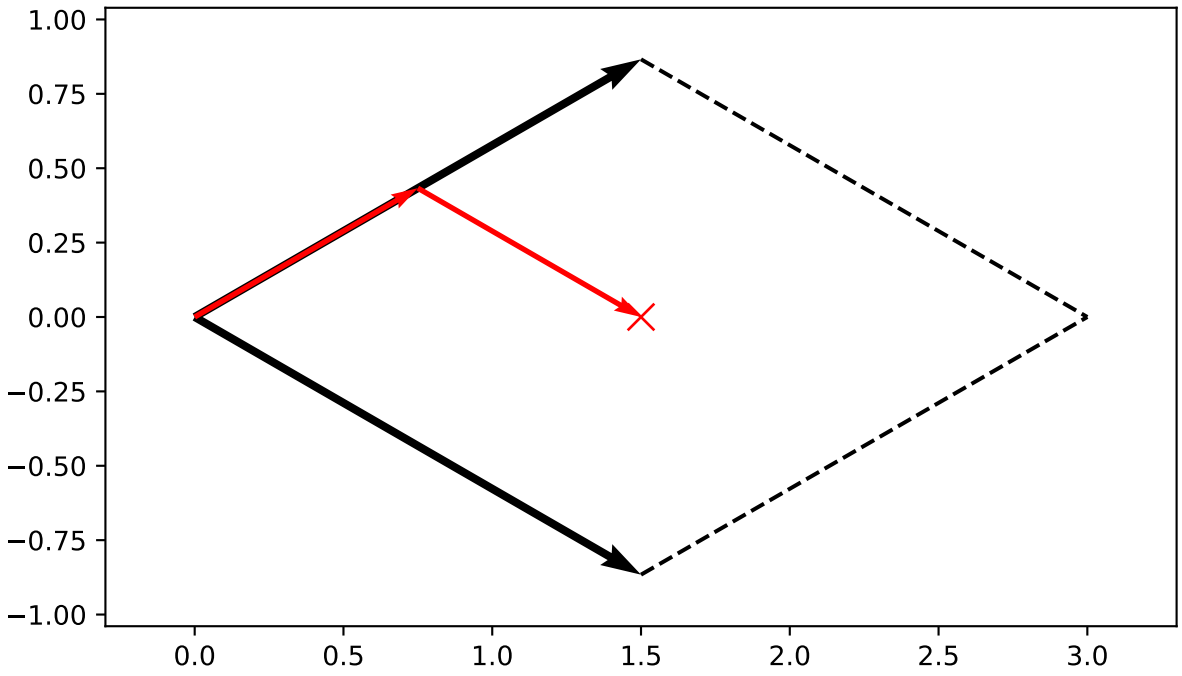
```
>>> n = [[0, 0] [1, 0], [2, 0]]
>>> x = latt.transform(n)
>>> x
[[0.         0.        ]
 [1.5        0.8660254 ]
 [3.         1.73205081]]
>>> latt.itransform(x)
[[ 0.00000000e+00  0.00000000e+00]
 [ 1.00000000e+00 -3.82105486e-17]
 [ 2.00000000e+00 -7.64210971e-17]]
```

Note: As can be seen in the last example, some inaccuracies can occur in the transformations depending on the data type due to machine precision.

Any point $\boldsymbol{r}$ in the cartesian cooridnates can be translated by a translation vector $\boldsymbol{n} = (n_1, \ldots, n_d)$:

$$\boldsymbol{x} = \boldsymbol{r} + \sum_{i=1}^{d} n_i \boldsymbol{a}_i.$$

The inverse operation is also available. It returns the translation vector $\boldsymbol{n} = (n_1, \ldots, n_d)$ and the point $\boldsymbol{r}$ such that $\boldsymbol{r}$ is the neareast possible point to the origin:


1.00
0.75
0.50
0.25
0.00
−0.25
−0.50
−0.75
−1.00
0.0
0.5
1.0
1.5
2.0
2.5
3.0

```
>>> n = [1, 0]
>>> r = [0.5, 0.0]
>>> x = latt.translate(n, r)
>>> x
[2.        0.8660254]
>>> latt.itransform(x)
(array([1, 0]), array([0.5, 0. ]))
```

Again, both methods are vectorized:

```
>>> n = [[0, 0], [1, 0], [2, 0]]
>>> r = [0.5, 0]
>>> x = latt.translate(n, r)
>>> x
[[0.5        0.        ]
 [2.         0.8660254 ]
 [3.5        1.73205081]]
>>> n2, r2 = latt.itranslate(x)
>>> n2
[[0 0]
 [1 0]
 [2 0]]
>>> r2
[[0.5 0. ]
 [0.5 0. ]
 [0.5 0. ]]
```

Specifiying the index of the atom in the unit cell `alpha` the positions of a translated atom can be obtained via the translation vector $\boldsymbol{n}$:

```
>>> latt.get_position([0, 0], alpha=0)
[0. 0.]
>>> latt.get_position([1, 0], alpha=0)
[1.5       0.8660254]
>>> latt.get_position([2, 0], alpha=0)
[3.         1.73205081]
```

Multiple positions can be computed by the `get_positions` method. The argument is a list of lattice indices, consisting of the translation vector `n` and the atom index `alpha` as a single array. Note the last column of `indices` in the following example, where all atom indices `alpha=0`:

```
>>> indices = [[0, 0, 0], [1, 0, 0], [2, 0, 0]]
>>> latt.get_positions(indices)
[[0.         0.        ]
 [1.5        0.8660254 ]
 [3.         1.73205081]]
```

3.2.3 Neighbors

The maximal number of neighbors of the atoms in the unit cell for *all* distance levels can be accessed by the property `num_neighbors`:

```
>>> latt.num_neighbors
[6]
```

Since the lattice only contains one atom in the unit cell a array with one element is returned. Similar to the position of a lattice site, the neighbors of a site can be obatined by the translation vector of the unit cell and the atom index. Additionaly, the distance level has to be specified via an index. The nearest neighbors of the site at the origin can, for example, be computed by calling

```
>>> neighbors = latt.get_neighbors([1, 0], alpha=0, distidx=0)
>>> neighbors
[[ 2 -1  0]
 [ 0  1  0]
 [ 0  0  0]
 [ 2  0  0]
 [ 1 -1  0]
 [ 1  1  0]]
```

The results ara again arrays cvontaining translation vectors plus the atom index `alpha`:

```
>>> neighbor = neighbors[0]
>>> n, alpha = neighbor[:-1], neighbor[-1]
>>> n
[ 2 -1]
>>> alpha
0
```

In addition to the lattice indices the positions of the neighbors can be computed:

```
>>> latt.get_neighbor_positions([1, 0], alpha=0, distidx=0)
[[ 1.5         2.59807621]
 [ 1.5        -0.8660254 ]
 [ 0.          0.        ]
 [ 3.          1.73205081]
 [ 0.          1.73205081]
 [ 3.          0.        ]]
```

or the vectors from the site to the neighbors

```
>>> latt.get_neighbor_positions(alpha=0, distidx=0)
[[ 0.          1.73205081]
 [ 0.         -1.73205081]
 [-1.5        -0.8660254 ]
 [ 1.5         0.8660254 ]
 [-1.5         0.8660254 ]
 [ 1.5        -0.8660254 ]]
```

Here no translation vector is needed since the vectors from a site to it's neighbors are translational invariant.

3.2.4 Reciprocal lattice

The reciprocal lattice vectors of the `Lattice` instance can be computed via

```
>>> latt.reciprocal_vectors()
[[ 2.0943951   3.62759873]
 [ 2.0943951  -3.62759873]]
```

Also, the reciprocal lattice can be constrcuted, which has the reciprocal vectors from the current lattice as basis vectors:

```
>>> rlatt = latt.reciprocal_lattice()
>>> rlatt.vectors
[[ 2.0943951   3.62759873]
 [ 2.0943951  -3.62759873]]
```

The reciprocal lattice can be used to construct the 1. Brillouin zone of a lattice, whcih is defined as the Wigner-Seitz cell of the reciprocal lattice:

```
>>> bz = rlatt.wigner_seitz_cell()
```

Additionally, a explicit method is available:

```
>>> bz = latt.brillouin_zone()
```

The 1. Brillouin zone can be visualized:

```
>>> latt = lp.Lattice.hexagonal()
>>> bz = latt.brillouin_zone()
>>> bz.draw()
>>> plt.show()
```

3.3 Finite lattice models

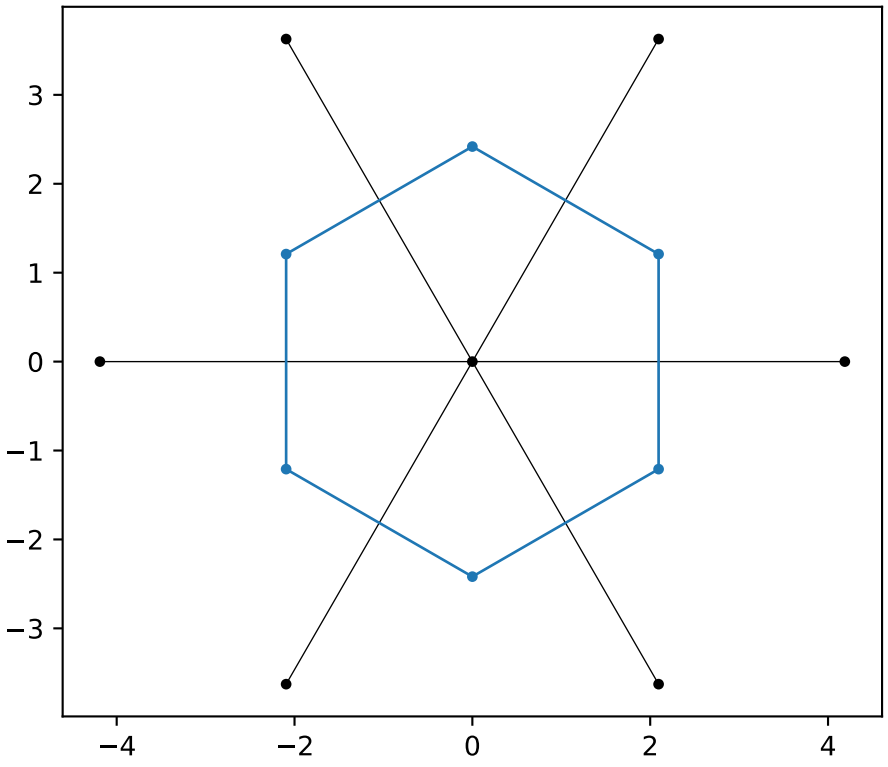
So far only abstract, infinite lattices have been discussed. In order to construct a finite sized model of the configured lattice structure we have to build the lattice.

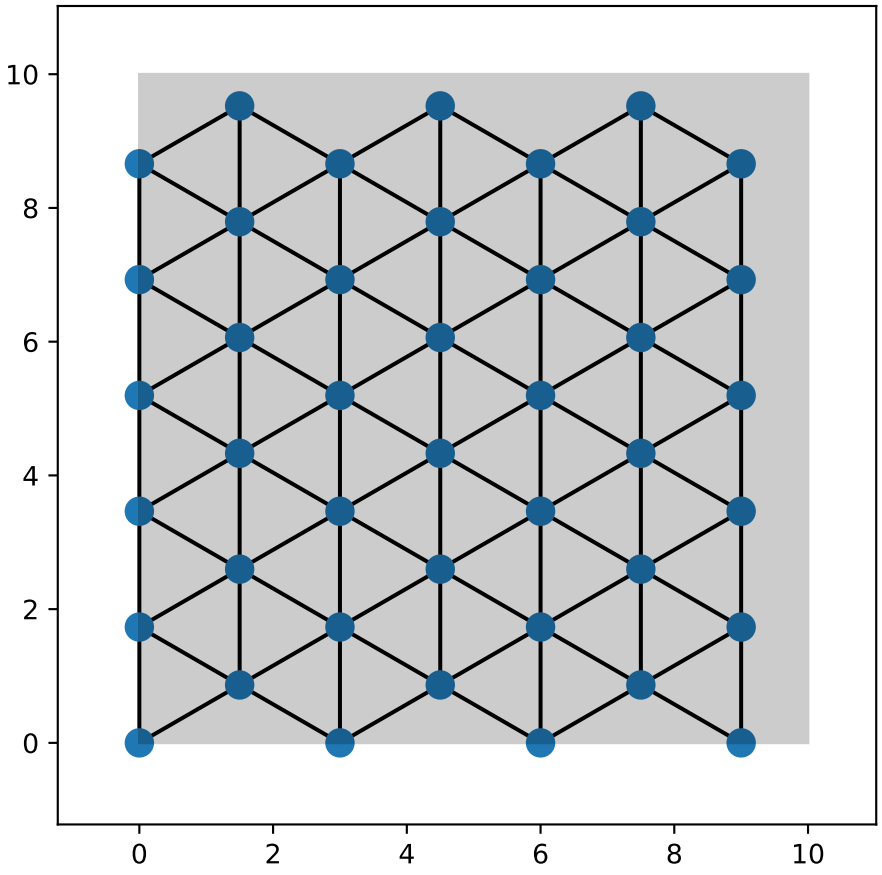
3.3.1 Build geometries

By default, the shape passed to the `build` is used to create a box in cartesian coordinates. Alternatively, the geometry can be constructed in the basis of the lattice by setting `primitive=True`. As an example, consider the hexagonal lattice. We can build the lattice in a box of the specified shape:

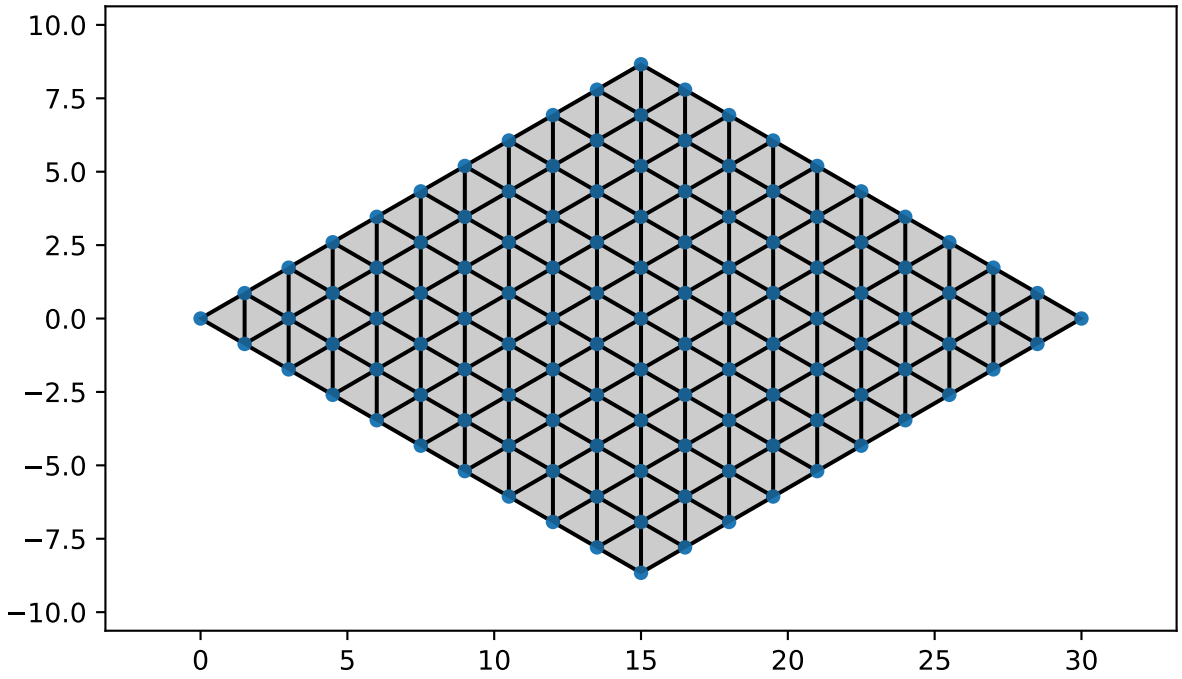
```
>>> latt = lp.Lattice.hexagonal()
>>> latt.add_atom()
>>> latt.add_connections()
>>> s = latt.build((10, 10))
>>> ax = latt.plot()
>>> s.plot(ax)
>>> plt.show()
```

or in the coordinate system of the lattice, which results in


3
2
1
0
−1
−2
−3
−4
−2
0
2
4

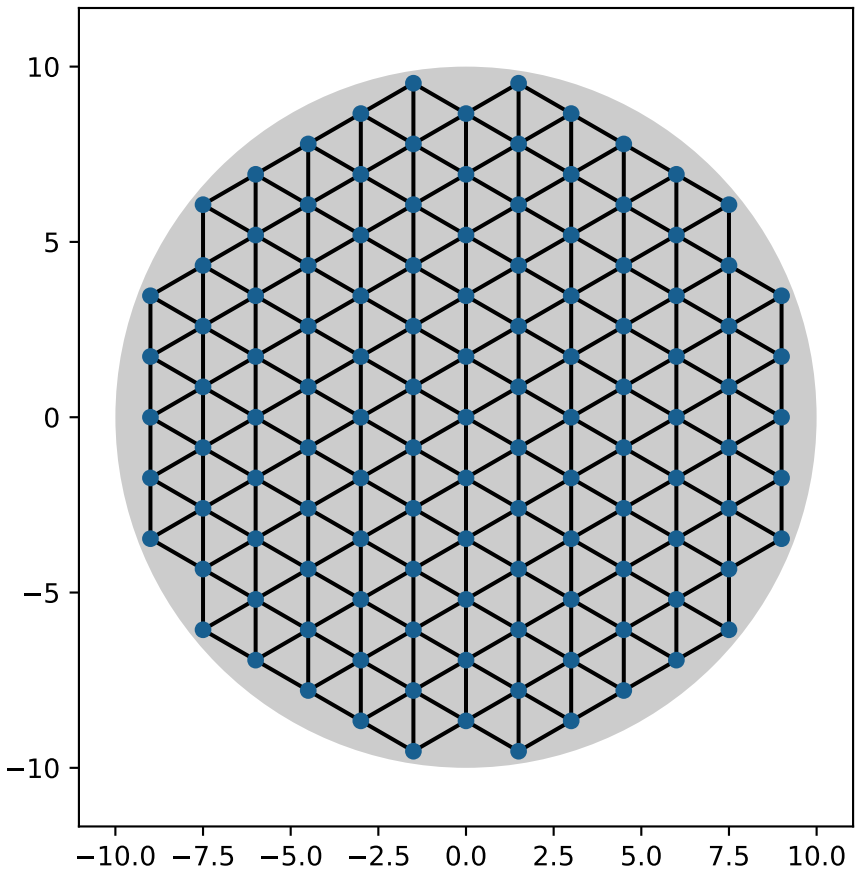

10
8
6
4
2
0
0
2
4
6
8
10

```
>>> latt = lp.Lattice.hexagonal()
>>> latt.add_atom()
>>> latt.add_connections()
>>> s = latt.build((10, 10), primitive=True)
>>> ax = latt.plot()
>>> s.plot(ax)
>>> plt.show()
```



Other geometries can be build by using `AbstractShape` ojects:

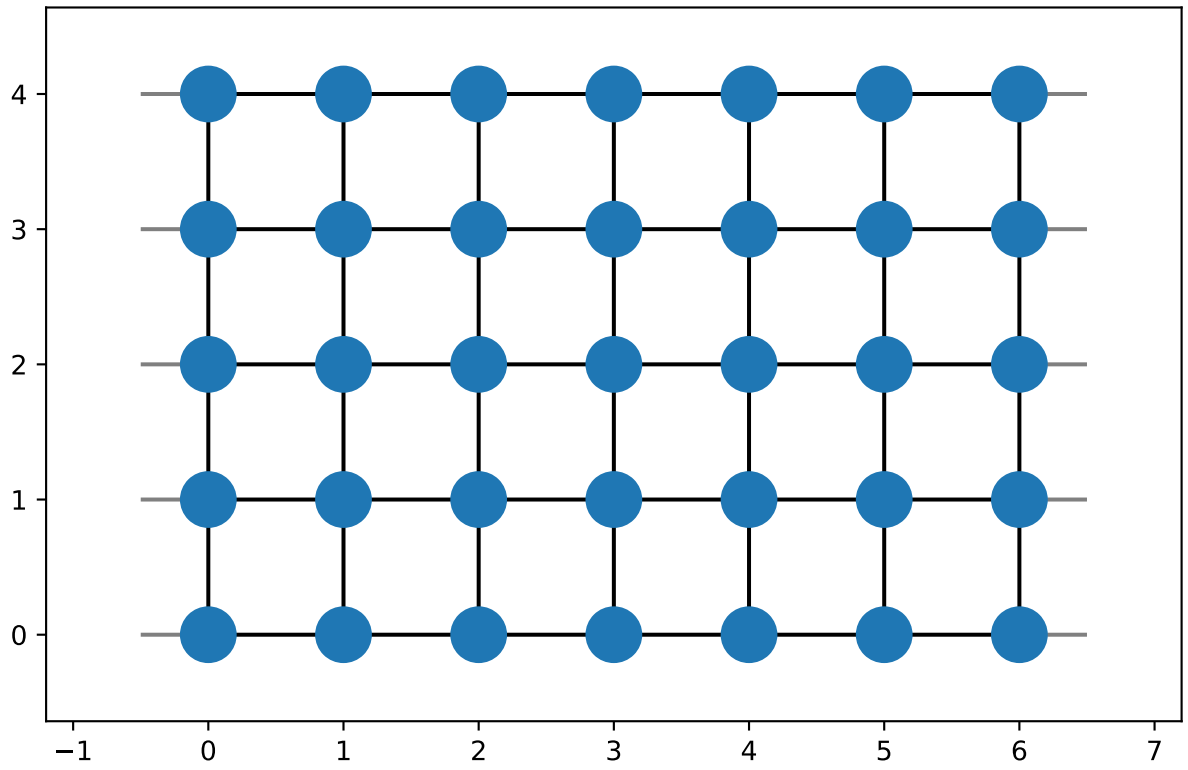
```
>>> latt = lp.Lattice.hexagonal()
>>> latt.add_atom()
>>> latt.add_connections()
>>> s = lp.Circle((0, 0), radius=10)
>>> latt.build(s, primitive=True)
>>> ax = latt.plot()
>>> s.plot(ax)
>>> plt.show()
```


10
5
0
−5
−10
−10.0
−7.5
−5.0
−2.5
0.0
2.5
5.0
7.5
10.0

3.3.2 Periodic boundary conditions

After a finite size lattice model has been buildt periodic boundary conditions can be configured by specifying the axis of the periodic boundary conditions. The periodic boundary conditions can be set up for each axes individually, for example

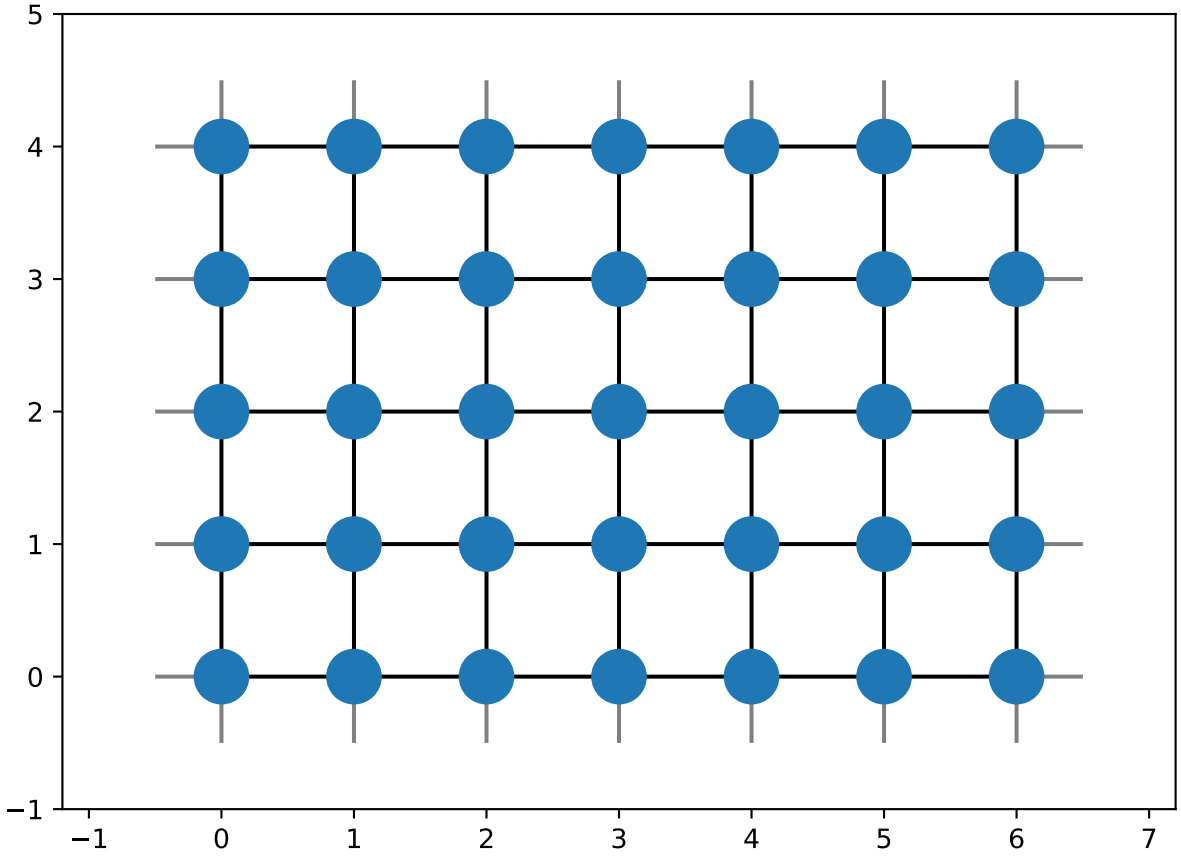
```
>>> latt = lp.simple_square()
>>> latt.build((6, 4))
>>> latt.set_periodic(0)
>>> latt.plot()
>>> plt.show()
```



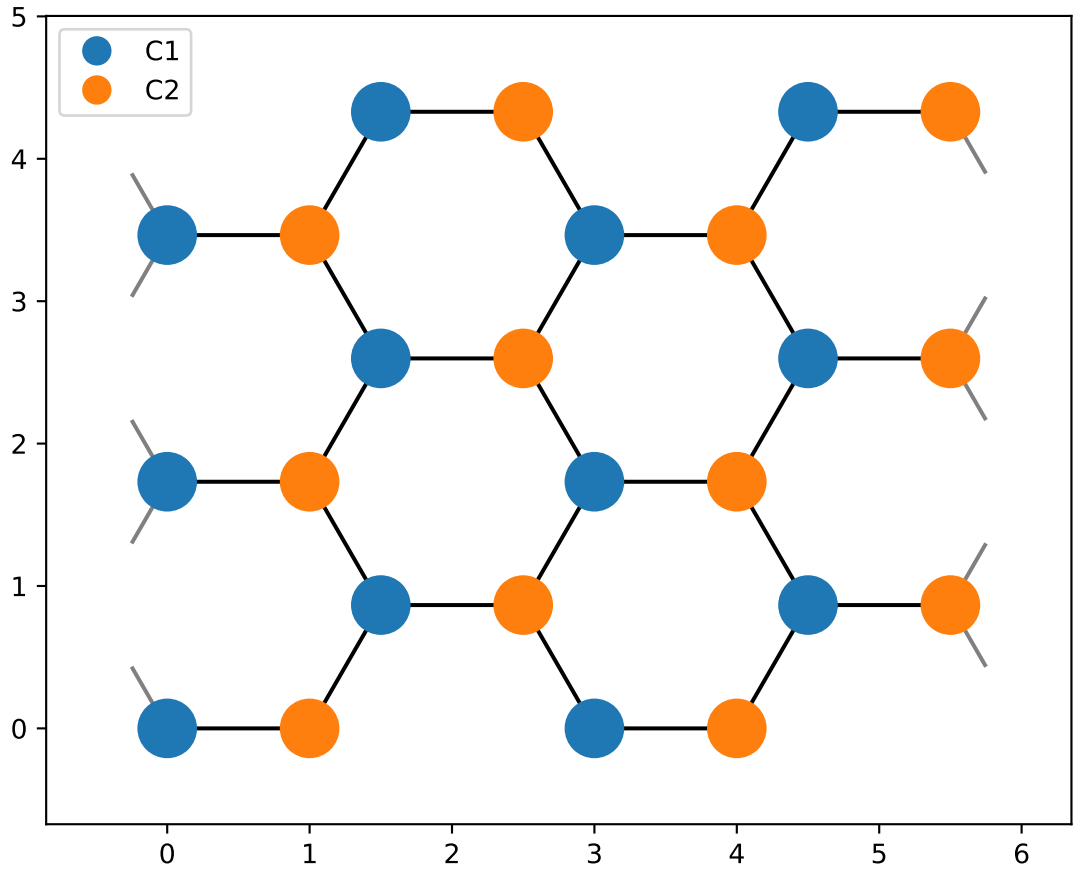
or for multiple axes at once:

```
>>> latt = lp.simple_square()
>>> latt.build((6, 4))
>>> latt.set_periodic([0, 1])
>>> latt.plot()
>>> plt.show()
```

The periodic boundary conditions are computed in the same coordinate system chosen for building the model. If `primitive=False`, i.e. world coordinates, the box around the buildt lattice is repeated periodically:


5
4
3
2
1
0
−1
−1
0
1
2
3
4
5
6
7

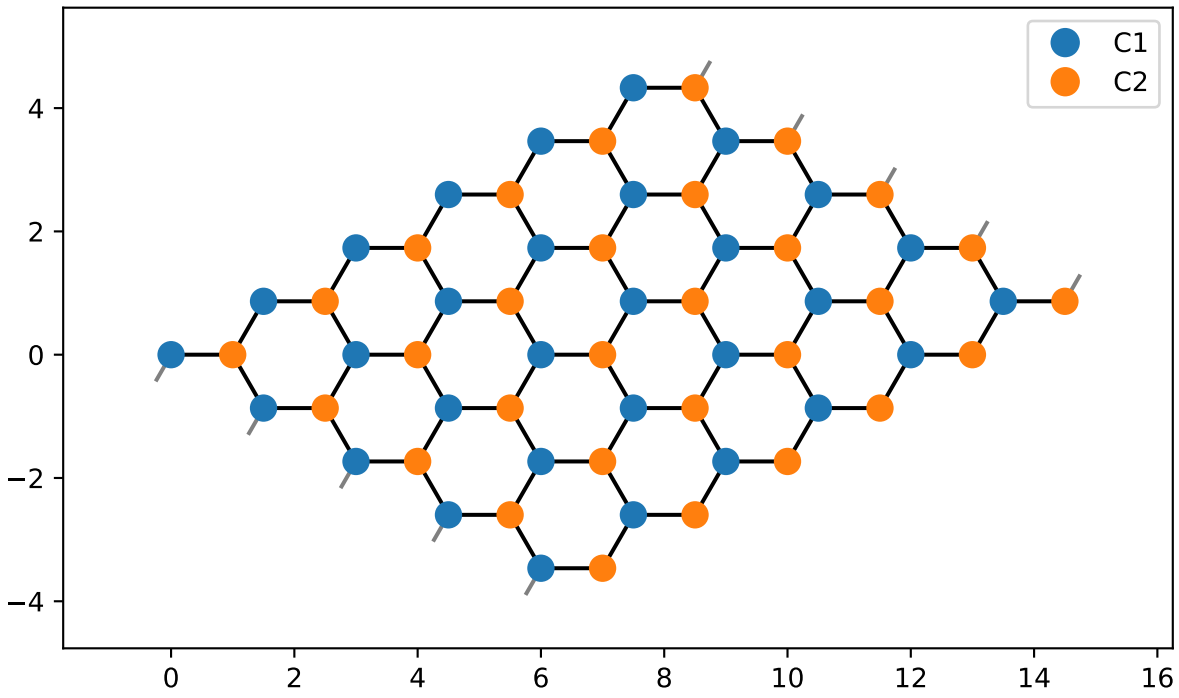
```
>>> latt = lp.graphene()
>>> latt.build((5.5, 4.5))
>>> latt.set_periodic(0)
>>> latt.plot()
>>> plt.show()
```



Here, the periodic boundary conditions again are set up along the x-axis, even though the basis vectors of the hexagonal lattice define a new basis. In the coordinate system of the lattice the periodic boundary contitions are set up along the basis vectors:

```
>>> latt = lp.graphene()
>>> latt.build((5.5, 4.5), primitive=True)
>>> latt.set_periodic(0)
>>> latt.plot()
>>> plt.show()
```

Warning: The `set_periodic` method assumes the lattice is build such that periodic boundary condtitions are possible. This is especially important if a lattice with multiple atoms in the unit cell is used. To correctly connect both sides of the lattice it has to be ensured that each cell in the lattice is fully contained. If, for example, the last unit cell in the x-direction is cut off in the middle no perdiodic boundary conditions will be computed since the distance between the two edges is larger than the other distances in the lattice. A future version will check if this

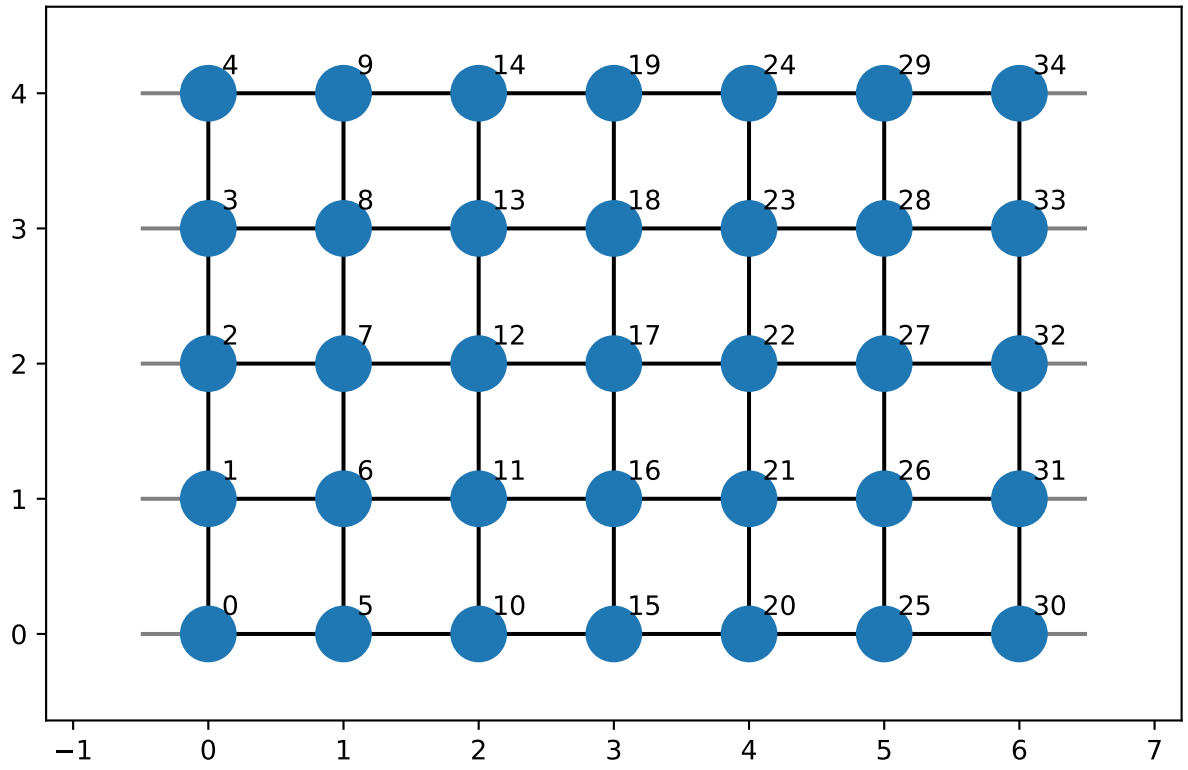

C1
C2
4
2
0
−2
−4
0
2
4
6
8
10
12
14
16

requirement is fulfilled, but until now the user is responsible for the correct configuration.

3.3.3 Position and neighbor data

After building the lattice and optionally setting periodic boundary conditions the information of the buildt lattice can be accessed. The data of the lattice model then can be accessed by a simple index `i`. The syntax is the same as before, just without the `get_` prefix. In order to find the right index, the `plot` method also supports showing the cooresponding super indices of the lattice sites:

```
>>> latt = lp.simple_square()
>>> latt.build((6, 4))
>>> latt.set_periodic(0)
>>> latt.plot(show_indices=True)
>>> plt.show()
```



The positions of the sites in the model can now be accessed via the super index `i`:

```
>>> latt.position(2)
[0. 2.]
```

Similarly, the neighbors can be found via

```
>>> latt.neighbors(2, distidx=0)
[3 1 7 32]
```

The nearest neighbors also can be found with the helper method

```
>>> latt.nearest_neighbors(2)
[3 1 7 32]
```

The position and neighbor data of the finite lattice model is stored in the `LatticeData` object, wich can be accessed via the `data` attribute. Additionally, the positions and (lattice) indices of the model can be directly fetched, for example

```
>>> latt.positions
[[0. 0.]
 [0. 1.]
 ...
 [6. 3.]
 [6. 4.]]
```

3.3.4 Data map

The lattice model makes it is easy to construct the (tight-binding) Hamiltonian of a non-interacting model:

```
>>> latt = simple_chain(a=1.0)
>>> latt.build(shape=4)
>>> n = latt.num_sites
>>> eps, t = 0., 1.
>>> ham = np.zeros((n, n))
>>> for i in range(n):
...     ham[i, i] = eps
...     for j in latt.nearest_neighbors(i):
...         ham[i, j] = t
>>> ham
[[0. 1. 0. 0. 0.]
 [1. 0. 1. 0. 0.]
 [0. 1. 0. 1. 0.]
 [0. 0. 1. 0. 1.]
 [0. 0. 0. 1. 0.]]
```

Since we loop over all sites of the lattice the construction of the hamiltonian is slow. An alternative way of mapping the lattice data to the hamiltonian is using the *DataMap* object returned by the *map()* method of the lattice data. This stores the atom-types, neighbor-pairs and corresponding distances of the lattice sites. Using the built-in masks the construction of the hamiltonian-data can be vectorized:

```
>>> from scipy import sparse
>>> eps, t = 0., 1.
>>> dmap = latt.data.map()                # Build datamap
>>> values = np.zeros(dmap.size)          # Initialize array for data of H
>>> values[dmap.onsite(alpha=0)] = eps    # Map onsite-energies to array
>>> values[dmap.hopping(distidx=0)] = t   # Map hopping-energies to array
>>> ham_s = sparse.csr_matrix((values, dmap.indices))
>>> ham_s.toarray()
[[0. 1. 0. 0. 0.]
```

(continues on next page)

(continued from previous page)

```
[1. 0. 1. 0. 0.]
[0. 1. 0. 1. 0.]
[0. 0. 1. 0. 1.]
[0. 0. 0. 1. 0.]]
```

CHAPTER

FOUR

LATTPY

Package for modeling Bravais lattices and finite lattice structures.

4.1 Submodules

lattpy.atom	Objects for representing atoms and the unitcell of a lattice.
lattpy.basis	Basis object for defining the coordinate system and unit cell of a lattice.
lattpy.data	This module contains objects for low-level representation of lattice systems.
lattpy.disptools	Tools for dispersion computation and plotting.
lattpy.lattice	This module contains the main *Lattice* object.
lattpy.shape	Objects for representing the shape of a finite lattice.
lattpy.spatial	Spatial algorithms and data structures.
lattpy.structure	Lattice structure object for defining the atom basis and neighbor connections.
lattpy.utils	Contains miscellaneous utility methods.

lattpy.**simple_chain**(*a=1.0*, *atom=None*, *neighbors=1*)

Creates a 1D lattice with one atom at the origin of the unit cell.

Parameters

a
[float, optional] The lattice constant (length of the basis-vector).

atom
[str or Atom, optional] The atom to add to the lattice. If a string is passed, a new `Atom` instance is created.

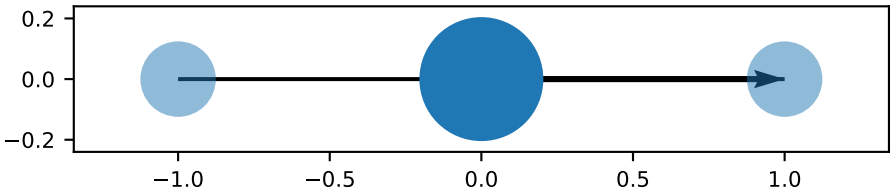
neighbors
[int, optional] The number of neighbor-distance levels, e.g. setting to 1 means only nearest neighbors. The default is nearest neighbors (1).

Returns

latt
[Lattice] The configured lattice instance.

Examples

```
>>> import matplotlib.pyplot as plt
>>> latt = lp.simple_chain()
>>> latt.plot_cell()
>>> plt.show()
```



lattpy.**alternating_chain**(*a=1.0*, *atom1=None*, *atom2=None*, *x0=0.0*, *neighbors=1*)

Creates a 1D lattice with two atoms in the unit cell.

Parameters

a
[float, optional] The lattice constant (length of the basis-vector).

atom1
[str or Atom, optional] The first atom to add to the lattice. If a string is passed, a new `Atom` instance is created.

atom2
[str or Atom, optional] The second atom to add to the lattice. If a string is passed, a new `Atom` instance is created.

x0
[float, optional] The offset of the atom positions in x-direction.

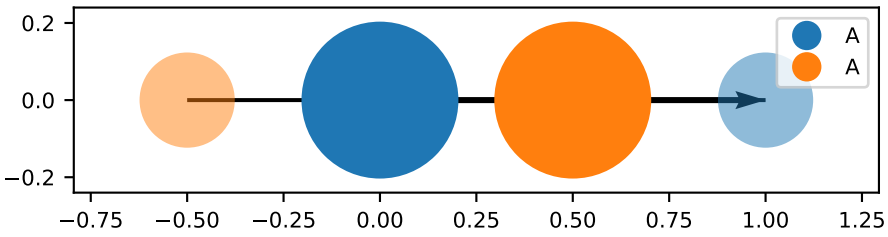
neighbors
[int, optional] The number of neighbor-distance levels, e.g. setting to 1 means only nearest neighbors. The default is nearest neighbors (1).

Returns

latt
[Lattice] The configured lattice instance.

Examples

```
>>> import matplotlib.pyplot as plt
>>> latt = lp.alternating_chain()
>>> latt.plot_cell()
>>> plt.show()
```



lattpy.**simple_square**(*a=1.0*, *atom=None*, *neighbors=1*)

Creates a square lattice with one atom at the origin of the unit cell.

Parameters

a
[float, optional] The lattice constant (length of the basis-vector).

atom
[str or Atom, optional] The atom to add to the lattice. If a string is passed, a new `Atom` instance is created.

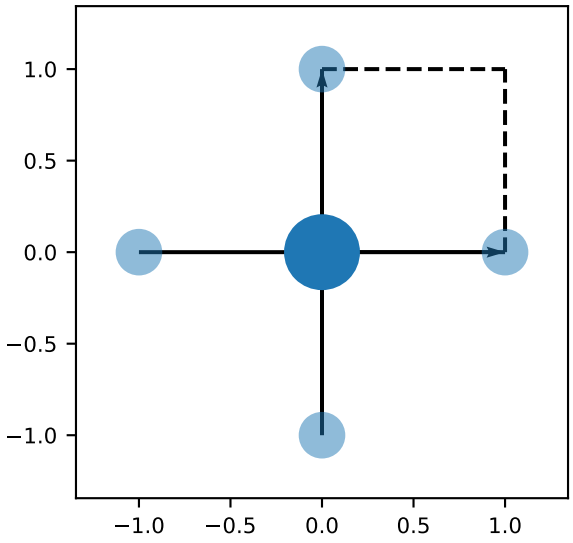
neighbors
[int, optional] The number of neighbor-distance levels, e.g. setting to 1 means only nearest neighbors. The default is nearest neighbors (1).

Returns

latt
[Lattice] The configured lattice instance.

Examples

```
>>> import matplotlib.pyplot as plt
>>> latt = lp.simple_square()
>>> latt.plot_cell()
>>> plt.show()
```



lattpy.**simple_rectangular**(*a1=1.5*, *a2=1.0*, *atom=None*, *neighbors=2*)

Creates a rectangular lattice with one atom at the origin of the unit cell.

Parameters

a1
[float, optional] The lattice constant in the x-direction.

a2
[float, optional] The lattice constant in the y-direction.

atom
[str or Atom, optional] The atom to add to the lattice. If a string is passed, a new `Atom` instance is created.

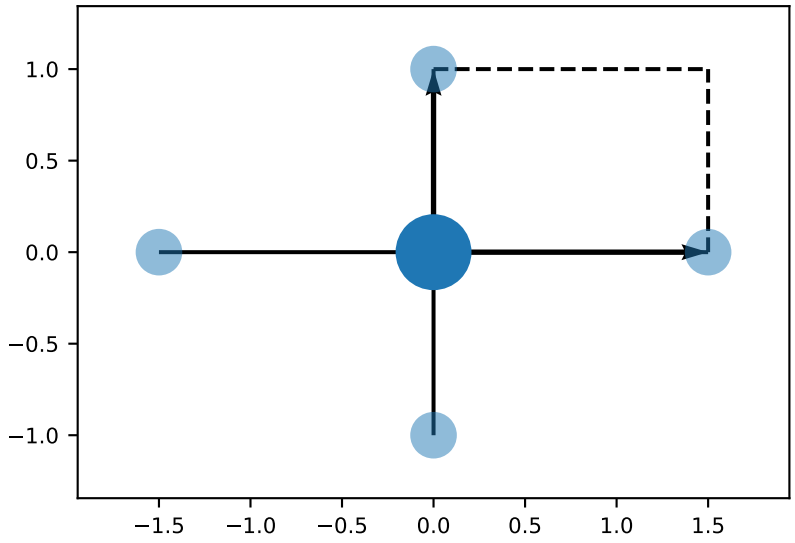
neighbors
[int, optional] The number of neighbor-distance levels, e.g. setting to 1 means only nearest neighbors. The default is nearest neighbors (2).

Returns

latt
[Lattice] The configured lattice instance.

Examples

```
>>> import matplotlib.pyplot as plt
>>> latt = lp.simple_rectangular()
>>> latt.plot_cell()
>>> plt.show()
```



lattpy.**simple_hexagonal**(*a=1.0*, *atom=None*, *neighbors=1*)

Creates a hexagonal lattice with one atom at the origin of the unit cell.

Parameters

a
[float, optional] The lattice constant (length of the basis-vector).

atom
[str or Atom, optional] The atom to add to the lattice. If a string is passed, a new `Atom` instance is created.

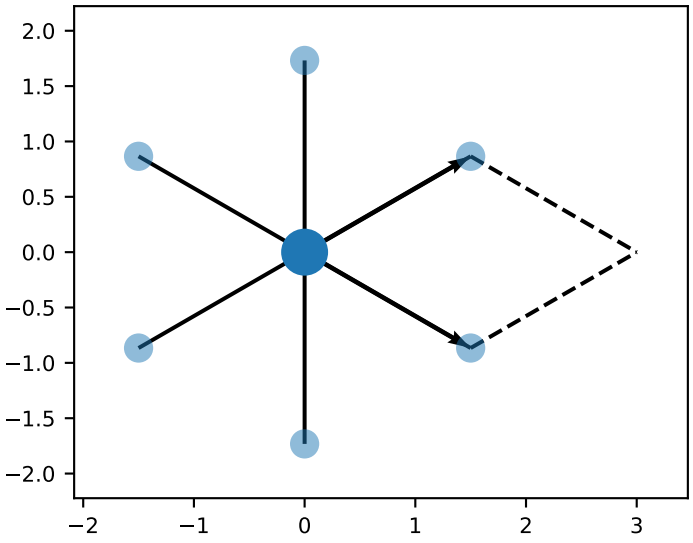
neighbors
[int, optional] The number of neighbor-distance levels, e.g. setting to 1 means only nearest neighbors. The default is nearest neighbors (1).

Returns

latt
[Lattice] The configured lattice instance.

Examples

```
>>> import matplotlib.pyplot as plt
>>> latt = lp.simple_hexagonal()
>>> latt.plot_cell()
>>> plt.show()
```



lattpy.**honeycomb**(*a=1.0*, *atom=None*)

Creates a honeycomb lattice with two identical atoms in the unit cell.

Parameters

a
: [float, optional] The lattice constant (length of the basis-vector).

atom
: [str or Atom, optional] The atom to add to the lattice. If a string is passed, a new `Atom` instance is created.

Returns

latt
: [Lattice] The configured lattice instance.

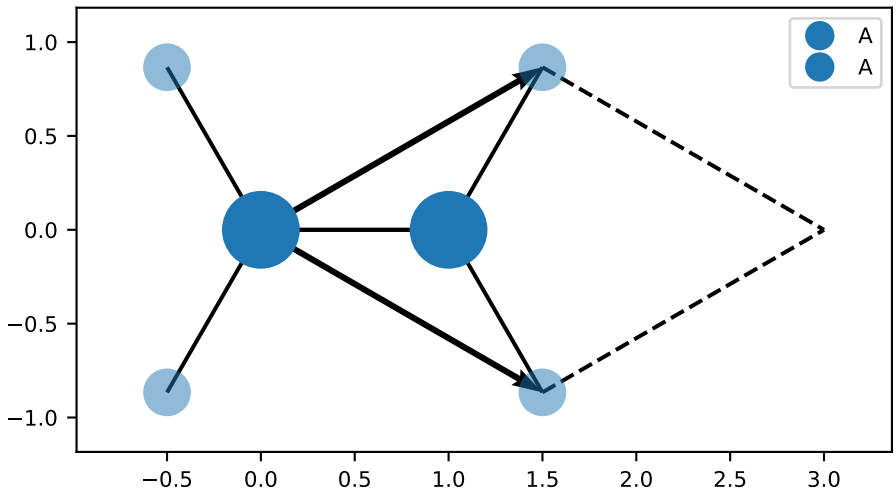
Examples

```
>>> import matplotlib.pyplot as plt
>>> latt = lp.honeycomb()
>>> latt.plot_cell()
>>> plt.show()
```

lattpy.**graphene**(*a=1.0*)

Creates a hexagonal lattice with two atoms in the unit cell.

Parameters



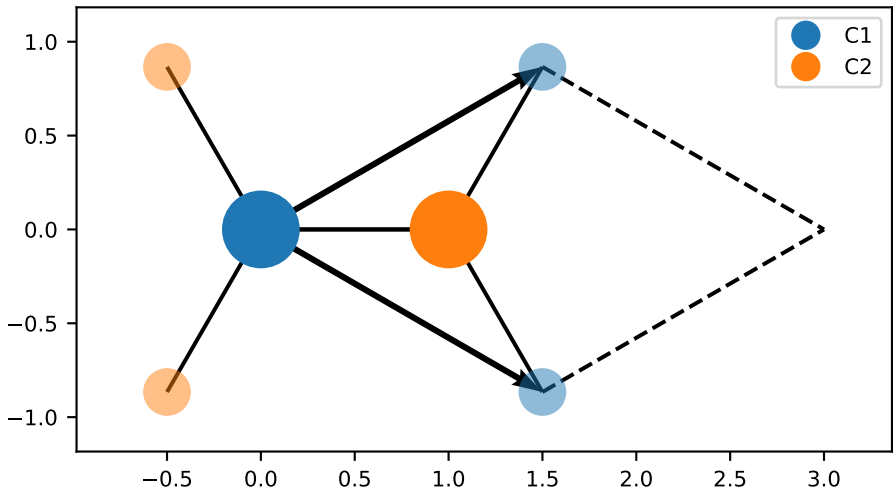
a
: [float, optional] The lattice constant (length of the basis-vectors).

Returns

latt
: [Lattice] The configured lattice instance.

Examples

```
>>> import matplotlib.pyplot as plt
>>> latt = lp.graphene()
>>> latt.plot_cell()
>>> plt.show()
```



lattpy.**simple_cubic**(*a=1.0*, *atom=None*, *neighbors=1*)

Creates a cubic lattice with one atom at the origin of the unit cell.

Parameters

a
[float, optional] The lattice constant (length of the basis-vector).

atom
[str or Atom, optional] The atom to add to the lattice. If a string is passed, a new `Atom` instance is created.

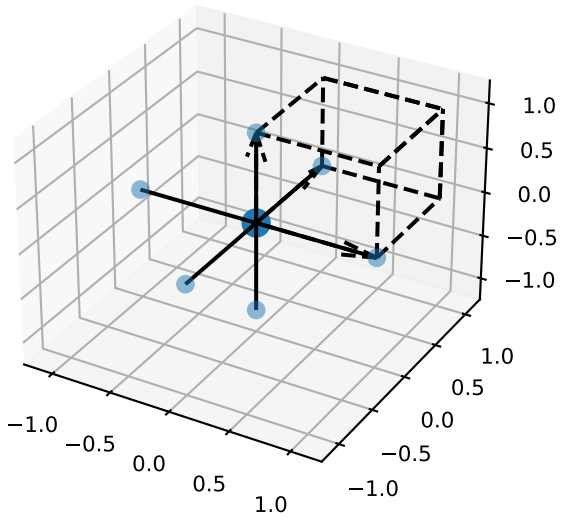
neighbors
[int, optional] The number of neighbor-distance levels, e.g. setting to 1 means only nearest neighbors. The default is nearest neighbors (1).

Returns

latt
[Lattice] The configured lattice instance.

Examples

```
>>> import matplotlib.pyplot as plt
>>> latt = lp.simple_cubic()
>>> latt.plot_cell()
>>> plt.show()
```



lattpy.**nacl_structure**(*a=1.0*, *atom1='Na'*, *atom2='Cl'*, *neighbors=1*)

Creates a NaCl lattice structure.

Parameters

a
[float, optional] The lattice constant (length of the basis-vector).

atom1
[str or Atom, optional] The first atom to add to the lattice. If a string is passed, a new `Atom` instance is created. The default name is *Na*.

atom2
[str or Atom, optional] The second atom to add to the lattice. If a string is passed, a new `Atom` instance is created.. The default name is *Cl*.

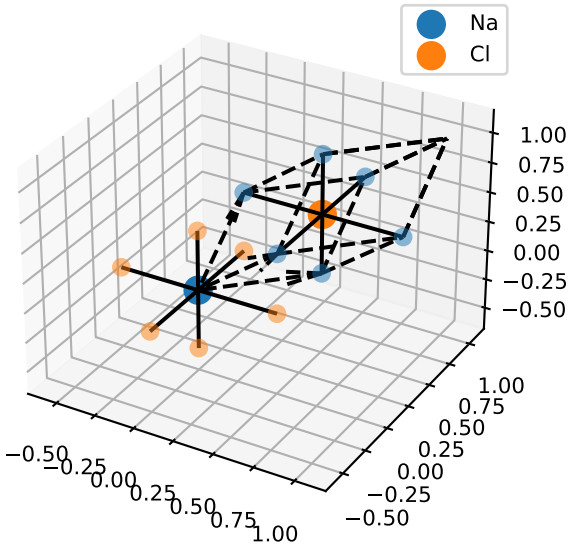
neighbors
[int, optional] The number of neighbor-distance levels, e.g. setting to 1 means only nearest neighbors. The default is nearest neighbors (1).

Returns

latt
[Lattice] The configured lattice instance.

Examples

```
>>> import matplotlib.pyplot as plt
>>> latt = lp.nacl_structure()
>>> latt.plot_cell()
>>> plt.show()
```



lattpy.**finite_hypercubic**(*s*, *a=1.0*, *atom=None*, *neighbors=1*, *primitive=True*, *periodic=None*)

Creates a d-dimensional finite lattice model with one atom in the unit cell.

Parameters

s
[float or Sequence[float] or AbstractShape] The shape of the finite lattice. This also defines the dimensionality.

a
[float, optional] The lattice constant (length of the basis-vectors).

atom
[str or Atom, optional] The atom to add to the lattice. If a string is passed, a new `Atom` instance is created.

neighbors
[int, optional] The number of neighbor-distance levels, e.g. setting to 1 means only nearest neighbors. The default is nearest neighbors (1).

primitive
[bool, optional] If True the shape will be multiplied by the cell size of the model. The default is True.

periodic
[bool or int or (N,) array_like] One or multiple axises to apply the periodic boundary conditions. If the axis is `None` no perodic boundary conditions will be set.

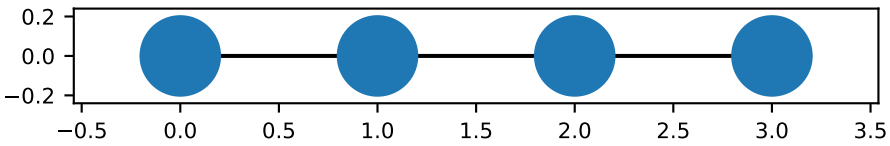
Returns

latt
[Lattice] The configured lattice instance.

Examples

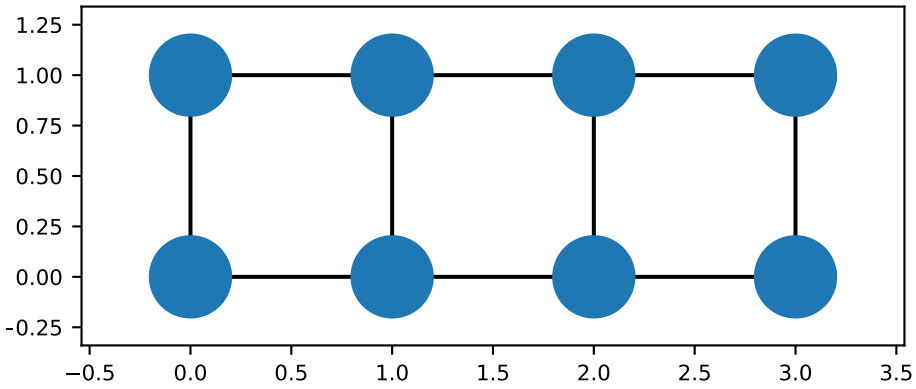
Simple chain:

```
>>> import matplotlib.pyplot as plt
>>> latt = lp.finite_hypercubic(4)
>>> latt.plot()
>>> plt.show()
```



Simple square:

```
>>> import matplotlib.pyplot as plt
>>> latt = lp.finite_hypercubic((4, 2))
>>> latt.plot()
>>> plt.show()
```


1.25
1.00
0.75
0.50
0.25
0.00
-0.25
−0.5
0.0
0.5
1.0
1.5
2.0
2.5
3.0
3.5

CHAPTER

FIVE

LATTPY.ATOM

Objects for representing atoms and the unitcell of a lattice.

class lattpy.atom.**Atom**(*name=None*, *radius=0.2*, *color=None*, *weight=1.0*, ***kwargs*)

Bases: MutableMapping

Object representing an atom of a Bravais lattice.

Parameters

name
[str, optional] The name of the atom. The default is 'A'.

radius
[float, optional] The radius of the atom in real space.

color
[str or float or array_like, optional] The color used to visualize the atom.

weight
[float, optional] The weight of the atom.

****kwargs**
Additional attributes of the atom.

property id

The id of the atom.

property index

Return the index of the Atom instance.

property name

Return the name of the Atom instance.

property weight

Return the weight or the Atom instance.

dict()

Returns the data of the Atom instance as a dictionary.

copy()

Creates a deep copy of the Atom instance.

get(*k*[, *d*]) → D[k] if k in D, else d. d defaults to None.

is_identical(*other*)

Checks if the other Atom is identical to this one.

CHAPTER

SIX

LATTPY.BASIS

Basis object for defining the coordinate system and unit cell of a lattice.

class lattpy.basis.**LatticeBasis**(*basis*, ***kwargs*)

Bases: object

Lattice basis for representing the coordinate system and unit cell of a lattice.

The `LatticeBasis` object is the core of any lattice model. It defines the basis vectors and subsequently the coordinate system of the lattice and provides the necessary basis transformations between the world and lattice coordinate system.

Parameters

basis: array_like or float or LatticeBasis
The primitive basis vectors that define the unit cell of the lattice. If a `LatticeBasis` instance is passed it is copied and used as the new basis.

****kwargs**
Key-word arguments. Used only when subclassing `LatticeBasis`.

Examples

```
>>> import lattpy as lp
>>> import matplotlib.pyplot as plt
>>> basis = lp.LatticeBasis.square()
>>> _ = basis.plot_basis()
>>> plt.show()
```

Attributes

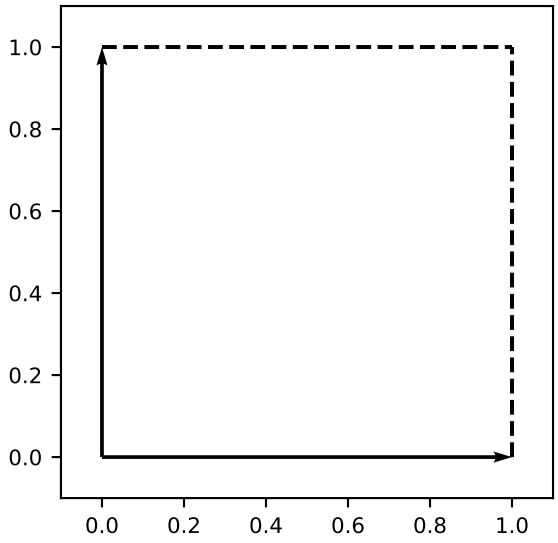
dim
int: The dimension of the lattice.

vectors
np.ndarray: Array containing the basis vectors as rows.

vectors3d
np.ndarray: The basis vectors expanded to three dimensions.

norms
np.ndarray: Lengths of the basis vectors.

cell_size
np.ndarray: The shape of the box spawned by the basis vectors.



cell_volume
float: The volume of the unit cell defined by the basis vectors.

Methods

chain([a])	Initializes a one-dimensional lattice.
square([a])	Initializes a 2D lattice with square basis vectors.
rectangular([a1, a2])	Initializes a 2D lattice with rectangular basis vectors.
oblique(alpha[, a1, a2])	Initializes a 2D lattice with oblique basis vectors.
hexagonal([a])	Initializes a 2D lattice with hexagonal basis vectors.
hexagonal3d([a, az])	Initializes a 3D lattice with hexagonal basis vectors.
sc([a])	Initializes a 3D simple cubic lattice.
fcc([a])	Initializes a 3D face centered cubic lattice.
bcc([a])	Initializes a 3D body centered cubic lattice.
itransform(world_coords)	Transform the world coords `(x, y, ...)` into the basis coords `(n, m, ...)`.
transform(basis_coords)	Transform the basis-coords `(n, m, ...)` into the world coords `(x, y, ...)`.
itranslate(x)	Returns the translation vector and atom position of the given position.
translate(nvec[, r])	Translates the given postion vector `r` by the translation vector `n`.
is_reciprocal(vecs[, tol])	Checks if the given vectors are reciprocal to the lattice vectors.
reciprocal_vectors([tol, check])	Computes the reciprocal basis vectors of the bravais lattice.
reciprocal_lattice([min_negative])	Creates the lattice in reciprocal space.
get_neighbor_cells([distidx, ...])	Find all neighboring unit cells of the unit cell at the origin.
wigner_seitz_cell()	Computes the Wigner-Seitz cell of the lattice structure.
brillouin_zone([min_negative])	Computes the first Brillouin-zone of the lattice structure.
plot_basis([lw, ls, margins, grid, ...])	Plot the lattice basis.

RVEC_TOLERANCE = 1e-06

classmethod chain(*a=1.0*, ***kwargs*)

Initializes a one-dimensional lattice.

classmethod square(*a=1.0*, ***kwargs*)

Initializes a 2D lattice with square basis vectors.

classmethod rectangular(*a1=1.0*, *a2=1.0*, ***kwargs*)

Initializes a 2D lattice with rectangular basis vectors.

classmethod oblique(*alpha*, *a1=1.0*, *a2=1.0*, ***kwargs*)

Initializes a 2D lattice with oblique basis vectors.

classmethod hexagonal(*a=1.0*, ***kwargs*)

Initializes a 2D lattice with hexagonal basis vectors.

classmethod hexagonal3d(*a=1.0*, *az=1.0*, ***kwargs*)

Initializes a 3D lattice with hexagonal basis vectors.

classmethod sc(*a=1.0*, ***kwargs*)

Initializes a 3D simple cubic lattice.

classmethod fcc(*a=1.0*, ***kwargs*)

Initializes a 3D face centered cubic lattice.

classmethod bcc(*a=1.0*, ***kwargs*)

Initializes a 3D body centered cubic lattice.

classmethod hypercubic(*dim*, *a=1.0*, ***kwargs*)

Creates a d-dimensional cubic lattice.

property dim

int: The dimension of the lattice.

property vectors

np.ndarray: Array containing the basis vectors as rows.

property vectors3d

np.ndarray: The basis vectors expanded to three dimensions.

property norms

np.ndarray: Lengths of the basis vectors.

property cell_size

np.ndarray: The shape of the box spawned by the basis vectors.

property cell_volume

float: The volume of the unit cell defined by the basis vectors.

itransform(*world_coords*)

Transform the world coords `(x, y, ...)` into the basis coords `(n, m, ...)`.

Parameters

world_coords
[(..., N) array_like] The coordinates in the world coordinate system that are transformed into the lattice coordinate system.

Returns

basis_coords
[(..., N) np.ndarray] The coordinates in the lattice coordinate system.

Examples

Construct a lattice with basis vectors $a_1 = (2, 0)$ and $a_2 = (0, 1)$:

```
>>> latt = LatticeBasis([[2, 0], [0, 1]])
```

Transform points into the coordinate system of the lattice:

```
>>> latt.itransform([2, 0])
[1. 0.]
```

```
>>> latt.itransform([4, 0])
[2. 0.]
```

```
>>> latt.itransform([0, 1])
[0. 1.]
```

transform(*basis_coords*)

Transform the basis-coords `(n, m, ...)` into the world coords `(x, y, ...)`.

Parameters

basis_coords
[(..., N) array_like] The coordinates in the lattice coordinate system that are transformed into the world coordinate system.

Returns

world_coords
[(..., N) np.ndarray] The coordinates in the cartesian coordinate system.

Examples

Construct a lattice with basis vectors $a_1 = (2, 0)$ and $a_2 = (0, 1)$:

```
>>> latt = LatticeBasis([[2, 0], [0, 1]])
```

Transform points into the world coordinat system:

```
>>> latt.itransform([1, 0])
[2. 0.]
```

```
>>> latt.itransform([2, 0])
[4. 0.]
```

```
>>> latt.itransform([0, 1])
[0. 1.]
```

translate(*nvec*, *r=0.0*)

Translates the given postion vector `r` by the translation vector `n`.

The position is calculated using the translation vector n and the atom position in the unitcell r:

$$R = \sum_i n_i v_i + r$$

Parameters

nvec
[(..., N) array_like] Translation vector in the lattice coordinate system.

r
[(N) array_like, optional] The position in cartesian coordinates. If no vector is passed only the translation is returned.

Returns

r_trans
[(..., N) np.ndarray] The translated position.

Examples

Construct a lattice with basis vectors $a_1 = (2, 0)$ and $a_2 = (0, 1)$:

```
>>> latt = LatticeBasis([[2, 0], [0, 1]])
```

Translate the origin:

```
>>> n = [1, 0]
>>> latt.translate(n)
[2. 0.]
```

Translate a point:

```
>>> p = [0.5, 0.5]
>>> latt.translate(n, p)
[2.5 0.5]
```

Translate a point by multiple translation vectors:

```
>>> p = [0.5, 0.5]
>>> nvecs = [[0, 0], [1, 0], [2, 0]]
>>> latt.translate(nvecs, p)
[[0.5 0.5]
 [2.5 0.5]
 [4.5 0.5]]
```

itranslate(x)

Returns the translation vector and atom position of the given position.

Parameters

x
[(..., N) array_like or float] Position vector in cartesian coordinates.

Returns

nvec
[(..., N) np.ndarray] Translation vector in the lattice basis.

r
[(..., N) np.ndarray, optional] The position in real-space.

Examples

Construct a lattice with basis vectors $a_1 = (2, 0)$ and $a_2 = (0, 1)$:

```
>>> latt = LatticeBasis([[2, 0], [0, 1]])
>>> latt.itranslate([2, 0])
(array([1., 0.]), array([0., 0.]))
```

```
>>> latt.itranslate([2.5, 0.5])
(array([1., 0.]), array([0.5, 0.5]))
```

is_reciprocal(*vecs*, *tol=1e-06*)

Checks if the given vectors are reciprocal to the lattice vectors.

The lattice- and reciprocal vectors a_i and b_i must satisfy the relation

$$a_i \cdot b_i = 2\pi\delta_{ij}$$

To check the given vectors, the difference of each dot-product is compared to 2π with the given tolerance.

Parameters

vecs
[array_like or float] The vectors to check. Must have the same dimension as the lattice.

tol
[float, optional] The tolerance used for checking the result of the dot-products.

Returns

is_reciprocal
[bool] Flag if the vectors are reciprocal to the lattice basis vectors.

reciprocal_vectors(*tol=1e-06*, *check=False*)

Computes the reciprocal basis vectors of the bravais lattice.

The lattice- and reciprocal vectors a_i and b_i must satisfy the relation

$$a_i \cdot b_i = 2\pi\delta_{ij}$$

Parameters

tol
[float, optional] The tolerance used for checking the result of the dot-products.

check
[bool, optional] Check the result and raise an exception if it does not satisfy the definition.

Returns

v_rec
[np.ndarray] The reciprocal basis vectors of the lattice.

Examples

Reciprocal vectors of the square lattice:

```
>>> latt = LatticeBasis(np.eye(2))
>>> latt.reciprocal_vectors()
[[6.28318531 0.        ]
 [0.         6.28318531]]
```

reciprocal_lattice(*min_negative=False*)

Creates the lattice in reciprocal space.

Parameters

min_negative
[bool, optional] If True the reciprocal vectors are scaled such that there are fewer negative elements than positive ones.

Returns

rlatt
[LatticeBasis] The lattice in reciprocal space

See also:

reciprocal_vectors
Constructs the reciprocal vectors used for the reciprocal lattice

Examples

Reciprocal lattice of the square lattice:

```
>>> latt = LatticeBasis(np.eye(2))
>>> rlatt = latt.reciprocal_lattice()
>>> rlatt.vectors
[[6.28318531 0.        ]
 [0.         6.28318531]]
```

`get_neighbor_cells`(*distidx=0*, *include_origin=True*, *comparison=<function isclose>*)

Find all neighboring unit cells of the unit cell at the origin.

Parameters

distidx
[int, default] Index of distance to neighboring cells, default is 0 (nearest neighbors).

include_origin
[bool, optional] If True the origin is included in the set.

comparison
[callable, optional] The method used for comparing distances.

Returns

indices
[np.ndarray] The lattice indeices of the neighboring unit cells.

Examples

```
>>> latt = LatticeBasis(np.eye(2))
>>> latt.get_neighbor_cells(distidx=0, include_origin=False)
[[-1  0]
 [ 0 -1]
 [ 0  1]
 [ 1  0]]
```

`wigner_seitz_cell`()

Computes the Wigner-Seitz cell of the lattice structure.

Returns

ws_cell
[WignerSeitzCell] The Wigner-Seitz cell of the lattice.

`brillouin_zone`(*min_negative=False*)

Computes the first Brillouin-zone of the lattice structure.

Constructs the Wigner-Seitz cell of the reciprocal lattice

Parameters

min_negative
[bool, optional] If True the reciprocal vectors are scaled such that there are fewer negative elements than positive ones.

Returns

ws_cell
[WignerSeitzCell] The Wigner-Seitz cell of the reciprocal lattice.

`get_cell_superindex`(*index*, *shape*)

Converts a cell index to a super-index for the given shape.

The index of a unit cell is the translation vector.

Parameters

index
[array_like or (N, D) np.ndarray] One or multiple cell indices. If a numpy array is passed the result will be an array of super-indices, otherwise an integer is returned.

shape
[(D,) array_like] The shape for converting the index.

Returns

super_index
[int or (N,) int np.ndarray] The super index. If *index* is a numpy array, *super_index* is an array of super indices, otherwise it is an integer.

`get_cell_index`(*super_index*, *shape*)

Converts a super index to the corresponding cell index for the given shape.

The index of a unit cell is the translation vector.

Parameters

super_index
[int or (N,) int np.ndarray] One or multiple super indices.

shape
[(D,) array_like] The shape for converting the index.

Returns

index
[np.ndarray] The cell index. Of shape (D,) if *super_index* is an integer, otherwise of shape (N, D).

`plot_basis`(*lw=None*, *ls='--'*, *margins=0.1*, *grid=False*, *show_cell=True*, *show_vecs=True*, *adjustable='box'*, *ax=None*, *show=False*)

Plot the lattice basis.

Parameters

lw
[float, optional] The line width used for plotting the unit cell outlines.

ls
[str, optional] The line style used for plotting the unit cell outlines.

margins
[Sequence[float] or float, optional] The margins of the plot.

grid
[bool, optional] If True, draw a grid in the plot.

show_vecs
[bool, optional] If True the first unit-cell is drawn.

show_cell
[bool, optional] If True the outlines of the unit cell are plotted.

adjustable
[None or {'box', 'datalim'}, optional] If not None, this defines which parameter will be adjusted to meet the equal aspect ratio. If 'box', change the physical dimensions of the Axes. If 'datalim', change the x or y data limits. Only applied to 2D plots.

ax
[plt.Axes or plt.Axes3D or None, optional] Parent plot. If None, a new plot is initialized.

show
[bool, optional] If True, show the resulting plot.

CHAPTER

SEVEN

LATTPY.DATA

This module contains objects for low-level representation of lattice systems.

class lattpy.data.**DataMap**(*alphas*, *pairs*, *distindices*)

Bases: object

Object for low-level representation of sites and site-pairs.

Parameters

alphas
[(N) np.ndarray] The atom indices of the sites.

pairs
[(M, 2) np.ndarray] An array of index-pairs of the lattice sites.

distindices
[(M) np.ndarray] The distance-indices for each pair

Notes

This object is not intended to be instantiated by the user. Use the `map` method of `latticeData` or the `dmap` method of the main ``Lattice```object.

property size

The number of the data points (sites + neighbor pairs)

property indices

The indices of the data points as rows and collumns.

property rows

The rows of the data points.

property cols

The columns of the data points.

property nbytes

The number of bytes stored in the datamap.

indices_indptr()

Constructs the *indices* and *indptr* arrays used for CSR/BSR matrices.

CSR/BSR sparse matrix format: The block column indices for row i are stored in `indices[indptr[i]:indptr[i+1]]` and their corresponding block values are stored in `data[indptr[i]: indptr[i+1]]`.

Returns

indices
[(N,) np.ndarray] CSR/BSR format index array.

indptr
[(M,) np.ndarray] CSR/BSR format index pointer array.

See also:

`scipy.sparse.csr_matrix`
Compressed Sparse Row matrix

`scipy.sparse.bsr_matrix`
Block Sparse Row matrix

`onsite`(*alpha=None*)

Creates a mask of the site elements for the atoms with the given index.

Parameters

alpha
[int, optional] Index of the atom in the unitcell. If *None* a mask for all atoms is returned. The default is *None*.

Returns

mask
[np.ndarray]

`hopping`(*distidx=None*)

Creates a mask of the site-pair elements with the given distance index.

Parameters

distidx
[int, optional] Index of distance to neighboring sites, default is 0 (nearest neighbors). If *None* a mask for neighbor-connections is returned. The default is *None*.

Returns

mask
[np.ndarray]

`zeros`(*norb=None*, *dtype=None*)

Creates an empty data-arary.

Parameters

norb
[int, optional] The number of orbitals M. By default, only a single orbital is used.

dtype
[int or str or np.dtype, optional] The data type of the array. By default, it is set automatically.

Returns

data
[np.ndarray] The empty data array. If a single orbital is used the array is one-dimensional, otherwise the array has the shape (N, M, M).

build_csr(*data*, *shape=None*, *dtype=None*)

Constructs a CSR matrix using the given data and the indices of the data map.

Parameters

data
[(N,) np.ndarray] The input data for constructing the CSR matrix. The data array should be filled using the built-in mask methods of the *DataMap* class.

shape
[tuple, optional] The shape of the resulting matrix. If None (default), the shape is inferred from the data and indices of the matrix.

dtype
[int or str or np.dtype, optional] The data type of the matrix. By default, it is set automatically.

Returns

sparse_mat
[(M, M) scipy.sparse.csr.csr_matrix] The sparse matrix representing the lattice data.

build_bsr(*data*, *shape=None*, *dtype=None*)

Constructs a BSR matrix using the given data and the indices of the data map.

Parameters

data
[(N, B, B) np.ndarray] The input data for constructing the BSR matrix. The array must be 3-dimensional, where the first axis N represents the number of blocks and the last two axis B the size of each block. The data array should be filled using the built-in mask methods of the *DataMap* class.

shape
[tuple, optional] The shape of the resulting matrix. If None (default), the shape is inferred from the data and indices of the matrix.

dtype
[int or str or np.dtype, optional] The data type of hte matrix. By default, it is set automatically.

Returns

sparse_mat
[(M, M) scipy.sparse.csr.bsr_matrix] The sparse matrix representing the lattice data.

class lattpy.data.**LatticeData**(**args*)

Bases: object

Object for storing the indices, positions and neighbors of lattice sites.

Parameters

indices
[array_like of iterable of int] The lattice indices of the sites.

positions
[array_like of iterable of int] The positions of the sites.

neighbors
[iterable of iterable of of int] The neighbors of the sites.

distances
[iterabe of iterable of int] The distances of the neighbors.

`property dim`
The dimension of the data points.

`property num_sites`
The number of sites stored.

`property num_distances`
The number of distances of the neighbor data.

`property nbytes`
Returns the number of bytes stored.

`copy()`
Creates a deep copy of the instance.

`reset()`
Resets the *LatticeData* instance.

`set(`*indices*, *positions*, *neighbors*, *distances*`)`
Sets the data of the *LatticeData* instance.

Parameters

indices
[(N, D+1) np.ndarray] The lattice indices of the sites.

positions
[(N, D) np.ndarray] The positions of the sites.

neighbors
[(N, M) np.ndarray] The neighbors of the sites.

distances
[(N, M) iterabe of iterable of int] The distances of the neighbors.

`remove(`*sites*`)`

`get_limits()`
Computes the geometric limits of the positions of the stored sites.

Returns

limits
[np.ndarray] The minimum and maximum value for each axis of the position data.

`get_index_limits()`
Computes the geometric limits of the lattice indices of the stored sites.

Returns

limits: np.ndarray
The minimum and maximum value for each axis of the lattice indices.

`get_cell_limits()`
Computes the geometric limits of the lattice cells of the stored sites.

Returns

limits: np.ndarray
The minimum and maximum value for each axis of the translation indices.

`get_translation_limits()`

Computes the geometric limits of the translation vectors of the stored sites.

Returns

limits
[np.ndarray] The minimum and maximum value for each axis of the lattice indices.

`neighbor_mask(site, distidx=None, periodic=None, unique=False)`

Creates a mask for the valid neighbors of a specific site.

Parameters

site
[int] The index of the site.

distidx
[int, optional] The index of the distance. If *None* the data for all distances is returned. The default is *None* (all neighbors).

periodic
[bool, optional] Periodic neighbor flag. If *None* the data for all neighbors is returned. If a bool is passed either the periodic or non-periodic neighbors are masked. The default is *None* (all neighbors).

unique
[bool, optional] If 'True', each unique pair is only return once. The defualt is *False*.

Returns

mask
[np.ndarray]

`set_periodic(indices, distances, nvecs, axes)`

Adds periodic neighbors to the invalid slots of the neighbor data

Parameters

indices
[dict] Indices of the periodic neighbors. All dictionaries have the site as key and a list of `np.ndarray` as values.

distances
[dict] The distances of the periodic neighbors.

nvecs
[dict] The translation vectors of the periodic neighbors.

axes
[dict] Index of the translation axis of the periodic neighbors.

`sort(ax=None, indices=None, reverse=False)`

`remove_periodic()`

`sort_neighbors()`

`add_neighbors(site, neighbors, distances)`

`append(*args, copy=False)`

`get_positions`(*alpha*)

Returns the atom positions of a sublattice.

`get_neighbors`(*site*, *distidx=None*, *periodic=None*, *unique=False*)

Returns the neighbors of a lattice site.

See the *neighbor_mask*-method for more information on parameters

Returns

neighbors
[np.ndarray] The indices of the neighbors.

`iter_neighbors`(*site*, *unique=False*)

Iterates over the neighbors of all distance levels.

See the *neighbor_mask*-method for more information on parameters

Yields

distidx
[int]

neighbors
[np.ndarray]

`map`()

Builds a map containing the atom-indices, site-pairs and distances.

Returns

datamap
[DataMap]

`site_mask`(*mins=None*, *maxs=None*, *invert=False*)

Creates a mask for the position data of the sites.

Parameters

mins
[sequence or float or None, optional] Optional lower bound for the positions. The default is no lower bound.

maxs
[sequence or float or None, optional] Optional upper bound for the positions. The default is no upper bound.

invert
[bool, optional] If *True*, the mask is inverted. The default is *False*.

Returns

mask
[np.ndarray] The mask containing a boolean value for each site.

`find_sites`(*mins=None*, *maxs=None*, *invert=False*)

Returns the indices of sites inside or outside the given limits.

Parameters

mins
[sequence or float or None, optional] Optional lower bound for the positions. The default is no lower bound.

maxs
[sequence or float or None, optional] Optional upper bound for the positions. The default is no upper bound.

invert
[bool, optional] If *True*, the mask is inverted and the positions outside of the bounds will be returned. The default is *False*.

Returns

indices: np.ndarray
The indices of the masked sites.

`find_outer_sites`(*ax*, *offset*)

Returns the indices of the outer sites along a specific axis.

Parameters

ax
[int] The geometrical axis.

offset
[int] The width of the outer slices.

Returns

indices
[np.ndarray] The indices of the masked sites.

CHAPTER

EIGHT

LATTPY.DISPTOOLS

Tools for dispersion computation and plotting.

lattpy.disptools.**bandpath_subplots**(*ticks*, *labels*, *xlabel='k'*, *ylabel='$E(k)$'*, *grid='both'*)

lattpy.disptools.**plot_dispersion**(*disp*, *labels*, *xlabel='k'*, *ylabel='$E(k)$'*, *grid='both'*, *color=None*, *alpha=0.2*, *lw=1.0*, *scales=None*, *fill=False*, *ax=None*, *show=True*)

lattpy.disptools.**disp_dos_subplots**(*ticks*, *labels*, *xlabel='k'*, *ylabel='$E(k)$'*, *doslabel='$n(E)$'*, *wratio=(3, 1)*, *grid='both'*)

lattpy.disptools.**plot_disp_dos**(*disp*, *dos_data*, *labels*, *xlabel='k'*, *ylabel='E(k)'*, *doslabel='n(E)'*, *wratio=(3, 1)*, *grid='both'*, *color=None*, *fill=True*, *disp_alpha=0.2*, *dos_alpha=0.2*, *lw=1.0*, *scales=None*, *axs=None*, *show=True*)

lattpy.disptools.**plot_bands**(*kgrid*, *bands*, *k_label='k'*, *disp_label='E(k)'*, *grid='both'*, *contour_grid=False*, *bz=None*, *pi_ticks=True*, *ax=None*, *show=True*)

class lattpy.disptools.**DispersionPath**(*dim=0*)

Bases: `object`

Defines a dispersion path between high symmetry (HS) points.

Examples

Define a path using the add-method or preset points. To get the actual points the 'build'-method is called:

```
>>> path = DispersionPath(dim=3).add([0, 0, 0], 'Gamma').x(a=1.0).cycle()
>>> vectors = path.build(n_sect=1000)
```

Attributes

dim
[int]

labels
[list of str]

points
[list of array_like]

n_sect
[int]

classmethod **chain_path**(*a=1.0*)

classmethod **square_path**(*a=1.0*)

classmethod **cubic_path**(*a=1.0*)

property **num_points**

int: Number of HS points in the path

add(*point*, *name=''*)

Adds a new HS point to the path

This method returns the instance for easier path definitions.

Parameters

point: array_like
The coordinates of the HS point. If the dimension of the point is higher than the set dimension the point will be clipped.

name: str, optional
Optional name of the point. If not specified the number of the point is used.

Returns

self: DispersionPath

add_points(*points*, *names=None*)

Adds multiple HS points to the path

Parameters

points: array_like
The coordinates of the HS points.

names: list of str, optional
Optional names of the points. If not specified the number of the point is used.

Returns

self: DispersionPath

cycle()

Adds the first point of the path.

This method returns the instance for easier path definitions.

Returns

self: DispersionPath

gamma()

DispersionPath: Adds the .math:'Gamma=(0, 0, 0)' point to the path

x(*a=1.0*)

DispersionPath: Adds the .math:'X=(pi, 0, 0)' point to the path

m(*a=1.0*)

DispersionPath: Adds the ,math:'M=(pi, pi, 0)' point to the path

r(*a=1.0*)

DispersionPath: Adds the .math:'R=(pi, pi, pi)' point to the path

build(*n_sect=1000*)

Builds the vectors defining the path between the set HS points.

Parameters

n_sect: int, optional
Number of points between each pair of HS points.

Returns

path: (N, D) np.ndarray

get_ticks()

Get the positions of the points of the last buildt path.

Mainly used for setting ticks in plot.

Returns

ticks: (N) np.ndarray
labels: (N) list

edges()

Constructs the edges of the path.

distances()

Computes the distances between the edges of the path.

scales()

Computes the scales of the the edges of the path.

draw(*ax*, *color=None*, *lw=1.0*, ***kwargs*)

subplots(*xlabel='k'*, *ylabel='E(k)'*, *grid='both'*)

Creates an empty matplotlib plot with configured axes for the path.

Parameters

xlabel: str, optional
ylabel: str, optional
grid: str, optional

Returns

fig: plt.Figure
ax: plt.Axis

plot_dispersion(*disp*, *ax=None*, *show=True*, ***kwargs*)

plot_disp_dos(*disp*, *dos*, *axs=None*, *show=True*, ***kwargs*)

CHAPTER

NINE

LATTPY.LATTICE

This module contains the main *Lattice* object.

class lattpy.lattice.**Lattice**(*basis*, ***kwargs*)

Bases: *LatticeStructure*

Main lattice object representing a Bravais lattice model.

Combines the `LatticeBasis` and the `LatticeStructure` class and adds the ability to construct finite lattice models.

Inheritance

Parameters

basis: array_like or float or LatticeBasis
The primitive basis vectors that define the unit cell of the lattice. If a `LatticeBasis` instance is passed it is copied and used as the new basis of the lattice.

****kwargs**
Key-word arguments. Used for quickly configuring a `Lattice` instance. Allowed key-words are:

Properties: atoms: Dictionary containing the atoms to add to the lattice. cons: Dictionary containing the connections to add to the lattice. shape: int or tuple defining the shape of the finite size lattice to build. periodic: int or list defining the periodic axes to set up.

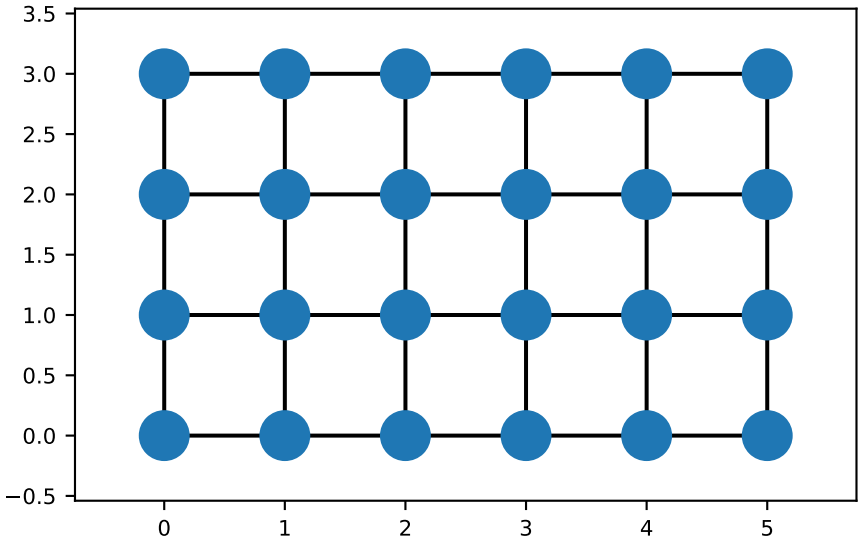
Examples

Two dimensional lattice with one atom in the unit cell and nearest neighbors

```
>>> import lattpy as lp
>>> latt = lp.Lattice(np.eye(2))
>>> latt.add_atom()
>>> latt.add_connections(1)
>>> _ = latt.build((5, 3))
>>> latt
Lattice(dim: 2, num_base: 1, num_neighbors: [4], shape: [5. 3.])
```

Quick-setup of the same lattice:

```
>>> import lattpy as lp
>>> import matplotlib.pyplot as plt
>>> latt = lp.Lattice.square(atoms={(0.0, 0.0): "A"}, cons={("A", "A"): 1})
>>> _ = latt.build((5, 3))
>>> _ = latt.plot()
>>> plt.show()
```



property num_sites

int: Number of sites in lattice data (if lattice has been built).

property num_cells

int: Number of unit-cells in lattice data (if lattice has been built).

property indices

np.ndarray: The lattice indices of the cached lattice data.

property positions

np.ndarray: The lattice positions of the cached lattice data.

volume()

The total volume (number of cells x cell-volume) of the built lattice.

Returns

vol
[float] The volume of the finite lattice structure.

`alpha`(*idx*)

Returns the atom component of the lattice index for a site in the lattice.

Parameters

idx
[int] The super-index of a site in the cached lattice data.

Returns

alpha
[int] The index of the atom in the unit cell.

`atom`(*idx*)

Returns the atom of a given site in the cached lattice data.

Parameters

idx
[int] The super-index of a site in the cached lattice data.

Returns

atom
[Atom]

`position`(*idx*)

Returns the position of a given site in the cached lattice data.

Parameters

idx
[int] The super-index of a site in the cached lattice data.

Returns

pos
[(D,) np.ndarray] The position of the lattice site.

`limits`()

Returns the spatial limits of the lattice model.

Returns

limits
[(2, D) np.ndarray] An array of the limits of the lattice positions. The first axis contains the minimum position and the second the maximum position for all dimensions.

`relative_position`(*fractions*)

Computes a relitve position in the lattice model.

Parameters

fractions
[float or (D,) array_like] The position relatice to the size of the lattice. The center of the lattice is returned for the fractions *[0.5, …, 0.5]*.

Returns

relpos
[(D,) np.ndarray] The relative position in the lattice model.

`center`()

Returns the spatial center of the lattice.

Returns

center
[(D,) np.ndarray] The position of the spatial center of the lattice.

`center_of_gravity`()

Computes the center of gravity of the lattice model.

Returns

center
[(D,) np.ndarray] The center of gravity of the lattice model.

Notes

Requires that the *mass* attribute is set for each atom. If no mass is set the default mass of *1.0* is used.

`superindex_from_pos`(*pos*, *atol=0.0001*)

Returns the super-index of a given position.

Parameters

pos
[(D,) array_like] The position of the site in cartesian coordinates.

atol
[float, optional] The absolute tolerance for comparing positions.

Returns

index
[int or None] The super-index of the site in the cached lattice data.

`superindex_from_index`(*ind*)

Returns the super-index of a site defined by the lattice index.

Parameters

ind
[(D + 1,) array_like] The lattice index `(n_1, ..., n_D, alpha)` of the site.

Returns

index
[int or None] The super-index of the site in the cached lattice data.

`neighbors`(*site*, *distidx=None*, *unique=False*)

Returns the neighours of a given site in the cached lattice data.

Parameters

site
[int] The super-index of a site in the cached lattice data.

distidx
[int, optional] Index of distance to the neighbors, default is 0 (nearest neighbors).

unique
[bool, optional] If True, each unique pair is only returned once.

Returns

indices
[np.ndarray of int] The super-indices of the neighbors.

`nearest_neighbors`(*idx*, *unique=False*)

Returns the nearest neighors of a given site in the cached lattice data.

Parameters

idx
[int] The super-index of a site in the cached lattice data.

unique
[bool, optional] If True, each unique pair is only return once.

Returns

indices
[(N,) np.ndarray of int] The super-indices of the nearest neighbors.

`iter_neighbors`(*site*, *unique=False*)

Iterates over the neighbors of all distances of a given site.

Parameters

site
[int] The super-index of a site in the cached lattice data.

unique
[bool, optional] If True, each unique pair is only return once.

Yields

distidx
[int] The distance index of the neighbor indices.

neighbors
[(N,) np.ndarray] The super-indices of the neighbors for the corresponding distance level.

`check_neighbors`(*idx0*, *idx1*)

Checks if two sites are neighbors and returns the distance level if they are.

Parameters

idx0
[int] The first super-index of a site in the cached lattice data.

idx1
[int] The second super-index of a site in the cached lattice data.

Returns

distidx
[int or None] The distance index of the two sites if they are neighbors.

`build`(*shape*, *primitive=False*, *pos=None*, *check=True*, *min_neighbors=None*, *num_jobs=-1*, *periodic=None*, *callback=None*, *dtype=None*)

Constructs the indices and neighbors of a finite size lattice.

Parameters

shape
[(N,) array_like or float or AbstractShape] Shape of finite size lattice to build.

primitive
[bool, optional] If True the shape will be multiplied by the cell size of the model. The default is False.

pos
[(N,) array_like or int, optional] Optional position of the section to build. If `None` the origin is used.

check
[bool, optional] If True the positions of the translation vectors are checked and filtered. The default is True. This should only be disabled if filtered later.

min_neighbors
[int, optional] The minimum number of neighbors a site must have. This can be used to remove dangling sites at the edge of the lattice.

num_jobs
[int, optional] Number of jobs to schedule for parallel processing of neighbors. If `-1` is given all processors are used. The default is `-1`.

periodic
[int or array_like, optional] Optional periodic axes to set. See `set_periodic` for mor details.

callback
[callable, optional] The indices and positions are passed as arguments.

dtype
[int or str or np.dtype, optional] Optional data-type for storing the lattice indices. Using a smaller bit-size may help reduce memory usage. By default, the given limits are checked to determine the smallest possible data-type.

Raises

ValueError
Raised if the dimension of the position doesn't match the dimension of the lattice.

NoConnectionsError
Raised if no connections have been set up.

NotAnalyzedError
Raised if the lattice distances and base-neighbors haven't been computed.

`periodic_translation_vectors`(*axes*, *primitive=False*)

Constrcuts all translation vectors for periodic boundary conditions.

Parameters

axes
[int or (N,) array_like] One or multiple axises to compute the translation vectors for.

primitive
[bool, optional] Flag if the specified axes are in cartesian or lattice coordinates. If `True` the passed position will be multiplied with the lattice vectors. The default is `False` (cartesian coordinates).

Returns

nvecs
[list of tuple] The translation vectors for the periodic boundary conditions. The first item of each element is the axis, the second the corresponding translation vector.

`kdtree`(*positions=None*, *eps=0.0*, *boxsize=None*)

`set_periodic`(*axis=None*, *primitive=None*)

Sets periodic boundary conditions along the given axis.

Parameters

axis
[bool or int or (N,) array_like] One or multiple axises to apply the periodic boundary conditions. If the axis is `None` the perodic boundary conditions will be removed.

primitive
[bool, optional] Flag if the specified axes are in cartesian or lattice coordinates. If `True` the passed position will be multiplied with the lattice vectors. The default is `False` (cartesian coordinates). .. deprecated:: 0.8.0

The *primitive* argument will be removed in lattpy 0.9.0

Raises

NotBuiltError
Raised if the lattice hasn't been built yet.

Notes

The lattice has to be built before applying the periodic boundarie conditions. The lattice also has to be at least three atoms big in the specified directions. Uses the same coordinate system (cartesian or primtive basis vectors) as chosen for building the lattice.

`compute_connections`(*latt*)

Computes the connections between the current and another lattice.

Parameters

latt
[Lattice] The other lattice.

Returns

neighbors
[(N, 2) np.ndarray] The connecting pairs between the two lattices. The first index of each row is the index in the current lattice data, the second one is the index for the other lattice `latt`.

distances
[(N) np.ndarray] The corresponding distances for the connections.

`minimum_distances`(*site*, *primitive=None*)

Computes the minimum distances between one site and the other lattice sites.

This method can be used to find the distances in a lattice with periodic boundary conditions.

Parameters

site
[int] The super-index i of a site in the cached lattice data.

primitive
[bool, optional] Flag if the periopdic boundarey conditions are set up along cartesian or primitive basis vectors. The default is `False` (cartesian coordinates). .. deprecated:: 0.8.0

The *primitive* argument will be removed in lattpy 0.9.0

Returns

min_dists
[(N,) np.ndarray] The minimum distances between the lattice site i and the other sites.

Notes

Uses the same coordinate system (cartesian or primtive basis vectors) as chosen for building the lattice.

append(*latt*, *ax=0*, *side=1*, *sort_ax=None*, *sort_reverse=False*, *primitive=None*)

Append another *Lattice*-instance along an axis.

Parameters

latt
[Lattice] The other lattice to append to this instance.

ax
[int, optional] The axis along the other lattice is appended. The default is 0 (x-axis).

side
[int, optional] The side at which the new lattice is appended. If, for example, axis 0 is used, the other lattice is appended on the right side if `side=+1` and on the left side if `side=-1`.

sort_ax
[int, optional] The axis to sort the lattice indices after the other lattice has been added. The default is the value specified for `ax`.

sort_reverse
[bool, optional] If True, the lattice indices are sorted in reverse order.

primitive
[bool, optional] Flag if the periopdic boundarey conditions are set up along cartesian or primitive basis vectors. The default is `False` (cartesian coordinates). .. deprecated:: 0.8.0

The *primitive* argument will be removed in lattpy 0.9.0

Notes

Uses the same coordinate system (cartesian or primtive basis vectors) as chosen for building the lattice.

Examples

```
>>> latt = Lattice(np.eye(2))
>>> latt.add_atom(neighbors=1)
>>> latt.build((5, 2))
>>> latt.shape
[5. 2.]
```

```
>>> latt2 = Lattice(np.eye(2))
>>> latt2.add_atom(neighbors=1)
>>> latt2.build((2, 2))
>>> latt2.shape
[2. 2.]
```

```
>>> latt.append(latt2, ax=0)
>>> latt.shape
[8. 2.]
```

extend(*size*, *ax=0*, *side=1*, *num_jobs=1*, *sort_ax=None*, *sort_reverse=False*)

Extend the lattice along an axis.

Parameters

size
[float] The size of which the lattice will be extended in direction of `ax`.

ax
[int, optional] The axis along the lattice is extended. The default is 0 (x-axis).

side
[int, optional] The side at which the new lattice is appended. If, for example, axis 0 is used, the lattice is extended to the right side if `side=+1` and to the left side if `side=-1`.

num_jobs
[int, optional] Number of jobs to schedule for parallel processing of neighbors for new sites. If `-1` is given all processors are used. The default is `-1`.

sort_ax
[int, optional] The axis to sort the lattice indices after the lattice has been extended. The default is the value specified for `ax`.

sort_reverse
[bool, optional] If True, the lattice indices are sorted in reverse order.

Examples

```
>>> latt = Lattice(np.eye(2))
>>> latt.add_atom(neighbors=1)
>>> latt.build((5, 2))
>>> latt.shape
[5. 2.]
```

```
>>> latt.extend(2, ax=0)
[8. 2.]
```

```
>>> latt.extend(2, ax=1)
[8. 5.]
```

repeat(*num=1*, *ax=0*, *side=1*, *sort_ax=None*, *sort_reverse=False*)

Repeat the lattice along an axis.

Parameters

num
[int] The number of times the lattice will be repeated in direction `ax`.

ax
[int, optional] The axis along the lattice is extended. The default is 0 (x-axis).

side
[int, optional] The side at which the new lattice is appended. If, for example, axis 0 is used, the lattice is extended to the right side if `side=+1` and to the left side if `side=-1`.

sort_ax
[int, optional] The axis to sort the lattice indices after the lattice has been extended. The default is the value specified for `ax`.

sort_reverse
[bool, optional] If True, the lattice indices are sorted in reverse order.

Examples

```
>>> latt = Lattice(np.eye(2))
>>> latt.add_atom(neighbors=1)
>>> latt.build((5, 2))
>>> latt.shape
[5. 2.]
```

```
>>> latt.repeat()
[11.  2.]
```

```
>>> latt.repeat(3)
[35.  2.]
```

```
>>> latt.repeat(ax=1)
[35.  5.]
```

dmap()

DataMap : Returns the data-map of the lattice model.

neighbor_pairs(*unique=False*)

Returns all neighbor pairs with their corresponding distances in the lattice.

Parameters

unique
[bool, optional] If True, only unique pairs with i < j are returned. The default is False.

Returns

pairs
[(N, 2) np.ndarray] An array containing all neighbor pairs of the lattice. If *unique=True*, the first index is always smaller than the second index in each element.

distindices
[(N,) np.ndarray] The corresponding distance indices of the neighbor pairs.

Examples

```
>>> latt = Lattice.chain()
>>> latt.add_atom(neighbors=1)
>>> latt.build(5)
>>> idx, distidx = latt.neighbor_pairs()
>>> idx
array([[0, 1],
       [1, 2],
       [1, 0],
       [2, 3],
       [2, 1],
       [3, 2]], dtype=uint8)
```

```
>>> distidx
array([0, 0, 0, 0, 0, 0], dtype=uint8)
```

```
>>> idx, distidx = latt.neighbor_pairs(unique=True)
>>> idx
array([[0, 1],
       [1, 2],
       [2, 3]], dtype=uint8)
```

adjacency_matrix()

Computes the adjacency matrix for the neighbor data of the lattice.

Returns

adj_mat
[(N, N) csr_matrix] The adjacency matrix of the lattice.

See also:

neighbor_pairs
Generates a list of neighbor indices.

Examples

```
>>> latt = Lattice.chain()
>>> latt.add_atom(neighbors=1)
>>> latt.build(5)
>>> adj_mat = latt.adjacency_matrix()
>>> adj_mat.toarray()
array([[0, 1, 0, 0],
       [1, 0, 1, 0],
```

(continues on next page)

(continued from previous page)

```
       [0, 1, 0, 1],
       [0, 0, 1, 0]], dtype=int8)
```

`copy`()

Lattice : Creates a (deep) copy of the lattice instance.

`todict`()

Creates a dictionary containing the information of the lattice instance.

Returns

d
[dict] The information defining the current instance.

`dumps`()

Creates a string containing the information of the lattice instance.

Returns

s
[str] The information defining the current instance.

`dump`(*file*)

Save the data of the `Lattice` instance.

Parameters

file
[str or int or bytes] File name to store the lattice. If `None` the hash of the lattice is used.

`classmethod load`(*file*)

Load data of a saved `Lattice` instance.

Parameters

file
[str or int or bytes] File name to load the lattice.

Returns

latt
[Lattice] The lattice restored from the file content.

`plot`(*lw=None*, *margins=0.1*, *legend=None*, *grid=False*, *pscale=0.5*, *show_periodic=True*, *show_indices=False*, *index_offset=0.1*, *con_colors=None*, *adjustable='box'*, *ax=None*, *show=False*)

Plot the cached lattice.

Parameters

lw
[float, optional] Line width of the neighbor connections.

margins
[Sequence[float] or float, optional] The margins of the plot.

legend
[bool, optional] Flag if legend is shown

grid
[bool, optional] If True, draw a grid in the plot.

pscale
: [float, optional] The scale for drawing periodic connections. The default is half of the normal length.

show_periodic
: [bool, optional] If True the periodic connections will be shown.

show_indices
: [bool, optional] If True the index of the sites will be shown.

index_offset
: [float, optional] The positional offset of the index text labels. Only used if *show_indices=True*.

con_colors
: [Sequence[tuple], optional] list of colors to override the defautl connection color. Each element has to be a tuple with the first two elements being the atom indices of the pair and the third element the color, for example `[(0, 0, 'r')]`.

adjustable
: [None or {'box', 'datalim'}, optional] If not None, this defines which parameter will be adjusted to meet the equal aspect ratio. If 'box', change the physical dimensions of the Axes. If 'datalim', change the x or y data limits. Only applied to 2D plots.

ax
: [plt.Axes or plt.Axes3D or None, optional] Parent plot. If None, a new plot is initialized.

show
: [bool, optional] If True, show the resulting plot.

CHAPTER

TEN

LATTPY.PLOTTING

Contains plotting tools for the lattice and other related objects.

lattpy.plotting.**subplot**(*dim*, *adjustable='box'*, *ax=None*)

Generates a two- or three-dimensional subplot with a equal aspect ratio

Parameters

dim
[int] The dimension of the plot.

adjustable
[None or {'box', 'datalim'}, optional] If not None, this defines which parameter will be adjusted to meet the equal aspect ratio. If 'box', change the physical dimensions of the Axes. If 'datalim', change the x or y data limits. Only applied to 2D plots.

ax
[Axes, optional] Existing axes to format. If an existing axes is passed no new figure is created.

Returns

fig
[Figure] The figure of the subplot.

ax
[Axes] The newly created or formatted axes of the subplot.

lattpy.plotting.**draw_line**(*ax*, *points*, ***kwargs*)

Draw a line segment between multiple points.

Parameters

ax
[Axes] The axes for drawing the line segment.

points
[(N, D) np.ndarray] A list of points between the line is drawn.

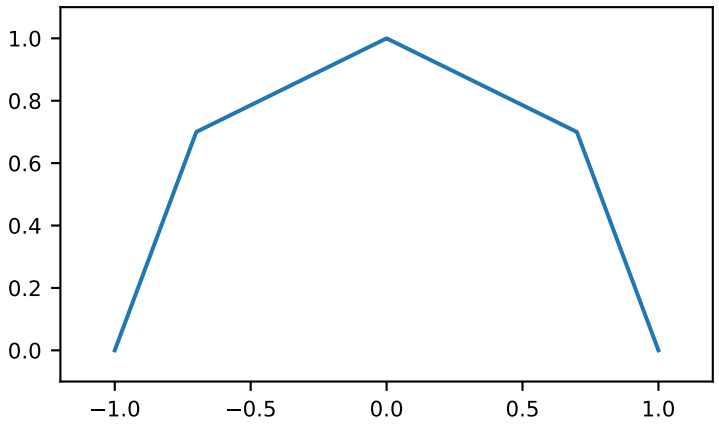
****kwargs**
Additional keyword arguments for drawing the line.

Returns

coll
[Line2D or Line3D] The created line.

Examples

```
>>> from lattpy import plotting
>>> import matplotlib.pyplot as plt
>>> fig, ax = plt.subplots()
>>> points = np.array([[1, 0], [0.7, 0.7], [0, 1], [-0.7, 0.7], [-1, 0]])
>>> _ = plotting.draw_line(ax, points)
>>> ax.margins(0.1, 0.1)
>>> plt.show()
```



lattpy.plotting.**draw_lines**(*ax*, *segments*, ***kwargs*)

Draw multiple line segments between points.

Parameters

ax
[Axes] The axes for drawing the lines.

segments
[array_like of (2, D) np.ndarray] A list of point pairs between the lines are drawn.

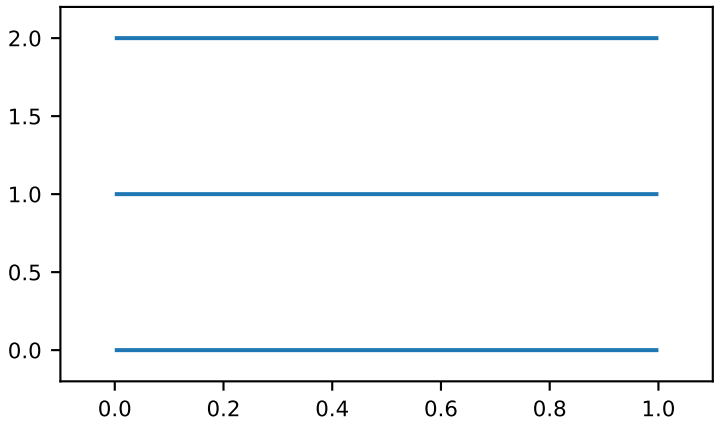
****kwargs**
Additional keyword arguments for drawing the lines.

Returns

coll: LineCollection or Line3DCollection
The created line collection.

Examples

```
>>> from lattpy import plotting
>>> import matplotlib.pyplot as plt
>>> fig, ax = plt.subplots()
>>> segments = np.array([
...     [[0, 0], [1, 0]],
...     [[0, 1], [1, 1]],
...     [[0, 2], [1, 2]]
... ])
>>> _ = plotting.draw_lines(ax, segments)
>>> ax.margins(0.1, 0.1)
>>> plt.show()
```



lattpy.plotting.**hide_box**(*ax*, *axis=False*)

Remove the box and optionally the axis of a plot.

Parameters

ax
[Axes] The axes to remove the box.

axis
[bool, optional] If True the axis are hiden as well as the box.

lattpy.plotting.**draw_arrows**(*ax*, *vectors*, *pos=None*, ***kwargs*)

Draws multiple arrows from an optional starting point in the given directions.

Parameters

ax
[Axes] The axes for drawing the arrows.

vectors
[(N, D) np.ndarray] The vectors to draw.

pos
[(D,) np.ndarray, optional] The starting position of the vectors. The default is the origin.

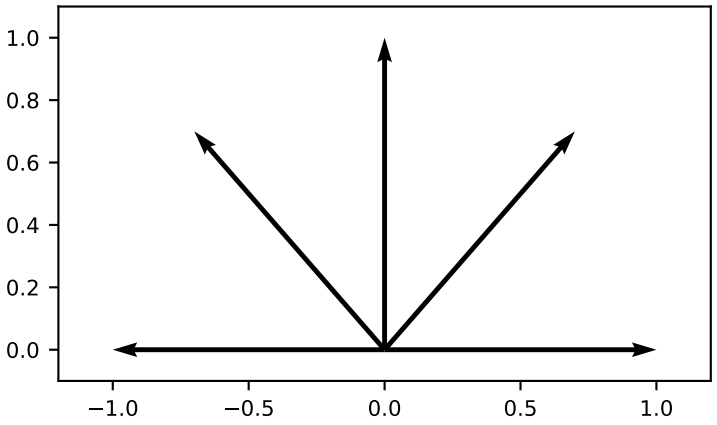
****kwargs**
Additional keyword arguments for drawing the arrows.

Returns

coll: LineCollection or Line3DCollection
The created line collection.

Examples

```
>>> from lattpy import plotting
>>> import matplotlib.pyplot as plt
>>> fig, ax = plt.subplots()
>>> vectors = np.array([[1, 0], [0.7, 0.7], [0, 1], [-0.7, 0.7], [-1, 0]])
>>> _ = plotting.draw_arrows(ax, vectors)
>>> ax.margins(0.1, 0.1)
>>> plt.show()
```



lattpy.plotting.**draw_vectors**(*ax*, *vectors*, *pos=None*, ***kwargs*)

Draws multiple lines from an optional starting point in the given directions.

Parameters

ax
[Axes] The axes for drawing the lines.

vectors
[(N, D) np.ndarray] The vectors to draw.

pos
[(D,) np.ndarray, optional] The starting position of the vectors. The default is the origin.

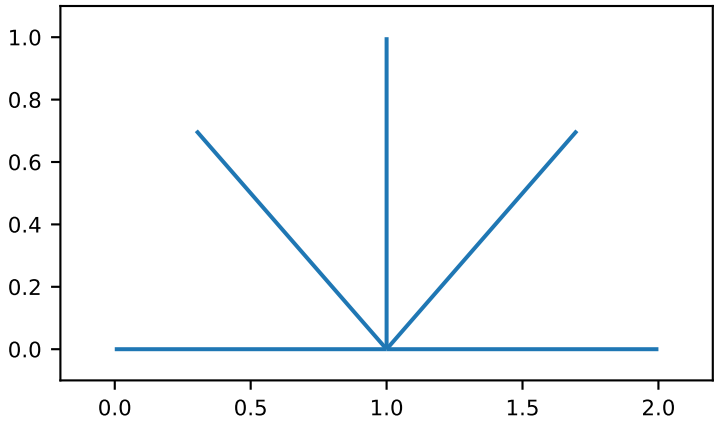
****kwargs**
Additional keyword arguments for drawing the lines.

Returns

coll: LineCollection or Line3DCollection
The created line collection.

Examples

```
>>> from lattpy import plotting
>>> import matplotlib.pyplot as plt
>>> fig, ax = plt.subplots()
>>> vectors = np.array([[1, 0], [0.7, 0.7], [0, 1], [-0.7, 0.7], [-1, 0]])
>>> _ = plotting.draw_vectors(ax, vectors, [1, 0])
>>> ax.margins(0.1, 0.1)
>>> plt.show()
```



lattpy.plotting.**draw_points**(*ax*, *points*, *size=10*, ***kwargs*)

Draws multiple points as scatter plot.

Parameters

ax
[Axes] The axes for drawing the points.

points
[(N, D) np.ndarray] The positions of the points to draw.

size
[float, optional] The size of the markers of the points.

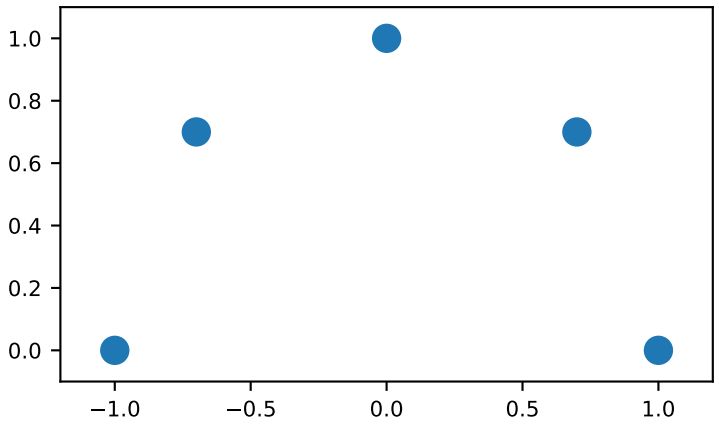
****kwargs**
Additional keyword arguments for drawing the points.

Returns

scat
[PathCollection] The scatter plot item.

Examples

```
>>> from lattpy import plotting
>>> import matplotlib.pyplot as plt
>>> fig, ax = plt.subplots()
>>> points = np.array([[1, 0], [0.7, 0.7], [0, 1], [-0.7, 0.7], [-1, 0]])
>>> _ = plotting.draw_points(ax, points)
>>> ax.margins(0.1, 0.1)
>>> plt.show()
```



lattpy.plotting.**draw_indices**(*ax*, *positions*, *offset=0.05*, ***kwargs*)

Draws the indices of the given positions on the plot.

Parameters

ax
[Axes] The axes for drawing the text.

positions
[(..., D) array_like] The positions of the texts.

offset
[float or (D,) array_like] The offset of the positions of the texts.

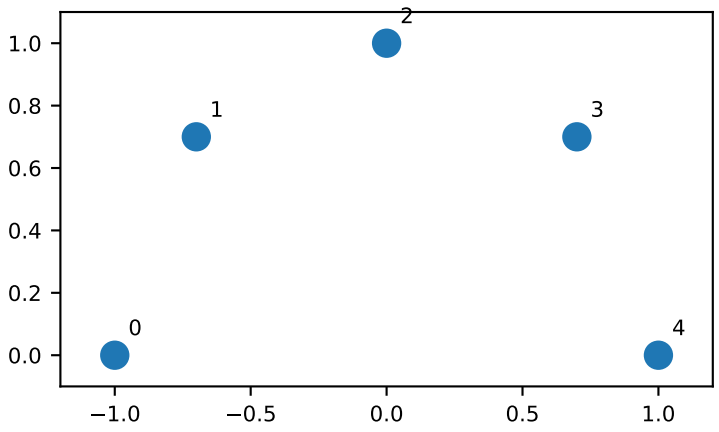
****kwargs**
Additional keyword arguments for drawing the text.

Returns

texts
[list] The text items.

Examples

```
>>> from lattpy import plotting
>>> import matplotlib.pyplot as plt
>>> points = np.array([[-1, 0], [-0.7, 0.7], [0, 1], [0.7, 0.7], [1, 0]])
>>> fig, ax = plt.subplots()
>>> _ = plotting.draw_points(ax, points)
>>> _ = plotting.draw_indices(ax, points)
>>> ax.margins(0.1, 0.1)
>>> plt.show()
```



lattpy.plotting.**draw_unit_cell**(*ax*, *vectors*, *outlines=True*, ***kwargs*)

Draws the basis vectors and unit cell.

Parameters

ax
[Axes] The axes for drawing the text.

vectors
[float or (D, D) array_like] The vectors defining the basis.

outlines
[bool, optional] If True the box define dby the basis vectors (unit cell) is drawn.

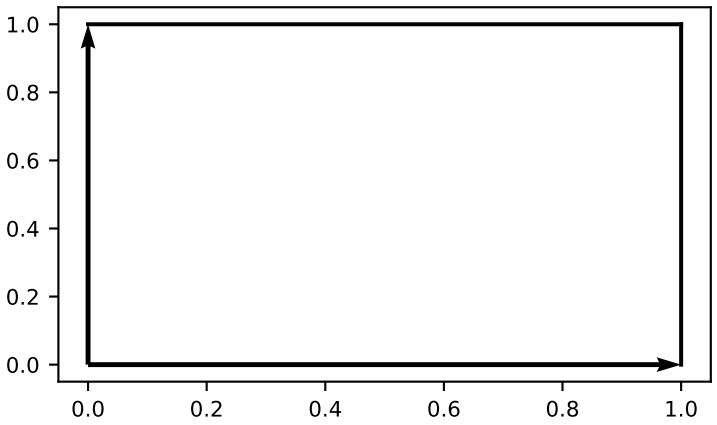
****kwargs**
Additional keyword arguments for drawing the lines.

Returns

lines
[list] A list of the plotted lines.

Examples

```
>>> from lattpy import plotting
>>> import matplotlib.pyplot as plt
>>> vectors = np.array([[1, 0], [0, 1]])
>>> fig, ax = plt.subplots()
>>> _ = plotting.draw_unit_cell(ax, vectors)
>>> plt.show()
```



lattpy.plotting.**draw_surfaces**(*ax*, *vertices*, ***kwargs*)

Draws a 3D surfaces defined by a set of vertices.

Parameters

ax
[Axes3D] The axes for drawing the surface.

vertices
[array_like] The vertices defining the surface.

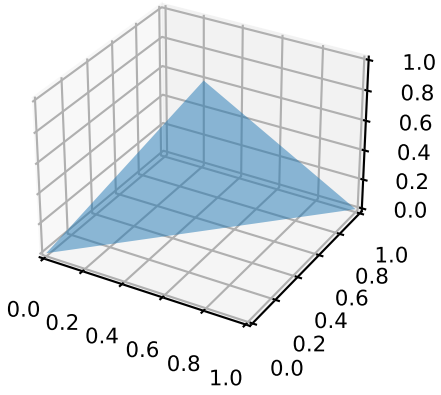
****kwargs**
Additional keyword arguments for drawing the lines.

Returns

surf
[Poly3DCollection] The surface object.

Examples

```
>>> from lattpy import plotting
>>> import matplotlib.pyplot as plt
>>> vertices = [[0, 0, 0], [1, 1, 0], [0.5, 0.5, 1]]
>>> fig = plt.figure()
>>> ax = fig.add_subplot(111, projection="3d")
>>> _ = plotting.draw_surfaces(ax, vertices, alpha=0.5)
>>> plt.show()
```



lattpy.plotting.**interpolate_to_grid**(*positions*, *values*, *num=(100, 100)*, *offset=(0.0, 0.0)*, *method='linear'*, *fill_value=nan*)

lattpy.plotting.**draw_sites**(*ax*, *points*, *radius=0.2*, ***kwargs*)

Draws multiple circles with a scaled radius.

Parameters

ax
[Axes] The axes for drawing the points.

points
[(N, D) np.ndarray] The positions of the points to draw.

radius
[float] The radius of the points. Scaling is only supported for 2D plots!

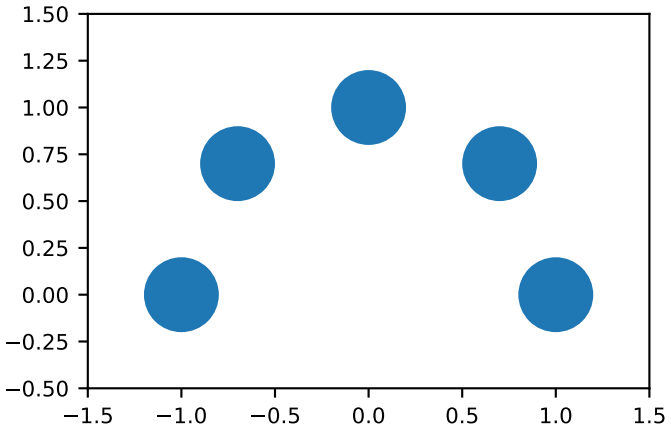
****kwargs**
Additional keyword arguments for drawing the points.

Returns

point_coll
[CircleCollection or PathCollection] The circle or path collection.

Examples

```
>>> from lattpy import plotting
>>> import matplotlib.pyplot as plt
>>> fig, ax = plt.subplots()
>>> points = np.array([[1, 0], [0.7, 0.7], [0, 1], [-0.7, 0.7], [-1, 0]])
>>> _ = plotting.draw_sites(ax, points, radius=0.2)
>>> _ = ax.set_xlim(-1.5, +1.5)
>>> _ = ax.set_ylim(-0.5, +1.5)
>>> plotting.set_equal_aspect(ax)
>>> plt.show()
```



lattpy.plotting.**connection_color_array**(*num_base*, *default='k'*, *colors=None*)

Construct color array for the connections between all atoms in a lattice.

Parameters

num_base
[int] The number of atoms in the unit cell of a lattice.

default
[str or int or float or tuple] The default color of the connections.

colors
[Sequence[tuple], optional] list of colors to override the defautl connection color. Each element has to be a tuple with the first two elements being the atom indices of the pair and the third element the color, for example `[(0, 0, 'r')]`.

Returns

color_array
[List of List] The connection color array

Examples

```
>>> connection_color_array(2, "k", colors=[(0, 1, "r")])
[['k', 'r'], ['r', 'k']]
```

CHAPTER

ELEVEN

LATTPY.SHAPE

Objects for representing the shape of a finite lattice.

class lattpy.shape.**AbstractShape**(*dim*, *pos=None*)

Bases: ABC

Abstract shape object.

abstract limits()

Returns the limits of the shape.

abstract contains(*points*, *tol=0.0*)

Checks if the given points are contained in the shape.

abstract plot(*ax*, *color='k'*, *lw=0.0*, *alpha=0.2*, ***kwargs*)

Plots the contour of the shape.

class lattpy.shape.**Shape**(*shape*, *pos=None*, *basis=None*)

Bases: *AbstractShape*

General shape object.

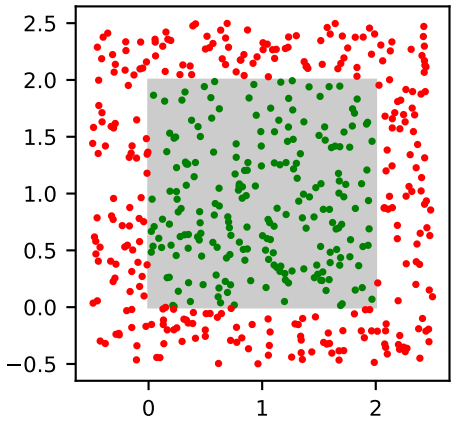
Examples

Cartesian coordinates

```
>>> points = np.random.uniform(-0.5, 2.5, size=(500, 2))
>>> s = lp.Shape((2, 2))
>>> s.limits()
[[0.  2. ]
 [0.  2.]]
```

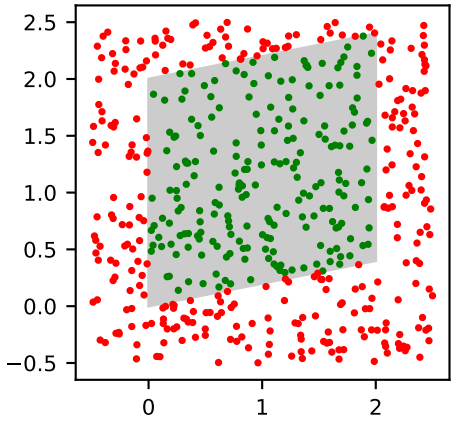
```
>>> import matplotlib.pyplot as plt
>>> mask = s.contains(points)
>>> s.plot(plt.gca())
>>> plt.scatter(*points[mask].T, s=3, color="g")
>>> plt.scatter(*points[~mask].T, s=3, color="r")
>>> plt.gca().set_aspect("equal")
>>> plt.show()
```

Angled coordinate system




```
>>> s = lp.Shape((2, 2), basis=[[1, 0.2], [0, 1]])
>>> s.limits()
[[0.  2. ]
 [0.  2.4]]
```

```
>>> mask = s.contains(points)
>>> s.plot(plt.gca())
>>> plt.scatter(*points[mask].T, s=3, color="g")
>>> plt.scatter(*points[~mask].T, s=3, color="r")
>>> plt.gca().set_aspect("equal")
>>> plt.show()
```



limits()

Returns the limits of the shape.

contains(*points*, *tol=0.0*)

Checks if the given points are contained in the shape.

plot(*ax*, *color='k'*, *lw=0.0*, *alpha=0.2*, ***kwargs*)

Plots the contour of the shape.

class lattpy.shape.**Circle**(*pos*, *radius*)

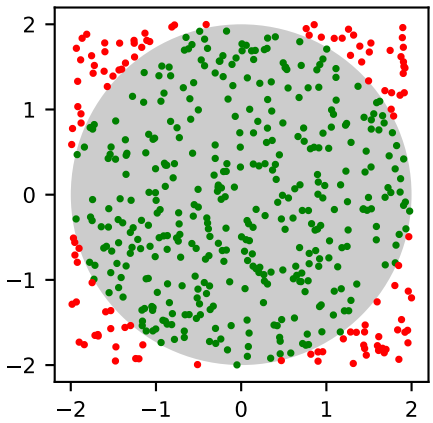
Bases: *AbstractShape*

Circle shape.

Examples

```
>>> s = lp.Circle((0, 0), radius=2)
>>> s.limits()
[[-2.  2.]
 [-2.  2.]]
```

```
>>> import matplotlib.pyplot as plt
>>> points = np.random.uniform(-2, 2, size=(500, 2))
>>> mask = s.contains(points)
>>> s.plot(plt.gca())
>>> plt.scatter(*points[mask].T, s=3, color="g")
>>> plt.scatter(*points[~mask].T, s=3, color="r")
>>> plt.gca().set_aspect("equal")
>>> plt.show()
```



limits()

Returns the limits of the shape.

contains(*points*, *tol=0.0*)

Checks if the given points are contained in the shape.

plot(*ax*, *color='k'*, *lw=0.0*, *alpha=0.2*, ***kwargs*)

Plots the contour of the shape.

class lattpy.shape.**Donut**(*pos*, *radius_outer*, *radius_inner*)

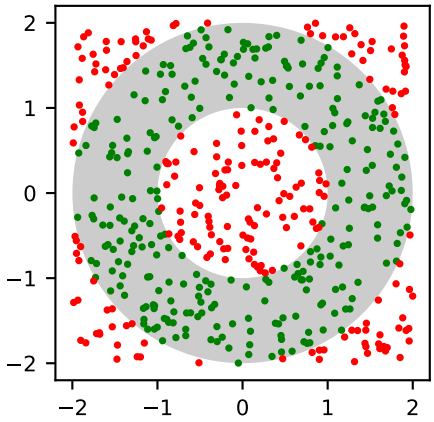
Bases: *AbstractShape*

Circle shape with cut-out in the middle.

Examples

```
>>> s = lp.Donut((0, 0), radius_outer=2, radius_inner=1)
>>> s.limits()
[[-2.  2.]
 [-2.  2.]]
```

```
>>> import matplotlib.pyplot as plt
>>> points = np.random.uniform(-2, 2, size=(500, 2))
>>> mask = s.contains(points)
>>> s.plot(plt.gca())
>>> plt.scatter(*points[mask].T, s=3, color="g")
>>> plt.scatter(*points[~mask].T, s=3, color="r")
>>> plt.gca().set_aspect("equal")
>>> plt.show()
```



limits()

Returns the limits of the shape.

contains(*points*, *tol=1e-10*)

Checks if the given points are contained in the shape.

plot(*ax*, *color='k'*, *lw=0.0*, *alpha=0.2*, ***kwargs*)

Plots the contour of the shape.

class lattpy.shape.**ConvexHull**(*points*)

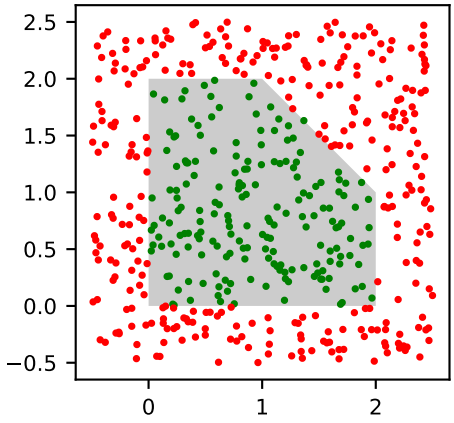
Bases: *AbstractShape*

Shape defined by convex hull of arbitrary points.

Examples

```
>>> s = lp.ConvexHull([[0, 0], [2, 0], [2, 1], [1, 2], [0, 2]])
>>> s.limits()
[[0.  2.]
 [0.  2.]]
```

```
>>> import matplotlib.pyplot as plt
>>> points = np.random.uniform(-0.5, 2.5, size=(500, 2))
>>> mask = s.contains(points)
>>> s.plot(plt.gca())
>>> plt.scatter(*points[mask].T, s=3, color="g")
>>> plt.scatter(*points[~mask].T, s=3, color="r")
>>> plt.gca().set_aspect("equal")
>>> plt.show()
```



limits()

Returns the limits of the shape.

contains(*points*, *tol=1e-10*)

Checks if the given points are contained in the shape.

plot(*ax*, *color='k'*, *lw=0.0*, *alpha=0.2*, ***kwargs*)

Plots the contour of the shape.

CHAPTER

TWELVE

LATTPY.SPATIAL

Spatial algorithms and data structures.

lattpy.spatial.**distance**(*r1*, *r2*, *decimals=None*)

Calculates the euclidian distance bewteen two points.

Parameters

r1
[(D,) np.ndarray] First input point.

r2
[(D,) np.ndarray] Second input point of matching size.

decimals: int, optional
Decimals for rounding the output.

Returns

distance: float
The distance between the input points.

lattpy.spatial.**distances**(*r1*, *r2*, *decimals=None*)

Calculates the euclidian distance between multiple points.

Parameters

r1
[(N, D) array_like] First input point.

r2
[(N, D) array_like] Second input point of matching size.

decimals: int, optional
Optional decimals to round distance to.

Returns

distances
[(N,) np.ndarray] The distances for each pair of input points.

lattpy.spatial.**interweave**(*arrays*)

Interweaves multiple arrays along the first axis

Parameters

arrays: (M) Sequence of (N, ...) array_like
The input arrays to interwave. The shape of all arrays must match.

Returns

interweaved: (M*N, ...) np.ndarray

lattpy.spatial.**vindices**(*limits*, *sort_axis=0*, *dtype=None*)

Return an array representing the indices of a d-dimensional grid.

Parameters

limits: (D, 2) array_like
The limits of the indices for each axis.

sort_axis: int, optional
Optional axis that is used to sort indices.

dtype: int or str or np.dtype, optional
Optional data-type for storing the lattice indices. By default the given limits are checked to determine the smallest possible data-type.

Returns

vectors: (N, D) np.ndarray

lattpy.spatial.**vrange**(*start=None*, **args*, *dtype=None*, *sort_axis=0*, ***kwargs*)

Return evenly spaced vectors within a given interval.

Parameters

start: array_like, optional
The starting value of the interval. The interval includes this value. The default start value is 0.

stop: array_like
The end value of the interval.

step: array_like, optional
Spacing between values. If *start* and *stop* are sequences and the *step* is a scalar the given step size is used for all dimensions of the vectors. The default step size is 1.

sort_axis: int, optional
Optional axis that is used to sort indices.

dtype: dtype, optional
The type of the output array. If *dtype* is not given, infer the data type from the other input arguments.

Returns

vectors: (N, D) np.ndarray

lattpy.spatial.**cell_size**(*vectors*)

Computes the shape of the box spawned by the given vectors.

Parameters

vectors: array_like
The basis vectors defining the cell.

Returns

size: np.ndarray

lattpy.spatial.**cell_volume**(*vectors*)

Computes the volume of the unit cell defined by the primitive vectors.

The volume of the unit-cell in two and three dimensions is defined by

$$V_{2d} = a_1 a_2, \quad V_{3d} = a_1 \cdot a_2 a_3$$

For higher dimensions the volume is computed using the determinant:

$$V_d = \sqrt{\det A A^T}$$

where A is the array of vectors.

Parameters

vectors: array_like
The basis vectors defining the cell.

Returns

vol: float
The volume of the unit cell.

`lattpy.spatial.`**`compute_vectors`**(*a*, *b=None*, *c=None*, *alpha=None*, *beta=None*, *gamma=None*, *decimals=0*)

Computes lattice vectors by the lengths and angles.

`class` `lattpy.spatial.`**`KDTree`**(*points*, *k=1*, *max_dist=inf*, *eps=0.0*, *p=2*, *boxsize=None*)

Bases: `cKDTree`

Simple wrapper of scipy's cKTree with global query settings.

`query_ball_point`(*self*, *x*, *r*, *p=2.0*, *eps=0*, *workers=1*, *return_sorted=None*, *return_length=False*)

Find all points within distance r of point(s) x.

Parameters

x
[array_like, shape tuple + (self.m,)] The point or points to search for neighbors of.

r
[array_like, float] The radius of points to return, shall broadcast to the length of x.

p
[float, optional] Which Minkowski p-norm to use. Should be in the range [1, inf]. A finite large p may cause a ValueError if overflow can occur.

eps
[nonnegative float, optional] Approximate search. Branches of the tree are not explored if their nearest points are further than `r / (1 + eps)`, and branches are added in bulk if their furthest points are nearer than `r * (1 + eps)`.

workers
[int, optional] Number of jobs to schedule for parallel processing. If -1 is given all processors are used. Default: 1.

Changed in version 1.6.0: The "n_jobs" argument was renamed "workers". The old name "n_jobs" is deprecated and will stop working in SciPy 1.8.0.

return_sorted
[bool, optional] Sorts returned indicies if True and does not sort them if False. If None, does not sort single point queries, but does sort multi-point queries which was the behavior before this option was added.

New in version 1.2.0.

return_length: bool, optional
Return the number of points inside the radius instead of a list of the indices. .. versionadded:: 1.3.0

Returns

results
[list or array of lists] If *x* is a single point, returns a list of the indices of the neighbors of *x*. If *x* is an array of points, returns an object array of shape tuple containing lists of neighbors.

Notes

If you have many points whose neighbors you want to find, you may save substantial amounts of time by putting them in a cKDTree and using query_ball_tree.

Examples

```
>>> from scipy import spatial
>>> x, y = np.mgrid[0:4, 0:4]
>>> points = np.c_[x.ravel(), y.ravel()]
>>> tree = spatial.cKDTree(points)
>>> tree.query_ball_point([2, 0], 1)
[4, 8, 9, 12]
```

Query multiple points and plot the results:

```
>>> import matplotlib.pyplot as plt
>>> points = np.asarray(points)
>>> plt.plot(points[:,0], points[:,1], '.')
>>> for results in tree.query_ball_point(([2, 0], [3, 3]), 1):
...     nearby_points = points[results]
...     plt.plot(nearby_points[:,0], nearby_points[:,1], 'o')
>>> plt.margins(0.1, 0.1)
>>> plt.show()
```

`query_ball_tree`(*self*, *other*, *r*, *p=2.0*, *eps=0*)

Find all pairs of points between *self* and *other* whose distance is at most r

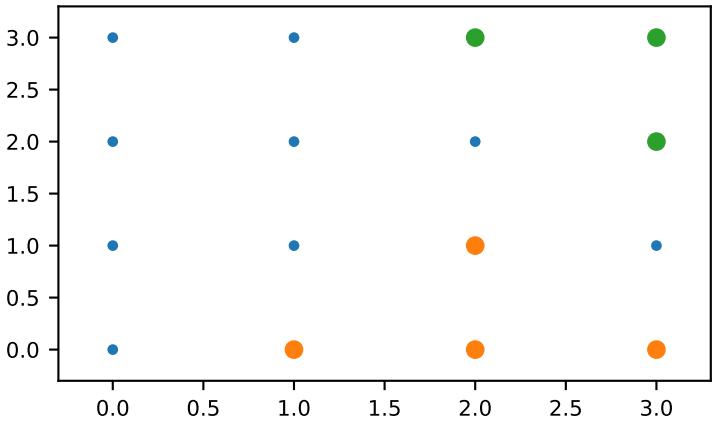
Parameters

other
[cKDTree instance] The tree containing points to search against.

r
[float] The maximum distance, has to be positive.

p
[float, optional] Which Minkowski norm to use. *p* has to meet the condition `1 <= p <= infinity`. A finite large p may cause a ValueError if overflow can occur.

eps
[float, optional] Approximate search. Branches of the tree are not explored if their nearest points are further than `r/(1+eps)`, and branches are added in bulk if their furthest points are nearer than `r * (1+eps)`. *eps* has to be non-negative.



Returns

results
[list of lists] For each element `self.data[i]` of this tree, `results[i]` is a list of the indices of its neighbors in `other.data`.

Examples

You can search all pairs of points between two kd-trees within a distance:

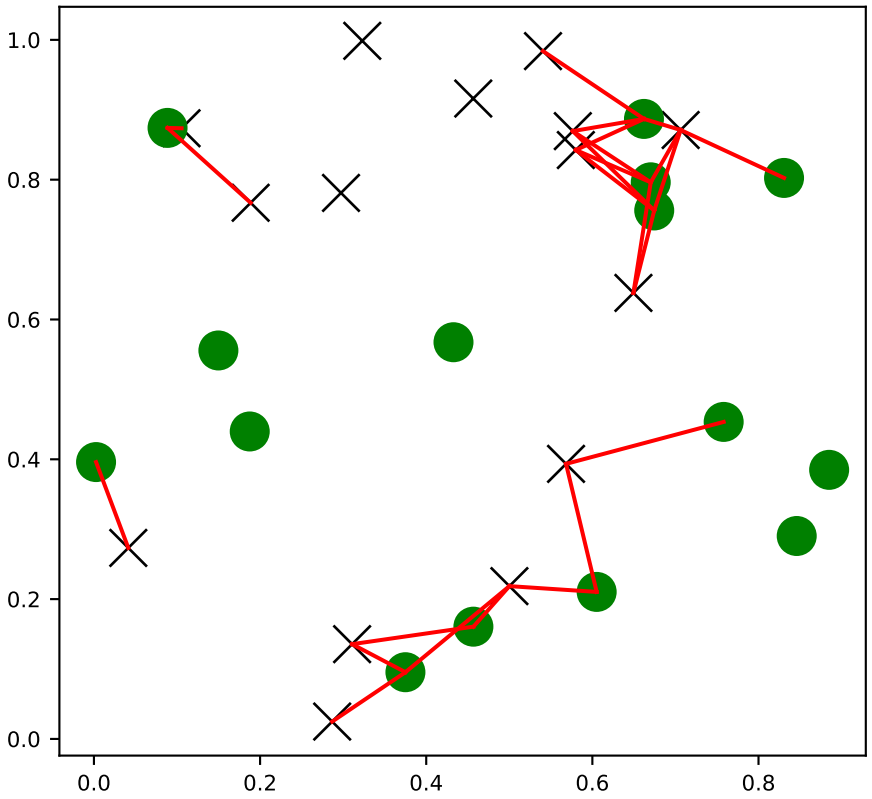
```
>>> import matplotlib.pyplot as plt
>>> import numpy as np
>>> from scipy.spatial import cKDTree
>>> rng = np.random.default_rng()
>>> points1 = rng.random((15, 2))
>>> points2 = rng.random((15, 2))
>>> plt.figure(figsize=(6, 6))
>>> plt.plot(points1[:, 0], points1[:, 1], "xk", markersize=14)
>>> plt.plot(points2[:, 0], points2[:, 1], "og", markersize=14)
>>> kd_tree1 = cKDTree(points1)
>>> kd_tree2 = cKDTree(points2)
>>> indexes = kd_tree1.query_ball_tree(kd_tree2, r=0.2)
>>> for i in range(len(indexes)):
...     for j in indexes[i]:
...         plt.plot([points1[i, 0], points2[j, 0]],
...             [points1[i, 1], points2[j, 1]], "-r")
>>> plt.show()
```

query_pairs(*self*, *r*, *p=2.0*, *eps=0*)

Find all pairs of points in *self* whose distance is at most r.

Parameters

r
[positive float] The maximum distance.


1.0
0.8
0.6
0.4
0.2
0.0
0.0
0.2
0.4
0.6
0.8

p
: [float, optional] Which Minkowski norm to use. `p` has to meet the condition `1 <= p <= infinity`. A finite large p may cause a ValueError if overflow can occur.

eps
: [float, optional] Approximate search. Branches of the tree are not explored if their nearest points are further than `r/(1+eps)`, and branches are added in bulk if their furthest points are nearer than `r * (1+eps)`. *eps* has to be non-negative.

output_type
: [string, optional] Choose the output container, 'set' or 'ndarray'. Default: 'set'

Returns

results
: [set or ndarray] Set of pairs `(i,j)`, with `i < j`, for which the corresponding positions are close. If output_type is 'ndarray', an ndarry is returned instead of a set.

Examples

You can search all pairs of points in a kd-tree within a distance:

```
>>> import matplotlib.pyplot as plt
>>> import numpy as np
>>> from scipy.spatial import cKDTree
>>> rng = np.random.default_rng()
>>> points = rng.random((20, 2))
>>> plt.figure(figsize=(6, 6))
>>> plt.plot(points[:, 0], points[:, 1], "xk", markersize=14)
>>> kd_tree = cKDTree(points)
>>> pairs = kd_tree.query_pairs(r=0.2)
>>> for (i, j) in pairs:
...     plt.plot([points[i, 0], points[j, 0]],
...              [points[i, 1], points[j, 1]], "-r")
>>> plt.show()
```

query(*self*, *x*, *k=1*, *eps=0*, *p=2*, *distance_upper_bound=np.inf*, *workers=1*)

Query the kd-tree for nearest neighbors

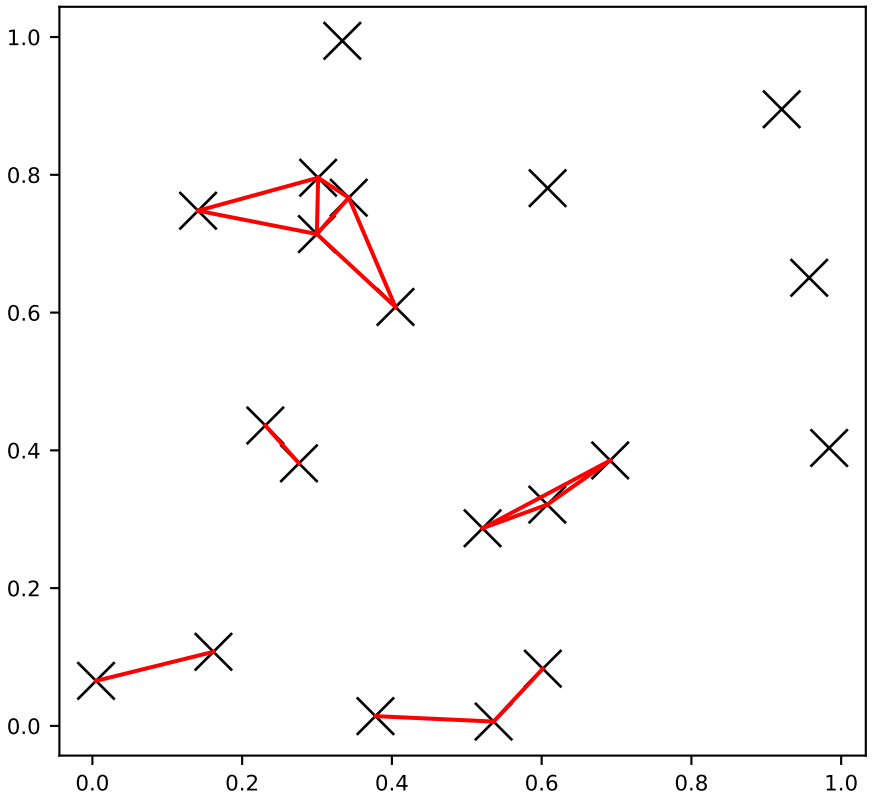
Parameters

x
: [array_like, last dimension self.m] An array of points to query.

k
: [list of integer or integer] The list of k-th nearest neighbors to return. If k is an integer it is treated as a list of [1, … k] (range(1, k+1)). Note that the counting starts from 1.

eps
: [non-negative float] Return approximate nearest neighbors; the k-th returned value is guaranteed to be no further than (1+eps) times the distance to the real k-th nearest neighbor.

p
: [float, 1<=p<=infinity] Which Minkowski p-norm to use. 1 is the sum-of-absolute-values "Manhattan" distance 2 is the usual Euclidean distance infinity is the maximum-coordinate-difference distance A finite large p may cause a ValueError if overflow can occur.


1.0
0.8
0.6
0.4
0.2
0.0
0.0
0.2
0.4
0.6
0.8
1.0

distance_upper_bound
: [nonnegative float] Return only neighbors within this distance. This is used to prune tree searches, so if you are doing a series of nearest-neighbor queries, it may help to supply the distance to the nearest neighbor of the most recent point.

workers
: [int, optional] Number of workers to use for parallel processing. If -1 is given all CPU threads are used. Default: 1.

  Changed in version 1.6.0: The "n_jobs" argument was renamed "workers". The old name "n_jobs" is deprecated and will stop working in SciPy 1.8.0.

Returns

d
: [array of floats] The distances to the nearest neighbors. If `x` has shape `tuple+(self.m,)`, then `d` has shape `tuple+(k,)`. When k == 1, the last dimension of the output is squeezed. Missing neighbors are indicated with infinite distances.

i
: [ndarray of ints] The index of each neighbor in `self.data`. If `x` has shape `tuple+(self.m,)`, then `i` has shape `tuple+(k,)`. When k == 1, the last dimension of the output is squeezed. Missing neighbors are indicated with `self.n`.

Notes

If the KD-Tree is periodic, the position `x` is wrapped into the box.

When the input k is a list, a query for arange(max(k)) is performed, but only columns that store the requested values of k are preserved. This is implemented in a manner that reduces memory usage.

Examples

```
>>> import numpy as np
>>> from scipy.spatial import cKDTree
>>> x, y = np.mgrid[0:5, 2:8]
>>> tree = cKDTree(np.c_[x.ravel(), y.ravel()])
```

To query the nearest neighbours and return squeezed result, use

```
>>> dd, ii = tree.query([[0, 0], [2.2, 2.9]], k=1)
>>> print(dd, ii, sep='\n')
[2.         0.2236068]
[ 0 13]
```

To query the nearest neighbours and return unsqueezed result, use

```
>>> dd, ii = tree.query([[0, 0], [2.2, 2.9]], k=[1])
>>> print(dd, ii, sep='\n')
[[2.        ]
 [0.2236068]]
[[ 0]
 [13]]
```

To query the second nearest neighbours and return unsqueezed result, use

```
>>> dd, ii = tree.query([[0, 0], [2.2, 2.9]], k=[2])
>>> print(dd, ii, sep='\n')
[[2.23606798]
 [0.80622577]]
[[ 6]
 [19]]
```

To query the first and second nearest neighbours, use

```
>>> dd, ii = tree.query([[0, 0], [2.2, 2.9]], k=2)
>>> print(dd, ii, sep='\n')
[[2.         2.23606798]
 [0.2236068  0.80622577]]
[[ 0  6]
 [13 19]]
```

or, be more specific

```
>>> dd, ii = tree.query([[0, 0], [2.2, 2.9]], k=[1, 2])
>>> print(dd, ii, sep='\n')
[[2.         2.23606798]
 [0.2236068  0.80622577]]
[[ 0  6]
 [13 19]]
```

class lattpy.spatial.**VoronoiTree**(*points*)

Bases: object

query(*x*, *k=1*, *eps=0*)

draw(*ax=None*, *color='C0'*, *size=3*, *lw=1*, *alpha=0.15*, *point_color='k'*, *point_size=3*, *draw_data=True*, *points=True*, *draw=True*, *fill=True*)

class lattpy.spatial.**WignerSeitzCell**(*points*)

Bases: *VoronoiTree*

property limits

property size

check(*points*)

arange(*steps*, *offset=0.0*)

linspace(*nums*, *offset=0.0*, *endpoint=False*)

meshgrid(*nums=None*, *steps=None*, *offset=0.0*, *check=True*, *endpoint=False*)

symmetry_points()

lattpy.spatial.**rx**(*theta*)

X-Rotation matrix.

lattpy.spatial.**ry**(*theta*)

Y-Rotation matrix.

lattpy.spatial.**rz**(*theta*)

Z-Rotation matrix.

lattpy.spatial.**rotate2d**(*a*, *theta*, *degree=True*)

Applies the z-rotation matrix to a 2D point

lattpy.spatial.**rotate3d**(*a*, *thetax=0.0*, *thetay=0.0*, *thetaz=0.0*, *degree=True*)

Applies the general rotation matrix to a 3D point

CHAPTER

THIRTEEN

LATTPY.STRUCTURE

Lattice structure object for defining the atom basis and neighbor connections.

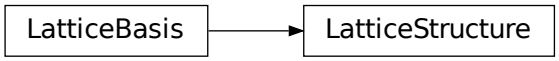
class lattpy.structure.**LatticeStructure**(*basis*, ***kwargs*)

Bases: *LatticeBasis*

Structure object representing a infinite Bravais lattice.

Combines the `LatticeBasis` with a set of atoms and connections between them to define a general lattice structure.

Inheritance



Parameters

basis
[array_like or float or LatticeBasis] The primitive basis vectors that define the unit cell of the lattice. If a `LatticeBasis` instance is passed it is copied and used as the new basis of the lattice.

****kwargs**
Key-word arguments. Used for quickly configuring a `LatticeStructure` instance. Allowed keywords are:

Properties: atoms: Dictionary containing the atoms to add to the lattice. cons: Dictionary containing the connections to add to the lattice.

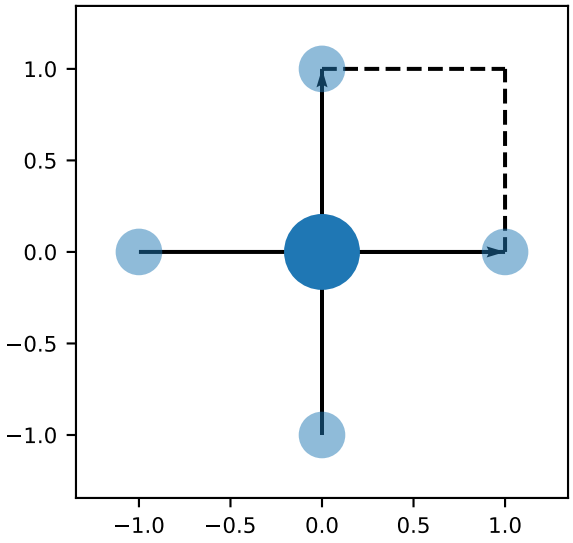
Examples

Two dimensional lattice with one atom in the unit cell and nearest neighbors

```
>>> import lattpy as lp
>>> latt = lp.LatticeStructure(np.eye(2))
>>> latt.add_atom()
>>> latt.add_connections(1)
>>> latt
LatticeStructure(dim: 2, num_base: 1, num_neighbors: [4])
```

Quick-setup of the same lattice:

```
>>> import lattpy as lp
>>> import matplotlib.pyplot as plt
>>> latt = lp.LatticeStructure.square(atoms={(0.0, 0.0): "A"}, cons={("A", "A"): 1})
>>> _ = latt.plot_cell()
>>> plt.show()
```



DIST_DECIMALS = 6

property num_base

int: The number of atoms in the unit cell.

property atoms

list of Atom: List of the atoms in the unit cell.

property atom_positions

list of np.ndarray: List of positions of the atoms in the unit cell.

property num_distances

int: The maximal number of distances between the lattice sites.

property num_neighbors

np.ndarray: The number of neighbors of each atom in the unitcell.

property base_neighbors

np.ndarray: The neighbors of the unitcell at the origin.

property distances

np.ndarray: List of distances between the lattice sites.

add_atom(*pos=None*, *atom=None*, *primitive=False*, *neighbors=0*, ***kwargs*)

Adds a site to the basis of the lattice unit cell.

Parameters

pos
[(N) array_like or float, optional] Position of site in the unit-cell. The default is the origin of the cell. The size of the array has to match the dimension of the lattice.

atom
[str or dict or Atom, optional] Identifier of the site. If a string is passed, a new Atom instance is created.

primitive
[bool, optional] Flag if the specified position is in cartesian or lattice coordinates. If True the passed position will be multiplied with the lattice vectors. The default is `False` (cartesian coordinates).

neighbors
[int, optional] The number of neighbor distance to calculate. If the number is 0 the distances have to be calculated manually after configuring the lattice basis.

****kwargs**
Keyword arguments for ´Atom´ constructor. Only used if a new `Atom` instance is created.

Returns

atom
[Atom] The `Atom`-instance of the newly added site.

Raises

ValueError
Raised if the dimension of the position does not match the dimension of the lattice.

ConfigurationError
Raised if the position of the new atom is already occupied.

Examples

Construct a square lattice

```
>>> latt = LatticeStructure(np.eye(2))
```

Create an atom and add it to the origin of the unit cell of the lattice

```
>>> atom1 = Atom(name="A")
>>> latt.add_atom([0.0, 0.0], atom=atom1)
>>> latt.get_atom(0)
Atom(A, size=10, 0)
```

An `Atom` instance can also be created by passing the name of the (new) atom and optional keyword arguments for the constructor:

```
>>> latt.add_atom([0.5, 0.5], atom="B", size=15)
>>> latt.get_atom(1)
Atom(B, size=15, 1)
```

get_alpha(*atom*)

Returns the index of the atom in the unit-cell.

Parameters

atom
[int or str or Atom] The argument for getting the atom. If a `int` is passed it is interpreted as the index, if a `str` is passed as the name of an atom.

Returns

alpha
[int or list of int] The atom indices. If a string was passed multiple atoms with the same name can be returned as list.

Examples

Construct a lattice with two identical atoms and a third atom in the unit cell:

```
>>> latt = LatticeStructure(np.eye(2))
>>> latt.add_atom([0, 0], atom="A")
>>> latt.add_atom([0.5, 0], atom="B")
>>> latt.add_atom([0.5, 0.5], atom="B")
```

Get the atom index of atom A:

```
>>> latt.get_alpha("A")
[0]
```

Get the indices of the atoms B:

```
>>> latt.get_alpha("B")
[1, 2]
```

Since there are two atoms B in the unit cell of the lattice both indices are returned.

get_atom(*atom*)

Returns the `Atom` instance of the given atom in the unit cell.

Parameters

atom
[int or str or Atom] The argument for getting the atom. If a `int` is passed it is interpreted as the index, if a `str` is passed as the name of an atom.

Returns

atom
[Atom] The `Atom` instance of the given site.

See also:

get_alpha
Get the index of the given atom.

Examples

Construct a lattice with one atom in the unit cell

```
>>> latt = LatticeStructure(np.eye(2))
>>> latt.add_atom([0, 0], atom="A")
```

Get the atom instance by the name

```
>>> latt.get_atom("A")
Atom(A, size=10, 0)
```

or by the index:

```
>>> latt.get_atom(0)
Atom(A, size=10, 0)
```

add_connection(*atom1*, *atom2*, *num_distances=1*, *analyze=False*)

Sets the number of distances for a specific connection between two atoms.

Parameters

atom1
[int or str or Atom] The first atom of the connected pair.

atom2
[int or str or Atom] The second atom of the connected pair.

num_distances
[int, optional] The number of neighbor-distance levels, e.g. setting to 1 means only nearest neighbors. The default are nearest neighbor connections.

analyze
[bool, optional] If True the lattice basis is analyzed after adding connections. If `False` the `analyze`-method needs to be called manually. The default is `False`.

See also:

add_connections
Set up connections for all atoms in the unit cell in one call.

analyze
Called after setting up all the lattice connections.

Examples

Construct a square lattice with two atoms, A and B, in the unit cell:

```
>>> latt = LatticeStructure(np.eye(2))
>>> latt.add_atom([0.0, 0.0], atom="A")
>>> latt.add_atom([0.5, 0.5], atom="B")
```

Set next nearest and nearest neighbors between the A atoms:

```
>>> latt.add_connection("A", "A", num_distances=2)
```

Set nearest neighbors between A and B:

```
>>> latt.add_connection("A", "B", num_distances=1)
```

Set nearest neighbors between the B atoms:

```
>>> latt.add_connection("B", "B", num_distances=1)
```

add_connections(*num_distances=1*, *analyze=True*)

Sets the number of distances for all possible atom-pairs of the unitcell.

Parameters

num_distances
[int, optional] The number of neighbor-distance levels, e.g. setting to 1 means only nearest neighbors. The default are nearest neighbor connections.

analyze
[bool, optional] If True the lattice basis is analyzed after adding connections. If `False` the `analyze`-method needs to be called manually. The default is True.

See also:

add_connection
Set up connection between two specific atoms in the unit cell.

analyze
Called after setting up all the lattice connections.

Examples

Construct a square lattice with one atom in the unit cell:

```
>>> latt = LatticeStructure(np.eye(2))
>>> latt.add_atom()
```

Set nearest neighbor hopping:

```
>>> latt.add_connections(num_distances=1)
```

analyze()

Analyzes the structure of the lattice and stores neighbor data.

Check's distances between all sites of the bravais lattice and saves the `n` lowest values. The neighbor lattice-indices of the unit-cell are also stored for later use. This speeds up many calculations like finding nearest neighbors.

Raises

NoAtomsError
Raised if no atoms where added to the lattice. The atoms in the unit cell are needed for computing the neighbors and distances of the lattice.

NoConnectionsError
Raised if no connections have been set up.

Notes

Before calling the `analyze` function all connections in the lattice have to be set up.

Examples

Construct a square lattice with one atom in the unit cell and nearest neighbors:

```
>>> latt = LatticeStructure(np.eye(2))
>>> latt.add_atom()
>>> latt.add_connections(num_distances=1)
```

Call *analyze* after setting up the connections

```
>>> latt.analyze()
```

get_position(*nvec=None*, *alpha=0*)

Returns the position for a given translation vector `nvec` and atom `alpha`.

Parameters

nvec
[(N) array_like or int] The translation vector.

alpha
[int, optional] The atom index, default is 0.

Returns

pos
[(N) np.ndarray] The position of the transformed lattice site.

Examples

```
>>> latt = LatticeStructure(np.eye(2))
>>> latt.add_atom()
>>> latt.add_connections(1, analyze=True)
>>> latt.get_position([1, 0], alpha=0)
[1. 0.]
```

get_positions(*indices*)

Returns the positions for multiple lattice indices.

Parameters

indices
[(N, D+1) array_like or int] List of lattice indices in the format $(n_1, ..., n_d,)$.

Returns

pos
[(N, D) np.ndarray] The positions of the lattice sites.

Examples

```
>>> latt = LatticeStructure(np.eye(2))
>>> latt.add_atom()
>>> latt.add_connections(1)
>>> ind = [[0, 0, 0], [1, 0, 0], [1, 1, 0]]
>>> latt.get_positions(ind)
[[0. 0.]
 [1. 0.]
 [1. 1.]]
```

estimate_index(*pos*)

Returns the nearest matching lattice index (n, alpha) for a position.

Parameters

pos
[array_like or float] The position of the site in world coordinates.

Returns

nvec
[np.ndarray] The estimated translation vector n.

Examples

```
>>> latt = LatticeStructure(np.eye(2))
>>> latt.add_atom()
>>> latt.add_connections(1)
>>> latt.estimate_index([1.2, 0.2])
[1 0]
```

get_neighbors(*nvec=None*, *alpha=0*, *distidx=0*)

Returns the neighour indices of a given site by transforming neighbor data.

Parameters

nvec: (D) array_like or int, optional
The translation vector, the default is the origin.

alpha: int, optional
The atom index, default is 0.

distidx: int, optional
The index of distance to the neighbors, default is 0 (nearest neighbors).

Returns

indices: (N, D+1) np.ndarray
The lattice indices of the neighbor sites.

Raises

NotAnalyzedError
Raised if the lattice distances and base-neighbors haven't been computed.

Examples

```
>>> latt = LatticeStructure(np.eye(2))
>>> latt.add_atom()
>>> latt.add_connections(1)
>>> latt.get_neighbors(nvec=[0, 0], alpha=0, distidx=0)
[[ 1  0  0]
 [ 0 -1  0]
 [-1  0  0]
 [ 0  1  0]]
```

get_neighbor_positions(*nvec=None*, *alpha=0*, *distidx=0*)

Returns the neighour positions of a given site by transforming neighbor data.

Parameters

nvec: (D) array_like or int, optional
The translation vector, the default is the origin.

alpha: int, optional
The site index, default is 0.

distidx: int, default
The index of distance to the neighbors, default is 0 (nearest neighbors).

Returns

positions: (N, D) np.ndarray
The positions of the neighbor sites.

Raises

NotAnalyzedError
Raised if the lattice distances and base-neighbors haven't been computed.

Examples

```
>>> latt = LatticeStructure(np.eye(2))
>>> latt.add_atom()
>>> latt.add_connections(1)
>>> latt.get_neighbor_positions(nvec=[0, 0], alpha=0, distidx=0)
[[ 1.  0.]
 [ 0. -1.]
 [-1.  0.]
 [ 0.  1.]]
```

get_neighbor_vectors(*alpha=0*, *distidx=0*, *include_zero=False*)

Returns the vectors to the neigbor sites of an atom in the unit cell.

Parameters

alpha
[int, optional] Index of the base atom. The default is the first atom in the unit cell.

distidx
[int, default] Index of distance to the neighbors, default is 0 (nearest neighbors).

include_zero
[bool, optional] Flag if zero-vector is included in result. The default is `False`.

Returns

vectors
[np.ndarray] The vectors from the site of the atom $alpha$ to the neighbor sites.

Raises

NotAnalyzedError
Raised if the lattice distances and base-neighbors haven't been computed.

Examples

```
>>> latt = LatticeStructure(np.eye(2))
>>> latt.add_atom()
>>> latt.add_connections(1)
>>> latt.get_neighbor_vectors(alpha=0, distidx=0)
[[ 1.  0.]
 [ 0. -1.]
 [-1.  0.]
 [ 0.  1.]]
```

`fourier_weights`(*k*, *alpha=0*, *distidx=0*)

Returns the Fourier-weight for a given vector.

Parameters

k
[array_like] The wavevector to compute the lattice Fourier-weights.

alpha
[int, optional] Index of the base atom. The default is the first atom in the unit cell.

distidx
[int, default] Index of distance to the neighbors, default is 0 (nearest neighbors).

Returns

weight
[np.ndarray]

`get_base_atom_dict`(*atleast2d=True*)

Returns a dictionary containing the positions for eatch of the base atoms.

Parameters

atleast2d
[bool, optional] If True, one-dimensional coordinates will be cast to 2D vectors.

Returns

atom_pos
[dict] The positions of the atoms as a dictionary.

Examples

```
>>> latt = LatticeStructure(np.eye(2))
>>> latt.add_atom([0, 0], atom="A")
>>> latt.add_atom([0.5, 0], atom="B")
>>> latt.add_atom([0.5, 0.5], atom="B")
>>> latt.get_base_atom_dict()
{
    Atom(A, radius=0.2, 0): [array([0, 0])],
    Atom(B, radius=0.2, 1): [array([0.5, 0. ]), array([0.5, 0.5])]
}
```

check_points(*points*, *shape*, *relative=False*, *pos=None*, *tol=0.001*)

Returns a mask for the points in the given shape.

Parameters

points: (M, N) np.ndarray
The points in cartesian coordinates.

shape: (N) array_like or int or AbstractShape
shape of finite size lattice to build.

relative: bool, optional
If True the shape will be multiplied by the cell size of the model. The default is True.

pos: (N) array_like or int, optional
Optional position of the section to build. If `None` the origin is used.

tol: float, optional
The tolerance for checking the points. The default is `1e-3`.

Returns

mask: (M) np.ndarray
The mask for the points inside the shape.

Examples

```
>>> latt = LatticeStructure(np.eye(2))
>>> shape = (2, 2)
>>> points = np.array([[0, 0], [2, 2], [3, 2]])
>>> latt.check_points(points, shape)
[ True  True False]
```

build_translation_vectors(*shape*, *primitive=False*, *pos=None*, *check=True*, *dtype=None*, *oversample=0.0*)

Constructs the translation vectors n in a given shape.

Parameters

shape: (N) array_like or int
shape of finite size lattice to build.

primitive: bool, optional
If True the shape will be multiplied by the cell size of the model. The default is True.

pos: (N) array_like or int, optional
Optional position of the section to build. If `None` the origin is used.

check: bool, optional
If *True* the positions of the translation vectors are checked and filtered. The default is True. This should only be disabled if filtered later.

dtype: int or np.dtype, optional
Optional data-type for storing the lattice indices. By default, the given limits are checked to determine the smallest possible data-type.

oversample: float, optional
Faktor for upscaling limits for initial index grid. This ensures that all positions are included. Only needed if corner points are missing. The default is 0.

Returns

nvecs: (M, N) np.ndarray
The translation-vectors in lattice-coordinates.

Raises

ValueError
Raised if the dimension of the position doesn't match the dimension of the lattice.

Examples

```
>>> latt = LatticeStructure(np.eye(2))
>>> latt.build_translation_vectors((2, 2))
[[0 0]
 [0 1]
 [0 2]
 [1 0]
 [1 1]
 [1 2]
 [2 0]
 [2 1]
 [2 2]]
```

`build_indices`(*shape*, *primitive=False*, *pos=None*, *check=True*, *callback=None*, *dtype=None*, *return_pos=False*)

Constructs the lattice indices .math:*(n,)* in the given shape.

Parameters

shape: (N) array_like or int or AbstractShape
shape of finite size lattice to build.

primitive: bool, optional
If True the shape will be multiplied by the cell size of the model. The default is True.

pos: (N) array_like or int, optional
Optional position of the section to build. If `None` the origin is used.

check
[bool, optional] If True the positions of the translation vectors are checked and filtered. The default is True. This should only be disabled if filtered later.

callback: callable, optional
Optional callable for filtering sites. The indices and positions are passed as arguments.

dtype: int or str or np.dtype, optional
Optional data-type for storing the lattice indices. By default, the given limits are checked to determine the smallest possible data-type.

return_pos: bool, optional
Flag if positions should be returned with the indices. This can speed up the building process, since the positions have to be computed here anyway. The default is `False`.

Returns

indices: (M, N+1) np.ndarray
The lattice indices of the sites in the format .math:*(n_1, …, n_d,)*.

positions: (M, N) np.ndarray
Corresponding positions. Only returned if `return_positions=True`.

Raises

ValueError
Raised if the dimension of the position doesn't match the dimension of the lattice.

Examples

Build indices of a linear chain with two atoms in the unit cell:

```
>>> latt = LatticeStructure(np.eye(2))
>>> latt.add_atom([0.0, 0.0], "A")
>>> latt.add_atom([0.5, 0.5], "B")
>>> indices, positions = latt.build_indices((2, 1), return_pos=True)
```

The indices contain the translation vector and the atom index

```
>>> indices
[[0 0 0]
 [0 0 1]
 [0 1 0]
 [1 0 0]
 [1 0 1]
 [1 1 0]
 [2 0 0]
 [2 1 0]]
```

The positions are the positions of the atoms in the same order of the indices:

```
>>> positions
[[0.  0. ]
 [0.5 0.5]
 [0.  1. ]
 [1.  0. ]
 [1.5 0.5]
 [1.  1. ]
 [2.  0. ]
 [2.  1. ]]
```

compute_neighbors(*indices*, *positions*, *num_jobs=1*)

Computes the neighbors for the given points.

Parameters

indices
[(N, D+1) array_like] The lattice indices of the sites in the format .math:*(n_1, ..., n_D,)* where N is the number of sites and D the dimension of the lattice.

positions
[(N, D) array_like] The positions of the sites in cartesian coordinates where N is the number of sites and D the dimension of the lattice.

num_jobs
[int, optional] Number of jobs to schedule for parallel processing. If `-1` is given all processors are used. The default is 1.

Returns

neighbors: (N, M) np.ndarray
The indices of the neighbors in `positions`. M is the maximum number of neighbors previously computed in the `analyze` method.

distances: (N, M) np.ndarray
The corresponding distances of the neighbors.

See also:

analyze
Used to pre-compute the base neighbors of the unit cell.

Examples

Construct indices of a one dimensional lattice:

```
>>> latt = LatticeStructure(1)
>>> latt.add_atom()
>>> latt.add_connections()
>>> indices, positions = latt.build_indices(3, return_pos=True)
>>> positions
[[0.]
 [1.]
 [2.]
 [3.]]
```

Compute the neighbors of the constructed sites

```
>>> neighbors, distances = latt.compute_neighbors(indices, positions)
>>> neighbors
[[1 4]
 [2 0]
 [3 1]
 [2 4]]
```

```
>>> indices
[[ 1. inf]
```

(continues on next page)

(continued from previous page)

```
[ 1.  1.]
[ 1.  1.]
[ 1. inf]]
```

The neighbor indices and distances of sites with less than the maximum number of neighbors are filled up with an invalid index (here: 4) and `np.inf` as distance.

copy()

LatticeStructure : Creates a (deep) copy of the lattice instance.

todict()

Creates a dictionary containing the information of the lattice instance.

Returns

d
[dict] The information defining the current instance.

classmethod fromdict(*d*)

Creates a new instance from information stored in a dictionary.

Parameters

d
[dict] The information defining the current instance.

Returns

latt
[LatticeStructure] The restored lattice instance.

dumps()

Creates a string containing the information of the lattice instance.

Returns

s
[str] The information defining the current instance.

dump(*file*)

Save the data of the `LatticeStructure` instance.

Parameters

file
[str or int or bytes] File name to store the lattice. If `None` the hash of the lattice is used.

classmethod load(*file*)

Load data of a saved `LatticeStructure` instance.

Parameters

file
[str or int or bytes] File name to load the lattice.

Returns

latt
[LatticeStructure] The lattice restored from the file content.

plot_cell(*lw=None*, *alpha=0.5*, *cell_lw=None*, *cell_ls='--'*, *margins=0.1*, *legend=None*, *grid=False*, *show_cell=True*, *show_vecs=True*, *show_neighbors=True*, *con_colors=None*, *adjustable='box'*, *ax=None*, *show=False*)

Plot the unit cell of the lattice.

Parameters

lw
[float, optional] Line width of the neighbor connections.

alpha
[float, optional] The alpha value of the neighbor sites.

cell_lw
[float, optional] The line width used for plotting the unit cell outlines.

cell_ls
[str, optional] The line style used for plotting the unit cell outlines.

margins
[Sequence[float] or float, optional] The margins of the plot.

legend
[bool, optional] Flag if legend is shown.

grid
[bool, optional] If True, draw a grid in the plot.

show_neighbors
[bool, optional] If True the neighbors are plotted.

show_vecs
[bool, optional] If True the first unit-cell is drawn.

show_cell
[bool, optional] If True the outlines of the unit cell are plotted.

con_colors
[Sequence[tuple], optional] list of colors to override the defautl connection color. Each element has to be a tuple with the first two elements being the atom indices of the pair and the third element the color, for example `[('A', 'A', 'r')]`.

adjustable
[None or {'box', 'datalim'}, optional] If not None, this defines which parameter will be adjusted to meet the equal aspect ratio. If 'box', change the physical dimensions of the Axes. If 'datalim', change the x or y data limits. Only applied to 2D plots.

ax
[plt.Axes or plt.Axes3D or None, optional] Parent plot. If None, a new plot is initialized.

show
[bool, optional] If True, show the resulting plot.

CHAPTER

FOURTEEN

LATTPY.UTILS

Contains miscellaneous utility methods.

exception lattpy.utils.**LatticeError**

Bases: Exception

exception lattpy.utils.**ConfigurationError**

Bases: *LatticeError*

property msg

property hint

exception lattpy.utils.**SiteOccupiedError**(*atom*, *pos*)

Bases: *ConfigurationError*

exception lattpy.utils.**NoAtomsError**

Bases: *ConfigurationError*

exception lattpy.utils.**NoConnectionsError**

Bases: *ConfigurationError*

exception lattpy.utils.**NotAnalyzedError**

Bases: *ConfigurationError*

exception lattpy.utils.**NotBuiltError**

Bases: *ConfigurationError*

lattpy.utils.**min_dtype**(*a*, *signed=True*)

Returns the minimum required dtype to store the given values.

Parameters

a
[array_like] One or more values for determining the dtype. Should contain the maximal expected values.

signed
[bool, optional] If *True* the dtype is forced to be signed. The default is *True*.

Returns

dtype
[dtype] The required dtype.

lattpy.utils.**chain**(*items*, *cycle=False*)

Creates a chain between items

Parameters

items
[Sequence] items to join to chain

cycle
[bool, optional] cycle to the start of the chain if True, default: False

Returns

chain: list
chain of items

lattpy.utils.**create_lookup_table**(*array*, *dtype=<class 'numpy.uint8'>*)

Converts the given array to an array of indices linked to the unique values.

Parameters

array
[array_like]

dtype
[int or np.dtype, optional] Optional data-type for storing the indices of the unique values. By default *np.uint8* is used, since it is assumed that the input-array has only a few unique values.

Returns

values
[np.ndarray] The unique values occuring in the input-array.

indices
[np.ndarray] The corresponding indices in the same shape as the input-array.

lattpy.utils.**frmt_num**(*num*, *dec=1*, *unit=''*, *div=1000.0*)

Returns a formatted string of a number.

Parameters

num
[float] The number to format.

dec
[int, optional] Number of decimals. The default is 1.

unit
[str, optional] Optional unit suffix. By default no unit-strinmg is used.

div
[float, optional] The divider used for units. The default is 1000.

Returns

num_str: str

lattpy.utils.**frmt_bytes**(*num*, *dec=1*)

Returns a formatted string of the number of bytes.

lattpy.utils.**frmt_time**(*seconds*, *short=False*, *width=0*)

Returns a formated string for a given time in seconds.

Parameters

seconds
[float] Time value to format

short
[bool, optional] Flag if short representation should be used.

width
[int, optional] Optional minimum length of the returned string.

Returns

time_str: str

CHAPTER

FIFTEEN

WHAT'S NEW

15.1 0.7.7 - 2022-02-11

15.1.1 New Features

- add additional spatial methods to lattice object
- reuse coordinate system argument used for building lattice in other `Lattice`-methods

15.1.2 Improvements/Bug Fixes

- remove deprecated `set_num_neighbors` method
- remove deprecated `fill` method from `DataMap`
- remove deprecated `get_neighbor_pos` method from `LatticeData`

15.2 0.7.6 - 2022-12-06

15.2.1 New Features

- add method for getting limits of unit cells to `LatticeData` object
- add conversion methods between cell index and super index for regular shapes to `LatticeBasis`
- add `hypercubic` constructor to `LatticeBasis` object.
- add `np.zeros` wrapper to `DataMap`

15.2.2 Improvements/Bug Fixes

- rename index methods of `Lattice` object to use superindex naming convention
- cast `distidx` to full numpy array instead of list of arrays
- replace deprecated `np.bool` type with the builtin `bool`
- add endpoint argument to WignerSeitzCell meshgrid method
- add endpoint argument to linspace of `WignerSeitzCell`

15.2.3 BREAKING CHANGE

`index_from_position` and `index_from_lattice_index` have been renamed to `superindex_from_pos` and `superindex_from_index`.

15.3 0.7.5 - 2022-25-05

15.3.1 Improvements/Bug Fixes

- fix error when setting periodic neighbors twice
- set periodic axes only if size is big enough (#67)

15.4 0.7.4 - 2022-10-05

15.4.1 New Features

- add method `neighbor_pairs` for generating a list of neighbor indices

15.4.2 Documentation

- add example to `adjacency_matrix`
- add docstring to `neighbor_pairs`

15.5 0.7.3 - 2022-06-05

15.5.1 Improvements/Bug Fixes

- `adjacency_matrix` is now vectorized and returns a `csr_matrix`
- passing a False boolean as axis to `set_periodic` now removes the periodic boundaries

15.6 0.7.2 - 2022-04-05

15.6.1 New Features

- add prefabs for the hexagonal (triangular) and honeycomb lattice.
- add methods for building sparse matrices to `DataMap` class

15.6.2 Improvements/Bug Fixes

- add argument for building in primitive basis to the `finite_hypercubic` method.

15.7 0.7.1 - 2022-29-03

15.7.1 New Features

- add argument for setting periodic boundary conditions to the `finite_hypercubic` method.
- add method for computing minimum distances in a lattice with periodic boundary conditions
- add `shape` keyword argument to Lattice constructor
- add CSR/BSR sparse matrix format of indices and index-pointers to DataMap

15.7.2 Code Refactoring

- rename `distance` variables to `distances_` to prevent same name as method

15.8 0.7.0 - 2022-21-02

15.8.1 New Features

- Add method for computing the adjacency matrix of the lattice graph
- Split the lattice structure into separate object `LatticeStructure` and use it as base class for `Lattice`
- Split the lattice basis into separate object `LatticeBasis` and use it as base class for `Lattice`

15.8.2 Code Refactoring

- use black code formatter

15.8.3 Documentation

- add inheritance diagram to `LatticeStructure` and fix some docstrings
- add inheritance diagram to `Lattice`
- add example to `LatticeBasis` docstring
- add attributes to docstring of `LatticeBasis`
- improve docstring of `Lattice` class

15.9 0.6.7 - 2022-16-02

15.9.1 New Features

- add method for hiding the box and axis of a plot
- Add `finite_hypercubic` lattice prefab
- use git-chglog to auto-generate changelogs

15.9.2 Improvements/Bug Fixes

- add `use_mplstyle` to plotting module
- change atom parameter order and fix resulting errors
- use `box` for plot aspect ratio
- improve lattice plotting and fix scaling/auto-limit issues
- update change log template and include old entries

15.9.3 Code Refactoring

- rename unitcell module to atom

15.9.4 Documentation

- fix limits of plot in configuration tutorial
- update index page of docs
- fix docstrings of `DataMap`
- add hamiltonian section to tutorial
- add change log contributing to documentation

15.10 0.6.6 - 2022-12-02

15.10.1 Improved/Fixed

- improve build process
- improve periodic neighbor computation
- improve documentation
- improve CI/Tests
- minor fixes

15.11 0.6.5 - 2022-03-02

15.11.1 New Features

- 2D/3D `Shape` object for easier lattice construction.
- repeat/extend built lattices.

15.11.2 Improved/Fixed

- improve build process
- improve periodic neighbor computation (still not stable)
- add/improve tests
- improve plotting
- add more docstrings
- fix multiple bugs

CHAPTER

SIXTEEN

CONTRIBUTING

Thank you for contributing to `lattpy` :tada:

16.1 Pre-commit Hooks

We are using the pre-commit framework to automatically run some checks and the Black code formatter at commit time. This ensures that every commit fulfills the basic requirements to be mergeable and follows the coding style of the project.

The pre-commit hooks can be installed via

```
$ pre-commit install
pre-commit installed at .git/hooks/pre-commit
```

16.2 Commit Message Format

A format influenced by Angular commit message.

```
<type>: <subject>
<BLANK LINE>
<body>
<BLANK LINE>
<footer>
```

16.2.1 Type

Must be one of the following:

- **feat:** A new feature
- **fix:** Bug fixes or improvements
- **perf:** A code change that improves performance
- **refactor:** Code refactoring
- **ci:** Changes to CI configuration files and scripts
- **docs:** Documention changes
- **build:** Updating Makefile etc, no production code changes

- **test:** Adding missing tests or correcting existing tests
- **update** Other configurations updates
- **auto** Mostly used by automatic commits (for example from GitHub workflows)

16.2.2 Subject

Use the summary field to provide a succinct description of the change:

- use the imperative, present tense: "change" not "changed" nor "changes"
- don't capitalize the first letter
- no dot (.) at the end

16.2.3 Body (optional)

Just as in the summary, use the imperative, present tense: "fix" not "fixed" nor "fixes".

Explain the motivation for the change in the commit message body. This commit message should explain why you are making the change. You can include a comparison of the previous behavior with the new behavior in order to illustrate the impact of the change.

16.2.4 Footer (optional)

The footer can contain information about breaking changes and deprecations and is also the place to reference GitHub issues, Jira tickets, and other PRs that this commit closes or is related to. For example:

```
BREAKING CHANGE: <breaking change summary>
<BLANK LINE>
<breaking change description + migration instructions>
<BLANK LINE>
<BLANK LINE>
Fixes #<issue number>
```

or

```
DEPRECATED: <what is deprecated>
<BLANK LINE>
<deprecation description + recommended update path>
<BLANK LINE>
<BLANK LINE>
Closes #<pr number>
```

Breaking Change section should start with the phrase "BREAKING CHANGE: " followed by a summary of the breaking change, a blank line, and a detailed description of the breaking change that also includes migration instructions.

Similarly, a Deprecation section should start with "DEPRECATED: " followed by a short description of what is deprecated, a blank line, and a detailed description of the deprecation that also mentions the recommended update path.

CHAPTER

SEVENTEEN

INDICES AND TABLES

- genindex
- modindex
- search

PYTHON MODULE INDEX

l

INDEX

D

E

F

G

H

I

K

L

M

N

O

P

Q

R

S

T

V

W

X

Z