Tyjal DeWolf-Moura

MBSE 2019, spring 2024.

The Tight Binding Model, an Introduction in Python.

Introduction

I created Python code for this project that allows for simulations using the tight-binding model. The tight binding, or linear combination of atomic orbitals, model is an approximate quantum mechanical model of electron states commonly used to simulate the band diagram structure of crystal solids. The model assumes that the basis states of the electrons are approximately the same as the atomic orbitals of the atom on its own in the cell, ignoring the influence of the potentials of other nearby atoms. The Schrodinger equation of the electron in a lattice of atoms can be reduced to a series of linear equations using the tight binding assumption. In this series of linear equations, each state has an onsite energy term, and each pair has a hopping term. A matrix is then constructed from these linear equations and can be diagonalized to find the relation between the energy and wave vector, allowing for the construction of band diagrams. For this project, I created Python code from scratch that, given a series of orbital positions, onsite terms, hopping terms, and points in reciprocal space, will construct the tight-binding Hamiltonian of the system, plot out a path in k space, and create band diagrams for that path by solving the Hamiltonian around it.

In this paper, I will explain the tight-binding model, with appropriate background, how I wrote the code to use this model, how to use the code, and several examples of inputs to the

model and the produced band diagrams. Throughout this code, we will follow the formalism outlined in [*Formalism—PythTB 1.8.0 documentation*], as this project is essentially an attempt to recreate the functionality of this program from scratch. Many of the parameters of our examples were also taken from this project so we could compare our results to theirs.

A Jupyter notebook or PDF of a Jupyter notebook containing the relevant code should be attached to and viewed alongside this document.

Mathematical formalism

We will follow the same mathematical formalism used in [*Formalism—PythTB 1.8.0 documentation*]; however, we will expand on it in slightly more depth.

We may define our crystal lattice using its lattice vectors, a_1, a_2, and a_3, in three dimensions. Any given crystal unit cell may then be referred to by the number of units cells away from the origin it is along each of these lattice vectors, represented by lattice coordinate vector:

$$\vec{\mathbf{R}} = n_1\vec{\mathbf{a}}_1 + n_2\vec{\mathbf{a}}_2 + n_3\vec{\mathbf{a}}_3$$

Where the n's are integers. The lattice coordinate vector will be the zero vector for the unit cell at the origin. The reciprocal lattice

$$\vec{\mathbf{G}} = m_1\vec{\mathbf{b}}_1 + m_2\vec{\mathbf{b}}_2 + m_3\vec{\mathbf{b}}_3$$

is then defined such that its vectors, b_i, relate to the real-space lattice vectors by the relation:

$$\vec{\mathbf{a}}_i \cdot \vec{\mathbf{b}}_j = 2\pi\delta_{ij}$$

Meaning also that

$$\exp(\mathbf{i}\vec{\mathbf{G}} \cdot \vec{\mathbf{R}}) = 1$$

Note that we use the symbol i to refer to the complex number, as we will use "i" to refer to indices later. In three dimensions, these reciprocal lattice vectors can be found from the real-space lattice vectors by the following formula

$$\mathbf{b}_1 = \frac{2\pi \vec{\mathbf{a}}_2 \times \vec{\mathbf{a}}_3}{\vec{\mathbf{a}}_1 \cdot (\vec{\mathbf{a}}_2 \times \vec{\mathbf{a}}_3)},\ \mathbf{b}_2 = \frac{2\pi \mathbf{a}_3 \times \mathbf{a}_1}{\vec{\mathbf{a}}_1 \cdot (\vec{\mathbf{a}}_2 \times \vec{\mathbf{a}}_3)},\ \mathbf{b}_3 = \frac{2\pi \mathbf{a}_1 \times \mathbf{a}_2}{\vec{\mathbf{a}}_1 \cdot (\vec{\mathbf{a}}_2 \times \vec{\mathbf{a}}_3)}$$

A different formula can be used for two dimensions. However, the previous formula also works if one defines one of the lattice vectors as a unit vector orthogonal to the others. Remembering that the a_i and b_i vectors are not necessarily cartesian or unit vectors is crucial. A vector in reciprocal space, such as the wavevector of the electron, can be written in either cartesian coordinates

$$\vec{\mathbf{k}} = k_x \hat{x} + k_y \hat{y} + k_z \hat{z}$$

or reciprocal lattice coordinates

$$\vec{\mathbf{k}} = k_1 \vec{\mathbf{b}}_1 + k_2 \vec{\mathbf{b}}_2 + k_3 \vec{\mathbf{b}}_3$$

Throughout our formalism and code, we will use the real and reciprocal lattice coordinates, referred to as "internal coordinates," to differentiate them from cartesian coordinates.

We now begin constructing our state formalism. The tight binding model assumes that the potentials of atoms in the unit cell overlap only minimally, such that the states of the electron around an atom in the lattice are the same as those around the atom in the single-atom case. Further, it is assumed that the electrons are tightly bound to each atom enough that they do not interact. Our model is, in a sense, a single electron model. Other electrons only affect the state of the electron so much as their presence dictates which orbital states we include in our simulation. We can define one of these orbital states in the unit cell as

$$\varphi_i(\vec{\mathbf{r}} - \vec{\mathbf{R}} - \vec{\mathbf{t}}_i) = \phi_{\vec{\mathbf{R}}i}(\vec{\mathbf{r}})$$

Or in bra-ket notation

$$|\phi_{\vec{\mathbf{R}}i}\rangle$$

Referring to orbital "i" in unit cell $\vec{R}$ around the atom at position

$$\vec{\mathbf{t}}_i = a_{i,1}\vec{\mathbf{a}}_1 + a_{i,2}\vec{\mathbf{a}}_2 + a_{i,3}\vec{\mathbf{a}}_3$$

In the unit cell. As part of the tight binding assumption, we assume that these states are orthonormal, both between states in the unit cell and across unit cells

$$\langle\phi_{\vec{\mathbf{R}}i}|\phi_{\vec{\mathbf{R}}'j}\rangle = \delta_{\vec{R}\vec{R}'}\delta_{ij}$$

As our crystal lattice has translational symmetry, we can reasonably assume that our Hamiltonian also has translation symmetry such that the inner product with the Hamiltonian of any two orbitals in any two unit cells is

$$\langle\phi_{\vec{\mathbf{R}}'i}|\,\hat{H}\,|\phi_{\vec{\mathbf{R}}'+\vec{\mathbf{R}}j}\rangle = \langle\phi_{\vec{\mathbf{0}}i}|\,\hat{H}\,|\phi_{\vec{\mathbf{R}}j}\rangle = H_{ij}(\vec{\boldsymbol{R}})$$

This translational symmetry means we can treat the unit cell of the first state as always the origin unit cell. These will be the hopping and onsite terms we will eventually input into our code. It is important to note that as the Hamiltonian is hermitian, our reverse hopping terms can be found simply from the forward hopping terms by taking the complex conjugate

$$\langle\phi_{\vec{\mathbf{0}}j}|\,\hat{H}\,|\phi_{-\vec{\mathbf{R}}i}\rangle = H_{ji}(-\vec{\boldsymbol{R}}) = H_{ij}(\vec{\boldsymbol{R}})^* = (\langle\phi_{\vec{\mathbf{0}}i}|\,\hat{H}\,|\phi_{\vec{\mathbf{R}}j}\rangle)^*$$

We may also assume that the value of our Hamiltonian decays rapidly with increasing $\vec{R}$, which allows us only to consider nearest-neighbor interactions, which we will do in our examples later. However, this assumption is unnecessary for our math formulas or code.

The next step of applying the tight binding model is transitioning to the Bloch basis.

The Bloch basis functions are constructed from a linear combination of the atomic orbital states, with each term differing by only a position and k-vector-based phase.

$$|\chi_i^{\vec{k}}\rangle = \sum_{\vec{\boldsymbol{R}}} e^{\mathrm{i}\vec{\boldsymbol{k}}\cdot(\vec{\boldsymbol{R}}+\vec{\mathrm{t}}_\mathrm{i})}\,|\phi_{\vec{\mathbf{R}}i}\rangle$$

This Bloch state consists of a linear combination over all unit cells of the single orbital state "i" in the unit cell. We have as many different Bloch basis states for each wave vector as we have defined orbitals in the unit cell. The states are also all equally weighted in the basis we use, which would correspond to a delocalized electron having equal wavefunction amplitude, but not phase, in every unit cell. This equal weighting is because we can assume the electron wavefunction amplitude is periodic from cell to cell due to the periodic nature of crystals[Simon, S. H. (2013)].

The spatial normalization of the Bloch basis states is arbitrary for our purposes, but we can decide to normalize it to the volume of a single unit cell

$$\langle\chi_i^{\vec{k}}|\chi_i^{\vec{k}}\rangle = \iiint_{V_{cell}} \chi_i^{\vec{k}*}(\vec{r})\chi_i^{\vec{k}}(\vec{r})d^3r = 1$$

As a result of our early assumption of the orthogonality of our orbital states, our bloch states are also orthonormal

$$\langle\chi_i^{\vec{k}}|\chi_j^{\vec{k}}\rangle = \delta_{ij}$$

This basis will allow us to conveniently define our tight-binding Hamiltonian.

Now consider an arbitrary electron state composed of an arbitrary linear combination of bloch basis states

$$|\psi_{n,\vec{k}}\rangle = \sum_j C_{j,n,\vec{k}}\,|\chi_j^{\vec{k}}\rangle = \sum_j C_{j,n,\vec{k}} \sum_{\vec{\boldsymbol{R}}} e^{\mathrm{i}\vec{\boldsymbol{k}}\cdot(\vec{\boldsymbol{R}}+\vec{\mathrm{t}}_\mathrm{j})}\,|\phi_{\vec{\mathbf{R}}j}\rangle$$

If we apply the Hamiltonian operator to this arbitrary state and take the inner product with an arbitrary Bloch basis state, we begin the process of creating our Hamiltonian matrix

$$\hat{H}\left|\psi_{n,\vec{k}}\right\rangle = \sum_j C_{j,n,\vec{k}} \hat{H}\left|\chi_j^{\vec{k}}\right\rangle = \sum_j C_{j,n,\vec{k}} \sum_{\vec{\boldsymbol{R}}} e^{\mathrm{i}\vec{\boldsymbol{k}}\cdot(\vec{\boldsymbol{R}}+\vec{\mathrm{t}}_\mathrm{j})} \hat{H}\left|\phi_{\vec{\mathbf{R}}j}\right\rangle$$
$$\left\langle\chi_i^{\vec{k}}\right|\hat{H}\left|\psi_{n,\vec{k}}\right\rangle = \sum_j C_{j,n,\vec{k}} \left\langle\chi_i^{\vec{k}}\right|\hat{H}\left|\chi_j^{\vec{k}}\right\rangle = \sum_j C_{j,n,\vec{k}} \sum_{\vec{\boldsymbol{R}}} e^{\mathrm{i}\vec{\boldsymbol{k}}\cdot(\vec{\boldsymbol{R}}+\vec{\mathrm{t}}_\mathrm{j})} \left\langle\chi_i^{\vec{k}}\right|\hat{H}\left|\phi_{\vec{\mathbf{R}}j}\right\rangle$$

If we fully expand our Bloch basis, we can see that we have

$$\sum_j C_{j,n,\vec{k}} \sum_{\vec{\boldsymbol{R}}} e^{\mathrm{i}\vec{\boldsymbol{k}}\cdot(\vec{\boldsymbol{R}}+\vec{\mathrm{t}}_\mathrm{j})} \sum_{\vec{\boldsymbol{R}}'} e^{-\mathrm{i}\vec{\boldsymbol{k}}\cdot(\vec{\boldsymbol{R}}'+\vec{\mathrm{t}}_\mathrm{i}')} \left\langle\phi_{\vec{\mathbf{R}}'i}\right|\hat{H}\left|\phi_{\vec{\mathbf{R}}j}\right\rangle = \sum_j C_{j,n,\vec{k}} \sum_{\vec{\boldsymbol{R}}} \sum_{\vec{\boldsymbol{R}}'} e^{\mathrm{i}\vec{\boldsymbol{k}}\cdot(\vec{\boldsymbol{R}}-\vec{\boldsymbol{R}}'+\vec{\mathrm{t}}_\mathrm{j}-\vec{\mathrm{t}}_\mathrm{i})} \left\langle\phi_{\vec{\mathbf{R}}'i}\right|\hat{H}\left|\phi_{\vec{\mathbf{R}}j}\right\rangle$$

We can simply rewrite our lattice position vectors using $\vec{R} - \vec{R'} = \vec{R''}$, which allows us to rewrite our sum as

$$\sum_j C_{j,n,\vec{k}} \sum_{\vec{\boldsymbol{R}}} e^{\mathrm{i}\vec{\boldsymbol{k}}\cdot(\vec{\boldsymbol{R}}+\vec{\mathrm{t}}_\mathrm{j})} \sum_{\vec{\boldsymbol{R}}'} e^{-\mathrm{i}\vec{\boldsymbol{k}}\cdot(\vec{\boldsymbol{R}}'+\vec{\mathrm{t}}_\mathrm{i}')} \left\langle\phi_{\vec{\mathbf{R}}'i}\right|\hat{H}\left|\phi_{\vec{\mathbf{R}}j}\right\rangle = \sum_j C_{j,n,\vec{k}} \sum_{\vec{\boldsymbol{R}}} \sum_{\vec{\boldsymbol{R}}'} e^{\mathrm{i}\vec{\boldsymbol{k}}\cdot(\vec{\boldsymbol{R}}''+\vec{\mathrm{t}}_\mathrm{j}-\vec{\mathrm{t}}_\mathrm{i})} \left\langle\phi_{\vec{\mathbf{R}}'i}\right|\hat{H}\left|\phi_{\vec{\boldsymbol{R}}''+\vec{\boldsymbol{R}}'j}\right\rangle$$

As discussed earlier, our Hamiltonian has translational symmetry, which allows us to rewrite this as

$$= \sum_j C_{j,n,\vec{k}} \sum_{\vec{\boldsymbol{R}}} \sum_{\vec{\boldsymbol{R}}'} e^{\mathrm{i}\vec{\boldsymbol{k}}\cdot(\vec{\boldsymbol{R}}''+\vec{\mathrm{t}}_\mathrm{j}-\vec{\mathrm{t}}_\mathrm{i})} \left\langle\phi_{\vec{\mathbf{0}}i}\right|\hat{H}\left|\phi_{\vec{\boldsymbol{R}}''j}\right\rangle = \sum_j C'_{j,n,\vec{k}} \sum_{\vec{\boldsymbol{R}}''} e^{\mathrm{i}\vec{\boldsymbol{k}}\cdot(\vec{\boldsymbol{R}}''+\vec{\mathrm{t}}_\mathrm{j}-\vec{\mathrm{t}}_\mathrm{i})} \left\langle\phi_{\vec{\mathbf{0}}i}\right|\hat{H}\left|\phi_{\vec{\boldsymbol{R}}''j}\right\rangle$$

Where we have also combined our summation. Finally, As symbols and names are arbitrary, we will rename our $\vec{R}''$ vector $\vec{R}''$ to increase clarity. In summary, we then have found that

$$\left\langle\chi_i^{\vec{k}}\right|\hat{H}\left|\psi_{n,\vec{k}}\right\rangle = \sum_j C_{j,n,\vec{k}} \left\langle\chi_i^{\vec{k}}\right|\hat{H}\left|\chi_j^{\vec{k}}\right\rangle = \sum_j C_{j,n,\vec{k}} \sum_{\vec{\boldsymbol{R}}} e^{\mathrm{i}\vec{\boldsymbol{k}}\cdot(\vec{\boldsymbol{R}}+\vec{\mathrm{t}}_\mathrm{j}-\vec{\mathrm{t}}_\mathrm{i})} \left\langle\phi_{\vec{\mathbf{0}}i}\right|\hat{H}\left|\phi_{\vec{\boldsymbol{R}}j}\right\rangle =$$
$$\sum_j C_{j,n,\vec{k}} \sum_{\vec{\boldsymbol{R}}} e^{\mathrm{i}\vec{\boldsymbol{k}}\cdot(\vec{\boldsymbol{R}}+\vec{\mathrm{t}}_\mathrm{j}-\vec{\mathrm{t}}_\mathrm{i})} H_{ij}(\vec{\boldsymbol{R}})$$

Which is an eigenvalue problem we can solve by constructing and diagonalizing a Hamiltonian matrix with elements

$$H_{ij,\vec{k}} = \left\langle\chi_i^{\vec{k}}\right|\hat{H}\left|\chi_j^{\vec{k}}\right\rangle = \sum_{\vec{\boldsymbol{R}}} e^{\mathrm{i}\vec{\boldsymbol{k}}\cdot(\vec{\boldsymbol{R}}+\vec{\mathrm{t}}_\mathrm{j}-\vec{\mathrm{t}}_\mathrm{i})} \left\langle\phi_{\vec{\mathbf{0}}i}\right|\hat{H}\left|\phi_{\vec{\boldsymbol{R}}j}\right\rangle = \sum_j C_{j,n,\vec{k}} \sum_{\vec{\boldsymbol{R}}} e^{\mathrm{i}\vec{\boldsymbol{k}}\cdot(\vec{\boldsymbol{R}}+\vec{\mathrm{t}}_\mathrm{j}-\vec{\mathrm{t}}_\mathrm{i})} H_{ij}(\vec{\boldsymbol{R}})$$

As can be seen from this formula, the diagonal elements of this matrix are the onsite energy terms, real numbers, of each orbital, and the off-diagonal elements are the hopping terms, which are input into the code as any chosen complex number between each orbital summed over as many neighboring cells as one cares to include. In our test simulations, we limit ourselves to only nearest neighbors, but our code is not limited to nearest-neighbor interactions.

The dimensions of our matrix, and so the number of eigenvalues or bands we calculate, is equal to the number of orbitals we define in our unit cell. However, this is also arbitrary, as we can redefine our unit cell to be as many primitive cells of the crystal as we want. Our examples here restrict ourselves to the primitive cell, but the code can be used for compound cells.

Throughout this code, we have not taken account of the electron's spin, making this a spineless electron model. As a result, every band structure resulting from this model accommodates double the number of electrons implied.

This model can be adapted to incorporate spinors. However, that was beyond the scope and time constraints of this project.

Usage and structure of code

Now that we understand our mathematical formalism, we can review the code I wrote to implement it.

The project's first and most crucial function is "Hamiltonian(k,lattice, orbitals, onsite, hopping)." This function accepts a reciprocal space vector in reciprocal internal coordinates in the form of a list or array [a,b,c], the real space lattice vectors of your unit cell, in the form of a list of lists such as [[a,b,c],[d,e,f],[g,h,i]](three lattice vectors should always be provided even for two or one-dimensional crystals, and a vector of zeros should represent the unused dimensions), the positions of your orbitals in internal coordinates in the form of a list of vectors, the onsite

energy terms, which should be a list of real numbers, and finally the hopping terms. The hopping terms are more complicated to specify; the input should be a list of lists, with each sublist representing one hopping term of the form [h,i,j,[a,b,c]] where h is a real of complex number representing the hopping term, i and j are orbitals i and j in the given list of orbitals, and [a,b,c] is the coordinate of the lattice cell to which the hopping term applies. The code will then return the tight-binding Hamiltonian matrix. There is no need to specify reverse hopping terms; if hopping from "i" to "j" in unit cell R is defined, the code will automatically find the hopping term from orbital "j" to orbital "i" in unit cell -R by taking the hopping term's conjugate and switching the sign of R. As such no redundant entries should be entered as these will be double counted.

As the Hamiltonian is a hermitian matrix, the code simplifies this by filling out the diagonal entries with the onsite terms and then filling out the top right triangle of the matrix with the entered hopping terms. Finally, it takes the conjugate of each top right entry to find the bottom left entries of the matrix.

The second function to understand is the path tracing function k_path(k_points,n_xvalues). This function accepts a list of points in reciprocal space k_path, each represented by a list of internal coordinate bases components, an integer n_xvalues. It will then output a list of reciprocal space vectors mapping out a path connecting each of the points of length n_xvalues. It will also output a list of values from 0 to n n_xvalues, a list of which point in that list corresponds to each initial given point, and a list of entries corresponding to the total number of vectors along each path segment for use in plotting. This list of reciprocal space vectors can then be used with the Hamiltonian function to find eigenvalues along that path. An example of a high symmetry path for graphene can be seen in Figure 1. The high symmetry paths for many crystal structures are available online at [*Materials Project—Home*.]. Alternatively, the

online tool [*Tools:seekpath [CP2K Open Source Molecular Dynamics]*] can be used to attain high symmetry paths in structures.

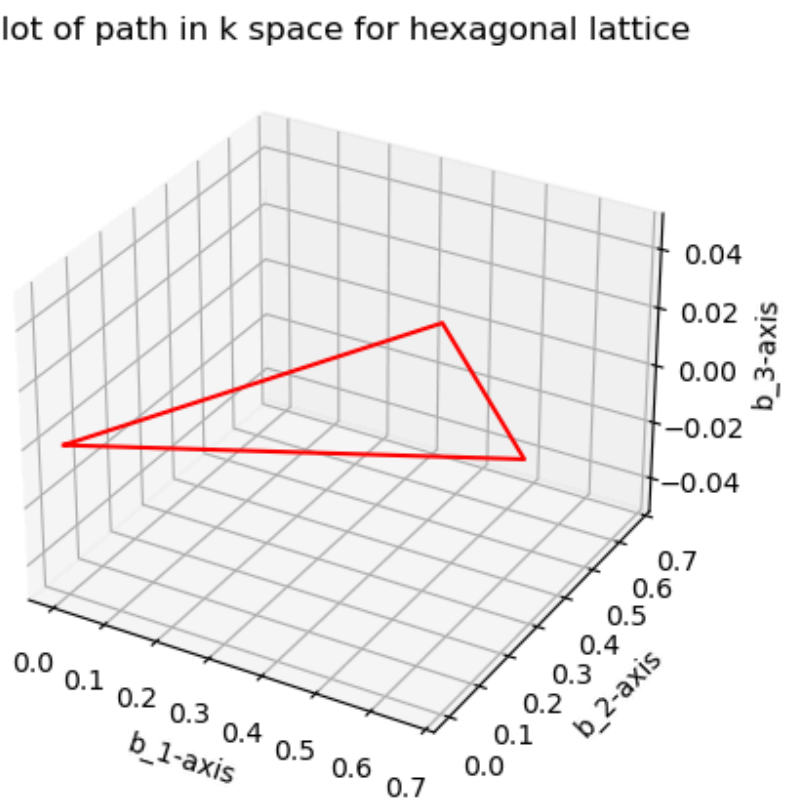


Figure 1. A high symmetry path in the reciprocal space of graphene. Note that the axes are distances in internal coordinates, so this is not a cartesian plot. The path starts at the origin, [0,0,0], and then sweeps out a path connecting the [⅔,⅓,0], [½,½,0], and [⅓,⅔,0] points before returning to the origin.

These two functions are then combined in a third function, "find_eigen(k_points,n_xvalues,lattice, orbitals, onsite,hopping)," to loop over the path in reciprocal space and find all eigenvalues for each point, which are then returned as a list of lists corresponding to each reciprocal space vector. This function is then used in our fourth function,

"plot_bands(k_points,n_xvalues,lattice, orbitals, onsite,hopping,title):" which will use all previous functions to create a band diagram, plotting the eigenvalues along the specified path in reciprocal space. The function automatically draws vertical labeled lines at each of the given points in reciprocal space that we used to define the segments of our path. The x-axis is an index of the point plotted along the path, and the y-axis is the magnitude of the eigenvalue, the unitless energy of the state in the band.

Examples

No units have been mentioned throughout our code and mathematical derivation, which is standard for tight-binding calculations. The calculations' unitless nature comes naturally from the fact that we are working in both reciprocal and real space. Therefore, our reciprocal units will simply be the inverse of our real space units. Likewise, our energies are unspecified, as the energy of our bands will have the same units as the unit we chose for our hopping and onsite energy terms. As such, throughout this paper, we will not refer to units but simply unitless numbers.

one-dimensional chain of atoms and 2d square lattice

To begin with, I plotted the band structure of some very simple lattices. First, I plotted a one-dimensional chain of atoms in Figure 2. This is the most basic structure possible, just a series of atoms in a line, and the resulting band structure is just a simple cosine function, as seen in the figure. Expanding this simple model to two dimensions, I found the band of a simple square lattice, as shown in Figure 3.

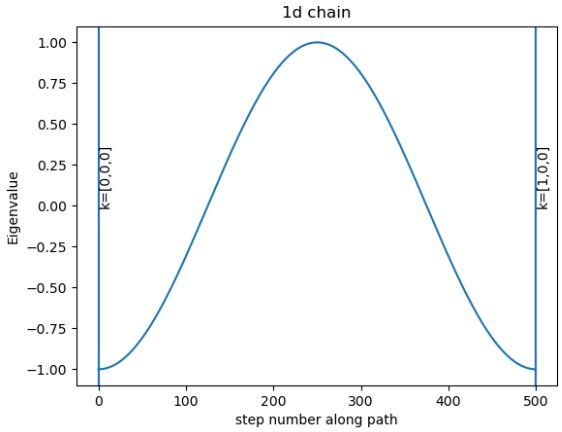


Figure 2. The band structure of a simple chain of atoms. The single atom's orbital is given an onsite energy of zero, and a hopping term of -1 is defined between it and its nearest neighbors.

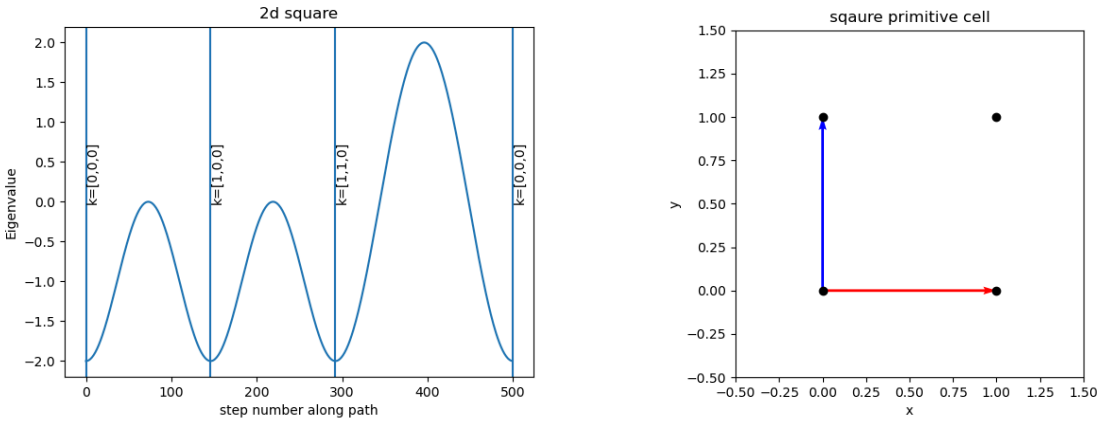


Figure 3. The band structure and primitive cell of a two-dimensional square lattice. The single atom's orbital is given an onsite energy of zero, and a hopping term of -1 is defined between it and its nearest neighbors. We define a path from the origin to [1,0,0], [1,1,0], and then back to the origin to find the band structure. On the primitive cell diagram, the atoms are shown in black, and the lattice vectors are shown as red and blue arrows.

Checkerboard model

Expanding on the square lattice model is the checkerboard model, a two-dimensional two-orbital model. It has a square lattice with lattice vectors [1,0,0] and [0,1,0]. The orbital

positions are the square unit cell's origin and point [0.5,0.5,0]. We give these two orbitals onsite energies of -1.1 and 1.1, respectively, and define a hopping term of 0.6 between each atom and its nearest neighbors. The result and primitive cell can be seen in Figure 4. As two different orbitals are now defined in our cell, we get two bands on our diagram.

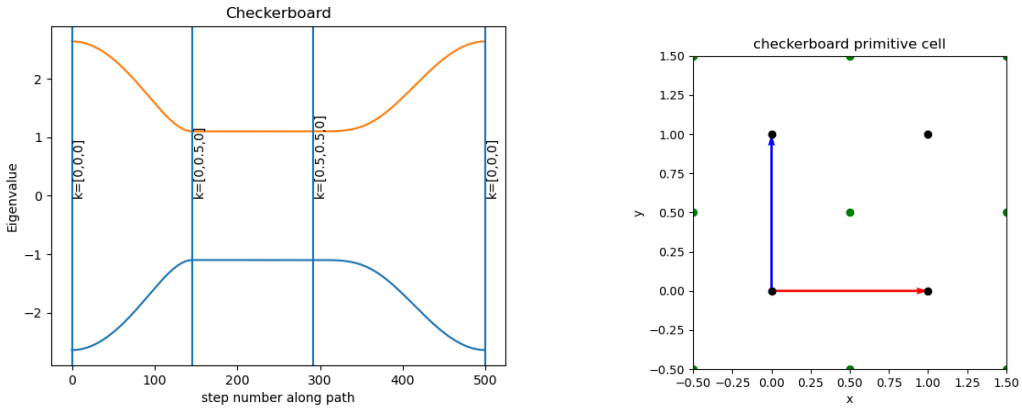


Figure 4. The band structure and primitive cell of a two-dimensional checkerboard lattice. To find the band structure, we define a path from the origin to [0,0.5,0], [0.5,0.5,0], and then back to the origin. On the primitive cell diagram, the two types of atoms are shown in black and green, and the lattice vectors are shown in red and blue arrows.

Graphene

Graphene consists of two-dimensional hexagonal sheets of carbon atoms and is a commonly used test model for simulations due to its simplicity. For our model, we define our lattice vectors as [1,0,0], [0.5,$\sqrt{3}$/2.0,0], and our atom/orbital positions as [1/3,1/3,0], [2/3,2/3,0] in the coordinates of this lattice. We define an onsite energy of 0 and a hopping term of -1 between nearest neighbors. The resulting band diagram has two bands that touch at a single point along our path, implying that the material may be a good conductor, which is true for graphene. Comparing this to the band structure of graphene using more advanced methods at [*mp-990448:*

C (Hexagonal, P6/mmm, 191)], we can see that though it is not an exact match, our diagram still closely matches the general shape of the valence and conduction bands of graphene.

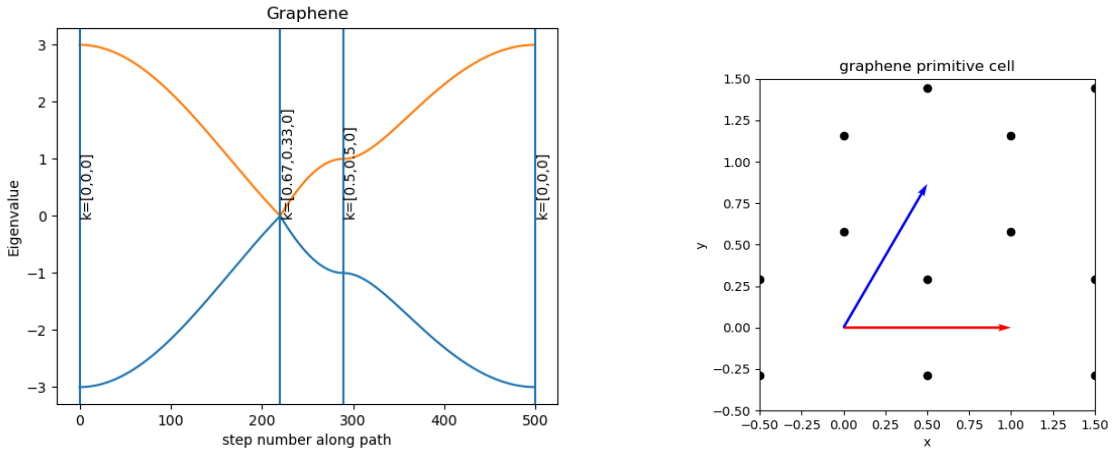


Figure 5. The band structure and primitive cell of two-dimensional graphene. To find the band structure, we define a path from the origin to [2/3,1/3,0], [0.5,0.5,0], and then back to the origin. On the primitive cell diagram, the atoms are shown in black, and the lattice vectors are shown as red and blue arrows.

Haldane

The Haldane model is similar to the graphene model in that it has a hexagonal crystal lattice, and we enter the same lattice vectors and positions for our code. The Haldane model is not intended to be a one-to-one representation of any particular material but instead could be a simple representation of many different materials with two types of atoms in a configuration similar to graphene. The two atoms in its primitive cell are no longer of a single type, and we give them two different onsite energies of -0.2 and 0.2, respectively. Additionally, we now define some hopping terms as complex numbers. Hopping between the two types of atoms is still defined as a simple -1, but hopping between atoms of the same type in different unit cells is now

defined as +/- 0.15i. These changes result in a band diagram resembling that of graphene but now has a distinct band gap.

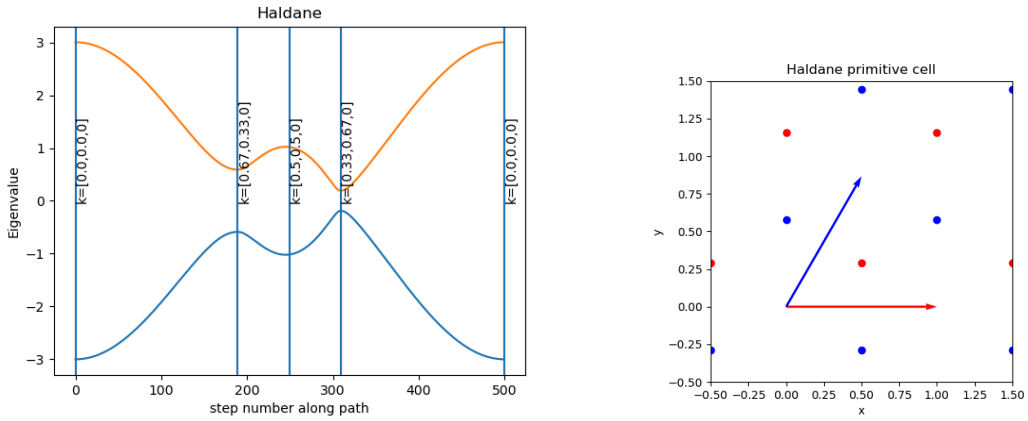


Figure 6. The band structure and primitive cell of a two-dimensional Haldane model. To find the band structure, we define a path from the origin to [⅔,⅓,0], [0.5,0.5,0], [⅓,⅔,0], and then back to the origin. On the primitive cell diagram, the atoms are shown as red and blue dots, and the lattice vectors are shown as red and blue arrows.

Conclusion

D

As shown in this paper, I have derived the basic mathematical formalism of the tight-binding model during this project. I then went on to demonstrate how Python could be used to implement this model and created robust functions that can be used to simulate a wide variety of structures. Lastly, I demonstrated several examples of the use of this code and the resulting plots. This code could be expanded to more general spinor models of electrons, but for now, I hope it can serve as a learning tool for the tight-binding model and materials simulation.

References

Formalism—PythTB 1.8.0 documentation. (n.d.). Retrieved April 15, 2024, from https://www.physics.rutgers.edu/pythtb/formalism.html

Materials Project—Home. (n.d.). Materials Project. Retrieved April 22, 2024, from https://next-gen.materialsproject.org/

mp-990448: C (Hexagonal, P6/mmm, 191). (n.d.). Materials Project. Retrieved April 22, 2024, from https://next-gen.materialsproject.org/materials/mp-990448?crystal_system=Hexagonal&formula=C

Simon, S. H. (2013). *The Oxford Solid State Basics*. OUP Oxford.

Tools:seekpath [CP2K Open Source Molecular Dynamics]. (n.d.). Retrieved April 11, 2024, from https://www.cp2k.org/tools:seekpath

Project Clean Code

April 22, 2024

[15]:
```
import numpy as np
import matplotlib.pyplot as plt
from numpy.linalg import eigvalsh
from numpy import linalg as LA
from mpl_toolkits import mplot3d
```

1 hamiltonian matrix creating function.

[2]:
```
#an attempt to allow the creation of any tight binding matrix
#k is a vector of the components of k in the internal basis, not the cartesian␣
 ↪basis. same for orbitals which should be in lattice coordinates.
def Hamiltonian(k,lattice, orbitals, onsite, hopping):

    k = np.array(k)


    #finding the matrix size
    n_orbitals = len(orbitals)

    #creating an empty matrix of the right side
    H = np.zeros((n_orbitals, n_orbitals), dtype = 'complex_')

    #adding in diagonals
    for i in range(n_orbitals):
        #set self energy. if none is given leave it at zero
        if i < len(orbitals):
            H[i,i]= onsite[i]

        #add any site hopping terms
        for hop in hopping:
            if hop[1] == i and hop[2] == i:
                R = -np.array(hop[3])
                H[i,i]=H[i,i]+hop[0]*np.exp(2j*np.pi*np.dot(k,R))

    #build top right part of matrix
    for i in range(n_orbitals):
        #loop over only the right side of the matrix
```

```
        for j in np.arange(i+1,n_orbitals):

            #looping through all terms and adding them if they correspond to␣
↪the current matrix element
            for hop in hopping:

                #hopping terms, only select the correct corresponding terms.
                if hop[1]==i and hop[2]==j and hop[1]!=hop[2]:

                    R = - np.array(orbitals[i]) + np.array(orbitals[j]) -  np.
↪array(hop[3])

                    H[i,j]=H[i,j]+hop[0]*np.exp(2j*np.pi*np.dot(k,R))
                if hop[1]==j and hop[2]==i and hop[1]!=hop[2]:

                    R = - np.array(orbitals[i]) + np.array(orbitals[j]) +  np.
↪array(hop[3])

                    H[i,j]=H[i,j]+np.conjugate(hop[0])*np.exp(2j*np.pi*np.
↪dot(k,R))


    #build bottom left of matrix by taking the conjugates of the top right.␣
↪This must be the case, or the matrix wouldn't be hermitian.
    #need to start at 1, not 0, because the 0th element has nothing to the left␣
↪of it. note: trying to use this for a matrix with only one element will␣
↪probably cause errors.
    if n_orbitals > 1:
        #starting with the second row
        for i in np.arange(1,n_orbitals):
            #loop over all elements to the left of j element
            for j in range(i):
                H[i,j]=np.conjugate(H[j,i])
    return H
```

Example of use using graphene parameters at the origin of the reciprocal space.

[3]:
```
#graphene

#lattice and positions
test_lattice = [np.array([1,0,0]),np.array([0.5,np.sqrt(3.0)/2.0,0]),np.
↪array([0,0,0])]
test_orbitals = [np.array([1/3,1/3,0]),np.array([2/3,2/3,0])]

# set model parameters
delta=0.0
t=-1.0
```

```
# set on-site energies
test_onsite=[-delta,delta]

# set hoppings (one for each connected pair of orbitals)
# (amplitude, i, j, [lattice vector to cell containing j])
test_hopping = [
[t, 0, 1, [ 0, 0,0]],
[t, 1, 0, [ 1, 0,0]],
[t, 1, 0, [ 0, 1,0]]]


#using the function
Hamiltonian(np.array([0,0,0]),test_lattice,␣
 ↪test_orbitals,test_onsite,test_hopping)
```

[3]:
```
array([[-0.+0.j, -3.+0.j],
       [-3.-0.j,  0.+0.j]])
```

2 Path producer

this code will take a list of points in K space, using internal coordinates, and output a list of k vector values along those points in sequence, with the number of values corresponding roughly to the length of the distance in k space in internal coordinates.

[4]:
```
#input list of points,including the origin. if you want it to loop back include␣
 ↪the origin at the end too, total numper of x values you want. Note things␣
 ↪are going to be rounded to integers a lot so don't put in tiny numbers for␣
 ↪number of x values
def k_path(k_points,n_xvalues):
    #find total path length
    total_length = 0
    #and the individual path lengths, this should always end up being a list of␣
 ↪length one less than the number of points
    segment_lengths=[]
    for i in range(len(k_points)-1):
        length = np.linalg.norm(np.array(k_points[i+1])-np.array(k_points[i]))
        total_length = total_length + length
        segment_lengths.append(length)


    #number of points in each segment.
    n_values = []
    for i in range(len(segment_lengths)-1):
        n_val = int(n_xvalues*segment_lengths[i]/total_length)
        n_values.append(n_val)
```

```
    #last n value is whatever is left out of the total desired and is used for␣
↪the final segment.
    n_val = int(n_xvalues - sum(n_values))
    n_values.append(n_val)

    #make a list of the total number of segments up to that point so we can␣
↪place markers. This list should should be equal in length to the number of k␣
↪points
    marker_n_values = [0]
    n_val = int(0)
    for n in n_values:
        n_val = n_val + n
        marker_n_values.append(n_val)

    #now finally we the code that creates our list of k_values
    k_list = []

    for i in range(len(segment_lengths)):
        #initial and final points of segment
        k0 = np.array(k_points[i])
        k1 = np.array(k_points[i+1])
        delta_k = k1 - k0
        n = n_values[i]
        for j in range(n):
            k = k0 +  delta_k*j/n
            k_list.append(k)

    x_list = t_list = range(len(k_list))
    return k_list, x_list,  marker_n_values, n_values
```

Example of use, using the high symmetry reciprocal space points of a graphene lattice

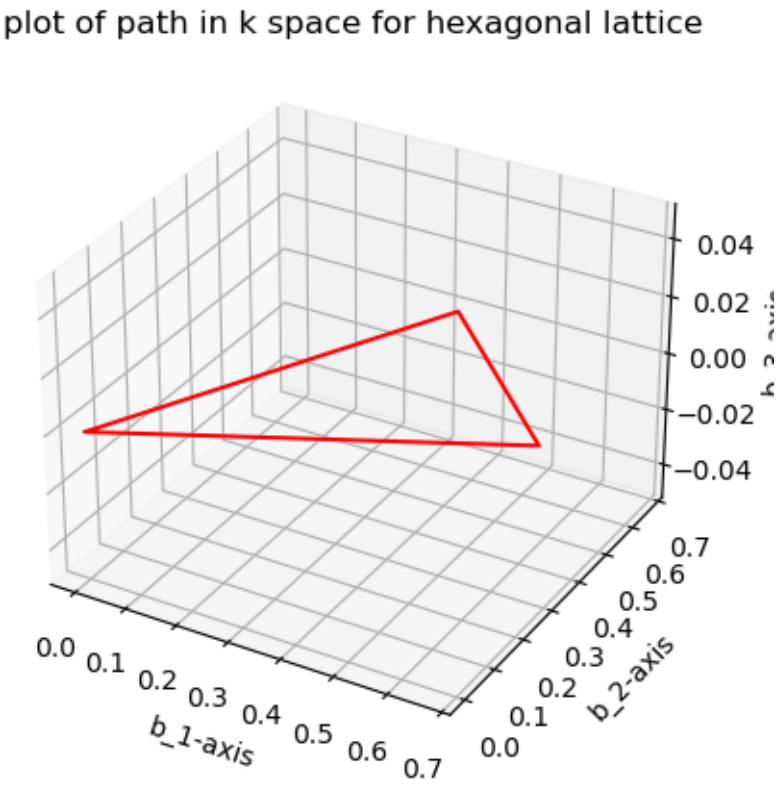
[6]:
```
path=[[0.,0.,0],[2./3.,1./3.,0],[.5,.5,0],[1./3.,2./3.,0], [0.,0.,0]]
test_k_list, test_t_list, test_marker_points, test_n_values = k_path(path,500)
```

[7]:
```
print(len(test_k_list),len(test_t_list), test_marker_points, test_n_values,␣
↪sum(test_n_values))
```

```
500 500 [0, 189, 249, 309, 500] [189, 60, 60, 191] 500
```

[16]:
```
test_k_transpose = np.transpose(np.array(test_k_list))
b1 = test_k_transpose[0]
b2 = test_k_transpose[1]
b3 = test_k_transpose[2]

fig = plt.figure()
ax = plt.axes(projection='3d')
```

```
ax.plot3D (b1, b2, b3, 'red')
ax.set_title('plot of path in k space for hexagonal lattice')
ax.set_xlabel('b_1-axis')
ax.set_ylabel('b_2-axis')
ax.set_zlabel('b_3-axis')
plt.savefig('plot of path in k space for hexagonal lattice.png')
plt.show()
```



3 function for finding the eigenvalues using the previous functions

[5]:
```
def find_eigen(k_points,n_xvalues,lattice, orbitals,onsite,hopping):
    k_list, t_list, marker_points, n_values= k_path(k_points,n_xvalues)

    dimensions = len(orbitals)
    E_list = []
    for i in range(dimensions):
        E_list.append([])

    for k in k_list:
        Hammy = Hamiltonian(k,lattice, orbitals, onsite, hopping)
```

```
        Eigens = LA.eigvalsh(Hammy)
        for i in range(dimensions):
            E_list[i].append(Eigens[i])

    return E_list, t_list, marker_points
```

4 Function using the above functions to fully plot out the badn diagram given the previous information, and save it to a figure.

[6]:
```
def plot_bands(k_points,n_xvalues,lattice, orbitals,onsite,hopping,title):
    E_list, t_list, marker_points = find_eigen(k_points,n_xvalues,lattice,␣
 ↪orbitals,onsite,hopping)
    dimensions = len(orbitals)
    title = str(title)

    for i in range(dimensions):
        plt.plot(t_list,E_list[i])

    for i in range(len(marker_points)):
        plt.axvline(marker_points[i])
        name = "k="+"["+str(round(k_points[i][0],␣
 ↪2))+","+str(round(k_points[i][1], 2))+","+str(round(k_points[i][2], 2))+"]"
        plt.text(marker_points[i],0,name,rotation=90)

    plt.xlabel("step number along path")
    plt.ylabel("Eigenvalue")
    plt.title(title)
    plt.show
    picname = title + ".png"
    plt.savefig(picname)
```

Function for drawing two dimensional primitive cells.

[62]:
```
def primitive_cell_plot_2d(lattice,orbitals,title, colors):
    real_orbitals = []

    for i in range(len(orbitals)):
        real_orbital = np.array(lattice[0])*orbitals[i][0] + np.
 ↪array(lattice[1])*orbitals[i][1] + np.array(lattice[2])*orbitals[i][2]
        real_orbitals.append(real_orbital)

    for i in range(len(orbitals)):
        plt.plot(real_orbitals[i][0],real_orbitals[i][1],'ok', color=colors[i])
```

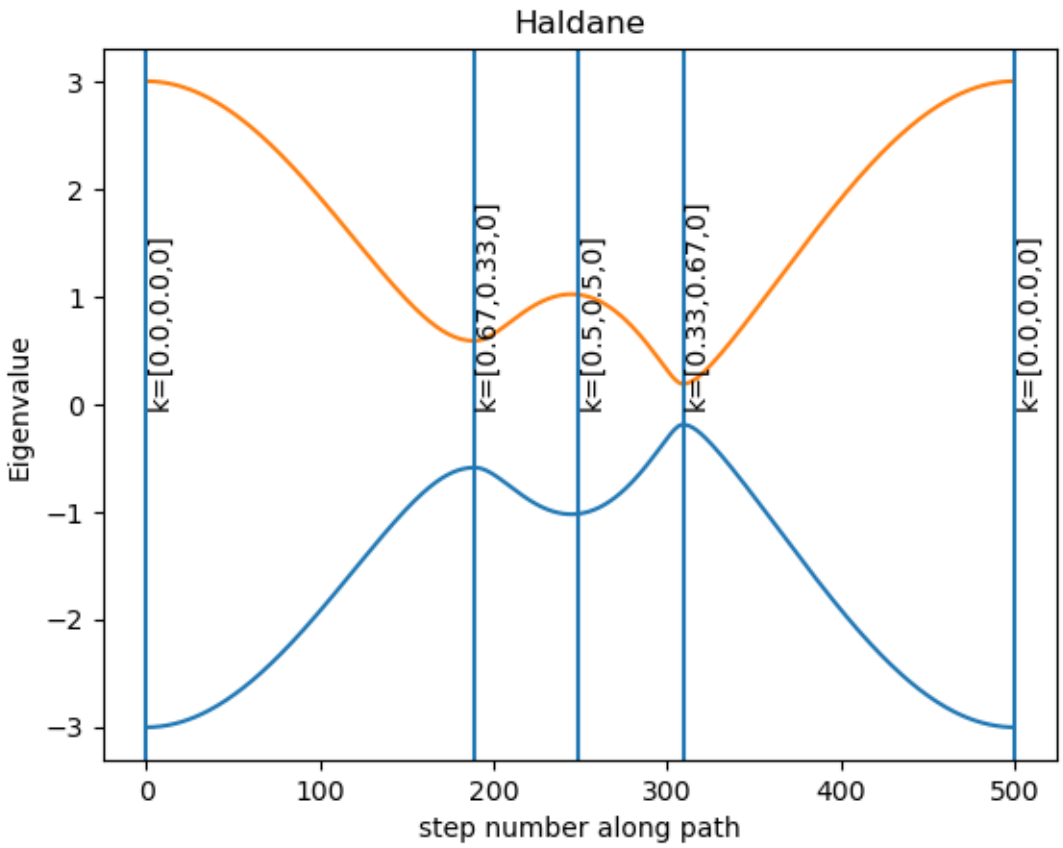
```
      plt.
↪plot(real_orbitals[i][0]+lattice[0][0],real_orbitals[i][1]+lattice[0][1],'ok',␣
↪color=colors[i])
      plt.
↪plot(real_orbitals[i][0]-lattice[0][0],real_orbitals[i][1]-lattice[0][1],'ok',␣
↪color=colors[i])
      plt.
↪plot(real_orbitals[i][0]+lattice[1][0],real_orbitals[i][1]+lattice[1][1],'ok',␣
↪color=colors[i])
      plt.
↪plot(real_orbitals[i][0]-lattice[1][0],real_orbitals[i][1]-lattice[1][1],'ok',␣
↪color=colors[i])
      plt.
↪plot(real_orbitals[i][0]+lattice[1][0]+lattice[0][0],real_orbitals[i][1]+lattice[1][1]+latt
↪color=colors[i])
      plt.
↪plot(real_orbitals[i][0]-lattice[1][0]-lattice[0][0],real_orbitals[i][1]-lattice[1][1]-latt
↪color=colors[i])
      plt.
↪plot(real_orbitals[i][0]+lattice[1][0]-lattice[0][0],real_orbitals[i][1]+lattice[1][1]-latt
↪color=colors[i])
      plt.
↪plot(real_orbitals[i][0]-lattice[1][0]+lattice[0][0],real_orbitals[i][1]-lattice[1][1]+latt
↪color=colors[i])


   plt.quiver(0,0,lattice[0][0],lattice[0][1], color='r', angles='xy',␣
↪scale_units='xy', scale=1)
   plt.quiver(0,0,lattice[1][0],lattice[1][1], color='b', angles='xy',␣
↪scale_units='xy', scale=1)

   plt.xlim(-0.5, 1.5)
   plt.ylim(-0.5, 1.5)

   ax = plt.gca()
   ax.set_aspect('equal', adjustable='box')



   plt.xlabel("x")
   plt.ylabel("y")
   plt.title(title)
   plt.show
   picname = title + ".png"
   plt.savefig(picname)
```

5 examples of use

Haldane model

[19]:
```
#testing haldane model
path=[[0.,0.,0],[2./3.,1./3.,0],[.5,.5,0],[1./3.,2./3.,0], [0.,0.,0]]

# define lattice vectors
test_lattice=[np.array([1.0,0.0,0]),np.array([0.5,np.sqrt(3.0)/2.0,0])]
# define coordinates of orbitals
test_orbitals=[np.array([1./3.,1./3.,0]),np.array([2./3.,2./3.,0])]


# set model parameters
delta=0.2
t=-1.0
t2 =0.15*np.exp((1.j)*np.pi/2.)
t2c=t2.conjugate()

# set on-site energies
test_onsite = [-delta,delta]
# set hoppings (one for each connected pair of orbitals)
# (amplitude, i, j, [lattice vector to cell containing j])
test_hopping = [
[t, 0, 1, [ 0, 0,0]],
[t, 1, 0, [ 1, 0,0]],
[t, 1, 0, [ 0, 1,0]],
[t2 , 0, 0, [ 1, 0,0]],
[t2 , 1, 1, [ 1,-1,0]],
[t2 , 1, 1, [ 0, 1,0]],
[t2c, 1, 1, [ 1, 0,0]],
[t2c, 0, 0, [ 1,-1,0]],
[t2c, 0, 0, [ 0, 1,0]]]
```

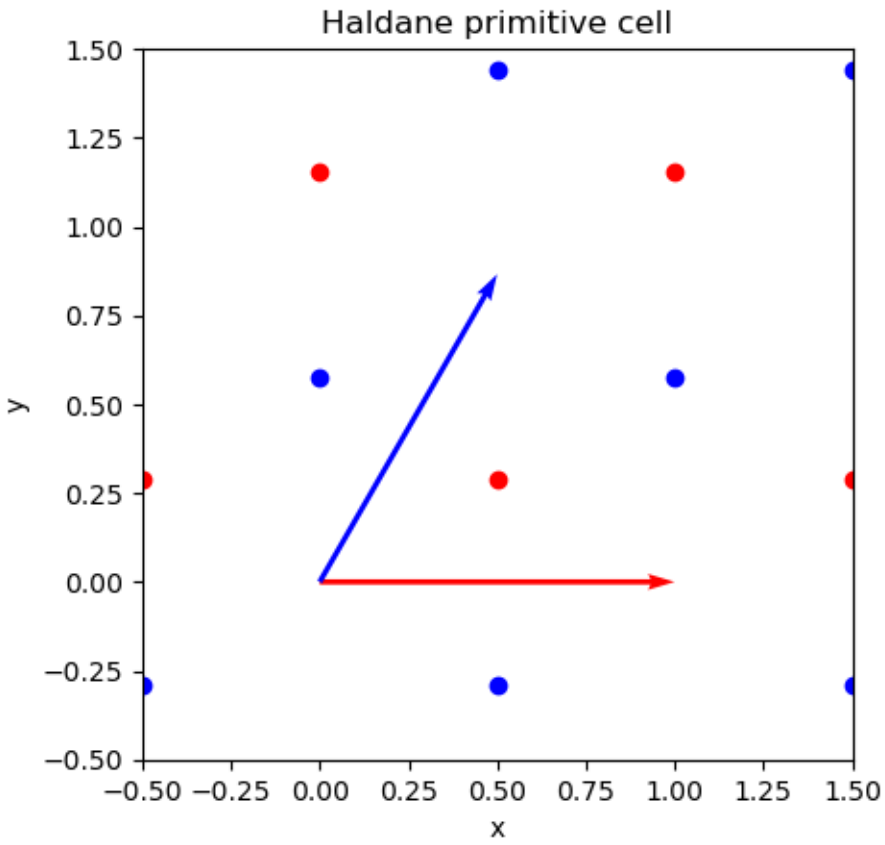
[20]:
```
plot_bands(path,500,test_lattice,␣
↪test_orbitals,test_onsite,test_hopping,"Haldane")
```



[65]:
```
#lattice and positions
test_lattice = [np.array([1,0,0]),np.array([0.5,np.sqrt(3.0)/2.0,0]),np.
 ↪array([0,0,0])]
test_orbitals = [np.array([1/3,1/3,0]),np.array([2/3,2/3,0])]

primitive_cell_plot_2d(test_lattice,test_orbitals,"Haldane primitive␣
 ↪cell",["r","b"])
```

```
C:\Users\Tyjal\AppData\Local\Temp\ipykernel_5260\2459911905.py:9: UserWarning:
color is redundantly defined by the 'color' keyword argument and the fmt string
"ok" (-> color='k'). The keyword argument will take precedence.
  plt.plot(real_orbitals[i][0],real_orbitals[i][1],'ok', color=colors[i])
C:\Users\Tyjal\AppData\Local\Temp\ipykernel_5260\2459911905.py:10: UserWarning:
color is redundantly defined by the 'color' keyword argument and the fmt string
"ok" (-> color='k'). The keyword argument will take precedence.
  plt.plot(real_orbitals[i][0]+lattice[0][0],real_orbitals[i][1]+lattice[0][1],'
ok', color=colors[i])
C:\Users\Tyjal\AppData\Local\Temp\ipykernel_5260\2459911905.py:11: UserWarning:
color is redundantly defined by the 'color' keyword argument and the fmt string
"ok" (-> color='k'). The keyword argument will take precedence.
```

```
  plt.plot(real_orbitals[i][0]-lattice[0][0],real_orbitals[i][1]-
lattice[0][1],'ok', color=colors[i])
C:\Users\Tyjal\AppData\Local\Temp\ipykernel_5260\2459911905.py:12: UserWarning:
color is redundantly defined by the 'color' keyword argument and the fmt string
"ok" (-> color='k'). The keyword argument will take precedence.
  plt.plot(real_orbitals[i][0]+lattice[1][0],real_orbitals[i][1]+lattice[1][1],'
ok', color=colors[i])
C:\Users\Tyjal\AppData\Local\Temp\ipykernel_5260\2459911905.py:13: UserWarning:
color is redundantly defined by the 'color' keyword argument and the fmt string
"ok" (-> color='k'). The keyword argument will take precedence.
  plt.plot(real_orbitals[i][0]-lattice[1][0],real_orbitals[i][1]-
lattice[1][1],'ok', color=colors[i])
C:\Users\Tyjal\AppData\Local\Temp\ipykernel_5260\2459911905.py:14: UserWarning:
color is redundantly defined by the 'color' keyword argument and the fmt string
"ok" (-> color='k'). The keyword argument will take precedence.
  plt.plot(real_orbitals[i][0]+lattice[1][0]+lattice[0][0],real_orbitals[i][1]+l
attice[1][1]+lattice[0][1],'ok', color=colors[i])
C:\Users\Tyjal\AppData\Local\Temp\ipykernel_5260\2459911905.py:15: UserWarning:
color is redundantly defined by the 'color' keyword argument and the fmt string
"ok" (-> color='k'). The keyword argument will take precedence.
  plt.plot(real_orbitals[i][0]-lattice[1][0]-lattice[0][0],real_orbitals[i][1]-
lattice[1][1]-lattice[0][1],'ok', color=colors[i])
C:\Users\Tyjal\AppData\Local\Temp\ipykernel_5260\2459911905.py:16: UserWarning:
color is redundantly defined by the 'color' keyword argument and the fmt string
"ok" (-> color='k'). The keyword argument will take precedence.
  plt.plot(real_orbitals[i][0]+lattice[1][0]-
lattice[0][0],real_orbitals[i][1]+lattice[1][1]-lattice[0][1],'ok',
color=colors[i])
C:\Users\Tyjal\AppData\Local\Temp\ipykernel_5260\2459911905.py:17: UserWarning:
color is redundantly defined by the 'color' keyword argument and the fmt string
"ok" (-> color='k'). The keyword argument will take precedence.
  plt.plot(real_orbitals[i][0]-lattice[1][0]+lattice[0][0],real_orbitals[i][1]-
lattice[1][1]+lattice[0][1],'ok', color=colors[i])
```

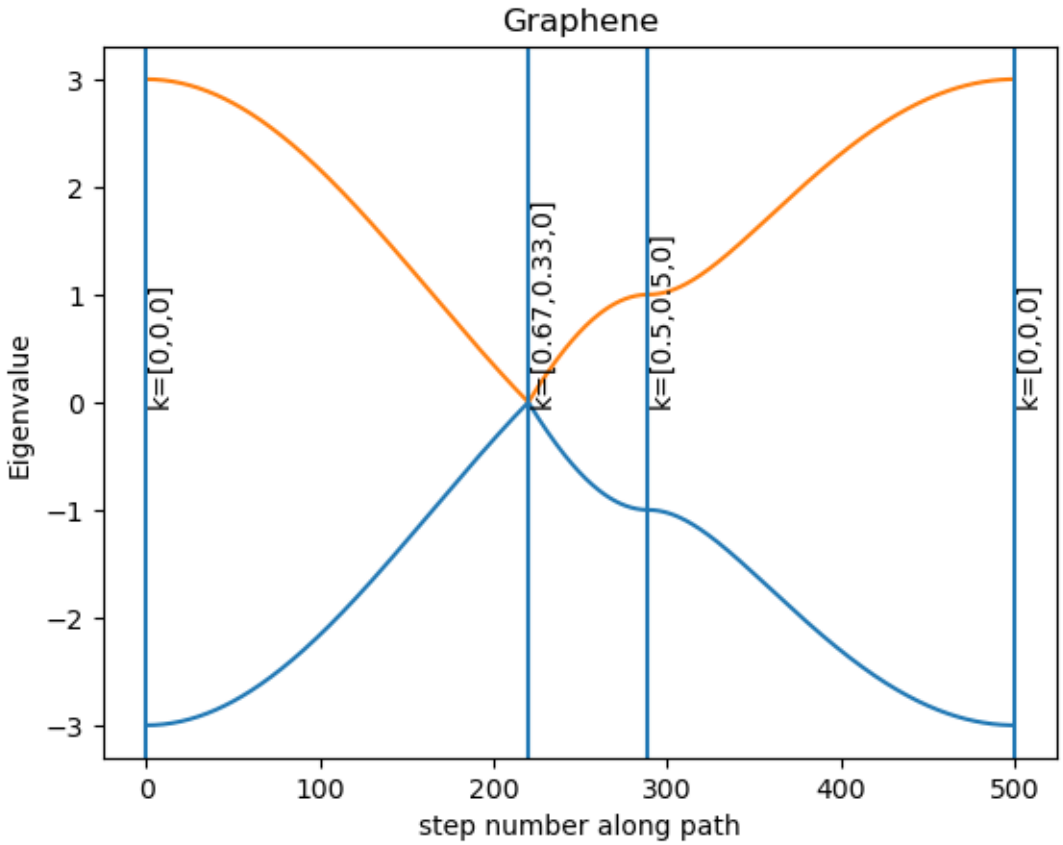


Graphene

[21]:
```
#graphene
path = [[0,0,0], [2/3,1/3,0], [0.5,0.5,0], [0,0,0]]
#lattice and positions
test_lattice = [np.array([1,0,0]),np.array([0.5,np.sqrt(3.0)/2.0,0]),np.
 ↪array([0,0,0])]
test_orbitals = [np.array([1/3,1/3,0]),np.array([2/3,2/3,0])]



# set model parameters
delta=0.0
t=-1.0

# set on-site energies
test_onsite=[-delta,delta]

# set hoppings (one for each connected pair of orbitals)
# (amplitude, i, j, [lattice vector to cell containing j])
```

```
test_hopping = [
[t, 0, 1, [ 0, 0,0]],
[t, 1, 0, [ 1, 0,0]],
[t, 1, 0, [ 0, 1,0]]]
```

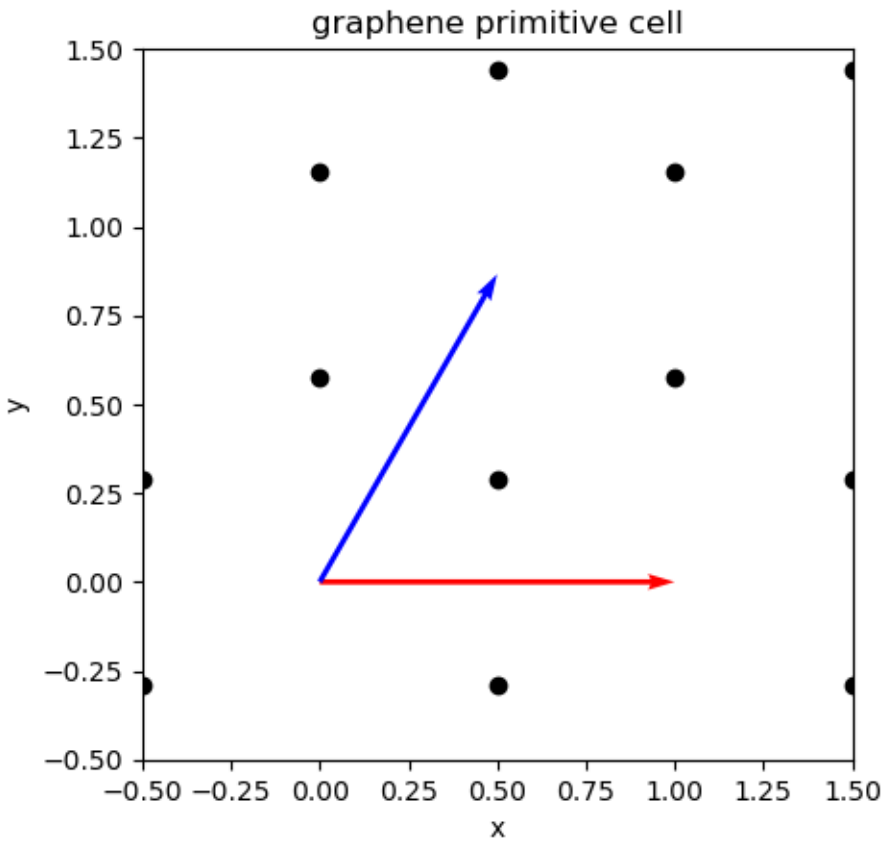
[22]:
```
plot_bands(path,500,test_lattice,␣
  ↪test_orbitals,test_onsite,test_hopping,"Graphene")
```



[63]:
```
#lattice and positions
test_lattice = [np.array([1,0,0]),np.array([0.5,np.sqrt(3.0)/2.0,0]),np.
  ↪array([0,0,0])]
test_orbitals = [np.array([1/3,1/3,0]),np.array([2/3,2/3,0])]

primitive_cell_plot_2d(test_lattice,test_orbitals,"graphene primitive␣
  ↪cell",["k","k"])
```

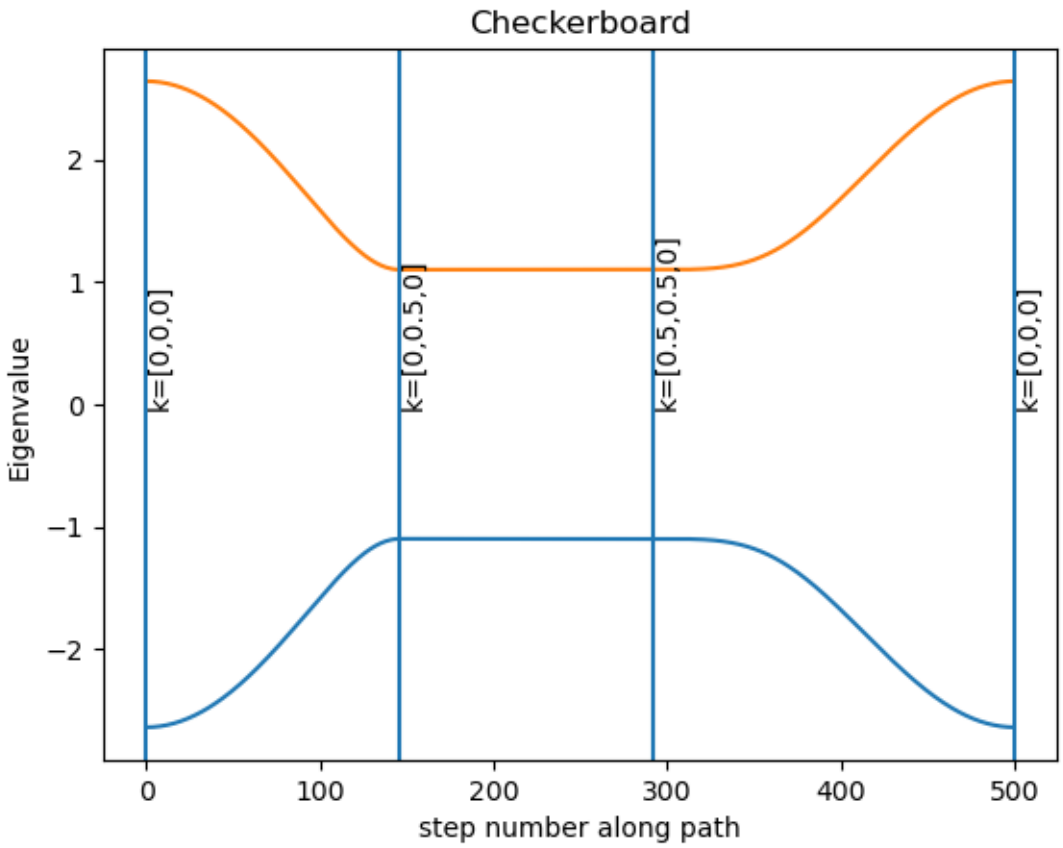
```
C:\Users\Tyjal\AppData\Local\Temp\ipykernel_5260\2459911905.py:9: UserWarning:
color is redundantly defined by the 'color' keyword argument and the fmt string
"ok" (-> color='k'). The keyword argument will take precedence.
  plt.plot(real_orbitals[i][0],real_orbitals[i][1],'ok', color=colors[i])
```

```
C:\Users\Tyjal\AppData\Local\Temp\ipykernel_5260\2459911905.py:10: UserWarning:
color is redundantly defined by the 'color' keyword argument and the fmt string
"ok" (-> color='k'). The keyword argument will take precedence.
  plt.plot(real_orbitals[i][0]+lattice[0][0],real_orbitals[i][1]+lattice[0][1],'
ok', color=colors[i])
C:\Users\Tyjal\AppData\Local\Temp\ipykernel_5260\2459911905.py:11: UserWarning:
color is redundantly defined by the 'color' keyword argument and the fmt string
"ok" (-> color='k'). The keyword argument will take precedence.
  plt.plot(real_orbitals[i][0]-lattice[0][0],real_orbitals[i][1]-
lattice[0][1],'ok', color=colors[i])
C:\Users\Tyjal\AppData\Local\Temp\ipykernel_5260\2459911905.py:12: UserWarning:
color is redundantly defined by the 'color' keyword argument and the fmt string
"ok" (-> color='k'). The keyword argument will take precedence.
  plt.plot(real_orbitals[i][0]+lattice[1][0],real_orbitals[i][1]+lattice[1][1],'
ok', color=colors[i])
C:\Users\Tyjal\AppData\Local\Temp\ipykernel_5260\2459911905.py:13: UserWarning:
color is redundantly defined by the 'color' keyword argument and the fmt string
"ok" (-> color='k'). The keyword argument will take precedence.
  plt.plot(real_orbitals[i][0]-lattice[1][0],real_orbitals[i][1]-
lattice[1][1],'ok', color=colors[i])
C:\Users\Tyjal\AppData\Local\Temp\ipykernel_5260\2459911905.py:14: UserWarning:
color is redundantly defined by the 'color' keyword argument and the fmt string
"ok" (-> color='k'). The keyword argument will take precedence.
  plt.plot(real_orbitals[i][0]+lattice[1][0]+lattice[0][0],real_orbitals[i][1]+l
attice[1][1]+lattice[0][1],'ok', color=colors[i])
C:\Users\Tyjal\AppData\Local\Temp\ipykernel_5260\2459911905.py:15: UserWarning:
color is redundantly defined by the 'color' keyword argument and the fmt string
"ok" (-> color='k'). The keyword argument will take precedence.
  plt.plot(real_orbitals[i][0]-lattice[1][0]-lattice[0][0],real_orbitals[i][1]-
lattice[1][1]-lattice[0][1],'ok', color=colors[i])
C:\Users\Tyjal\AppData\Local\Temp\ipykernel_5260\2459911905.py:16: UserWarning:
color is redundantly defined by the 'color' keyword argument and the fmt string
"ok" (-> color='k'). The keyword argument will take precedence.
  plt.plot(real_orbitals[i][0]+lattice[1][0]-
lattice[0][0],real_orbitals[i][1]+lattice[1][1]-lattice[0][1],'ok',
color=colors[i])
C:\Users\Tyjal\AppData\Local\Temp\ipykernel_5260\2459911905.py:17: UserWarning:
color is redundantly defined by the 'color' keyword argument and the fmt string
"ok" (-> color='k'). The keyword argument will take precedence.
  plt.plot(real_orbitals[i][0]-lattice[1][0]+lattice[0][0],real_orbitals[i][1]-
lattice[1][1]+lattice[0][1],'ok', color=colors[i])
```



checkerboard model

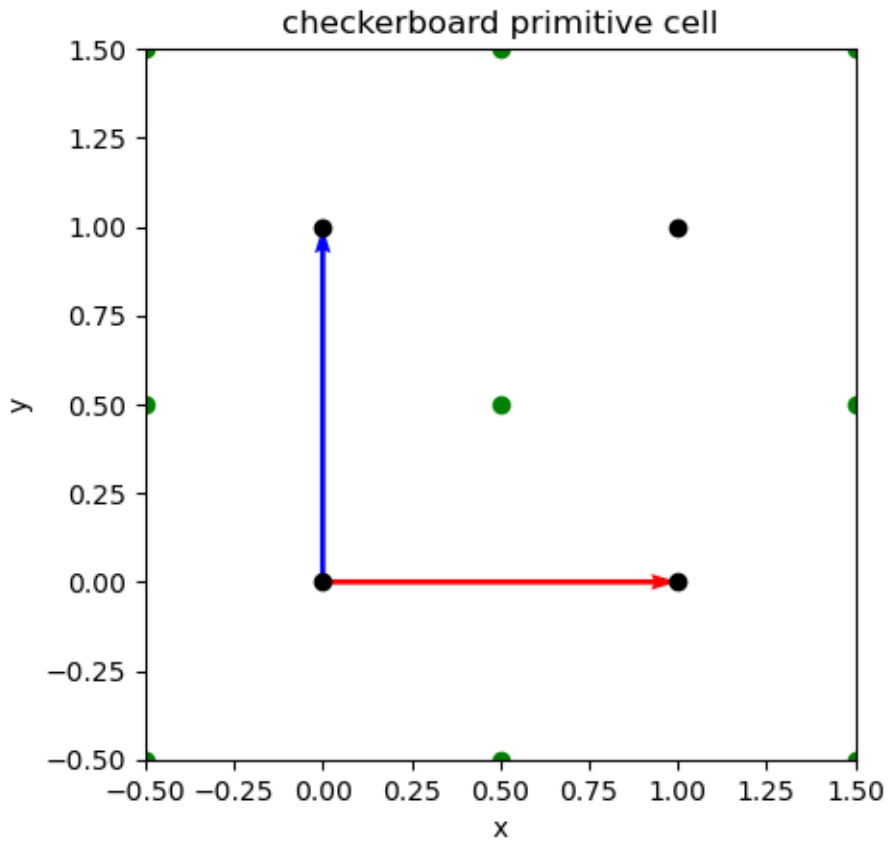
[23]:
```
#checkerboard
test_lattice = [np.array([1,0,0]),np.array([0,1,0]),np.array([0,0,0])]
test_orbitals = [np.array([0,0,0]),np.array([0.5,0.5,0])]
test_onsite = [-1.1,1.1]
test_hopping = [[0.6,0,1,[0,0,0]],[0.6,0,1,[1,0,0]],[0.6,0,1,[0,1,0]],[0.
↪6,0,1,[1,1,0]]]

path = [[0,0,0],[0,0.5,0],[0.5,0.5,0],[0,0,0]]
plot_bands(path,500,test_lattice,␣
↪test_orbitals,test_onsite,test_hopping,"Checkerboard")
```



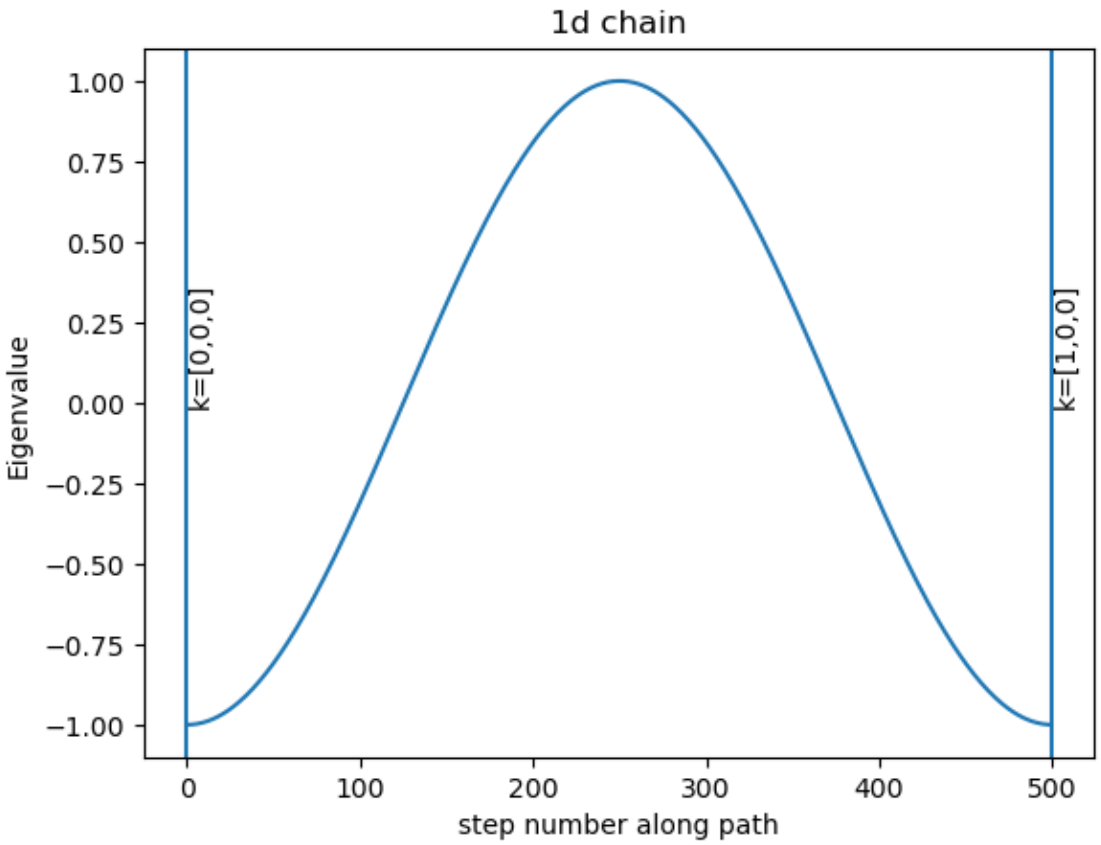
[64]:
```
#lattice and positions
test_lattice = [np.array([1,0,0]),np.array([0,1,0]),np.array([0,0,0])]
test_orbitals = [np.array([0,0,0]),np.array([0.5,0.5,0])]

primitive_cell_plot_2d(test_lattice,test_orbitals,"checkerboard primitive␣
↪cell",["k","g"])
```

```
C:\Users\Tyjal\AppData\Local\Temp\ipykernel_5260\2459911905.py:9: UserWarning:
color is redundantly defined by the 'color' keyword argument and the fmt string
"ok" (-> color='k'). The keyword argument will take precedence.
  plt.plot(real_orbitals[i][0],real_orbitals[i][1],'ok', color=colors[i])
C:\Users\Tyjal\AppData\Local\Temp\ipykernel_5260\2459911905.py:10: UserWarning:
color is redundantly defined by the 'color' keyword argument and the fmt string
"ok" (-> color='k'). The keyword argument will take precedence.
  plt.plot(real_orbitals[i][0]+lattice[0][0],real_orbitals[i][1]+lattice[0][1],'
ok', color=colors[i])
C:\Users\Tyjal\AppData\Local\Temp\ipykernel_5260\2459911905.py:11: UserWarning:
color is redundantly defined by the 'color' keyword argument and the fmt string
"ok" (-> color='k'). The keyword argument will take precedence.
  plt.plot(real_orbitals[i][0]-lattice[0][0],real_orbitals[i][1]-
```

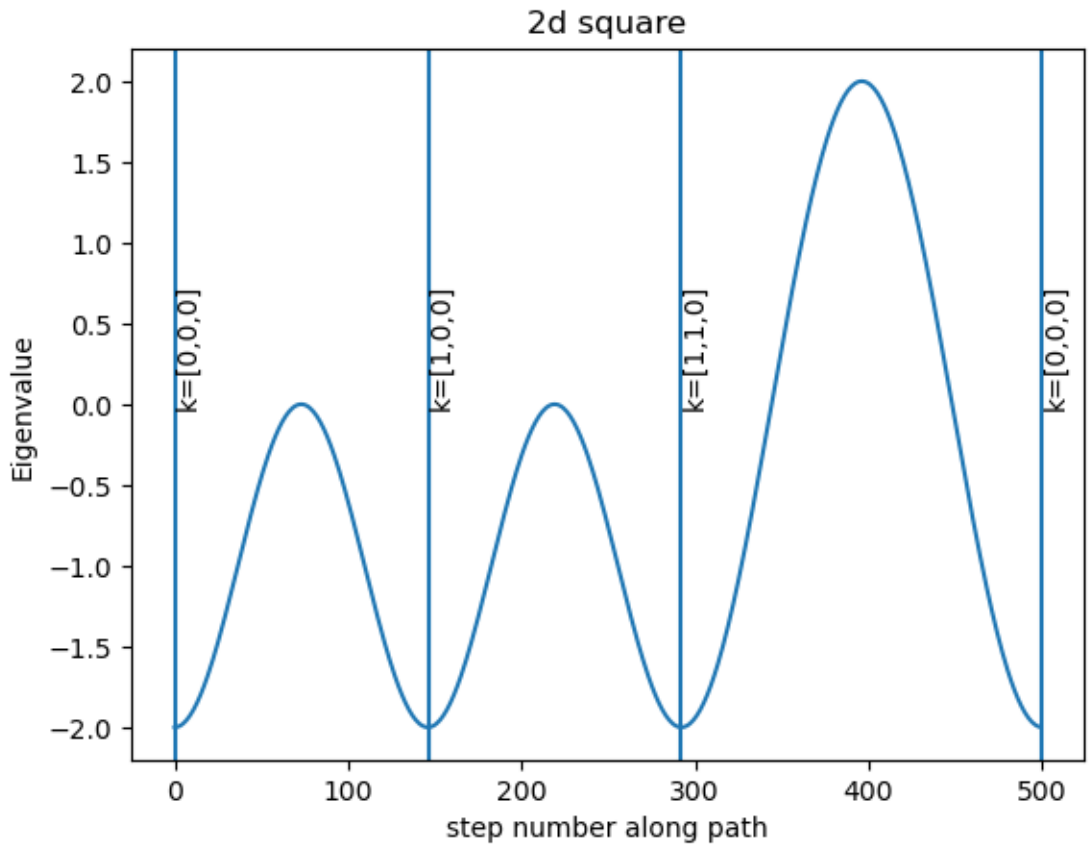
```
lattice[0][1],'ok', color=colors[i])
C:\Users\Tyjal\AppData\Local\Temp\ipykernel_5260\2459911905.py:12: UserWarning:
color is redundantly defined by the 'color' keyword argument and the fmt string
"ok" (-> color='k'). The keyword argument will take precedence.
  plt.plot(real_orbitals[i][0]+lattice[1][0],real_orbitals[i][1]+lattice[1][1],'
ok', color=colors[i])
C:\Users\Tyjal\AppData\Local\Temp\ipykernel_5260\2459911905.py:13: UserWarning:
color is redundantly defined by the 'color' keyword argument and the fmt string
"ok" (-> color='k'). The keyword argument will take precedence.
  plt.plot(real_orbitals[i][0]-lattice[1][0],real_orbitals[i][1]-
lattice[1][1],'ok', color=colors[i])
C:\Users\Tyjal\AppData\Local\Temp\ipykernel_5260\2459911905.py:14: UserWarning:
color is redundantly defined by the 'color' keyword argument and the fmt string
"ok" (-> color='k'). The keyword argument will take precedence.
  plt.plot(real_orbitals[i][0]+lattice[1][0]+lattice[0][0],real_orbitals[i][1]+l
attice[1][1]+lattice[0][1],'ok', color=colors[i])
C:\Users\Tyjal\AppData\Local\Temp\ipykernel_5260\2459911905.py:15: UserWarning:
color is redundantly defined by the 'color' keyword argument and the fmt string
"ok" (-> color='k'). The keyword argument will take precedence.
  plt.plot(real_orbitals[i][0]-lattice[1][0]-lattice[0][0],real_orbitals[i][1]-
lattice[1][1]-lattice[0][1],'ok', color=colors[i])
C:\Users\Tyjal\AppData\Local\Temp\ipykernel_5260\2459911905.py:16: UserWarning:
color is redundantly defined by the 'color' keyword argument and the fmt string
"ok" (-> color='k'). The keyword argument will take precedence.
  plt.plot(real_orbitals[i][0]+lattice[1][0]-
lattice[0][0],real_orbitals[i][1]+lattice[1][1]-lattice[0][1],'ok',
color=colors[i])
C:\Users\Tyjal\AppData\Local\Temp\ipykernel_5260\2459911905.py:17: UserWarning:
color is redundantly defined by the 'color' keyword argument and the fmt string
"ok" (-> color='k'). The keyword argument will take precedence.
  plt.plot(real_orbitals[i][0]-lattice[1][0]+lattice[0][0],real_orbitals[i][1]-
lattice[1][1]+lattice[0][1],'ok', color=colors[i])
```



[7]:
```
#1 d atom chain
test_lattice = np.array([1,0,0])
test_orbitals = np.array([0,0,0])
test_hopping = -1
test_onsite = 0

#checkerboard
test_lattice = [np.array([1,0,0]),np.array([0,0,0]),np.array([0,0,0])]
test_orbitals = [np.array([0,0,0])]
test_onsite = [0]
test_hopping = [[-1,0,0,[1,0,0]]]

path = [[0,0,0],[1,0,0]]
plot_bands(path,500,test_lattice, test_orbitals,test_onsite,test_hopping,"1d␣
↪chain")
```



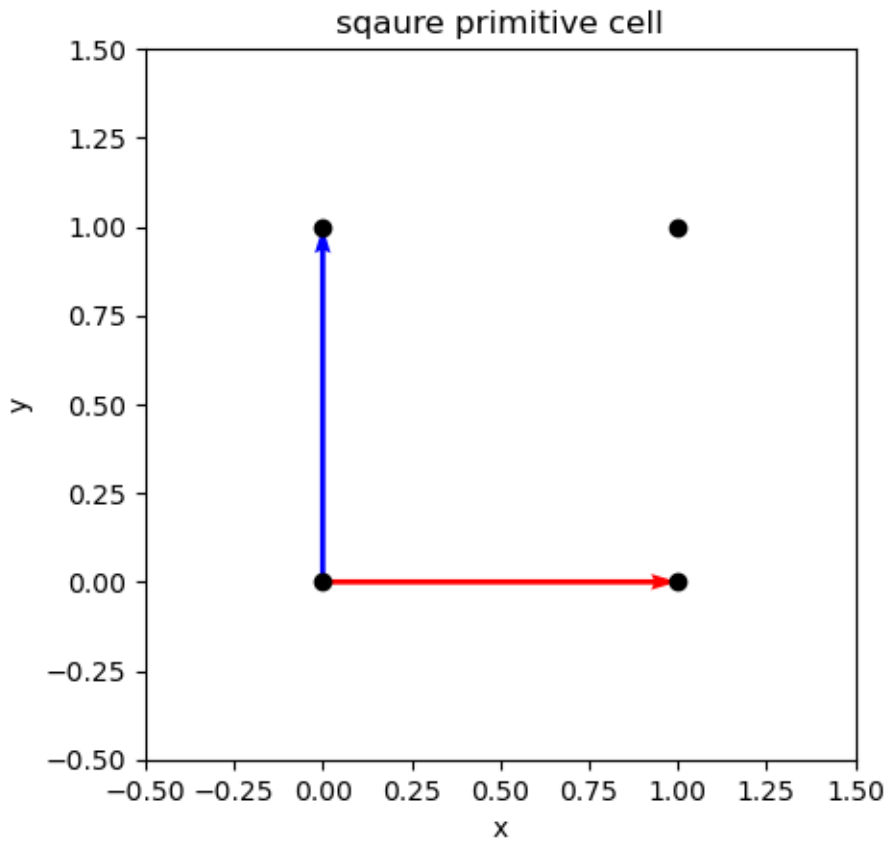
[9]:
```
#2d square
test_lattice = [np.array([1,0,0]),np.array([0,1,0]),np.array([0,0,0])]
test_orbitals = [np.array([0,0,0])]
test_onsite = [0]
test_hopping = [[-1,0,0,[1,0,0]],[-1,0,0,[0,1,0]]]

path = [[0,0,0],[1,0,0],[1,1,0],[0,0,0]]
plot_bands(path,500,test_lattice, test_orbitals,test_onsite,test_hopping,"2d␣
↪square")
```



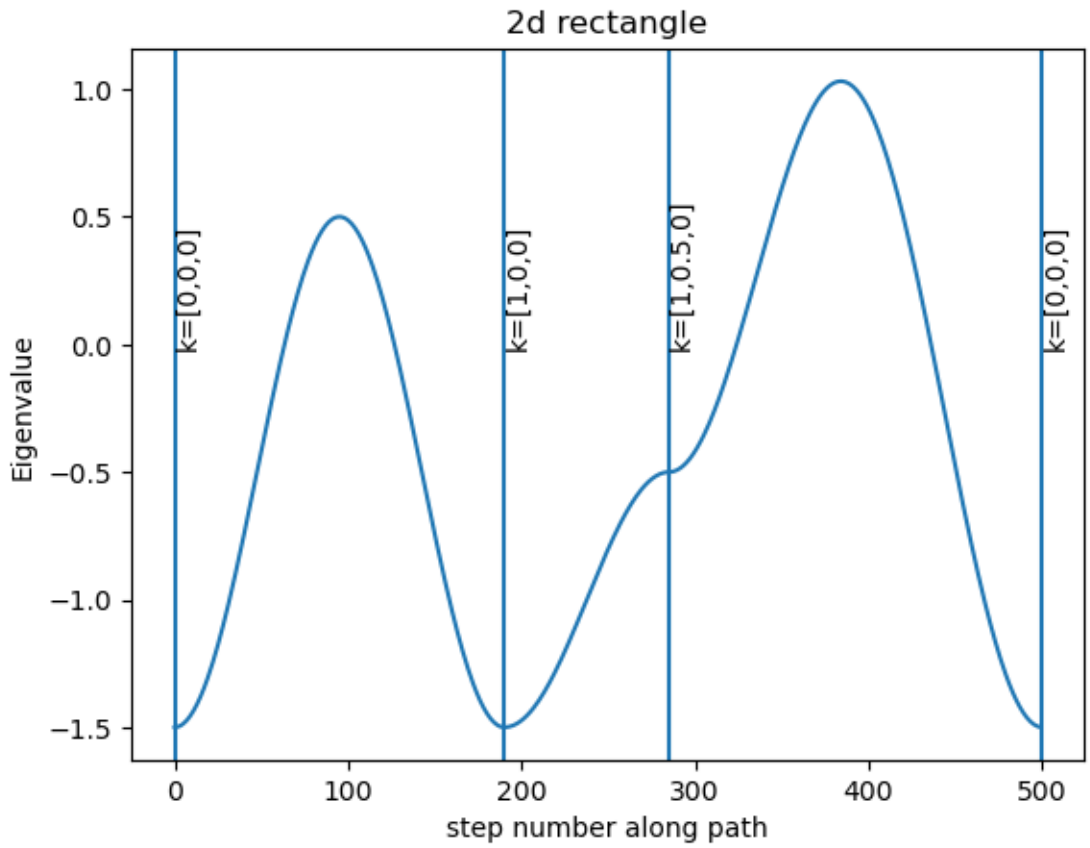
[66]:
```
#lattice and positions
test_lattice = [np.array([1,0,0]),np.array([0,1,0]),np.array([0,0,0])]
test_orbitals = [np.array([0,0,0])]

primitive_cell_plot_2d(test_lattice,test_orbitals,"sqaure primitive cell",["k"])
```

```
C:\Users\Tyjal\AppData\Local\Temp\ipykernel_5260\2459911905.py:9: UserWarning:
color is redundantly defined by the 'color' keyword argument and the fmt string
"ok" (-> color='k'). The keyword argument will take precedence.
  plt.plot(real_orbitals[i][0],real_orbitals[i][1],'ok', color=colors[i])
C:\Users\Tyjal\AppData\Local\Temp\ipykernel_5260\2459911905.py:10: UserWarning:
color is redundantly defined by the 'color' keyword argument and the fmt string
"ok" (-> color='k'). The keyword argument will take precedence.
  plt.plot(real_orbitals[i][0]+lattice[0][0],real_orbitals[i][1]+lattice[0][1],'
ok', color=colors[i])
C:\Users\Tyjal\AppData\Local\Temp\ipykernel_5260\2459911905.py:11: UserWarning:
color is redundantly defined by the 'color' keyword argument and the fmt string
"ok" (-> color='k'). The keyword argument will take precedence.
  plt.plot(real_orbitals[i][0]-lattice[0][0],real_orbitals[i][1]-
lattice[0][1],'ok', color=colors[i])
```

```
C:\Users\Tyjal\AppData\Local\Temp\ipykernel_5260\2459911905.py:12: UserWarning:
color is redundantly defined by the 'color' keyword argument and the fmt string
"ok" (-> color='k'). The keyword argument will take precedence.
  plt.plot(real_orbitals[i][0]+lattice[1][0],real_orbitals[i][1]+lattice[1][1],'
ok', color=colors[i])
C:\Users\Tyjal\AppData\Local\Temp\ipykernel_5260\2459911905.py:13: UserWarning:
color is redundantly defined by the 'color' keyword argument and the fmt string
"ok" (-> color='k'). The keyword argument will take precedence.
  plt.plot(real_orbitals[i][0]-lattice[1][0],real_orbitals[i][1]-
lattice[1][1],'ok', color=colors[i])
C:\Users\Tyjal\AppData\Local\Temp\ipykernel_5260\2459911905.py:14: UserWarning:
color is redundantly defined by the 'color' keyword argument and the fmt string
"ok" (-> color='k'). The keyword argument will take precedence.
  plt.plot(real_orbitals[i][0]+lattice[1][0]+lattice[0][0],real_orbitals[i][1]+l
attice[1][1]+lattice[0][1],'ok', color=colors[i])
C:\Users\Tyjal\AppData\Local\Temp\ipykernel_5260\2459911905.py:15: UserWarning:
color is redundantly defined by the 'color' keyword argument and the fmt string
"ok" (-> color='k'). The keyword argument will take precedence.
  plt.plot(real_orbitals[i][0]-lattice[1][0]-lattice[0][0],real_orbitals[i][1]-
lattice[1][1]-lattice[0][1],'ok', color=colors[i])
C:\Users\Tyjal\AppData\Local\Temp\ipykernel_5260\2459911905.py:16: UserWarning:
color is redundantly defined by the 'color' keyword argument and the fmt string
"ok" (-> color='k'). The keyword argument will take precedence.
  plt.plot(real_orbitals[i][0]+lattice[1][0]-
lattice[0][0],real_orbitals[i][1]+lattice[1][1]-lattice[0][1],'ok',
color=colors[i])
C:\Users\Tyjal\AppData\Local\Temp\ipykernel_5260\2459911905.py:17: UserWarning:
color is redundantly defined by the 'color' keyword argument and the fmt string
"ok" (-> color='k'). The keyword argument will take precedence.
  plt.plot(real_orbitals[i][0]-lattice[1][0]+lattice[0][0],real_orbitals[i][1]-
lattice[1][1]+lattice[0][1],'ok', color=colors[i])
```



[14]:
```
#2d square
test_lattice = [np.array([1,0,0]),np.array([0,2,0]),np.array([0,0,0])]
test_orbitals = [np.array([0,0,0])]
test_onsite = [0]
test_hopping = [[-1,0,0,[1,0,0]],[-0.5,0,0,[0,1,0]]]

path = [[0,0,0],[1,0,0],[1,0.5,0],[0,0,0]]
plot_bands(path,500,test_lattice, test_orbitals,test_onsite,test_hopping,"2d␣
↪rectangle")
```



6 this is code for converting betwean internal reciprocal lattice and cartesian coordinates.

it's somewhat of a work in progress, and not complete. But may be worked on more in the future. currently this function takes a series of lattice vectors and produces the corresponding reciprocal lattice internal coordinate vectors. These could be used to convert a k vector in internal coordinates into cartesian coordinates, for instance if you wanted the actual k_x k_y and k_z components of the wave vector.

[24]:
```
#functions for finding internal coordinates for reciprocal space
#this code will attempt to find the vectors for 1, 2, or 3 dimensional lattices.
 ↪ however for 2 dimensional lattices we use the formula for 3 dimensions, but␣
 ↪specify our own third unit vector using the cross product
def b_vecs(lattice_vecs):
    b_vecs = []
    #test dimensionality
    nonzero_lattices = []
    for i in range(len(lattice_vecs)):
        if any(lattice_vecs[i]):
```

```
            nonzero_lattices.append(np.array(lattice_vecs[i]))
    d = len(nonzero_lattices)

    if d == 0:
        print("error, cannot do zero dimensions")
    elif d > 3:
        print("error, cannot do more than three dimensions")
    elif d == 1:
        R = np.linalg.norm(nonzero_lattices[0])
        b_vecs.append(np.array(nonzero_lattices[0]/R**2))
        b_vecs.append(np.array([0,2*np.pi,0]))
        b_vecs.append(np.array([0,0,2*np.pi]))
    elif d >= 2:

        if d == 2:
            #creating a third unit vector to work with
            z = np.cross(nonzero_lattices[0],nonzero_lattices[1])
            z = z/np.linalg.norm(z)
            nonzero_lattices.append(z)


        V = np.dot(nonzero_lattices[0],np.
 ↪cross(nonzero_lattices[1],nonzero_lattices[2]))
        b_vecs.append(2*np.pi*np.cross(nonzero_lattices[1],nonzero_lattices[2])/
 ↪V)
        b_vecs.append(2*np.pi*np.cross(nonzero_lattices[2],nonzero_lattices[0])/
 ↪V)
        b_vecs.append(2*np.pi*np.cross(nonzero_lattices[0],nonzero_lattices[1])/
 ↪V)



    return np.array(b_vecs)
```

[25]:
```
#graphene
test_lattice = [np.array([1,0,0]),np.array([0.5,np.sqrt(3.0)/2.0,0]),np.
 ↪array([0,0,0])]
b_vecs(test_lattice)
```

[25]:
```
array([[ 6.28318531, -3.62759873,  0.        ],
       [ 0.        ,  7.25519746,  0.        ],
       [ 0.        ,  0.        ,  6.28318531]])
```