LSI, École Polytechnique

Arbitrary Brillouin Zones with PyMatGen

Juan L. Santana

June 24, 2024

Abstract

1. Introduction

I present my investigations on how to use the pymatgen package of python to generate arbitrary Fermi Surfaces constrained to First Brillouin Zones (FBZ), as well as band diagram along the high symmetry points on k-space.

2. Primitive Cells

Primitive cells are the irreducible representation of the material structure, i.e. the cell that contain the least amount of atoms while still describing all symmetries and periodicities. It is said to contain one and only one lattice point.

2.1. Structure class

The structure class grabs the information of the unit cell, meaning the three vectors describing the lattice and the motif, composed by the name of all atoms in the unit cell and their respective positions in cartesian coordinates:

```
    structure = Structure(lattice, ["A", "B"], [[a_1, a_2, a_3], [b_1, b_2, b_3]])

```

One can extract all this information from a .cif file. For example, take $PdCoO_2$. Running `Structure.from_file(cif_file)`, one can access the complete information of a lattice point. In the Materials Project website, there are many different options of visualization, as well as for dowloading as cif file. I present the visualization, method of extracting the cif file and final result is a table at the end of the document, **??**. I have checked that the only thing that changes is the order in which the atoms are taken into account, but the amount of them and their location is unchanged. The lattice vectors are also unchanged. Hence the cif files has the information of a single node, while the type of cell only depends on how the nodes are arranged inside the cell. For example the conventional cell may have a node in each corner while the primitive cell has it in the center.

A `structure` object has many properties. In principle one can manipulate the motif of the lattice, add or remove sites, get distances or angles, visualize and export the structure and so on.

From now on we will work with the Primitive Cell cif file.

2.2. SpaceGroupAnalyzer Class

Once a `structure` object has been created, we can feed it to the SpaceGroupAnalyzer class. The class provides a comprehensive set of tools for crystallographic symmetry analysis. It is called using `sga = SpaceGroupAnalyzer(structure)`. A non-exhaustive list of applications would be:

- Determining the space group symbol and number: `sga.get_space_group_symbol() // sga.get_space_group_number()`
- Obtaining the symmetry operations of the group: `sga.get_symmetry_operations()`
- Get the point group: `sga.get_point_group()`
- Get the primitive and conventional cell: `sga.get_primitive_standard_structure() // sga.get_conventional_standard_structure()`

Let us focus on this last option. The result of this method is a new `structure` object, as seen from the output. For the conventional cell, the result is the same as the initial cif file. However, the primitive cell has now been correctly generated.

```
sga.get_conventional_standard_structure(:
Full Formula (Co3 Pd3 O6)
Reduced Formula: CoPdO2
abc   :   2.863380   2.863380  17.873315
angles:  90.000000  90.000000 120.000000
pbc   :       True       True       True
Sites (12)
  #  SP           a         b         c
---  ----  --------  --------  --------
  0  Co2+  0.333333  0.666667  0.166667
  1  Co2+  1         1         0.5
  2  Co2+  0.666667  0.333333  0.833333
  3  Pd2+  0         0         0
  4  Pd2+  0.666667  0.333333  0.333333
  5  Pd2+  0.333333  0.666667  0.666667
  6  O2-   0.666667  0.333333  0.220857
  7  O2-   0         0         0.112476
  8  O2-   0.333333  0.666667  0.554191
  9  O2-   0.666667  0.333333  0.445809
 10  O2-   0         0         0.887524
 11  O2-   0.333333  0.666667  0.779143

```

```
sga.get_primitive_standard_structure():
Full Formula (Co1 Pd1 O2)
Reduced Formula: CoPdO2
abc   :   6.182881   6.182881   6.182881
angles:  26.777520  26.777520  26.777520
pbc   :       True       True       True
Sites (4)
  #  SP           a         b         c
---  ----  --------  --------  --------
  0  Co2+  0.5       0.5       0.5
  1  Pd2+  0         0         0
  2  O2-   0.887524  0.887524  0.887524
  3  O2-   0.112476  0.112476  0.112476

```

3. Brillouin Zone

A Brillouin Zone (BZ) can be generated from the reciprocal lattice vectors given a certain lattice. Using the method from before, we have two types of lattices:

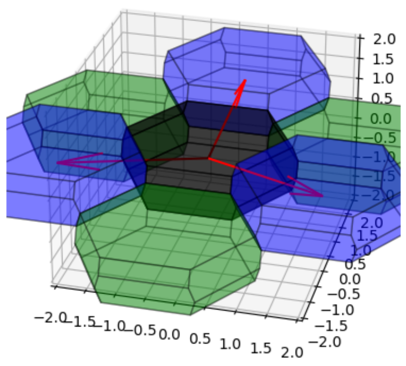

Figure 1. 1st (black), 2nd (blue) and 3rd (green) BZ'z calculated from the `lattice.get_brillouin_zone()` method and by translating these points by the positive and negative set of reciprocal basis vectors (in red).

```
Conventional Lattice Vectors :
 1.431690 -2.479760 0.000000
1.431690 2.479760 0.000000
0.000000 0.000000 17.873315
~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
Primitive Lattice Vectors :
 0.000000 1.653173 5.957772
1.431690 -0.826587 5.957772
-1.431690 -0.826587 5.957772

```

Here, each row is a different lattice vector. A BZ object can be obtained directly by the method `lattice.get_brillouin_zone()`, which produces a list of 3D points defining the vertices of the Brillouin Zone. The Brillouin Zone is by definition the primitive cell on reciprocal space, hence it must necessarily be calculated from the primitive basis. The faces of the BZ can be obtain simply by joining the vertices with the `Poly3DCollection(vertices)` method of matplotlib for example. The reciprocal basis can also be calculated from the structure using `structure.lattice.reciprocal_lattice.matrix` which returns a matrix whose rows are the reciprocal basis vectors. Fig. 1 is an example of all these plotted together.

4. Dispersion

To be able to plot Fermi Surfaces, we need to be able to describe dispersions inside this arbitrary Brillouin Zones. Suppose our BandStructure class generates a sets of points lying on the Fermi surface. We should be able to grab this set of points as points in a 3D space, and check wether or not the belong to the 1st BZ. If they do not, then one must check in which BZ they lie and perform the correct translation before accepting the point as part of the Fermi Surface. We separate then the coding of the BZ in 3 parts:

- Generation of the Brillouin Zone and checking if the points lie within
- Translation of such Brillouin Zone to generate the rest correctly
- Check algorithmically the location of each k-point on a Fermi Surface and posterior translation of such points so they fit inside the BZ.

4.1. Generation of the Brillouin Zone

The first step is obtaining the Brillouin Zone vertices using the method described in the previous section. The `get_brillouin_zone` method returns a nested array containing related vertices. To really generate a cloud of points, we need to produce a simple list of 3D points for which we can use `np.concatenate`. This is what is done to generate the polygonal view of Fig. 1.

Given these vertices, the idea would be to generate a surface and be able to distinguish points inside or outside the volume of the surface. To this end, first I generate a set of points linearly interpolating each pair of vertices. The result is a cloud of points that fills the volume of the Brillouin Zone. The function is not complicated to implement:

```python
def generate_equidistant_points(point1,
    point2, density):

    # Convert points to numpy arrays for
    easier manipulation
    p1 = np.array(point1)
    p2 = np.array(point2)

    # Generate linearly spaced numbers
    between 0 and 1
    t_values = np.linspace(0, 1, density +
    2)  # +2 to include the endpoints

    # Interpolate points
    points = [p1 + t * (p2 - p1) for t in
    t_values]

    return points

def generate_point_cloud(BZ,n_points = 5):
    vertices_BZ = np.concatenate(BZ)
    cloud = []
    for i, v_i in enumerate(vertices_BZ):
        for j , v_j in enumerate(
    vertices_BZ):
            if not(i == j):
                cloud += [
    generate_equidistant_points(v_i,v_j,
    n_points)]
    return np.concatenate(cloud)

```

Here, one function grabs two points and generates more in between, while the other grabs an object directly generated by the `get_brillouin_zone` method and generates a cloud of the selected density of points. 5 is generally enough.

The next point is key. Using the `Delaunay` algorithm of scipy, I generate polygons that join every single point of the cloud with each other. This way, we are essentially creating a volume in which almost every point is contained inside one of this polygons.

Finally, I can check for an arbitrary point in 3D, if it belongs to any of this polygons using `delaunay.find_simplex(p) >= 0`, where p is said point. This variable is `True` if p is inside the first Brillouin Zone, and negative otherwise.

I present some results in Fig. 2, in which I generate clouds of 2×10^4 random points and check if they are inside the BZ. One can clearly see how the shape of the BZ is drew by the correct points. One can also detect points in other BZ by translating the cloud previous to applying `Delaunay` by a reciprocal basis vector. In the future, it would be nice to check what is the errors in these method (points missed or included by mistake) in relation to the density of points in the interpolation. Also how it scales with time, since we would need to calculate quite a few of these BZ's, it could get computationally

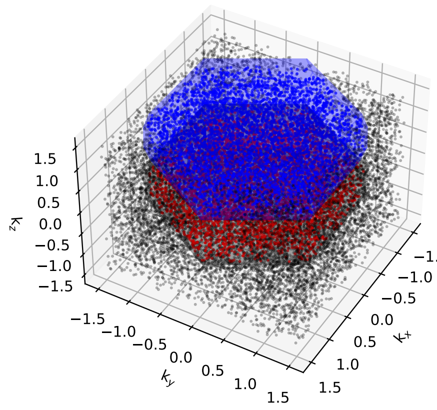

Figure 2. For a big cloud of points, one can clearly see the shape of the BZ. A close look will reveal the correct shape of all surfaces. The red points correspond to the fisrst BZ, while the blue ones correspond to the BZ generated by the reciprocal vector $\vec{G} = \vec{a}_1^* + \vec{a}_2^* + \vec{a}_3^*$
.

heavy.

4.2. Translation of k-points to the first Brillouin Zone

The last step, now that we can detect which points are inside which Brillouin Zone, is to map those outside the first one back to it. The idea is that, if a point $\vec{k}$ is detected inside a BZ that has been generated by an arbitrary reciprocal vector $\vec{G} = n_1\vec{a}_1^* + n_2\vec{a}_2^* + n_3\vec{a}_3^*$, with $\{n_i\}_{i=1,2,3} \in \mathbb{Z}$, then this point is completely equivalent to another point $\vec{k}' = \vec{k} - \vec{G}$ which lives in the 1st BZ. Hence, given a set of points, we will generate clouds of points like the previous one for every possible *translation*$\vec{G}$, use the `Delaunay` method to check which points are there, and translate back all positive cases by the known $\vec{G}$ translation.

Given a set of points K and the set of possible translations G, I wrote a program that loops over all G's generating the BZ, checking wheter the k-points are there, are translating them back to the 1st. The result is a list FS (from Fermi Surface) which contains points only inside the 1st BZ.

```
def Fermi_Surface(prim,k_points,
    density_of_cloud = 5,Visualize_quiver =
     False,savefig =False):
    primBZ = prim.lattice.
    get_brillouin_zone()
    RL_basis = prim.lattice.
    reciprocal_lattice.matrix

    if Visualize_quiver:
        cmap = mpl.cm.get_cmap("magma", len
    (G_vectors))
        colors = cmap(np.arange(len(
    G_vectors)))
        np.random.shuffle(colors)

    G_vectors = get_Gvectors()
    FS = np.empty((0,3))

    cloud = []
    for i, bz in enumerate(G_vectors):
        bz_vertices = []
        for vertex in primBZ:
            bz_vertices.append(vertex + np.
dot(RL_basis.T,bz))

        if k_points.size == 0:
            break
        print("\n Current Bz under
investigation :", bz,"\n")

        cloud = generate_point_cloud(
bz_vertices,n_points = density_of_cloud
)
        print("cloud translated \n")

        new_bz = Delaunay(np.array(cloud))

        is_inside_newBZ = new_bz.
find_simplex(k_points) >= 0
        print("New mask :  \n",
is_inside_newBZ)

        new_FS = k_points[is_inside_newBZ]-
np.dot(RL_basis.T,bz)

        if Visualize_quiver:
            if (new_FS.size !=0):
                print_BZ(ax,bz_vertices,
color=colors[i],alpha=0.1)

            for p in k_points[
is_inside_newBZ]:#new_FS:
                ax.scatter(*p,c=colors[i],
alpha = 0.3)
                p_tr = p - np.dot(RL_basis.
T,bz)
                dist = p_tr-p
                ax.scatter(*p_tr,c=colors[i
])
                ax.quiver(*p,*dist,length =
 1, color=colors[i], arrow_length_ratio
=0.1)

        FS = np.concatenate((FS,new_FS))
        print("New FS : \n", FS)

        not_inside = np.array([not(el) for
el in is_inside_newBZ])
        k_points = k_points[not_inside]

        print("Remaining points for
investigation : \n", k_points)

    if Visualize_quiver:
        print_BZ(ax,primBZ,alpha = 0.2)
        # print_Delaunay(ax,primBZ)

        ax.view_init(elev=45, azim=30)
        ax.set_xlim([-2,2])
        ax.set_ylim([-2,2])
        ax.set_zlim([-2,2])
        if savefig:
            fig.savefig("
Succesful_translation_to_1stBZ_5pts.pdf
")
        plt.show()
    return FS
```

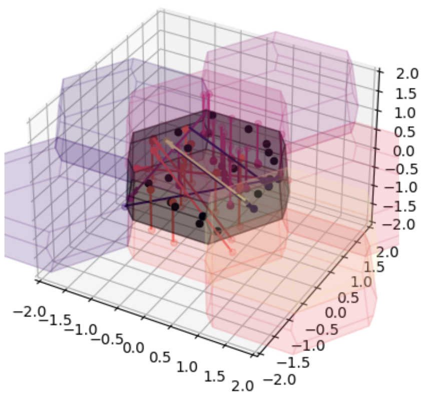

Figure 3. Translation of points from neighbouring BZ to the first one

We can also visualize that the program is correctly functioning by allowing `Visualize_quiver` to be True, which I show in Fig. 3

5. The marching cubes algorithm

One might think that it would be enough to simply produce a dense grid of points and clip it to the first BZ