1 What is PiLab?

PiLab[1] is a Scilab[2] based code that calculates condensed matter problems for periodic (so called π-Lab) lattice systems using tight-binding approaches.

What Can PiLab Do?

The current version PiLab-0.7.0 provides the following functions:

- Core Layer:

- Periodic lattice structure construction
- Generaion of hopping integrals using Slaster-Koster methods
- Basis transformation (relativistic/non-relativistic, cubic/spherical)
- Spin-orbit interactions
- Self-Consistent TB+U and Fermi level calculations.

- Application Layer:

- Band structure
- Density of state
- Floquet Hamiltonian
- Chern number
- Z2 invariant (Testing)
- Mapping of Multipolar Exchange Interaction (Testing)
- Defect state (Testing)
- Quantum Transport using surface Green's function method (Testing)

- Extension Layer:

- Floquet band structure
- Floquet Chern number
- Floquet Z2 invariant (Testing)

More functions will be added in the near future.

Features of PiLab

- Easy Graphing: Thank to the powerful graphic ability of Scilab, PiLab can let you view your calculation results immediately
- Easy Extracting: PiLab is designed to output all calculation results in an unified format. It also provides several functions to help you load and pass these data to other program. So you can easily extract the calculated data from PiLab and use your own codes for further data processing.
- Easy Extending: PiLab uses an open framework. It also provides tools for you to easily add your own code to PiLab. So you can extend PiLab base on the current functions to deal with tasks that are not supported without understanding the other part of PiLab.

[1]PiLab is a part of a library called the "PiLib". PiLab is to provide users an integrated input and output interface to solve particular tasks using the subroutines collected in PiLib. Therefore, "PiLib" may appear frequently in output files but it is not a typo.

[2]Scilab is a free and open-source matlab-like scientific computing language. Many resources can be found in http://www.scilab.org.

2 Installation

In the following, we will use a few notations to help the user understand how to use PiLab. 'ABC' means a string **ABC** in scilab console and we will leave "ABC" as the usually meaning of a quotation mark. " $\rightarrow$ *ABC* " means to input ABC as a command in the console of Scilab.

To install PiLab, you must install Scilab first. The latest version of Scilab can be found here: http://www.scilab.org. Once Scilab is installed, follow the instruction to install PiLib.

- Download PiLib source code from http://sites.google.com/site/pilabproject/
- Unzip the file and put it to somewhere you like, say, /User/PiLib/
- Open the file scilab.start. In windows, it should be in C:/Program Files/scilab-5.5.0/etc, in Linux, it should be in /scilab-5.4.0/share/scilab/etc.
- Find this comment in the file.

  // Protect variable previously defined

  Add the following command just below this comment and **before the keyword predef("all")**[3]

  PiLib_path='/User/PiLib/';

  PiLib=PiLib_path+'PiLib_loader.sce';

  exec(PiLib);

- Open Scilab, if there is no any error message, your installation should be correct. To test it, let's try to execute the cubic harmonic spin-1 tensor operator generator in PiLib:

  $\rightarrow$ *PIL_TO_gen(1,'c')*

  if it gives you nine 3x3 matrices, it works perfectly.

3 Framework of PiLab

3.1 Layer Sturcture

All tasks in PiLab are called by a file named by the format **project_task.plb**. For example, if one has a project called "Graphene" and wants to generate its lattice structure, it should be named as "Graphene_lat.plb" where "lat" is the keyword for lattice construction in PiLab. Other keywords for tasks like "hop" (hopping interal), "scc" (self-consistency), etc.

PiLab uses a layer structure which means all tasks are classified into a layer. Tasks in lower layer require the completion of higher layer. There are three layers, "core layer", "application layer" and "extension layer". Core layer is the basic layer. It defines the crystal lattice and generate the corresponding electronic structures. All the tasks in other layer will need the completion of all the tasks in this layer. In summary, it does three things: 1). lattice structure construction (lat), 2). generation of hopping integrals (hop) and 3). self-consistent calculation (scc). The order of the tasks in core layer cannot be changed, so you cannot do task **hop** before task **lat**.

As for the application layer, it contains several tasks that can calculate the physical properties form the results of core layer (electronic structure) directly. All the tasks in this layer are independent and will not influence each other. The only requirement is one has to complete all core layer calculations. Tasks in this layer are band structure calculation (ban), density of state (dsn), Floquet Hamiltonian (flq), Chern number (chn), etc.

Although the application layer can give us many physical properties, some properties cannot be obtained directly from the electronic structures. For example, Floquet problem, modification the electronic structure by shinning a light, requires one to calculate the Floquet Hamiltonian first. Once the Floquet Hamiltonian is obtained, the Floquet band and Floquet Chern number can be calculated. So those functions cannot be included in the application layer. Therefore, I introduce the extension layer. The

[3]If you are a developer of PiLib, you should not include *exec(PiLib);* because the *predef("all")* command will prohibit you to reload PiLib promptly. Instead, you should run $\rightarrow$ exec(PiLib) every time you want to use PiLib

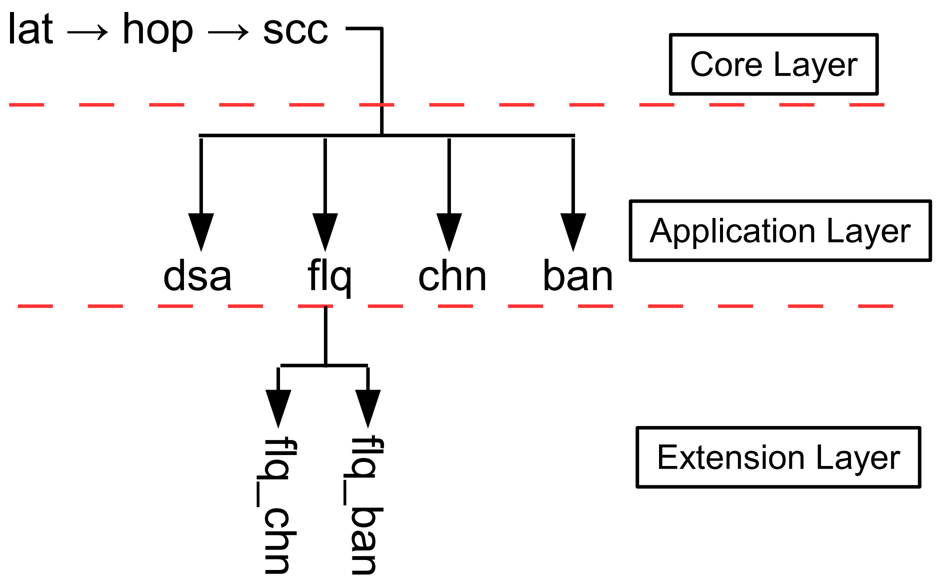

Figure 1: Task flow chart of PiLab-0.7.0

extension layer contains the extended calculations of a particular task in the application layer. To avoid confusion, all the task name in this layer will have the form "app_ext" where "app" means the corresponding task in the application layer and "ext" means the task in extension layer. For example, task name "flq_ban" means to calculate the Floquet band structure form the "flq" results. Also, "flq_chn" means to calculate the Floquet Chern number from the "flq" results. A flow chart of PiLab tasks of the version 0.7.0 is shown in Fig.1 and the meaning of each task keyword are listed below:

Core level

lat: lattice structure generation

hop: hopping integrals generation

scc: self-consistent calculation

Application level

ban: band structure calculation

dsa: density of state spectrum

flq: Floquet Hamiltonian generation

chn: Chern number calculation

Extension level

flq_ban: Floquet band structure calculation

flq_chn: Floquet Chern number calculation

Since this document just introduce the framework of PiLab, we will not change this flow chart or the list of task keywords if new function are released. Instead, we will just update **PiLab Dictionary**, another document that describes the details of each task.

3.2 Input and Output

The basic logic of PiLab is that one prepares an input file of a task. Then run PiLab to perform corresponding calculations. When it is done, all calculation results will be appended to the input file. Hence the task file **project_task.plb** will store not only the input but also the output information. If we perform next task, the task file will automatically read all necessary **.plb** files (at the same time, generates many temporary files with extension **.sod**) to continue new calculation. Note that, when performing a task, PiLab will generate many files with extension **.sod**. It stores the binary data of each **.plb** file. They are just temporary files during the calculations and you can delete it if you don't want them. It won't affect any calculation.

To demonstrate the input and output structure of PiLab, we will begin with a baby calculation, a lattice structure calculation of two-diemnsional Graphene. To this end, first you have to prepare an input file **Graphene_lat.plb**. PiLab has a useful function **PiLab_create** to help users generate template files for all tasks. For some tasks, this function will also set appropriate default values for some parameters.

To generate **Graphene_lat.plb**, type the command in Scilab console (you should create a folder to collect all your project file and *change the Scilab working folder to there*):

```
→ PiLab_ create('Graphene','lat')
```

Then PiLab will generate a file Graphene_ lat.plb in the working folder. This file can be opened and edited by any text editors (It is highly recommend to use Notepad++ or Pspad if you are using Windows platform because they can help you manage projects easily). In the file, there should be:

```
lat.Const=[ ]          // lattice constant, 1x1 real
lat.Primitive=[ ]      // Primitive vectors, (3x3/2x2/1x1)
lat.Sublatt=[ ]        // sublattice position, (nx3/nx2/nx1/)
lat.Order=[ ]          // Nearest Neighbor Order, 1x1 integer
```

A typical input file will look like this. There are few input syntax lat.Xxxx. The square brackets are where to put the input values. The words after "//" are comments which will not be read by PiLab. One may also notice that all sub-syntax are capitalized in their initial, e.g lat."C"onst, lat."P"rimitive, etc. This is a convention in PiLib. If you wants to generate a input file by yourself instead of using the built-in PiLab_create function, be sure to make it correct.

I'm not going to explain the meaning of each parameter at this stage. For the meaning and the details, please check PiLab Dictionary. Now, input the values for each parameters (Note, comma "," and semicolon ";" are different! If you are a Matlab user, you should know their difference):

```
lat.Const=[1]                                  // lattice constant, 1x1 real
lat.Primitive=[-3/2,sqrt(3)/2;3/2,sqrt(3)/2]   // Primitive vectors, (3x3/2x2/1x1)
lat.Sublatt=[0,0;1,0]                          //sublattice position, (nx3/nx2/nx1/)
lat.Order=[2]                                  // Nearest Neighbor Order, 1x1 integer
```

Save it, then type the command in Scilab console (make sure you are in correct working folder):

```
→ PiLab('Graphene','lat')
```

When the calculation is done, check **Graphene_lat.plb** again, you will find the file has been modified. The calculation results has been appended to the file as shown in Fig.2. In the appended part, one can immediately find all the output variables start with a statement:

```
============= PiLib Variable =============
```

This is a conventional statement of the PiLib. All the variable in PiLab is a matrix and PiLib will output them with this header. Just below it, there are two explanation line. Taking the first variable as an example:

```
lat.recip_vec, @full, the reciprocal lattice vectors
ORDER= 0, SIZE=[ 2, 2], TYPE=REAL
```

```
lat.Const=[1]                          // lattice constant, 1x1 real
lat.Primitive=[-3/2,sqrt(3)/2;3/2,sqrt(3)/2]   // Primitive vectors, (3x3/2x2/1x1)
lat.Sublatt=[0,0;1,0]                   // sublattice position, (nx3/nx2/nx1/)
lat.Order=[2]                          // Nearest Neighbor Order, 1x1 integer

============= PiLib Variable =============
lat.recip_vec, @full, the reciprocal lattice vectors
ORDER=    0, SIZE=[    2,    2], TYPE=REAL

         1          2

   -2.094395    3.627599
    2.094395    3.627599

============= PiLib Variable =============
lat.surr_site(1), @full, surrouding sites [order, dist, sublatt, n1, n2, n3, x, y, z]
ORDER=    0, SIZE=[   10,    9], TYPE=REAL

         1          2          3          4          5          6          7          8          9

    0.000000    0.000000    1.000000    0.000000    0.000000    0.000000    0.000000    0.000000    0.000000
    1.000000    1.000000    2.000000    0.000000   -1.000000    0.000000   -0.500000   -0.866025    0.000000
    1.000000    1.000000    2.000000    0.000000    0.000000    0.000000    1.000000    0.000000    0.000000
    1.000000    1.000000    2.000000    1.000000    0.000000    0.000000   -0.500000    0.866025    0.000000
    2.000000    1.732051    1.000000   -1.000000   -1.000000    0.000000    0.000000   -1.732051    0.000000
    2.000000    1.732051    1.000000   -1.000000    0.000000    0.000000    1.500000   -0.866025    0.000000
    2.000000    1.732051    1.000000    0.000000   -1.000000    0.000000   -1.500000   -0.866025    0.000000
    2.000000    1.732051    1.000000    0.000000    1.000000    0.000000    1.500000    0.866025    0.000000
    2.000000    1.732051    1.000000    1.000000    0.000000    0.000000   -1.500000    0.866025    0.000000
    2.000000    1.732051    1.000000    1.000000    1.000000    0.000000    0.000000    1.732051    0.000000

============= PiLib Variable =============
lat.surr_site(2), @full, surrouding sites [order, dist, sublatt, n1, n2, n3, x, y, z]
ORDER=    0, SIZE=[   10,    9], TYPE=REAL

         1          2          3          4          5          6          7          8          9

    0.000000    0.000000    2.000000    0.000000    0.000000    0.000000    1.000000    0.000000    0.000000
    1.000000    1.000000    1.000000   -1.000000    0.000000    0.000000    1.500000   -0.866025    0.000000
    1.000000    1.000000    1.000000    0.000000    0.000000    0.000000    0.000000    0.000000    0.000000
    1.000000    1.000000    1.000000    0.000000    1.000000    0.000000    1.500000    0.866025    0.000000
    2.000000    1.732051    2.000000   -1.000000   -1.000000    0.000000    1.000000   -1.732051    0.000000
    2.000000    1.732051    2.000000   -1.000000    0.000000    0.000000    2.500000   -0.866025    0.000000
    2.000000    1.732051    2.000000    0.000000   -1.000000    0.000000   -0.500000   -0.866025    0.000000
    2.000000    1.732051    2.000000    0.000000    1.000000    0.000000    2.500000    0.866025    0.000000
    2.000000    1.732051    2.000000    1.000000    0.000000    0.000000   -0.500000    0.866025    0.000000
    2.000000    1.732051    2.000000    1.000000    1.000000    0.000000    1.000000    1.732051    0.000000
```

Figure 2: Graphene_lat.plb

The first line contains three information. The first is the variable name: **lat.recip_vec**. It is the variable name that used in the code. "@full" means the output data is a full matrix rather than a sparse matrix (there are three output format, @full, @a-sp, @t-sp. Their difference will be introduced in next section). **the reciprocal lattice vectors** is a short comment of this variable, so users can understand the meaning of this output data. The second line contains the information of its numerical properties. Because all PiLib variables are output using scientific notation. Therefore, all values in PiLib variables range between $\pm 0.000000 \sim \pm 1.000000 \times 10^n$. Here **ORDER= 0** means all the output matrix elements have to multiply 10^0. **SIZE=[2, 2]** means the output matrix has a size of 2x2. **TYPE=REAL** means the output matrix are real values. There are other output types, e.g COMPLEX, SPARSE. For details, see next section. After the explanation line, comes the column number line. This is nothing but the column number of your output matrix. The matrix:

```
-2.094395    3.627599
2.094395    3.627599
```

is the output data. Combine all information, the first variable can be understood as: 1) the variable name is **lat.recip_vec** 2). it is a full matrix rather than a sparse matrix 3). this variable describes the reciprocal lattice vectors 4). this matrix has to multiple 10^0 5). This is a 2x2 matrix 6). This is a real value matrix. 7). The matrix elements are [-2.094395, 3.627599 ; 2.094395, 3.627599] which means the two reciprocal lattice vectors are [-2.094395, 3.627599] and [2.094395, 3.627599] respectively.

Therefore, to perform a task in PiLab, all you have to do is to understand the meaning of each input parameters and give them appropriate values. Then understand the meaning of each output variables. The details of each task should consult **PiLab dictionary** which can be found on the official website.

A very important concept in PiLab is that **one can always manually change the output data and perform all later calculations based on the manually changed results**. It allows users to deal with special problems where their hopping integrals or density matrices cannot be obtained by simple Slaster-Koster scheme. An interesting example is Haldane model where the second nearest hopping integrals are modified by a magnetic field with net flux equal to zero. This example can be found in the **PiLab example depot** on the official website.

4 Output Format

In the last section, we have mentioned there are several format of the output data. In fact, there are three types to output a matrix: @full, @t-sp, @a-sp, and four numerical types: STRING, REAL, COMPLEX, SPARSE. The reason that I introduce these types is to make the output data more concise and readable. In Fig.3, typical output format of each combination has been shown. Consider we have a NxM matrix. If this matrix has no special property, the easiest way is to output the whole matrix as found in the variable **lat.recip_vec** of Fig.3. However, it is possible that the matrix are zero everywhere except some matrix elements. If so, a better way to show the matrix is to output the locations and the values of non-zero matrix elements only. This idea is called sparse matrix. It is what @a-sp does (all + sparse, so called @a-sp). Futhermore, if the matrix is sparse and hermitian, we can even show the non-zero matrix elements of the upper triangle part only. It is what @t-sp does (triangle + sparse, so called @t-sp).

As for the numerical type: STRING, REAL, COMPLEX, SPARSE. there are nothing special and one should easily understand their meaning. Just remember that for @a-sp or @t-sp format, PiLib will show the indies and the values of the non-zero matrix elements rather than the whole matrix. In Fig.3, **hop.hop_mat(1)(:,:,3)** is a @a-sp matrix. For a sparse matrix, **SIZE=[2, 3]** means the matrix in sparse format is 2x3. It doesn't mean the full matrix is 2x3. In first row of the output data: 2 2 0.000000 0.000000 means the full matrix is 2x2 and 0.000000 0.000000 has no meaning (just for alignment). The below part shows the indies and values of the non-zero matrix elements. Therefore, the output data demonstrates the variable **hop.hop_mat(1)(:,:,3)** is a 2x2 matrix and the only non-zero matrix element is (1,2) and its value is 1.000000+0.000000i (by default, all sparse matrices are shown as complex values). Similar case can be found in **scc.DM_out**. However, this is a @t-sp variable, so the output data are just its upper triangle.

Examples of COMPLEX and STRING can be found in **ban.k_vec(:,:,1)** and **hop.state_info_text**. For COMPLEX, each matrix element is represented by two real values. The first one is the real part and the second one is the imaginary part. As for STRING, because a blank ' ' can also be a part of a string, so PiLib uses the notation "#" to separate each string.

```
============= PiLib Variable =============
lat.recip_vec, @full, the reciprocal lattice vectors
ORDER=    0, SIZE=[    2,    2], TYPE=REAL

         1          2

   -2.094395    3.627599
    2.094395    3.627599

============= PiLib Variable =============
hop.hop_mat(1)(:,:,3), @a-sp, hop_mat between site-1 and its 3-th neighbor
ORDER=    0, SIZE=[    2,    3], TYPE=SPARSE

      1       2                  3

      2       2    0.000000  0.000000
      1       2    1.000000  0.000000

============= PiLib Variable =============
ban.k_vec(:,:,1) , @full, eigenvectros at k_1=[0, 0, 0]
ORDER=   -1, SIZE=[    2,    2], TYPE=COMPLEX

                  1                    2

   -7.071068  0.000000    7.071068  0.000000
    7.071068  0.000000    7.071068  0.000000

============= PiLib Variable =============
scc.DM_out, @t-sp, the output density matrix
ORDER=    0, SIZE=[    3,    3], TYPE=SPARSE

      1       2                  3

      2       2    0.000000  0.000000
      1       1    0.500000  0.000000
      2       2    0.500000  0.000000

============= PiLib Variable =============
hop.state_info_text, @full, [state_label, site, n, l, SubOrb_text]
ORDER=    0, SIZE=[    2,    5], TYPE=STRING

1 # 1 # 1 # 1 # 5 P  z,d #
2 # 2 # 1 # 1 # 5 P  z,d #
```

Figure 3: output format

5 Exercise

In this section, I will lead you to have a quick exercise. Let's try to calculate the band structure, the density of state and the Chern number, etc, of a graphene. All the input files are prepared and can be downloaded from the official website (see Graphene folder in PiLab Example Depot section). You may find there are several files named **Graphene_xxx.plb**. The meaning of the input parameters and the output variables can be found in the **PiLab dictionary**. Here I will just lead the users to run this project. One should also reference PiLab dictionary when a task is performed.

First we will need to complete all core layer tasks. As mentioned in the previous sections, the tasks in core layer have to follow the order **lat** → **hop** → **scc**. Therefore, follow the steps:

- change the working folder of Scilab to the folder where you put all the **.plb** files
- → PiLab('Graphene','lat')
- → PiLab('Graphene','hop')
- → PiLab('Graphene','scc')

While perform each step, one should also check the changes of each **.plb** files and reference the **PiLab dictionary** for the meaning of each output variable. Once the above steps are done. We can now perform **.ban**, **.dsa**, **.chn** and **flq**. Since each task in application level are independent, one can perform these tasks with arbitrary order.

- → PiLab('Graphene','ban')
- → PiLab('Graphene','dsa')
- → PiLab('Graphene','chn')
- → PiLab('Graphene','flq')

While perform **ban** and **dsa**, PiLab will also draw the band structure and density of state. You can check them with the results obtained form the analytical solutions of the 2x2 graphene Hamiltonian which has been formulated by many textbooks. One can also perform Floquet band and Floquet Chern number calculations .

- → PiLab('Graphene','flq_ban')
- → PiLab('Graphene','flq_chn')

Note that, **flq_ban** and **flq_chn** are tasks in the extension layer. Their father task is **flq**. If it is not performed first, **flq_ban** and **flq_chn** will not be able to run.

6 Loading Data

It is likely that you just want to use part of the functions in PiLab and use your own codes for further data processing. If so, use the function: **PiLab_loader('project','task')**. For example, if you want to load the data in **graphene_hop.plb** file, just use

→ PiLab_load('graphene','hop')

Then all the data including inputs and outputs will automatically be stored in a file named **graphene_hop.sod**. This is a Scilab binary format file. To load it into the Scilab workspace, use:

→ load('graphehe_hop.sod')

Now it becomes a normal variable in Scilab workspace named **hop**, so can use any Scilab allowed methods to handle the data.

7 Conclusion

All the basic concepts of PiLab have been introduced in this article. The main idea of using PiLab is 1). prepare the input file using **PiLab_create** function 2). understand the meaning of output data 3). perform each task based on the flow chart as shown in Fig.1. PiLab will continue to develop new functions and make it more flexible to deal with various condensed matter problems. However, unless needed, we will not update this document frequently because the framework of PiLab will not change. Instead, we will update the **PiLab dictionary** and **PiLab example depot**. These resources can be found in the official website of PiLab.