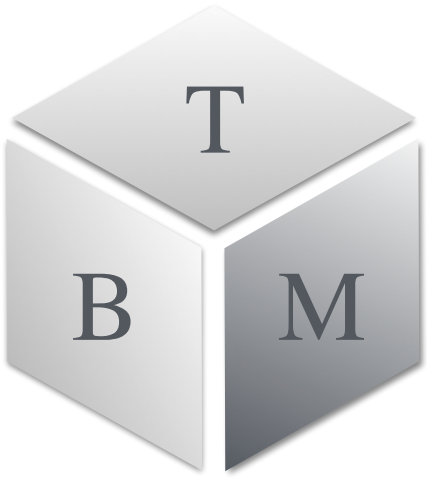
Tight Binding Model for Materials at Mesoscale: TBM^3

Yuan-Yen Tai



Contents

1 Introduction

"Tight Binding Model for Materials Mesoscale" is a C++ based package and framework that designed for construct 'any' kind of lattice structure with multi-orbital and spin degree of freedom to solve the basic nano-scale electronic structure. Our goal is to construct a highly flexible and reusable framework for anyone to easily solve the multi-scale quantum mechnical model, $H^{tot} = H^0 + H^{int}$.

$$H^0 = \sum_{i\delta,\alpha\beta,s} t_{i,i+\delta,\alpha\beta}\, c^{\dagger}_{i,\alpha s} c_{i+\delta,\beta s} \tag{1}$$

Listing. 1 shows a TBM^3 example code for implement the hopping term for the hopping term (Eq. 1), where this example describes a 3D model for the itinerant e_g-electron of a transition-metal-oxide. This example firstly add the chemical potential term to the Hamiltonian, and describe the multi-scal summation operation ($\sum_{ij,\alpha\beta,s}\cdots$) in the while-loop, and finally added each component of the Hopping parameter to the Hamiltonian.

Listing 1: Example for implement H^0

```
void Hamiltonian(){
    add_Chemical_Potential();
    while( pair_iteration() ){
        add_bond_hc( "Fe:1u  O:1u  +x  t1x");
        add_bond_hc( "Fe:1u  O:1u  +y  t1y");
        add_bond_hc( "Fe:1u  O:1u  +z  t1z");
        add_bond_hc( "Fe:1d  O:1d  +x  t1x");
        add_bond_hc( "Fe:1d  O:1d  +y  t1y");
        add_bond_hc( "Fe:1d  O:1d  +z  t1z");
        add_bond_hc( "Fe:2u  O:1u  +x  t2x");
        add_bond_hc( "Fe:2u  O:1u  +y  t2y");
        add_bond_hc( "Fe:2u  O:1u  +z  t2z");
        add_bond_hc( "Fe:2d  O:1d  +x  t2x");
        add_bond_hc( "Fe:2d  O:1d  +y  t2y");
        add_bond_hc( "Fe:2d  O:1d  +z  t2z");
        add_bond_hc( "Fe:1u  O:1u  -x  t1x");
        add_bond_hc( "Fe:1u  O:1u  -y  t1y");
        add_bond_hc( "Fe:1u  O:1u  -z  t1z");
        add_bond_hc( "Fe:1d  O:1d  -x  t1x");
        add_bond_hc( "Fe:1d  O:1d  -y  t1y");
        add_bond_hc( "Fe:1d  O:1d  -z  t1z");
        add_bond_hc( "Fe:2u  O:1u  -x  t2x");
        add_bond_hc( "Fe:2u  O:1u  -y  t2y");
        add_bond_hc( "Fe:2u  O:1u  -z  t2z");
        add_bond_hc( "Fe:2d  O:1d  -x  t2x");
        add_bond_hc( "Fe:2d  O:1d  -y  t2y");
        add_bond_hc( "Fe:2d  O:1d  -z  t2z");
    }
}
```

Without the TBM^3 package, such a piece of code needs hundreds of lines of programming to be completed. However, conveniency is not the only adventage of the package. We implemented the GPU computation for the matrix operation for the Hamiltonian which can effectiently boost the calculation speed for large lattice size problem.

2 Package setup and essentials

TBM^3 is based C++ and python programming languages, therefore we need to pre install several library for C++ and python for our construction.

2.1 C++: CUDA and MAGMA library

The cuda library is used to operate the nVidia graphic processing unit (Installed on hellgirl).
The magma library is a GPU based linear algebra libary to boost the matrix operation.

2.2 PYTHON: Numpy, matplotlib, vpython, wx

We need these library to visualize the input/output file and calculated results.

2.3 Setup the TBM^3 package

TBM^3 can be downloaded from here. Currently we are still in the *alpha-testing-stage* for the TBM^3 package development, therefore the link is limited only to our group member. However, we will make it open source in the near future.

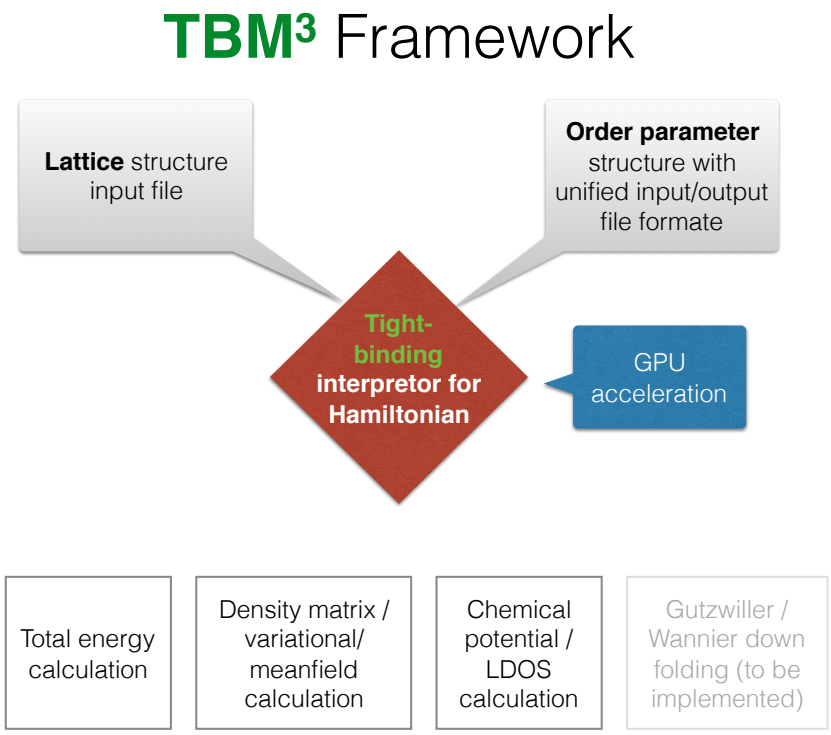
Listing 2: The folder structure of the package

```
"""
    source/
          |--> mat/  (GPU matrix operation library).
          |        |
          |        |--> include/ (include path for the compiler).
          |        |--> main.cpp (entry testing file).
          |        |--> src/ (source code).
          |        |--> ...
          |
          |--> lift/  (Lattice Interface For Tight binding).
                    |
                    |--> bin/ (place for the python code).
                    |        |
                    |        |--> lift.py (python script
                    |        |              for processing the lattice file).
                    |        |--> vlift.py (python script
                    |        |              for visualize input/output file).
                    |        |--> wxBand.py (python script
                    |        |              for visualize calculated bandstructure).
                    |        |--> mhg (simple script for compile the GPU based code).
                    |        |--> m16 (compile environment settings).
                    |
                    |--> include/ (include path for the compiler).
                    |--> src/ (TBM^cube source code).
                    |--> examples/ (several example code of the TBM^cube).
                    |             |
                    |             |--> main_cubic.cpp (the first example).
                    |             |--> ...
                    |
                    |--> doc/ (the place for this document).
                    |--> ...
"""
```

Listing. 2 shows the folder structure of the package. The easiest way to compile someone's own code is:

- 1. set up the **source/lift/bin/** for the system path.
- 2. modify the file **source/lift/bin/m16hg** for the environmental setting.

3 The structure of TBM^3



4 Input file format

The input file format contain several control blocks.
•0. Parameters, which can construct any parameter to be used in the program.
•1. Basis Vector, gives the dimension of the unitcell.
•2. Sub Atoms, describe the position and orbital/spin degree of freedom.
•3. Supercell Dim, makes it easy to construct repeated unitcell structure and convert into a large lattice structure.
•4. Bonding basis, describe the smallest linking vector for each atom.
•5. Bondings, describe the relations of all the atoms inside the lattice, periodic boundary condition is also considered.
The first thing to do with the input file is to use the 'lift.py' script to check the formate and translate the file into a '.lif' file:
$ >: **lift.py filename.lat**
After doing this, the program will automatically generate a file '**filename.lat.lif**' (if nothing wrong has been made), the input file is ready to be used for programming.

Listing 3: The input file format for a simple cubic structure

```
#0 Parameters
 isCalculateMu  = 1
 isCalculateVar = 1
 isCalculateLDOS= 1
 isCalculateBand= 1
```

```
 Temperature= 0.001
 Mu          = 0

 t0          = 1
 t           =-t0
 Jh          = 2*t0
 Nb1         = 4
 Nb2         = 4
 Nb3         = 4
 vdt         = 0.1

#1 Basis Vector
 a1         1.0                 0.0                 0.0
 a2         0.0                 1.0                 0.0
 a3         0.0                 0.0                 1.0

#2 Sub Atoms
 Fe   1s   0.0                 0.0                 0.0

#3 Supercell Dim
 2    2    2

#4 Bonding basis
 x          1.0                 0.0                 0.0
 y          0.0                 1.0                 0.0
 z          0.0                 0.0                 1.0

#5 Bondings
+x         -x         +y         -y         +z         -z
```

Note that, after execute 'lift.py filename.lat' the script automatically added two blocks into the input file:
•7. Cell, describe all the atom informations in a supercell.
•8. Neighbors, describe all the relations based on the inofrmation of •4, •5 and •7.

Once •7, •8 has been made, you can modify it such as change the atom name to make a hetero/interface structure.

Listing 4: The part after performing 'lift.py filename.lat'

```
#7 Cell
 0          0          Fe-0       0.0                 0.0                 0.0
 1          1          Fe-0       1.0                 0.0                 0.0
 2          2          Fe-0       0.0                 1.0                 0.0
 3          3          Fe-0       1.0                 1.0                 0.0
 4          4          Fe-0       0.0                 0.0                 1.0
 5          5          Fe-0       1.0                 0.0                 1.0
 6          6          Fe-0       0.0                 1.0                 1.0
 7          7          Fe-0       1.0                 1.0                 1.0

#8 Neighbors
 0         +x:1       -x:1       +y:2       -y:2       +z:4       -z:4
 1         +x:0       -x:0       +y:3       -y:3       +z:5       -z:5
 2         +x:3       -x:3       +y:0       -y:0       +z:6       -z:6
 3         +x:2       -x:2       +y:1       -y:1       +z:7       -z:7
 4         +x:5       -x:5       +y:6       -y:6       +z:0       -z:0
 5         +x:4       -x:4       +y:7       -y:7       +z:1       -z:1
 6         +x:7       -x:7       +y:4       -y:4       +z:2       -z:2
 7         +x:6       -x:6       +y:5       -y:5       +z:3       -z:3
```

You can use another python script to visualize the lattice structure and atom positions.
$ >: **vlift.py filename.lat**

5 The programming structure

In this section, I will show how to constructure the Hamiltonian and calculate the chemical potential, band structure, LDOS, total energy and LLG spin dynamics.

$$H = \sum_{i,j(=i+\delta),\alpha\beta,\sigma} t_{ij} c^{\dagger}_{i,\alpha,s} c_{j,\beta,s} - J_h \sum_{i\alpha} \vec{\sigma}_{i,\alpha} \cdot \vec{S}_i \tag{2}$$

Listing 5: TBM^3 programming example

```
#include <stdlib.h>
#include "lift.h"

using namespace lift;

// Define the model
class MyModel: public TBModel{
public:
    MyModel(Lattice lat):   TBModel(lat){ render(); }

    void init_order()       {
        cout<<"Initialize the calculation ..."<<endl<<endl;

        // Initialize the magnetic order
        while (site_iterate()) {
            auto si = getSite();
            int rx=-1, ry=-1, rz=-1;
            if (int(si.pos[0])%2 ) rx=1;
            if (int(si.pos[1])%2 ) ry=1;
            if (int(si.pos[2])%2 ) rz=1;

            int mag_sign = rx*ry*rz;

            double theta=0, phi=0;
            if (mag_sign > 0) {
                theta   = 0+0.4;
                phi     = 0;
            } else if (mag_sign < 0) {
                theta   = pi-0.4;
                phi     = 0;
            }

            double  Sx = sin(theta)*cos(phi);
            double  Sy = sin(theta)*sin(phi);
            double  Sz = cos(theta);

            initOrder(si, si.Name()+" Sx")  =Sx;
            initOrder(si, si.Name()+" Sy")  =Sy;
            initOrder(si, si.Name()+" Sz")  =Sz;
        }
        initOrder.save(Lat, "");
    }

    void Hamiltonian()      {
        add_Chemical_Potential();
```

```
    OrderParameter order = initOrder;
    double Jh = -VAR("Jh").real();

    //---site iteration---
    while (site_iterate()) {
        auto si = getSite();
        add_hund_spin("Fe 1", Jh, order.getVars(si, "Sx Sy Sz"));
    }

    //---pair iteration---
    while (pair_iterate()) {
        auto pit = getPair();
        auto si = pit.AtomI;
        auto sj = pit.AtomJ;

        add_bond( "Fe:1u  Fe:1u  +x", +VAR("t") );
        add_bond( "Fe:1u  Fe:1u  +y", +VAR("t") );
        add_bond( "Fe:1u  Fe:1u  +z", +VAR("t") );
        add_bond( "Fe:1d  Fe:1d  +x", +VAR("t") );
        add_bond( "Fe:1d  Fe:1d  +y", +VAR("t") );
        add_bond( "Fe:1d  Fe:1d  +z", +VAR("t") );

        add_bond( "Fe:1u  Fe:1u  -x", +VAR("t") );
        add_bond( "Fe:1u  Fe:1u  -y", +VAR("t") );
        add_bond( "Fe:1u  Fe:1u  -z", +VAR("t") );
        add_bond( "Fe:1d  Fe:1d  -x", +VAR("t") );
        add_bond( "Fe:1d  Fe:1d  -y", +VAR("t") );
        add_bond( "Fe:1d  Fe:1d  -z", +VAR("t") );
    }
}

void render()           {
    // Set the electron carrier for different atoms
    setElectronCarrier  (" Fe:1.0 ");
    initElectronDensity (" Fe:1.0 ");

    // Get the reciprocal lattice vector
    auto    B = Lat.get_reciprocal();
    auto    b1=B.row(0)*0.5;
    auto    b2=B.row(1)*0.5;
    auto    b3=B.row(2)*0.5;
    Mu          = VAR("Mu").real();         // Setup chemical potential from input file.
    Temperature = VAR("Temperature").real();// Setup Temperature from input file.

    init_order();

    // Set k-points for general purpose calculation
    clear_k_point();
    unsigned N1=VAR("Nb1").real(), N2=VAR("Nb2").real(), N3=VAR("Nb3").real();
    for (double i1=0; i1<N1; i1++)
        for (double i2=0; i2<N2; i2++)
            for (double i3=0; i3<N3; i3++) {
                add_k_point( (i1/N1)*b1 + (i2/N2)*b2 + (i3/N3)*b3 );
            }

    if (VAR("isCalculateMu").real() == 1) { calcChemicalPotential(0.01,0.1); }
    if (VAR("isCalculateVar").real() == 1){ calcVariation(); }

    if (VAR("isCalculateLDOS").real() == 1) {
        selectLDOSsite(0);
        selectLDOSsite(1);
```

```
            selectLDOSsite(2);
            calcLDOS(0.01, 0.1);
        }

        // Calculate the band structure through high symmetry points
        if (VAR("isCalculateBand").real() == 1) {
            add_ksymm_point( "", +0.0*b1 +0.0*b2 +0.0*b3);
            add_ksymm_point( "", +0.0*b1 +0.0*b2 +1.0*b3);
            add_ksymm_point( "", +0.0*b1 +1.0*b2 +1.0*b3);
            add_ksymm_point( "", +1.0*b1 +1.0*b2 +1.0*b3);
            add_ksymm_point( "", +0.0*b1 +0.0*b2 +0.0*b3);

            calculateBandStructure();
        }
    }
};

int main(int argc, char** argv) {
    // Handle program arguments
    vector<string> args;
    for (int i=0; i<argc; i++) args.push_back(string(argv[i]));

    // Construct lattice structure(class) from input.
    string  filename="";
    if      (args.size()==1) { filename = "input.lat"; }
    else if (args.size()==2) { filename = args[1]; }

    if (filename.size()>0) {
        string cmd = "lift.py "+filename;
        cout<<system(cmd.c_str())<<endl;

        Lattice Lat(filename, NORMAL);
        //render the model
        MyModel tbm(Lat);
    }


    return 0;
}
```