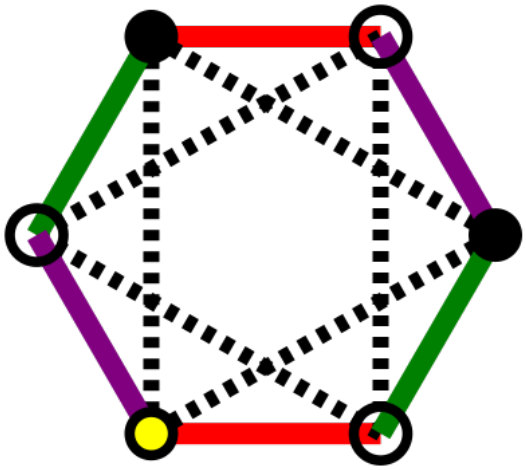


LatticeLab.jl v0.2
– A Julia package for lattice models in physics

Yunlong Lian
September 2024

Abstract

`LatticeLab.jl` is a Julia package for building lattice models in physics. It provides representations of theoretical lattice models for condensed matter, such as the tight-binding model for electrons, the atomic force model for lattice vibrations, and the Heisenberg model for localized spin. It can also be used to construct disordered lattice models for soft matter and artificial lattices for metamaterials. `LatticeLab.jl` is versatile and extendable. The source code is written in pure Julia, with self-evident names of variables and functions. Core modules are separated from extensions and can be maintained independently. The visualization facilities, including `BandStructures.jl` for ploting band structures, and `PlotSVG.jl` for displaying lattice structures, are packaged and maintained individually. Several interfaces from `LatticeLab.jl` to the post-processing packages for *abinitio* calculations, such as `Wannier90` and `Phonopy`, have been established. `LatticeLab.jl` is still under development for various applications. The core module and several components released in this version (v0.2) are stable and have already been used in several ongoing research projects. `LatticeLab.jl` will be published on GitHub when the version number reaches v1.0. Currectly it is a closed-source project with on-demand developments. Up to 28 Oct 2021, Dr. Yunlong Lian is the only developer of `LatticeLab.jl` and hold all the copyrights.

Contents

Chapter 1

Introduction

In this chapter, the motivation behind the development of Julia package `LatticeLab` is presented, followed by a brief description on the core data structures of the package and future plans on the development.

1.1 Lattice models in physics

Lattice models are widely used in theoretical studies of problems in physics. There are mainly two types of lattice models, namely the natural lattice models for materials with spatially periodic structures and the artificial lattice models originate from discretization of variable space. For the computer simulation with both types of lattice models, the first step always requires a representation of the lattice by appropriate data structures. The Julia package `LatticeLab` focuses on representation of lattices by the Julia programming language.

The natural lattice models are used to study crystalline and amorphous solids, liquids, metamaterials and artificial optical lattice. In these systems, the fundamental building blocks are atoms or wave resonators. The most important case is the crystalline solid, or *crystal* as an abbreviation. The atoms in a crystal are arranged periodically, within the minimal repeating unit called *unit cell*. Unit cells are repeated according to the *Bravais vectors*. It is well known that there are 230 distinct classes (called *space groups*) to arange atoms periodically in three-dimensional space. Wave resonators in metamaterials are arranged into a lattice in a similiar way.

Disorder can be introduced to the natrual periodic lattices in two ways, namely substitutional and geometrical. Substitutional disorder can be approximated by an ensemble of periodic lattices with supercells obtained by randomly replacing the atoms in a common unit cell. The virtual crystal approximation avoids supercells by mixing substitutions on

the same atom site, resulting in a "virtual crystal" with "interpolated atom" on the lattice sites with disorder.

Another important case of the natrual lattice is the geometrically disordered material, such as amorphous solids and liquids. One never finds repeated units in these systems. The lattice is a collection of randomly located building blocks, which satisfy certain geometrical constraints.

Quasi-crystal is an interesting type of lattice which has regular but non-repetitive structures. It can be viewed as an interpolation of regular and random lattices. The quasi-crystal lattice is generated either by projecting high-dimensional regular periodic lattice to two or three dimension, or by a set of generation rules. Systems on quasi-crystal lattice have interesting properties which are often valuable to the understanding of important aspects in regular or random lattices.

The artificial lattice model is often derived by a discretization procedure on a continuous model in theoretical studies. One well-known example is the lattice QCD, where spacetime is discretized and the gauge fields are assigned on nearest-neighbor links. The research on lattice QCD has inspired a lot of lattice models in statistical physics, which is an equally important class of lattice models in the study of critical phenomena and renormalization groups.

Recently, inspired by the research on graph neural networks, X have proposed the *crystal graph*, which is essentially a representation of the lattice with node and edge features.

The frequent appearances of lattice models motivate a detailed consideration of the computer representation of lattice. This is particularly important in the design of *abinitio* calculation programs (e.g. `Quantum Espresso`, `VASP`) and post-processing tools (e.g. `pymatgen`, `Phonopy`, `pyTB`, `WannierTools`), since in all of these programs one has to build the lattice based on user-specified crystal structures. The essential works to build up lattices in these tools are similar, however, in order to combine various tools to perform analysis, the user always have to write explicit code to build up a lattice. It is therefore necessary develop a package to coordinate numeric studies based on lattice models.

1.2 Representation of lattices in `LatticeLab`

The core data structure in `LatticeLab` is `mutable struct Lattice`. The types involved in the declaration is listed in 1.1. The lattice site `i` has coordinates `R0[:,i]` and sublattice id `SL[i]`. If site `i` is outside the bounding box `BBOX`, it may be pull back into site `EqV[i]` the box by translation. The site connectivity matrix `f` stores the links between lattice sites. It is the essential part of the lattice model. The lattice is generated by `UC::LatticeInfo` and `LN::LinkInfo`. The former generates the lattice coordinates `R0::Coordinates` and

Algorithm 1.1 Core data structure : `Lattice` and related types, `LatticeInfo`, `Coordinates`, `Indices`, `BoundingBox`

```
abstract type LatticeInfo end

Coordinates{TR} = Array{TR,2}

Index = Vector{Int64}
Indices = Array{Int64,2}

# (origin, trans, shift_units, pbc) = bbox
BoundingBox = Tuple{Vector,Matrix{Int64},Vector{Int64},Vector{Bool}}

mutable struct Lattice{TConnMat<:AbstractMatrix,TR<:Real}
    R0::Coordinates{TR}     # coordinates of lattice sites
    BBOX::BoundingBox       # bounding box of the lattice
    LN::LinkInfo            # link between pairs of lattice sites
    SL::Vector{Int64}       # sublattice id for each site
    f::TConnMat             # site connectivity matrix
    UC::LatticeInfo         # unit cell data or random property
    EqV::Index              # index of the equivalent vertex
    TransPerm::Indices
    TransNegPerm::Indices
end
```

Algorithm 1.2 Core data structure : `UnitCell` and related types `Masses`, `Orbits`

```
Var = Union{Number,Symbol}
Masses = Vector{Var}

Orbits = Vector{Symbol}

mutable struct UnitCell <: LatticeInfo
    dim::Int64          # dimension of the lattice, 2 or 3
    nsubl::Int64        # number of sublattices in a unit cell
    a::Coordinates      # Bravais basis
    delta::Coordinates  # relative coordinates of sublattices
    m::Masses           # mass / atom labels
    xi::Vector{Orbits}  # labels of atomic orbit
end
```

Algorithm 1.3 Core data structure : `LinkInfo` and related types `Direction`, `Spring`

```
Direction = Vector

Link = Dict{Tuple{Int64,Int64},Vector{Direction}}

mutable struct LinkInfo
    UC::LatticeInfo
    SPNB::Dict{Symbol,Link}
end
```

the latter generates the site connectivity matrix `f::TConnMat`. Details of the construction algorithms for `R0::Coordinates` and `f::TConnMat` is discussed in Sect.**??**.

The unit cell is often used to generate a periodic lattice. The information for a unit cell is organized by the data structure is `mutable` **`struct`** `UnitCell`. The spatial dimension and number of sites (usually called *sublattice*) are specified by `dim::Int64` and `nsubl::Int64`, respectively. The Bravais lattice vectors `a::Coordinates` is a `dim` by `dim` matrix with $\boldsymbol{a}_i$ =`a[:,i]`. The label, orbits and position of i'th sublattice site can be accessed via `m[i]`, `xi[i]` and $\boldsymbol{\delta}_i$ =`delta[:,i]`, respectively.

The data structure `LinkInfo` summarizes the pairwise interactions among lattice sites and is necessary to generate the connectivity matrix `Lattice.f`. The member `LinkInfo.SPNB` is a collection of `name::Symbol => link::Link` pairs, which describes

Algorithm 1.4 Core data structure : `MinimalLattice`

```
mutable struct MinimalLattice{NL,TN,TE}
    V::Vector{Pair{NL,TN}}
    E::Vector{Pair{Tuple{NL,NL},TE}}
end

function MinimalLattice(latt::Lattice)
    vl = make_featured_vertex_list(latt)
    el = make_featured_edge_list(latt)
    MinimalLattice(vl, el)
end
```

different types of links. A link is just a collection of `(i::Int,j::Int) => v::Direction` pairs, indicating that a link is starting at site `i` and pointing to site `j` at a location `v` away from it, i.e.

```
# meaning of type Link
```

To materialize the connectivity matrix `Lattice.f`, one has to match the link labels in `Lattice.f` to those in the data structure `mutable struct HoppingParameter` or `mutable struct ForceConstant`, which contains model-specific information on the pairwise interactions among lattice sites. The details of these data structures will be discussed in Sect.2.2.1.

In some applications, such as molecular dynamics and crystal graph neural networks, fast construction and modification of the lattice is crucial. The data structure `mutable struct MinimalLattice` is designed for this purpose. The lattice is viewed as a graph with features on vertices and edges, which are of type `TN` and `TE`. They are labelled by type `NL` and `Tuple{NL,NL}` variables, respectively.

1.3 Extension of the core module of `LatticeLab`

There are several extensions for core module of `LatticeLab` under development.

The current version of `LatticeLab` does not allow for degrees of freedoms on the links, therefore it does not provide any representations of edge models, such as the gauge field on a lattice, or the classical vertex models for spin ice. However, if all the degrees

of freedoms in the model are located on the edge, one can perform a dual transform of the lattice to map those edge degrees of freedoms onto the vertices in the dual lattice. Provided that the periodic boundary conditions are treated carefully, one can still use `LatticeLab` after this dual transform, at a cost of increased number of vertices (usually by a multiple of the coordinate number Z of the original lattice).

However, if the model includes both vertex *and* edge degrees of freedoms, the dual transform cannot be used and an extension of the lattice representation in `LatticeLab` is necessary. For example, Ref.[] studied a lattice model with electrons on vertices, interacting with the Z2 gauge fields on edges. One have to treat these degrees of freedoms on equal footing. Another example would be the graph neural network model of crystal structures. This model assigns features (degrees of freedoms) to both vertices and edges of a lattice and relies on neural message passing to train the graph neural network. The key to a successful training process is the interaction between the vertex and edge features. In these applications, essential extensions have to be made on the data structures in `LatticeLab`.

The current version of `LatticeLab` does not provide efficient edge-indexing algorithms. The present edge-index algorithm is basic and sequential. As a consequence, the application of this package on random lattices is limited. An update is scheduled for an advanced algorithm to organise and locate edges.

The cost of indexing and modification of lattice is significant in the current version of `LatticeLab`. The applications on molecular dynamics is thus limited. There is an ongoing work on the data structure `mutable struct MinimalLattice` of the core module, aiming at low-cost access of lattice.

Chapter 2

Design and implementations of `LatticeLab`

This chapter is a documentation of the development of `LatticeLab`. Important design ideas and implementation details are presented.

2.1 Principles and overview of the design

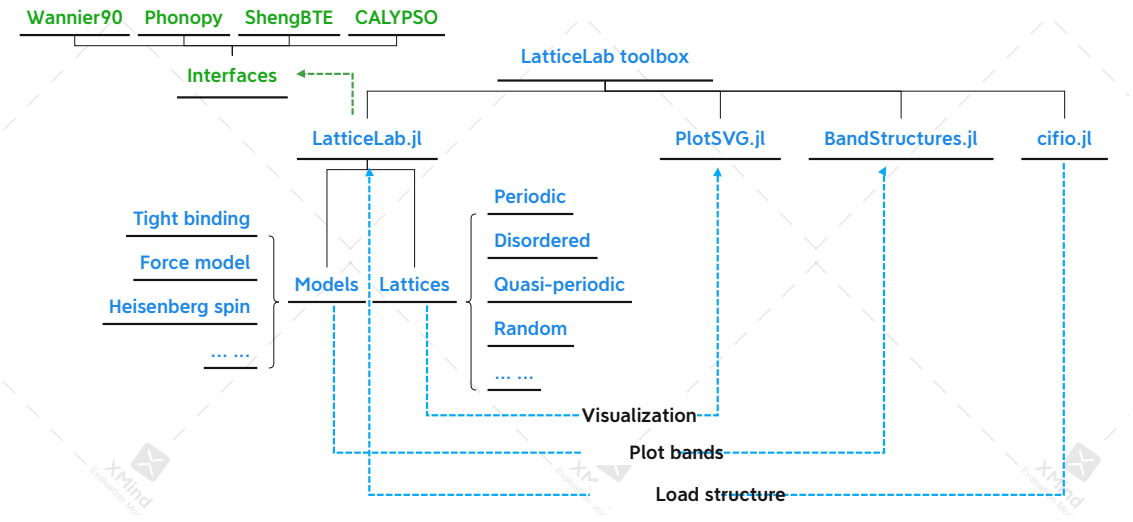


Figure 2.1: Design overview of `LatticeLab`.

The most important aspect of the design of `LatticeLab` is the unified view of lattice models — a finite collection, or a collection generated by a finite set, of sites V, combined

with a finite collection, or a collection generated by a finite set, of links E. The core comoponets of LatticeLab is designed to maximize the code reuse, so as to minimize the testing efforts and programming errors. Such unified view of lattice models has also broadened the applications of LatticeLab, since the efforts to adapt to a particular class of lattice models is minimal.

The visualizations of lattice and band structures do not, and should not have any impact on the design. They are developed as individual packages. The lattice can be visualized with the help of the package PlotSVG, which converts a graph to an SVG file. The band structure can be plotted by the package BandStrucutures, which organizes the band energies in a carefully designed data structure and generates figures using PyPlot.

The interface to read cif files is necessary. However, due to the complexity of this task and its potential application to other packages, it is also developed as an individual package cifio.

The interfaces to other tools, such as Wannier90, Phonopy, ShengBTE and CALYPSO, are designed case by case during several research projects. All interfaces parse the calculation results from the aforementioned softwares and build lattice accordingly.

2.2 Implementation details

2.2.1 Data structures

Figure.2.2 summarizes the major data structures in LatticeLab. The essential data structures to represent a lattice has been presented in Sect.1.2. They are:

```
abstract type LatticeInfo end
mutable struct Lattice{TConnMat<:AbstractMatrix,TR<:Real} end
mutable struct UnitCell <: LatticeInfo end
mutable struct LinkInfo end
```

	Hopping Hamiltonian	Dynamical matrix	Heisenberg hamiltonian
k-space	HoppingHamiltonianQ	DynamicalMatrixQ	HeisenbergHamiltonianQ
	HoppingHamiltonianenQ	DynamicalMatricesQ	HeisenbergHamiltonianenQ
r-space	HoppingHamiltonianR	DynamicalMatrixR	HeisenbergHamiltonianR
	HoppingHamiltonianenR	DynamicalMatricesR	HeisenbergHamiltonianenR

Table 2.1: Data structures for hopping Hamiltonian, dynamical matrix and Heisenberg hamiltonian.

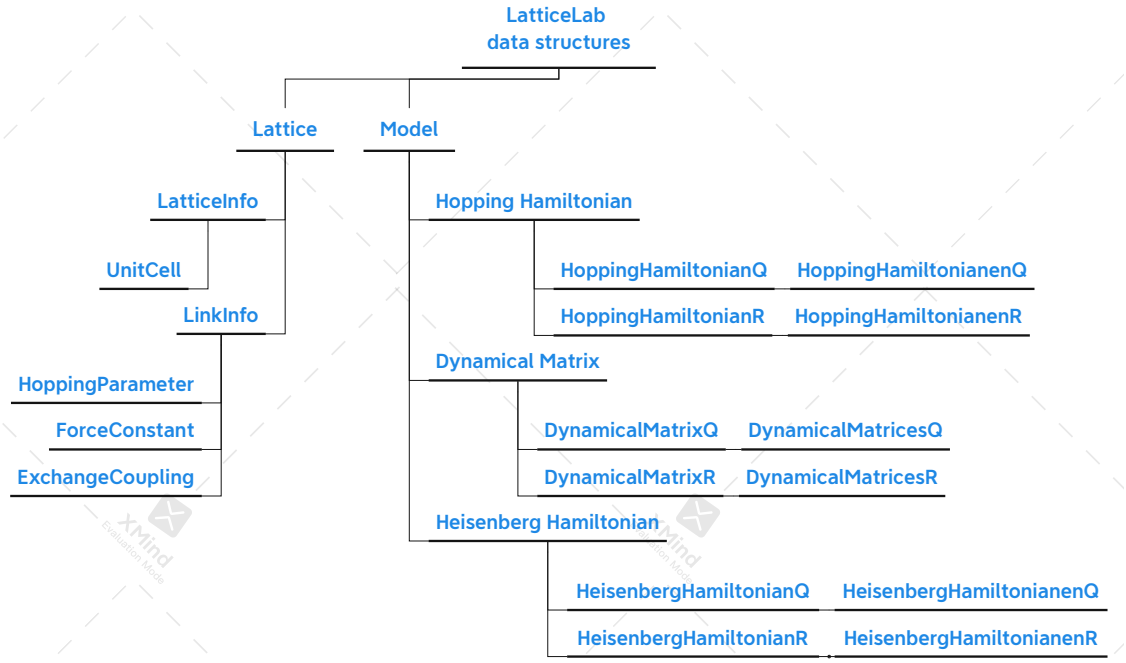


Figure 2.2: Data structures in `LatticeLab`.

`LatticeLab` provides data structures for the applications in condensed matter (Figure.2.2 and Table.2.1). [1] Their declarations are presented in Alg.2.1, Alg.2.2 and Alg.2.3, respectively. These data structures share some common features. The k-space hopping Hamiltonian, dynamical matrix and Heisenberg Hamiltonian have a common field `MAT::Dict`, which stores vector-matrix pairs of the following fomat:

```
[R1::Int, R2::Int, R3::Int] => A::Matrix
```

The Bloch Hamiltonian or dynamical matrix are then constructed by the following function:

```
qMAT(q, MAT::Dict, BASIS) = sum( cis(dot(q, BASIS * Rmnl)) .* Amnl
                                 for (Rmnl,Amnl) in MAT )
```

[1] The Heisenberg Hamiltonian for spin systems is currently under development.

which is simply

$$H(\boldsymbol{k}) = \sum_{mnl} \mathrm{e}^{i\boldsymbol{k}\cdot\boldsymbol{R}_{mnl}} A_{mnl}. \tag{2.1}$$

The field `COEFFS::Vector` in the data structures

```
HoppingHamiltonianenQ
HoppingHamiltonianenR
DynamicalMatricesQ
DynamicalMatricesR
HeisenbergHamiltonianenQ
HeisenbergHamiltonianenR
```

represents the coefficients of the list of Hamiltonians / dynamical matrices. One can perform additions and scalar multiplications on these data structures, for example

```
# H0::HoppingHamiltonianQ, Hnn::HoppingHamiltonianQ
# Htot::HoppingHamiltonianenQ
Htot = t0*H0 + t1*Hnn
```

This is very convenient in practice.

2.2.2 Indexing and updating

The indexing and updating operations on sites and edges are the most frequently used functions. `LatticeLab` provides both elementary and advanced site-indexing algorithms to meet demands from different applications. For most studies in band structures, the lattice is not modified after construction. During the construction of lattice, substantial amount of queries on neighboring sites are performed. In fact, this is the bottleneck of the performance of building and modifying the lattice. The speed of indexing algorithms is also crucial for the applications in Monte Carlo method. For typical molecular dynamics applications, the lattice is updated frequently. It is more challenging to develop a fast indexing algorithm in this case. The most demanding application of `LatticeLab` is the crystal graph neural network, which requires fast indexing and updating both sites and edges.

The current version of `LatticeLab` only provides the sequential indexing algorithms for lattice sites edges. In practice, this is sufficient for most applications in band structure calculations.

Algorithm 2.1 Data structure for hopping Hamiltonian.

```
mutable struct HoppingParameter
    UC::LatticeInfo
    HPLN::Dict
end

mutable struct HoppingHamiltonianQ
    LATT::Lattice
    MAT::Dict              # R_mn => Phi_mn
    T::HoppingParameter    #
    V::Dict                # on-site potential
end

mutable struct HoppingHamiltonianenQ
    LATT::Lattice
    MATS::Vector
    TS::Vector{HoppingParameter}
    VS::Vector             # on-site potentials
    COEFFS::Vector         # coeffs for linear combination
end

mutable struct HoppingHamiltonianR{TH<:AbstractMatrix}
    LATT::Lattice
    H::TH                  # explicit Hamiltonian matrix
    EIG::EigenModes        #
    T::HoppingParameter    #
    V::Dict                # on-site potential
end

# allows for t1*H1 + t2*H2 + t3*H3 + ...
mutable struct HoppingHamiltonianenR
    LATT::Lattice
    HS::Vector             # explicit Hamiltonian matrices
    EIG::EigenModes        #
    TS::Vector{HoppingParameter}
    VS::Vector             # on-site potentials
    COEFFS::Vector         # coeffs for linear combination
end
```

Algorithm 2.2 Data structure for dynamical matrix.

```
mutable struct ForceConstant
    UC::LatticeInfo
    FCLN::Dict
end

mutable struct DynamicalMatrixQ
    LATT::Lattice
    MAT::Dict          # R_mn => Phi_mn
    F::ForceConstant   #
    M::Dict            # mass
end

mutable struct DynamicalMatricesQ
    LATT::Lattice
    MATS::Vector       #
    FS::Vector{ForceConstant}
    M::Dict            # masses have to be the same
    COEFFS::Vector     # coeffs for linear combination
end

mutable struct DynamicalMatrixR{TD<:AbstractMatrix}
    LATT::Lattice
    D::TD              # explicit dynamical matrix
    EIG::EigenModes    #
    F::ForceConstant   #
    M::Dict            # mass
end

# allows for f1*D1 + f2*D2 + f3*D3 + ...
mutable struct DynamicalMatricesR
    LATT::Lattice
    DS::Vector         #
    EIG::EigenModes    #
    FS::Vector{ForceConstant}
    M::Dict            # masses have to be the same
    COEFFS::Vector     # coeffs for linear combination
end
```

Algorithm 2.3 Data structure for Heisenberg Hamiltonian.

```
mutable struct ExchangeCoupling
    UC::LatticeInfo
    EXLN::Dict
end

mutable struct HeisenbergHamiltonianQ
    LATT::Lattice
    MAT::Dict               # mn => M
    J::ExchangeCoupling     #
    Z::Dict                 # on-site Zeemann
end

mutable struct HeisenbergHamiltonianenQ
    LATT::Lattice
    MATS::Vector            # mn => M
    JS::Vector{ExchangeCoupling}
    ZS::Vector              # on-site Zeemann
    COEFFS::Vector          # coeffs for linear combination
end

mutable struct HeisenbergHamiltonianR{TH<:AbstractMatrix}
    LATT::Lattice
    H::TH                   # explicit Hamiltonian matrix
    J::ExchangeCoupling     #
    Z::Dict                 # on-site Zeemann
    G::Vector               # classical ground state
end

mutable struct HeisenbergHamiltonianenR
    LATT::Lattice
    HS::Vector              # explicit Hamiltonian matrices
    JS::Vector{ExchangeCoupling}
    ZS::Vector              # on-site Zeemann
    COEFFS::Vector          # coeffs for linear combination
    G::Vector               # classical ground state
end
```

2.2.3 Building lattice

One of the most important funciton in `LatticeLab` is `build_lattice()`. This function builds lattice according to the link information carried by `LnInfo::LinkInfo` and bounding box of the lattice specified by `bbox::BoundingBox`. It has the following pseudocode:

Algorithm 2.4 Building lattice.

```
function build_lattice(
    lninfo::LinkInfo,
    bbox::BoundingBox
    )::Lattice
    margin  = compute_margin(lninfo, bbox)
    R0,Subl = generate_R0(lninfo.UC, bbox, margin)
    EqV     = compute_EqV(lninfo.UC, bbox, margin)
    f       = generate_f(lninfo, R0, bbox, margin)
    shift_vectors = lninfo.UC.a * bbox[2]
    perm    = translate_permute(R0,   shift_vectors)
    Nperm   = translate_permute(R0, - shift_vectors)
    return Lattice( R0,
                    bbox,
                    copy(lninfo),
                    Subl,
                    f,
                    copy(lninfo.UC),
                    EqV,
                    perm,
                    Nperm )
end
```

The following subsections explain each step in detail.

Compute margin
`compute_margin(lninfo,bbox)`

A `Lattice` of finite collection of sites is always embedded in a larger container with a *margin*, in order to handle the boundary conditions and lattice translations. For periodic lattices, a margin simply means extra translations of the unit cell by the Bravais vectors to move the unit cell outside the bounding box.

Generating lattice sites
generate_R0(lninfo.UC,bbox,margin)

The coordinates of lattice sites are stored in `Lattice.R0[1:dim,1:nsites]`. It is generated as follows:

Algorithm 2.5 Generating lattice sites.

```
# Nsubl : number of sublattices in unit cell
# delta[:,s] : coordinate of sublattice s inside a unit cell
# a[:,i : Bravais basis vectors a_i
# iter_N_min_max : iteration range along the dimensions
all_sites = hcat([
    delta[:,s].+(a*[t...,])
        for t in iter_N_min_max
            for s in 1:Nsubl
]...)
```

Searching for equivalent lattice sites
compute_EqV(lninfo.UC, bbox, margin)

Each lattice site outside the bounding box has an equivalent site inside the box. This function searches for the equivalent sites in a straightforward way. Currently it is the major bottleneck of the function `build_lattice()`. I choose not to present the algorithm since it is still under improvement.

Constructing connectivity matrix
generate_f(lninfo, R0, bbox, margin)

The connectivity matrix `Lattice.f[1:nsites,1:nsites]` (a Julia sparse matrix) stores symbols that are used to identify links between lattice sites. The function `generate_f()` iterates on all sublattice pairs `(si, sj)` in the unit cell and all link labels `lb` and directions associated to the pair (via `lninfo.SPNB`), finds all pairs `(ri, rj)` in `Lattice.R0` which fit the setting and then assigns `Lattice.f[ri,rj] = lb`. The pseudocode is listed in Alg.2.6.

Algorithm 2.6 Constructing connectivity matrix.

```
function generate_f(
    lninfo::LinkInfo,
    R0::Array{T,2},
    bbox::BoundingBox,
    margin::Int,
    N_sites::Int
    ) where {T<:Real}
    #* f[1:N_sites,1:N_sites]  is a sparse array of symbols
    #% IJFD = reorganize lninfo.SPNB
    #% for each sublattice pair (si,sj)
        #% for each label lb in IJFD[(si,sj)]
            #% for each direction d
                #% case I
                #% for each site j0 in bounding box on sublattice sj
                    #% compute all i1 such that  d.+R0[:,j0] == R0[:,i1]
                    #% f[i1, j0] = lb
                #% case II
                #% for each site i0 in bounding box on sublattice si
                    #% compute all j1 such that  R0[:,j1] == R0[:,i0].+d
                    #% f[i0, j1] = lb

    #% return f
end
```

Generating permutation representation of lattice translations `translate_permute(R0, shift_vectors)` and `translate_permute(R0, - shift_vectors)`

Currently not implemented.

2.2.4 Constructing Hamiltonians and dynamical matrices

`LatticeLab` provides fast and convenient portals for constructing Hamiltonians and dynamical matrices in both k-space and r-space. The essential work is to assign numbers or small matrices to correct entries in the result matrix according to the link symbols stored in `Lattice.f`.

The basis ordering of the Hamiltonian and dynamical matrix is build-in and unique in `LatticeLab`. The matrix is divided into blocks according to lattice sites. Each block represents the connection between a pair of sites. The dimension of the block is determined

by the orbits on the source and destination sites. For tight-binding model, the orbits on each site have to be specified. For the force model of lattice vibration, each site has three directions of vibration. For the Heisenberg model of spin system, each site carries either classical degrees of freedom, which are the three spin directions, or quantum mechanical degrees of freedom, which is the Hilbert space of the S_z operator.

The representation of the matrix for Hamiltonian or dynamics in the code is listed in Alg.2.1, Alg.2.2 and Alg.2.3. For r-space Hamiltonians, the entire matrix is stored in the field `H` or `HS`. For k-space Hamiltonians, the matrix is stored in accordance to the Bravais vector. The vector-matrix pairs are stored in the field `MAT` or `MATS`.

2.2.4.1 k-space hamiltonians

The pseudocode to construct k-space matrix is listed in Alg.2.7.

Algorithm 2.7 Constructing k-space matrix.

```
function kspace_matrix(
    latt::Lattice,                          # lattice
    BLK::Vector{UnitRange{Int64}},          # block dimensions
    block_constructor::Function,            #
    block_constructor_diagonal::Function, #
    is_in_bounding_box::Function            #
    )
    #CONVENTION 1<--2
    #CONVENTION jj-->ii
    #% link_list <= latt
    #% delta <= sublattice coordinates w.r.t. unit cell origin
    #% MAT = empty dictionary
    #% for link in link_list
        #% for (sublattice_pair (ii,jj), direction d) in link
            #% if is_in_bounding_box(delta[:,jj] .+ d)
                #% R_mnl = compute_Bravais_vector(ii, jj, d)
                #% if ii != jj
                    #% B = block_constructor(ii, jj, d, link_symbol)
                    #% M = assign_matrix_to_pos(BLK[ii], BLK[jj], B)
                    #% MAT[R_mnl] += M
                #% else # ii==jj
                    #% B = block_constructor_diagonal(ii, d, link_symbol)
                    #% M = assign_matrix_to_pos(BLK[ii], BLK[ii], B)
                    #% MAT[R_mnl] += M

    #% return MAT
end
```

2.2.4.2 r-space hamiltonians

The pseudocode to construct r-space matrix is listed in Alg.2.8.

Algorithm 2.8 Constructing r-space matrix.

```
function rspace_matrix(
    latt::Lattice,
    BLK::Vector{UnitRange{Int64}},
    block_constructor::Function,
    block_constructor_diagonal::Function
    )
    #% M
    #% inner_id[i] <= the position in inner site list of lattice site i
    #% for (i,j,link_symbol) in findnz( latt.f )
        #% if j is inner site in latt
            #% if i is inner site in latt
                #% if i == j
                    #% A = block_constructor_diagonal(i,link_symbol)
                #% else
                    #% A = block_constructor(i,j,link_symbol)
            #% M[BLK[inner_id[i]], BLK[inner_id[i]]] += A

            #% elseif across_upper_border(i, j)
                #% ieqv = equivalent site of i
                #% if ieqv  == j
                    #% A = block_constructor_diagonal(i,link_symbol)
                    #% M[BLK[inner_id[j]], BLK[inner_id[j]]] += A
                #% else
                    #% A = block_constructor(i,j,link_symbol)
                    #% M[BLK[inner_id[ieqv]], BLK[inner_id[j]]] += M

        #% elseif (i is inner site in latt) && across_upper_border(j, i)
            #% jeqv = equivalent site of j
                #% if jeqv  == i
                    #% A = block_constructor_diagonal(i,link_symbol)
                    #% M[BLK[inner_id[i]], BLK[inner_id[i]]] += M
                #% else
                    #% A = block_constructor(i,j,link_symbol)
                    #% M[BLK[inner_id[i]], BLK[inner_id[jeqv]]] += M

    #% return M
end
```

2.2.5 Visualizing lattices and band structures

The lattices can be visualized with the package `PlotSVG.jl`. The band structure can be plotted with `BandStructures.jl`. These two packages are trivial to implement. Please examine the source code for usage and see Sect.**??** for examples.

2.2.6 Interfaces

`LatticeLab` provides interfaces to other *abinitio* calculation softwares and post-processing tools, namely `Wannier90`, `Phonopy`, `ShengBTE` and `CALYPSO`. The interface to `Phonopy` is completed. The interface to `Wannier90` is close to finish. The interfaces to `ShengBTE` and `CALYPSO` are still in progress. Please examine the source code for usage and see Sect.3.1.6, Sect.3.1.7 for examples.

Chapter 3

Applications of `LatticeLab`

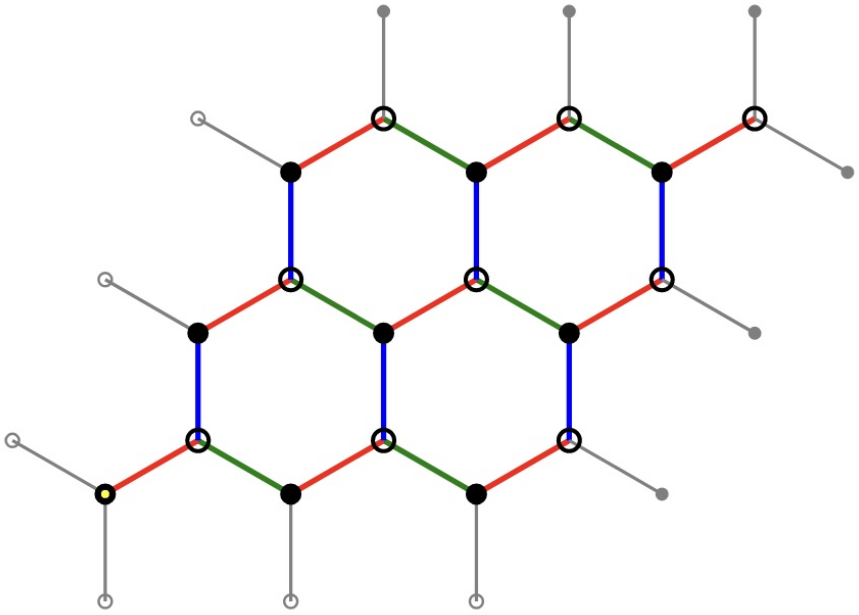
In this chapter, the calling method for some frequently used functions in `LatticeLab.jl` are documented by examples, which can be combined and adapted to perform various tasks.

3.1 Examples

3.1.1 Building honeycomb and kagome lattices

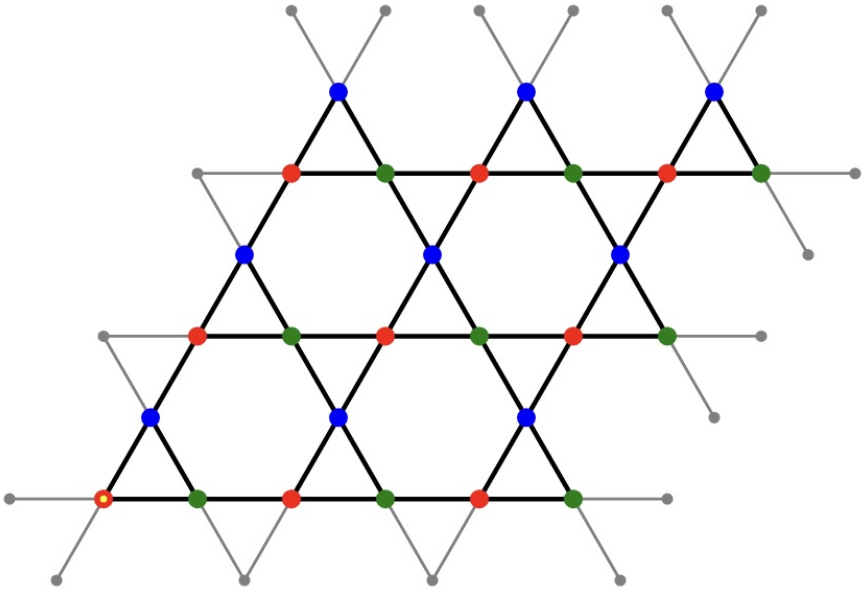
```
using LatticeLab
d1 = [ sqrt(3)/2, 1/2]
d2 = [-sqrt(3)/2, 1/2]
d3 = [         0,  -1] ;
BBOX(m,n) = ([-0.00021,-0.0001],  # origin
             [1 0 ; 0 1],        # supercell basis
             [m,n],               # supercell shifts
             [true,true])         # P.B.C. conditions
UC_Honeycomb = LatticeLab.UnitCell(
    2,                  # dimension
    2,                  # number of sublattices
    [sqrt(3) sqrt(3)/2
     0             3/2],# Bravais basis
    [0       sqrt(3)/2
     0             1/2],# sublattice coordinates
    [ :A,    :B  ],     # sublattice symbols
    [[:pz], [:pz]]      # orbits
)
```

```
UC_LN = LatticeLab.link_info_by_distance_direction(
    Dict( :tr => (1,[d1,-d1]),
          :tg => (1,[d2,-d2]),
          :tb => (1,[d3,-d3]), ), # symbol => (distance, (directions))
    UC_Honeycomb;
    bounding_box = BBOX(1,1),
    rounding_digits=12 ) ;

Graphene = build_lattice(UC_LN,BBOX(3,3)) ;

VSTYLE = Dict(
    :A=>("black",6.0,:dot),
    :B=>("black",6.0,:circle)
) ESTYLE = Dict(
    :tr=>("red",  3.0,:solid),
    :tg=>("green",3.0,:solid),
    :tb=>("blue", 3.0,:solid),
)
display("image/svg+xml",
    show_lattice_svg(Graphene, VSTYLE, ESTYLE; upscale=60))
```



```
using LatticeLab
```

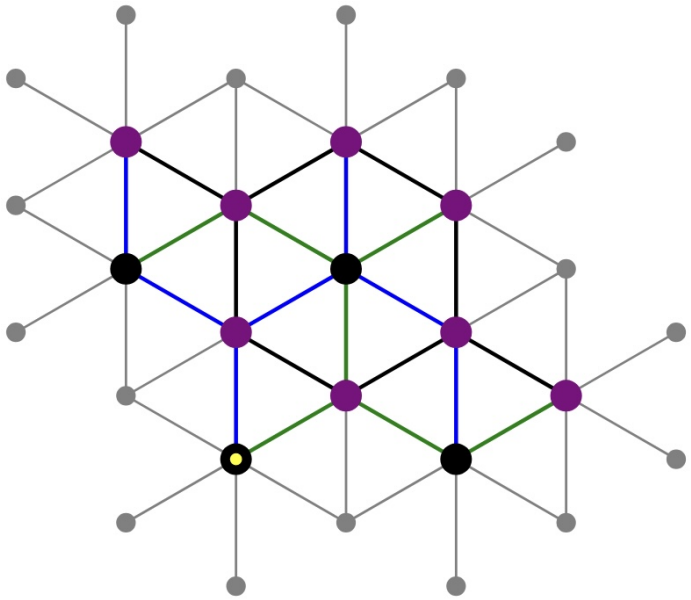
```
a1 = [1, 0]
a2 = [1/2, sqrt(3)/2] ;
BBOX(m,n) = ([-0.001,-0.001], # origin
             [1 0 ; 0 1],     # supercell basis
             [m,n],           # supercell shifts
             [true,true])     # P.B.C. conditions
aa = hcat( 2 .*a1, 2 .*a2 ) |> Coordinates
dd = hcat( [0,0], a1, a2) |> Coordinates
mm = [ :A,    :B,    :C     ] |> Masses
xx = [ [:pz,],[:pz,],[:pz,] ] .|> Orbits

UC_Kagome = UnitCell(2, 3, aa, dd, mm, xx)

LN_Kagome = LatticeLab.link_info_by_distance_direction(
    Dict(:t => (1,[]),), # symbol => (distance, (directions))
    UC_Kagome;
    bounding_box = BBOX(1,1),
    rounding_digits=6 )
Kagome = build_lattice(LN_Kagome,BBOX(3,3)) ;
VSTYLE = Dict(
    :A=>("red",  6.0,:dot),
    :B=>("green",6.0,:dot),
    :C=>("blue", 6.0,:dot)
) ESTYLE = Dict(
    :t=>("black", 3.0,:solid),
) display("image/svg+xml",
    show_lattice_svg(Kagome, VSTYLE, ESTYLE; upscale=60))
```



3.1.2 Load crystal structure from cif file via `cifio`

```
using LatticeLab
using cifio
function cellp2a(cellp; digits=12)
    @inline rd(x)  = round(x,digits=digits)
    @inline d2r(x) = Float64(x*pi/180)
    (a_Ang, b_Ang, c_Ang, alpha, beta, gamma) = cellp[1:6]
    alphar, betar, gammar = d2r.([alpha, beta, gamma])
    cell_px = a_Ang.*[1.0, 0.0, 0.0 ]
    cell_py = b_Ang.*[cos(gammar), sin(gammar), 0.0 ]
    cell_pz = c_Ang.*[
                cos(betar),
               (cos(alphar)-cos(betar)*cos(gammar))/sin(gammar),
                sqrt(1.0 - cos(alphar)^2 - cos(betar)^2 - cos(gammar)^2
                   + 2*cos(alphar)*cos(betar)*cos(gammar))/sin(gammar)
    ]
    return hcat(cell_px, cell_py, cell_pz) .|> rd
end
function CIF2UC(
    cif::CIF,
    orbs::Vector{Vector{Symbol}}
    )
    na = length(cif.frac)
    mm  = [Symbol("$c$i") for (i,c) in enumerate(first.(cif.frac))]
    aa  = LatticeLab.Coordinates(cellp2a(cif.cellp))
    ff  = hcat([f[2:4] for f in cif.frac]...)
    dd  = aa * ff |> LatticeLab.Coordinates  # Cartesian
    xx  = LatticeLab.Orbits.(orbs)
    UC = LatticeLab.UnitCell(3, na, aa, dd, mm, xx)
    return UC
end

Mo2N_CIF = split("""
Mo2N
_cell_length_a                          2.78373390
_cell_length_b                          2.78373390
_cell_length_c                         12.81473854
_cell_angle_alpha                      90.00000000
_cell_angle_beta                       90.00000000
_cell_angle_gamma                     120.00000000
_symmetry_Int_Tables_number            1
loop_
```

```
_space_group_symop_operation_xyz
   'x, y, z'
loop_
_atom_site_label
_atom_site_occupancy
_atom_site_fract_x
_atom_site_fract_y
_atom_site_fract_z
_atom_site_type_symbol
Mo1  1.0     0.66666667     0.33333333    0.60982427  Mo
Mo2  1.0     0.33333333     0.66666667    0.39017573  Mo
N1  1.0     0.00000000     0.00000000    0.50000000  N
""", "\n", keepempty=false) .|> string |> CIF ;

UC_Mo2N = CIF2UC(
    Mo2N_CIF,
    [[:d14,:d24,:d34,:d44,:d54,:s5],
     [:d14,:d24,:d34,:d44,:d54,:s5],
     [:px2,:py2,:pz2]]
)
BBOX(m,n) = ([-0.021,-0.01,-1e-6], # origin
             [1 0 0; 0 1 0; 0 0 1],    # supercell basis
             [m,n,1],                  # supercell shifts
             [true,true,false])        # P.B.C. conditions
LN_Mo2N = LatticeLab.link_info_sublattice_pairs_by_nth(
    Dict((1,3)=>(:f1,1),(2,3)=>(:f2,1),(1,2)=>(:g,1)),
    UC_Mo2N;
    bounding_box = BBOX(1,1),
    rounding_digits = 10 ) ;

Mo2N = build_lattice(LN_Mo2N,BBOX(2,2)) ;

VSTYLE = Dict(
    :Mo1=>("purple",12.0,:dot),
    :Mo2=>("purple",12.0,:dot),
    :N3 =>("black", 12.0,:dot)
) ESTYLE = Dict(
    :f1=>("green", 3.0,:solid),
    :f2=>("blue",  3.0,:solid),
    :g=> ("black", 3.0,:solid)
) display("image/svg+xml",
    show_lattice_svg(Mo2N, VSTYLE, ESTYLE; upscale=60))
```



3.1.3 Multiorbital hopping model

[Material proposed by Dr. Xinhai Tu]

```
using LatticeLab
include("../../include/tools.jl")
include("../../include/CIF2LN.jl")
include("../../include/wannier90_utils.jl")
Bi2Pd_hr_fn  = "Bi2Pd_soc_hr.dat"
Bi2Pd_cif_fn = "Bi2Pd.cif"
```

```
Porbits = [:pzu,:pzd,:pxu,:pxd,:pyu,:pyd]
Dorbits = [:dz2u,:dz2d,:dzxu,:dzxd,:dzyu,:dzyd,:dx2y2u,:dx2y2d,:dxyu,:dxyd] ;
P_u     = diagm(0=>vcat([[1,0] for i=1:11]...))
P_d     = diagm(0=>vcat([[0,1] for i=1:11]...))
P_Bi    = diagm(0=>vcat([[1 for i=1:12], [0 for i=1:10]]...))
P_Pd    = diagm(0=>vcat([[0 for i=1:12], [1 for i=1:10]]...))
P_Bi_l0 = diagm(0=>vcat([[1,1,0,0,0,0,1,1,0,0,0,0], [0 for i=13:22]]...))
P_Bi_l1 = P_Bi .- P_Bi_l0
P_Pd_l0 = diagm(0=>vcat([[0 for i=1:12], [1,1], [0 for i=15:22]]...))
P_Pd_l1 = diagm(0=>vcat([[0 for i=1:14], [1,1,1,1], [0 for i=19:22]]...))
P_Pd_l2 = P_Pd .- P_Pd_l0 .- P_Pd_l1
op(OP) = (k,v)->abs(v'*OP*v)
```

Constructing lattice

```
function Bi2Pd(fn)
    ORIG = [-8e-5, -1e-4, -7e-5]
    ID3  = [1 0 0; 0 1 0; 0 0 1]
    @inline select_non_empty(D) = Dict(k=>v for (k,v) in D if length(v)>0)
    xx = [Porbits, Porbits, Dorbits] .|> LatticeLab.Orbits
    UC = CIF2UC(CIF(fn), xx)
    @time begin
        ALL_LN     = all_links_between_sublattices(
                        UC;
                        maxlen=120, max_distance=200, large_enough_margin=20,
                        rounding_digits=6 ) |> select_non_empty
        ALL_LN_nnn = pick_nearest_by_ij(
                        ALL_LN,
                        merge( Dict((i,i)=>15 for i=1:3),
                               Dict((i,j)=>15 for i=1:3 for j=1:3 if i!=j) );
                        digits=6
        )
        # TODO needs more dedicated constructor, with x- and y-directions
        SP = all_links_dict_to_sp_no_directions(ALL_LN_nnn) ;
    end
    LN = LinkInfo(copy(UC), SP) ;
    return build_lattice(LN, BoundingBox((ORIG,ID3,[3,3,1],[true,true,false]))) ;
end

Bi2Pd_LATT = Bi2Pd(Bi2Pd_cif_fn) ;
```

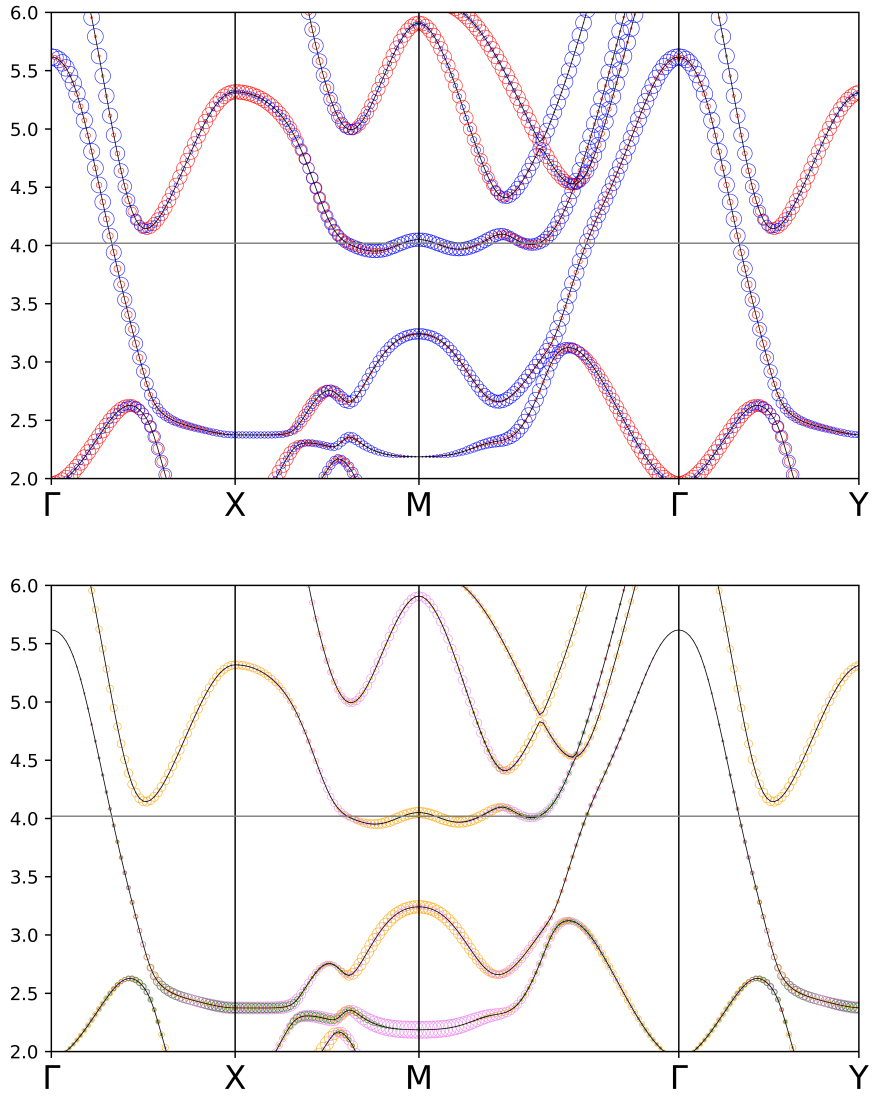
Computing bands from Wannier Hamiltonian

```
 const KPATH_REL = [
    "G"  =>  [0,    0,   0 ],
    "X"  =>  [1//2, 0,   0 ],
    "M"  =>  [1//2, 1//2,0 ],
    "G"  =>  [0,    0,   0 ],
    "Y"  =>  [0, 1//2,   0 ], ]
@inline kvec_rel2abs(rel,  b) = [k=>(b*vrel) for (k,vrel) in rel]
@inline HSP(b)  = kvec_rel2abs(KPATH_REL,b)

Bi2Pd_HQ_Wann = make_HWannQ(
        readlines(Bi2Pd_hr_fn),
        Bi2Pd_LATT;
        thr=0.001
) ;
BS_Wann = LatticeLab.band_structure_with_markers(
                HSP(reciprocal_basis(Bi2Pd_LATT)),
                Bi2Pd_HQ_Wann,
                [op(P_Bi_l0),op(P_Bi_l1),op(P_Pd_l0),op(P_Pd_l1),op(P_Pd_l2)],
                dk=1e-2
) |> BandStructures.LatticeLab_bands_BandStructure ;
```

Plot bands

```
plot_bands( "Bi2Pd_Wannier_Pd_lz=0(green),1(orange),2(violet).pdf",
            [BS_Wann,];
            SIZE  = [0.0,   0.0,    90.0,    90.0,     90.0     ],
            COLOR = ["red", "blue", "green", "orange", "violet" ],
            settings = Dict(
                  :line_colors=>["black"],
                  :lw=>[0.4],
                  :ref_levels=>[(4.0201,"gray"),(0.0,"gray")],
                  :range=>(2,6),
                  :K_sep=>-1,
                  :markerstrokealpha=>0.7,
                  :markerstrokewidth=>0.3,
                  :aspect_ratio=>0.6,
                  :figure_size=>(8,16),)
)
```

```
plot_bands( "Bi2Pd_Wannier_Bi_lz=0(red),1(blue).pdf",
            [BS_Wann,];
            SIZE  = [90.0,  90.0,   0.0,     0.0,       0.0     ],
            COLOR = ["red", "blue", "green", "orange", "violet" ],
            settings = Dict(
                  :line_colors=>["black"],
                  :lw=>[0.4],
                  :ref_levels=>[(4.0201,"gray"),(0.0,"gray")],
                  :range=>(2,6),
                  :K_sep=>-1,
                  :markerstrokealpha=>0.7,
                  :markerstrokewidth=>0.3,
                  :aspect_ratio=>0.6,
                  :figure_size=>(8,16),)
)
```


6.0
5.5
5.0
4.5
4.0
3.5
3.0
2.5
2.0
Γ
X
M
Γ
Y
6.0
5.5
5.0
4.5
4.0
3.5
3.0
2.5
2.0
Γ
X
M
Γ
Y

Extracting hopping parameters

```
function SP2T_raw(HQWannier; thr=0.02)
    @inline trim_empty(X) = [k=>v for (k,v) in X if length(v)>0]
    @inline trim_imag(X)  = (abs(imag(X))<1e-7 ? real(X) : X)
    LATT = HQWannier.LATT
    HQS = kspace_hopping_hamiltonian_symbolic(
                Dict(k=>(x->(:ALL,(x,))) for k in keys(LATT.LN.SPNB)),
                Dict(k=>[k,]             for k in keys(LATT.LN.SPNB)),
                Dict(m=>(x->x) for m in LATT.UC.m),
                Dict(m=>[m,]   for m in LATT.UC.m),
                LATT;
                da = 1.0
    ) ;
    ret = Dict(k=>[] for (k,dic) in HQS.MATS)
    ret_k = []
    MAT = HQWannier.MAT
    for (k,dic) in HQS.MATS
        ret_k = []
        for (q,m) in dic
            X,Y,V = findnz(m)
            push!(ret_k, q=>[ (x,y)=>trim_imag(MAT[q][x,y])
                                for (x,y) in zip(X,Y)
                                    if abs(MAT[q][x,y])>thr ])
        end
        ret[k] = trim_empty(ret_k)
    end
    return Dict(trim_empty(ret))
end

@inline show_hop(v) =
            println(
                join([ "   "*string(v[1])*" => ",
                       sort(["\t"*string(k=>p) for (k,p) in v[2]])...],
                     "\n") )
@inline show_t(k,v) = (println(string(k)*" => "); show_hop.(v))
R = SP2T_raw(Bi2Pd_HQ_Wann; thr=0.5) ;
[show_t(k,R[k]) for k in sort(collect(keys(R)))] ;
```

```
## ====================
##       RESULTS
## ====================
Bi1 =>
   [0, 0, 0] =>
    (1, 1) => 2.665081
    (2, 2) => 2.665081
    (3, 3) => 3.341092
    (4, 4) => 3.341092
    (5, 5) => 3.341092
    (6, 6) => 3.341092
Bi1_Bi1_1 =>
   [0, -1, 0] =>
    (1, 1) => -0.511186
    (2, 2) => -0.511186
    (5, 5) => 1.817241
    (6, 6) => 1.817241
   [1, 0, 0] =>
    (1, 1) => -0.511186
    (2, 2) => -0.511186
    (3, 3) => 1.817241
    (4, 4) => 1.817241
   [0, 1, 0] =>
    (1, 1) => -0.511186
    (2, 2) => -0.511186
    (5, 5) => 1.817241
    (6, 6) => 1.817241
   [-1, 0, 0] =>
    (1, 1) => -0.511186
    (2, 2) => -0.511186
    (3, 3) => 1.817241
    (4, 4) => 1.817241
Bi1_Bi2_1 =>
   [0, 0, 0] =>
    (1, 7) => 1.363654
    (2, 8) => 1.363654
    (7, 1) => 1.363654
    (8, 2) => 1.363654
Bi2 =>
   [0, 0, 0] =>
    (10, 10) => 3.341092
    (11, 11) => 3.341092
    (12, 12) => 3.341092
    (7, 7) => 2.665081
```

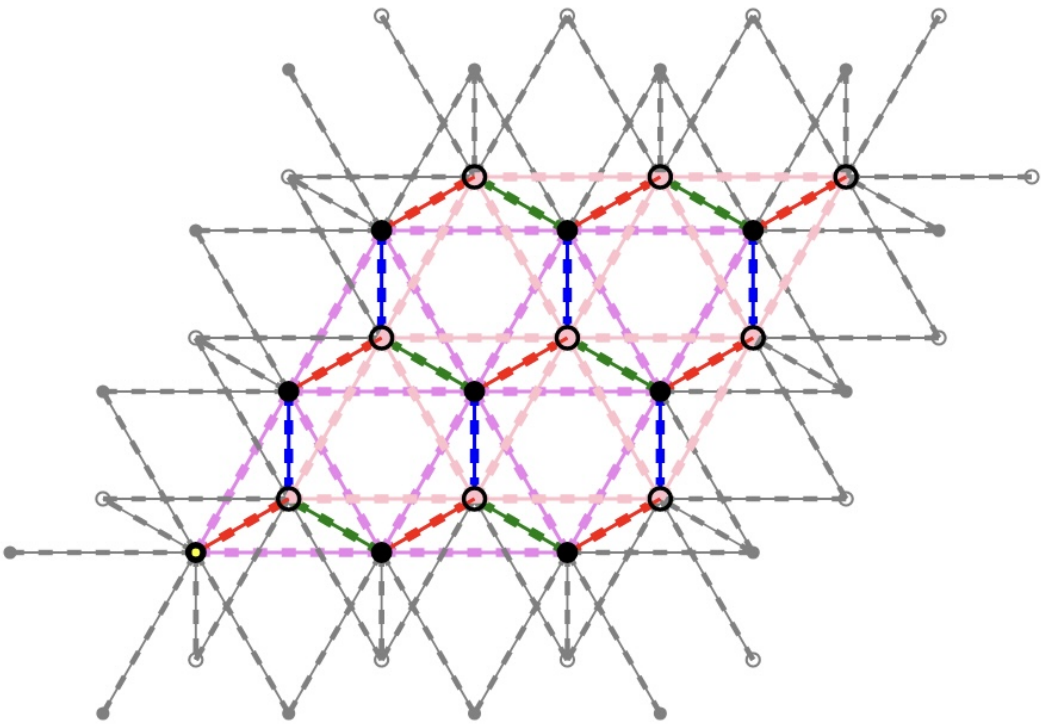
```
    (8, 8) => 2.665081
    (9, 9) => 3.341092
Bi2_Bi2_1 =>
   [0, -1, 0] =>
    (11, 11) => 1.817241
    (12, 12) => 1.817241
    (7, 7) => -0.511186
    (8, 8) => -0.511186
   [1, 0, 0] =>
    (10, 10) => 1.817241
    (7, 7) => -0.511186
    (8, 8) => -0.511186
    (9, 9) => 1.817241
   [0, 1, 0] =>
    (11, 11) => 1.817241
    (12, 12) => 1.817241
    (7, 7) => -0.511186
    (8, 8) => -0.511186
   [-1, 0, 0] =>
    (10, 10) => 1.817241
    (7, 7) => -0.511186
    (8, 8) => -0.511186
    (9, 9) => 1.817241
Pd3 =>
   [0, 0, 0] =>
    (13, 13) => 1.035689
    (14, 14) => 1.035689
    (15, 15) => 1.473082
    (16, 16) => 1.473082
    (17, 17) => 1.473082
    (18, 18) => 1.473082
    (19, 19) => 1.112115
    (20, 20) => 1.112115
    (21, 21) => 1.496155
    (22, 22) => 1.496155
```

3.1.4 Haldane model

```
using LatticeLab
using BandStructures
using LinearAlgebra
using PyPlot
a1 = [1,   0        ]
a2 = [1/2, sqrt(3)/2]
alpha = sqrt(3)
Atr = [a1, a2, a2.-a1, -a1, -a2, a1.-a2]
d1 = (alpha/3).*(Atr[1].+Atr[2])
d2 = (alpha/3).*(Atr[3].+Atr[4])
d3 = (alpha/3).*(Atr[5].+Atr[6])
delta = [d1,d2,d3]
BBOX(m,n) = ([-0.00021,-0.0001], # origin
             [1 0 ; 0 1],                # supercell basis
             [m,n],                      # supercell shifts
             [true,true])                # P.B.C. conditions
aa = hcat( alpha.*a1, alpha.*a2) |> LatticeLab.Coordinates
dd = hcat([0, 0], d1   ) |> LatticeLab.Coordinates
mm = [ :A,       :B      ] |> LatticeLab.Masses
xx = [[:pu,:pd],[:pu,:pd]] .|> LatticeLab.Orbits
UC_Honeycomb = LatticeLab.UnitCell( 2, 2, aa, dd, mm, xx )
nb1a(k) = Spring( (1,2) => [ delta[k],] )
nb1b(k) = Spring( (2,1) => [-delta[k],] )
nb2_cwise_s1  = Spring((1,1) => [alpha.*Atr[2], alpha.*Atr[4], alpha.*Atr[6]])
nb2_cwise_s2  = Spring((2,2) => [alpha.*Atr[2], alpha.*Atr[4], alpha.*Atr[6]])
nb2_ccwise_s1 = Spring((1,1) => [alpha.*Atr[1], alpha.*Atr[3], alpha.*Atr[5]])
nb2_ccwise_s2 = Spring((2,2) => [alpha.*Atr[1], alpha.*Atr[3], alpha.*Atr[5]])
NB_Haldane = [ nb1a(1), nb1a(2), nb1a(3), nb1b(1), nb1b(2), nb1b(3),
               nb2_cwise_s1, nb2_cwise_s2, nb2_ccwise_s1, nb2_ccwise_s2 ]
SP_Haldane = [ :t1Ar,   :t1Ag,   :t1Ab,   :t1Br,   :t1Bg,   :t1Bb,
               :t2ps1,  :t2ps2,  :t2ms1,  :t2ms2  ]
LN_Haldane = LatticeLab.LinkInfo(
    UC_Honeycomb,
    LatticeLab.zict(SP_Haldane, NB_Haldane) )
@assert check_compat(LN_Haldane)

Honeycomb_Haldane = build_lattice(LN_Haldane, BBOX(3,3))
```

Visualization of lattice

```
VSTYLE = Dict(
    :A=>("black",6.0,:dot),
    :B=>("black",6.0,:circle)
)
ESTYLE = Dict(
    :t1Ar => ("red",    2.0, :solid ),
    :t1Ag => ("green",  2.0, :solid ),
    :t1Ab => ("blue",   2.0, :solid ),
    :t1Br => ("red",    5.0, :dashed),
    :t1Bg => ("green",  5.0, :dashed),
    :t1Bb => ("blue",   5.0, :dashed),
    :t2ps1=> ("violet", 2.0, :solid ),
    :t2ps2=> ("pink",   2.0, :solid ),
    :t2ms1=> ("violet", 5.0, :dashed),
    :t2ms2=> ("pink",   5.0, :dashed),
)
display("image/svg+xml",
    show_lattice_svg(Honeycomb_Haldane, VSTYLE, ESTYLE; upscale=60))
```



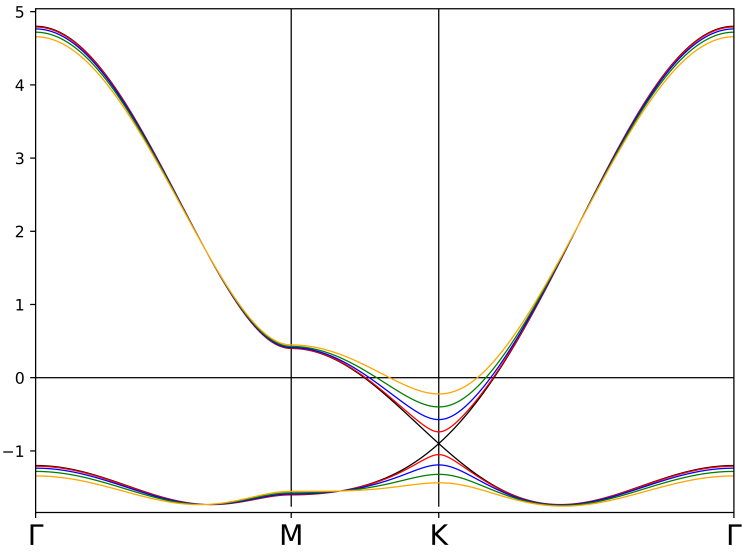
Construct k-space Hamiltonian

```
# the complicated way of assigning hopping parameters
func1() = (:ALL,(1,))
func(x) = (:ALL,(x,))
hHaldaneF  = Dict( :t1Ar=>func1, :t1Ag=>func1, :t1Ab=>func1,
                   :t1Br=>func1, :t1Bg=>func1, :t1Bb=>func1,
                   # the Haldanish lines
                   :t2ps1=>(x,y)->func(x*cis(y)),
                   :t2ps2=>(x,y)->func(x*cis(-y)),
                   :t2ms1=>(x,y)->func(x*cis(-y)),
                   :t2ms2=>(x,y)->func(x*cis(y)), )
hHaldaneP  = Dict( :t1Ar=>(), :t1Ag=>(), :t1Ab=>(),
                   :t1Br=>(), :t1Bg=>(), :t1Bb=>(),
                   :t2ps1=>(:tnn,:phi),:t2ms1=>(:tnn,:phi),
                   :t2ps2=>(:tnn,:phi),:t2ms2=>(:tnn,:phi), )
@inline hHaldane(tnn,phi) = LatticeLab.dispatch_params(
    Dict(:tnn=>tnn,:phi=>phi),
    hHaldaneF, hHaldaneP, 0.0 )
```

```
b = reciprocal_basis(Honeycomb_Haldane) ;
kpath = [ "G" => (0   ).*b[:,1],
          "M" => (1//2).*b[:,1] .+ (0//2).*b[:,2],
          "K" => (2//3).*b[:,1] .+ (1//3).*b[:,2],
          "G" => (0   ).*b[:,1], ] ;
HQs = [ kspace_hopping_hamiltonian(
            HoppingParameter(Honeycomb_Haldane.UC, hHaldane(0.3,phi)),
            zero_onsite_potential(Honeycomb_Haldane),
            Honeycomb_Haldane )
        for phi = 0:0.1:0.4 ] ;
BSs = [ LatticeLab.band_structure(
            kpath, HQ, dk=1e-3
            ) |> BandStructures.LatticeLab_bands_BandStructure
        for HQ in HQs ] ;
```

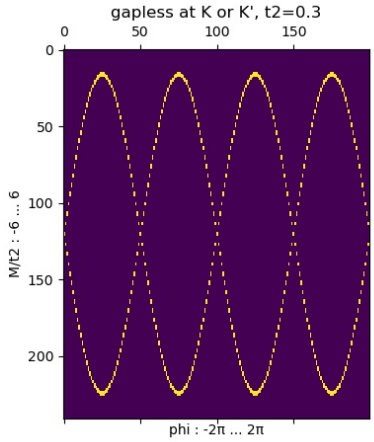
Plot band structure at $t_1 = 1$, $t_2 = 0.3$, $M = 0$, $\phi = 0.0, 0.1, 0.2, 0.3, 0.4$

```
plot_bands("Haldane_tnn=0.3.png",
           BSs,
           dpi = 1600,
           settings = Dict(
                :line_colors=>["black", "red", "blue", "green", "orange"],
                :lw=>0.8,
                :K_sep=>-1,
                :aspect_ratio=>0.6,
                :figure_size=>(16,20),)
)
```



Gapless states on the $M/t_2 - \phi$ plane

```
t2 = 0.3
K1 = (2//3).*b[:,1] .+ (1//3).*b[:,2]
K2 = (1//3).*b[:,1] .+ (2//3).*b[:,2]
function is_gapless(H,eps=1e-3)
    en1 = eigen(Matrix(HoppingHamiltonian(K1,H))).values
    en2 = eigen(Matrix(HoppingHamiltonian(K2,H))).values
    (  abs(maximum(en1)-minimum(en1))<eps
    || abs(maximum(en2)-minimum(en2))<eps  )
end
Ham(t2,M,phi) =
    kspace_hopping_hamiltonian(
        HoppingParameter(Honeycomb_Haldane.UC, hHaldane(t2,phi)),
        Dict(:A=>M,:B=>-M),
        Honeycomb_Haldane )
# compute phase diagram
@time Phase = [ is_gapless(Ham(t2,t2*r,phi),4e-2)
                for r = -6:0.05:6, phi = -2pi:(pi/50):2pi ] ;
## 8.839576 seconds (80.40 M allocations: 6.916 GiB,
## ... 8.97% gc time, 0.81% compilation time)

matshow(Phase)
xlabel("phi : -2pi ... 2pi")
ylabel("M/t2 : -6 ... 6")
title("gapless at K or K', t2=0.3")
savefig("Haldane_gapless.png")
```

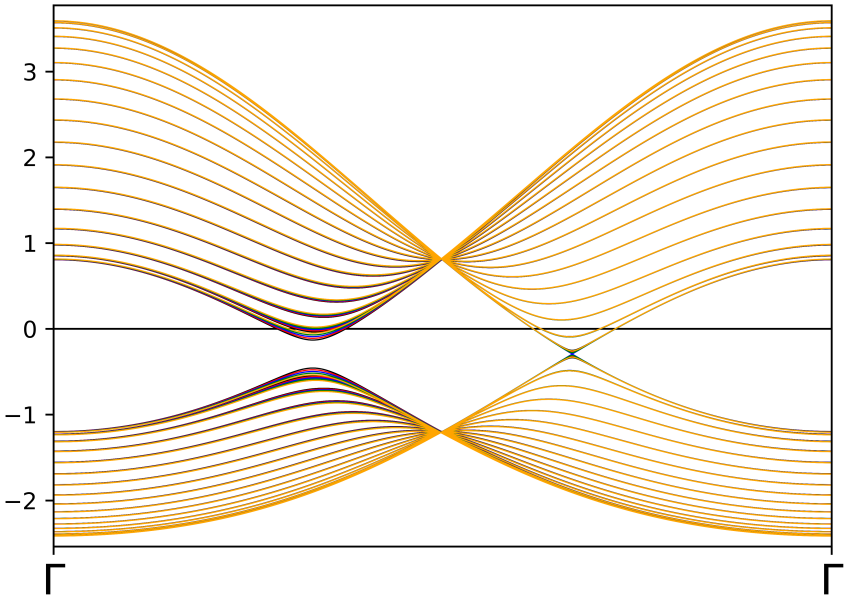


Stacking stripes: stripe along y-axis, stacking along x-axis

```
BBOX_ENL(m,n) = BoundingBox(
    ([-0.00021,-0.0001],  # origin
     [[1,0] [-1,2]],      # supercell basis
     [m,n],               # supercell shifts
     [true,true])         # P.B.C. conditions
)
BBOX_stack_stripe_along_x(Len) = BoundingBox(
    ([-0.00021,-0.0001],  # origin
     [[1,0] [0,1]],       # supercell basis
     [Len,1],             # supercell shifts
     [true,false])        # P.B.C. conditions
)
LATT00_y_Honeycomb_ENL = build_lattice(LN_Haldane, BBOX_ENL(1,16)) ;
LATTy_open = enlarge(LATT00_y_Honeycomb_ENL, BBOX_stack_stripe_along_x(3)) ;
b_y = reciprocal_basis(LATTy_open) ;
kpath_y = [ "G" => (0).*b_y[:,1],
            "G" => (1).*b_y[:,1], ] ;
HQy_open = [kspace_hopping_hamiltonian(
                HoppingParameter(LATTy_open.UC, hHaldane(0.1,0.2)),
                Dict(:A=>M,:B=>-M),
                LATTy_open)
            for M = 0.06:0.02:0.14 ] ;
BSy_open = [LatticeLab.band_structure(
                kpath_y, HQ, dk=2e-4
                ) |> BandStructures.LatticeLab_bands_BandStructure
            for HQ in HQy_open ] ;
```

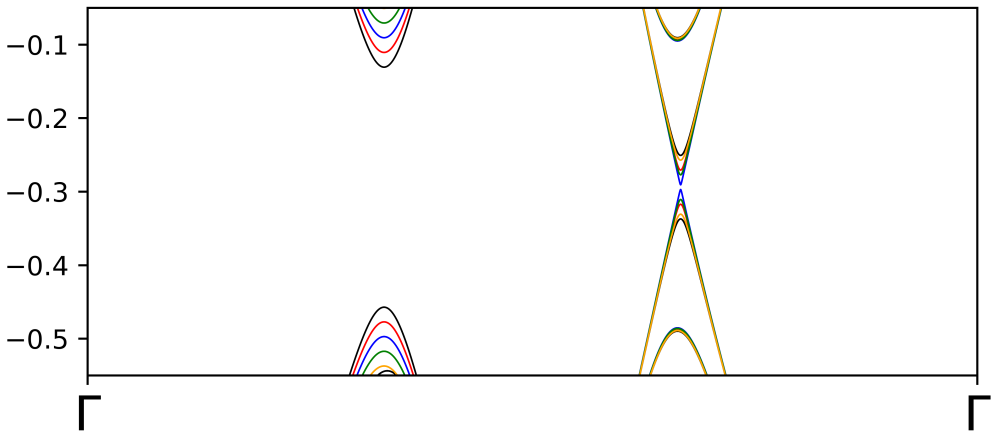
Plot band structure at $t_1 = 1$, $t_2 = 0.1$, $\phi = 0.2$, $M = 0.06, 0.08, 0.10, 0.12, 0.14$

```
plot_bands("Haldane_t2=0.1_phi=0.2_M=(0.06,0.02,0.14).png",
           BSy_open,
           dpi    = 1600,
           settings = Dict(
                :line_colors=>["black", "red", "blue", "green", "orange"],
                :K_sep=>-1,
                :lw=>0.8,
                :aspect_ratio=>0.4,
                :figure_size=>(16,24),)
)
```



Plot band structure at $t_1 = 1$, $t_2 = 0.1$, $\phi = 0.2$, $M = 0.06, 0.08, 0.10, 0.12, 0.14$, zoom in

```
plot_bands("Haldane_t2=0.1_phi=0.2_M=(0.06,0.02,0.14)__ZOOM.png",
           BSy_open,
           dpi    = 1600,
           settings = Dict(
                :line_colors=>["black", "red", "blue", "green", "orange"],
                :K_sep => -1,
                :range => (-0.55,-0.05),
                :lw    => 0.8,
                :aspect_ratio => 3,
                :figure_size  => (12,16),)
)
```

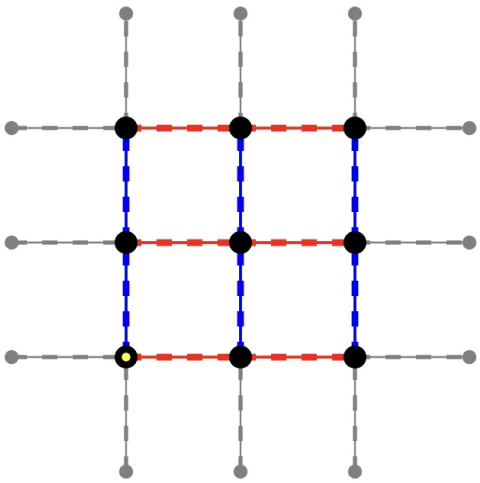


3.1.5 Square lattice BHZ model

```
using LatticeLab
using BandStructures
sigma0 = [1   0; 0  1]
sigmax = [0   1; 1  0]
sigmay = [0 -im; im 0]
sigmaz = [1   0; 0 -1]
I2 = sigma0 ;
BBOX(m,n) = ([-0.00021,-0.0001], # origin
             I2,                 # supercell basis
             [m,n],              # supercell shifts
             [true,true])        # P.B.C. conditions
UC_Sqlatt = LatticeLab.UnitCell(
    2,              # dimension
    1,              # number of sublattices
    I2,             # Bravais basis
    zeros(2,1),     # sublattice coordinates
    [:A, ],         # sublattice symbols
    [[:pzu, :pzd],] # orbits
)
@assert check_compat(UC_Sqlatt)
LN_Sqlatt = LatticeLab.link_info_by_distance_direction(
    Dict(:txp => (1,[[ 1,0]]),:typ => (1,[[0, 1]]),
         :txm => (1,[[-1,0]]),:tym => (1,[[0,-1]]), ),
    UC_Sqlatt;
    bounding_box = BBOX(1,1),
    rounding_digits=12
)
@assert check_compat(LN_Sqlatt)

Sqlatt = build_lattice(LN_Sqlatt,BBOX(3,3)) ;
```

Visualization of lattice

```
VSTYLE = Dict(:A=>("black",6.0,:dot))
ESTYLE = Dict(
    :txp=>("red",  1.5,:solid),
    :txm=>("red",  3.5,:dashed),
    :typ=>("blue", 1.5,:solid),
    :tym=>("blue", 3.5,:dashed),
)
display("image/svg+xml",
    show_lattice_svg(Sqlatt, VSTYLE, ESTYLE; upscale=60))
```



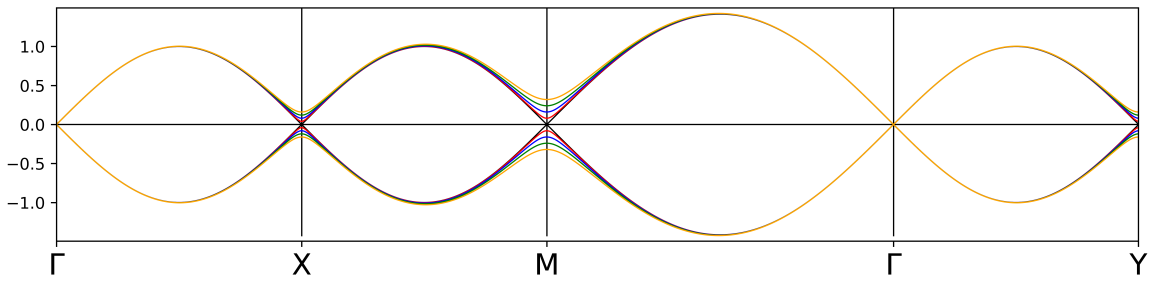
Construct k-space Hamiltonian

```
# the complicated way of assigning hopping parameters
func1() = (:MAT,(sigma0,))
func(x) = (:MAT,(x,     ))
hBHZF  = Dict(:txp => (a,b)->func((-im*a/2).*sigmax.+(b).*sigmaz),
              :typ => (a,b)->func((-im*a/2).*sigmay.+(b).*sigmaz),
              :txm => (a,b)->func(((-im*a/2).*sigmax.+(b).*sigmaz)'),
              :tym => (a,b)->func(((-im*a/2).*sigmay.+(b).*sigmaz)'),)
hBHZP  = Dict(:txp => (:tA, :tB), :typ => (:tA, :tB),
              :txm => (:tA, :tB), :tym => (:tA, :tB),)
@inline hBHZ(A,B) = LatticeLab.dispatch_params(
    Dict(:tA=>A,:tB=>B),
    hBHZF, hBHZP, 0.0 )
@inline vBHZ(Z,B) = Dict(:A=>((Z-4B).*sigmaz))
```

```
b = reciprocal_basis(Sqlatt) ;
kpath = [ "G" => (0   ).*b[:,1],
          "X" => (1//2).*b[:,1],
          "M" => (1//2).*b[:,1] .+ (1//2).*b[:,2],
          "G" => (0   ).*b[:,1],
          "Y" => (0   ).*b[:,1] .+ (1//2).*b[:,2],] ;
```

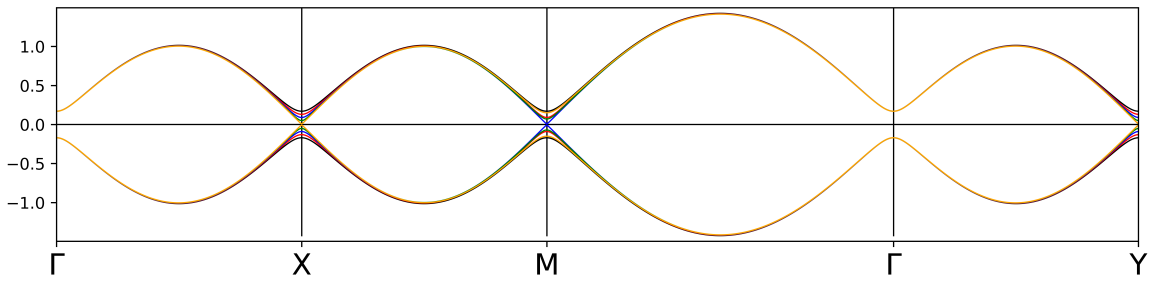
Plot band structure at $A = 1.0$, $B = 0.0, 0.01, 0.02, 0.03, 0.04$, $Z = 0.0$

```
Z = 0.0
HQs = [ kspace_hopping_hamiltonian(
            HoppingParameter(Sqlatt.UC, hBHZ(1.0,B)),
            vBHZ(Z,B),
            Sqlatt )
        for B = 0:0.01:0.04 ] ;
BSs = [ LatticeLab.band_structure(
            kpath, HQ, dk=5e-4
            ) |> BandStructures.LatticeLab_bands_BandStructure
        for HQ in HQs ] ;
plot_bands("SquareLattice_BHZ_Z=0.0.png",
           BSs,
           dpi  = 800,
           settings = Dict(
                :line_colors=>["black", "red", "blue", "green", "orange"],
                :lw=>0.8,
                :K_sep=>-1,
                :aspect_ratio=>1,
                :figure_size=>(12,6),)
)
```



Plot band structure at $A = 1.0$, $B = 0.0, 0.01, 0.02, 0.03, 0.04$, $Z = 0.17$

```
Z = 0.17
HQs = [ kspace_hopping_hamiltonian(
            HoppingParameter(Sqlatt.UC, hBHZ(1.0,B)),
            vBHZ(Z,B),
            Sqlatt )
        for B = 0:0.01:0.04 ] ;
BSs = [ LatticeLab.band_structure(
            kpath, HQ, dk=5e-4
            ) |> BandStructures.LatticeLab_bands_BandStructure
        for HQ in HQs ] ;
plot_bands("SquareLattice_BHZ_Z=0.17.png",
           BSs,
           dpi  = 800,
           settings = Dict(
                :line_colors=>["black", "red", "blue", "green", "orange"],
                :lw=>0.8,
                :K_sep=>-1,
                :aspect_ratio=>1,
                :figure_size=>(12,6),)
)
```



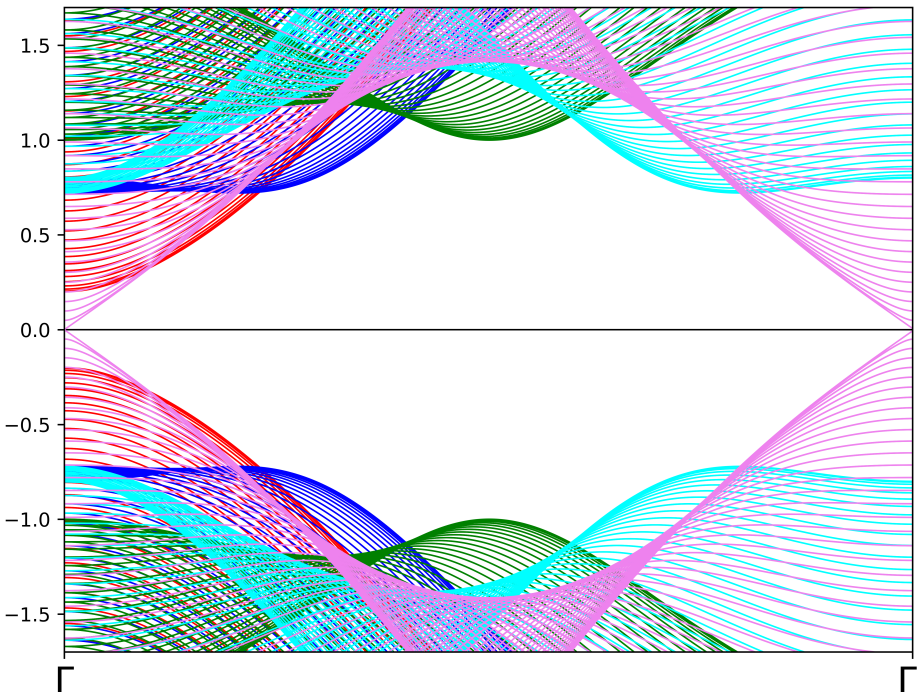
Stacking stripes: stripe along y-axis, stacking along x-axis

```
BBOX_open_x(m,n) = BoundingBox((
             [-0.00021,-0.0001], # origin
             I2,                 # supercell basis
             [m,n],              # supercell shifts
             [false,true]))      # P.B.C. conditions
BBOX_open_y(m,n) = BoundingBox((
             [-0.00021,-0.0001], # origin
             I2,                 # supercell basis
             [m,n],              # supercell shifts
             [true,false]))      # P.B.C. conditions
LATT00_y_SqLatt = build_lattice(LN_Sqlatt, BBOX(1,128)) ;
LATTy_open = enlarge(LATT00_y_SqLatt, BBOX_open_y(3,1)) ;
b_y = reciprocal_basis(LATTy_open)
kpath_y = [ "G" => (0.0).*b_y[:,1],
            "G " => (0.5).*b_y[:,1], ] ;
```

Plot band structure at $A = B = 1.0$, $Z = 0.2, 0.8, 2.0, 3.2, 4.0$ Reproducing Figure 10 in [Imura et al. Nanoscale Research Letters 2011, 6:358]

```
B = 1.0
HQy_open = [kspace_hopping_hamiltonian(
                HoppingParameter(LATTy_open.UC, hBHZ(1.0, B)),
                vBHZ(r*B, B),  LATTy_open)
            for r = [0.2,0.8,2.0,3.2,4.0] ] ;
BSy_open = [LatticeLab.band_structure(
                kpath_y, HQ, dk=1e-3
                ) |> BandStructures.LatticeLab_bands_BandStructure
            for HQ in HQy_open ] ;
# plot
# ref [Imura et al. Nanoscale Research Letters 2011, 6:358]
plot_bands("SquareLattice_BHZ_B=1.0_ZBratio=(0.2,0.8,2.0,3.2,4.0).png",
           BSy_open,
           dpi   = 800,
           settings = Dict(
                :line_colors=>["red", "blue", "green", "cyan", "violet"],
                :range => (-1.7,1.7),
                :K_sep=>-1, :lw=>0.8,
```

```
                    :aspect_ratio=>0.7,
                    :figure_size=>(10,6), ) )
```



3.1.6 Interface to Phonopy

[Material found in Phonopy database]

```
using LatticeLab
using BandStructures
using PyCall
phonopy = pyimport("phonopy")
sk = pyimport("seekpath")

function lattice_dynmat_from_phonopy(
    ph::PyCall.PyObject,
    POSCAR::String,
    FORSE_SETS::String,
    CONF::String;
    max_distance=30.0,
    maxlen=20,
    margin=6,
    bounding_box=([-0.001,-0.0004,-0.0006],
                  [1 0 0; 0 1 0; 0 0 1],
                  [1,1,1],
                  [true,true,true]),
    rounding_digits=5,
    PHONOPY_EPS=1e-6,
    PHONOPY_PREC_CTRL=8
    )
    #@info "LatticeLab.PhonopyCopy_from_vasp"
    @time PH = LatticeLab.PhonopyCopy_from_vasp(ph, POSCAR, FORSE_SETS, CONF)
    #@info "LatticeLab.all_links_between_sublattices"
    @time ALL_LN = LatticeLab.all_links_between_sublattices(
                                    PH.UC;
                                    maxlen=maxlen,
                                    max_distance=max_distance,
                                    large_enough_margin=margin,
                                    bounding_box=bounding_box,
                                    rounding_digits=rounding_digits )
    #@info "LatticeLab.extract_force_constants"
    @time SP, NB = LatticeLab.extract_force_constants(
                                    PH, ALL_LN;
                                    SVEC_PREC_CTRL=PHONOPY_PREC_CTRL,
                                    FC_PREC_CTRL=PHONOPY_PREC_CTRL,
                                    SVEC_NORM_EPS=PHONOPY_EPS,
                                    FC_EPS=PHONOPY_EPS )
```

```
    #@info "LatticeLab.build_lattice"
    @time LATT = LatticeLab.build_lattice(NB, bounding_box)
    #@info "LatticeLab.kspace_dynamical_matrix"
    @time DM = LatticeLab.kspace_dynamical_matrix(SP, PH.MASS, LATT)
    return PH, DM
end

function process_phonopy_database_element(
    ph::PyCall.PyObject,
    MP::String,
    FD::String;
    rounding_digits=4,
    margin=6,
    max_distance=20.0,
    maxlen=20,
    )
    POSCAR = "$FD/$MP/POSCAR-unitcell"
    FORcE = "$FD/$MP/" * (isfile("$FD/$MP/FORCE_SETS") ? "FORCE_SETS" : "FORCE_CONSTANTS
    @assert isfile(FORcE)
    CONF = "$FD/$MP/phonopy.conf"
    PH, DM, err = (nothing, nothing, nothing)
    try
        PH, DM = lattice_dynmat_from_phonopy(
                        ph, POSCAR, FORcE, CONF,
                        rounding_digits=rounding_digits,
                        max_distance=max_distance,
                        maxlen=maxlen,
                        margin=margin
        )
    catch _e_
        err = string(typeof(_e_))
    end
    return DM, err
end


function plot_band_mp(
    OPS,
    figure_file_name,
    DM,
    HighSymmRel;
    dk=0.01,
    COLOR=["red","blue","green","orange"],
    SIZE = 40.0,
    settings=Dict(  :colors=>["black",],
```

```
                    :lw=>0.4,
                    :range=>nothing,
                    :fontsize=>12,
                    :markerstrokealpha=>0.7,
                    :markerstrokewidth=>0.2,
                    :markersizetrimratio=>0.01,
                    :aspect_ratio=>3,
                    :figure_size => (10,4),    )
    )
    b    = LatticeLab.reciprocal_basis(DM.LATT.UC)
    HighSymmAbs = [[name=>b*qrel for (name,qrel) in HS] for HS in HighSymmRel]
    BS   = [ LatticeLab.band_structure_with_eigenvectors(
                HSA, DM;
                dk=dk, test_eigen=false, eps=1e-8)
            for HSA in HighSymmAbs]
    BS1 = [ compute_band_markers(
                b, OPS
            ) |> BandStructures.LatticeLab_bands_BandStructure
            for b in BS ];
    plot_bands(
            figure_file_name, BS1;
            dpi=800, COLOR=COLOR, SIZE=SIZE, settings=settings )
    return BS1
end
```

```
@inline l2b(UC) = BandStructures.UnitCell(3,UC.nsubl,UC.a,UC.d,UC.m,UC.x)

OPS_EL(UC,EL) = [ (k,v)->abs(v'*AtomProjector(EL,UC)*v) ]

# load and convert
DM, err = process_phonopy_database_element( phonopy,
                                            "mp-830-20180417", # CHEM = "GaN"
                                            "./",
                                            rounding_digits=5,
                                            max_distance=50.0,
                                            maxlen=40 )

# plot
HighSymmRel = kpath(call_get_path(sk, l2b(DM.LATT.UC); with_time_reversal=false))
ALONG  = [("Ga",LOPS_EL(DM.LATT.UC,:Ga),"red"),
          ("N", LOPS_EL(DM.LATT.UC,:N),"grey") ]
ATTRANS = [("Ga",TOPS_EL(DM.LATT.UC,:Ga),"red"),
           ("N", TOPS_EL(DM.LATT.UC,:N),"grey") ]
for (A,lt) in [(ALONG,"L"), (ATTRANS,"T")]
    for (atm, proj, col) in A
        plot_band_mp(
            proj,
            "GaN_$(lt)_$(atm).pdf",
            DM,
            HighSymmRel;
            dk=0.01,
            COLOR=[col,],
            SIZE = 50.0,
            settings=Dict(  :colors=>["black",],
                            :lw=>0.3,
                            :fontsize=>6,
                            :K_sep => 0.3,
                            :markerstrokealpha=>0.6,
                            :markerstrokewidth=>0.3,
                            :markersizetrimratio=>0.01,
                            :aspect_ratio=>6,
                            :figure_size => (8,4) ) )
    end
end
```

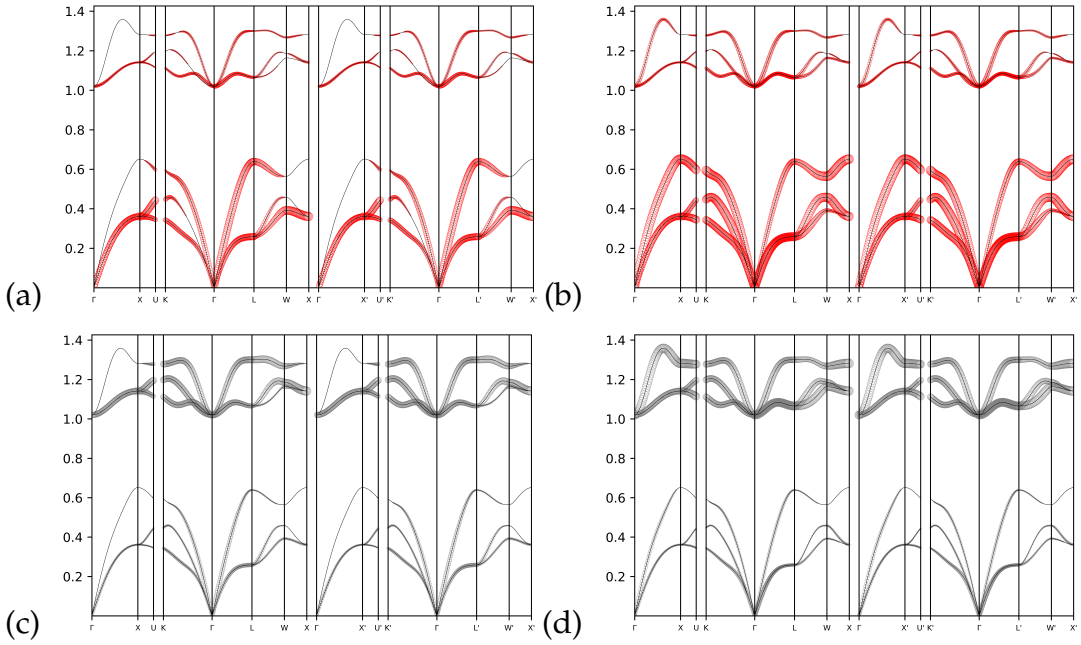


Figure 3.1: Phonon bands of GaN (Phonopy database, mp-830). Longitudinal modes of Ga (a) and N (c). Transverse modes of Ga (b) and N (d).

3.1.7 Interface to QuantumEspresso and Wannier90

[Material proposed by Dr. Pengfei Liu]

```
ENV["PSEUDO_ROOT"] = "../../../pseudopotentials"
include("../../include/using.jl")
include("../../include/tools.jl")
include("../../include/CIF2LN.jl")
include("../../include/wannier90_utils.jl")
global const WKSPC = "../../../MXene/Mo2N"

projfn           = "$WKSPC/Mo2N_projwfc/Mo2N.proj.projwfc_up"
proj_out_fn      = "$WKSPC/Mo2N_projwfc/Mo2N.projwfc.x.out"
bandfn           = "$WKSPC/Mo2N_nscf/Mo2N.pw.x.out"
band_xml_fn      = "$WKSPC/Mo2N_scf/Mo2N.band.xml"
band_highsymm_fn = "$WKSPC/Mo2N_band/Mo2N.pw.x.out"
scffn            = "$WKSPC/Mo2N_scf/Mo2N.pw.x.out"
band_w90_fn      = "$WKSPC/Mo2N_wannier90/Mo2N_band.dat"
hr_w90_fn        = "$WKSPC/Mo2N_wannier90/Mo2N_hr.dat"
MoN2_cif_fn      = "$WKSPC/../Mo2N.0.cif"
```

```
function pw_bands_from_xml_unit_eV(band_xml_file)
    hatree_unit_in_eV = 27.211324570273 # Electron Volt
    @inline parsenf(s) = parse.(Float64,split(s,r"\s",keepempty=false))
    xml = XMLDict.parse_xml(join(readlines(band_xml_file),"\n")) ;
    ks_energies_dics = xml["output"]["band_structure"]["ks_energies"] ;
    k_points = hcat([parsenf(ks_en["k_point"][""])
                     for ks_en in ks_energies_dics]...)
    ks_energies = hcat([hatree_unit_in_eV .* parsenf(ks_en["eigenvalues"][""])
                        for ks_en in ks_energies_dics]...)
    return k_points, ks_energies
end
```

```
function Mo2N(fn)
    ORIG = [-8e-5, -1e-4, -7e-5]
    ID3  = [1 0 0; 0 1 0; 0 0 1]
    @inline select_non_empty(D) = Dict(k=>v for (k,v) in D if length(v)>0)
    xx = [[:s,:d1,:d2,:d3,:d4,:d5],
          [:s,:d1,:d2,:d3,:d4,:d5],
          [:px,:py,:pz]] .|> LatticeLab.Orbits
    UC = CIF2UC(CIF(fn), xx)
    ALL_LN     = all_links_between_sublattices(
                    UC;
                    maxlen=12, max_distance=22,
                    rounding_digits=8 ) |> select_non_empty
                    ALL_LN_nnn = pick_nearest_by_ij(
                    ALL_LN,
                    merge( Dict((i,i)=>0 for i=1:6),
                           Dict((i,j)=>2 for i=1:6 for j=1:6 if i!=j) );
                    digits=6
    )
    SP = all_links_dict_to_sp_no_directions(ALL_LN_nnn) ;
    LN = LinkInfo(copy(UC), SP) ;
    return build_lattice(LN, BoundingBox((ORIG,ID3,[3,3,1],[true,true,false])))
end
```

```
function band_structure_from_pw_xml_projwfc_up(
    xml_fn::String,
    projwfc_up_fn::String,
    alat::Float64,
    kpath_abs,
    atom_orbit_list     )
    ## construct BS
    kp, en = pw_bands_from_xml_unit_eV(xml_fn)
    kpabs  = ((2pi/alat).*kp)
    BS = BandStructures.QE_bandsx_out(
                Pair{String, Tuple{Vector{T} where T, Int64}}[kpath_abs...,],
                [kpabs[:,i] => en[:,i] for i=1:size(en,2)]
    )
    ## add markers
    proj, states = projwfcx_output_projwfc_up(readlines(projwfc_up_fn))
    @info join(string.(states), "\n")
    nkp = size(proj,1)
    nbd = size(proj,2)
    nop = length(atom_orbit_list)
    @inline select_atom_l(atm, orb) =
        [ i for (i,t) in enumerate(states)
            if occursin(atm,t.atom) && t.orbit==orb ]
    @inline weight_atm_l(atm, orbit) =
        reshape( sum(proj[:,:,select_atom_l(atm, orbit)], dims=3), nkp, nbd )
    wt = zeros(Float64, nkp, nbd, nop)
    for (i,(atm,orb)) in enumerate(atom_orbit_list)
        wt[:,:,i] = weight_atm_l(atm,orb)
    end
    ik = 0
    BS.Markers = [
        kp_lb => [((ik+=1); (k, Matrix{ComplexF64}(wt[ik,:,:])) )
                    for (k,e) in kp_en_pairs]
        for (kp_lb,kp_en_pairs) in BS.Bands     ]
    ## return
    return BS
end
```

```
const KPATH_REL = [
    "G"  =>  [0,    0,   0 ],
    "K"  =>  [1//3, 1//3,0 ],
    "M"  =>  [1//2, 0,   0 ],
    "G"  =>  [0,    0,   0 ],  ]
const KPATH_REL_I = [
    "G"  =>  ([0,    0,   0 ], 100),
    "K"  =>  ([1//3, 1//3,0 ], 50 ),
    "M"  =>  ([1//2, 0,   0 ], Int(floor(50*sqrt(3)))),
    "G"  =>  ([0,    0,   0 ], 1), ]
@inline kvec_rel2abs(rel,  b) = [k=>(b*vrel) for (k,vrel) in rel]
@inline kveci_rel2abs(rel, b) = [k=>(b*vrel[1],vrel[2]) for (k,vrel) in rel]
@inline kvec_rel(rel, b) = [k=>(b*vrel) for (k,vrel) in rel]
@inline HSP(b)  = kvec_rel2abs(KPATH_REL,b)
@inline HSPi(b) = kveci_rel2abs(KPATH_REL_I,b)
```

```
Mo2N_LATT = Mo2N(MoN2_cif_fn)

HQw = make_HWannQ(
        readlines(hr_w90_fn),
        Mo2N_LATT;
        thr=0.001 ) ;

BS_Wann = LatticeLab.band_structure(
    HSP(reciprocal_basis(Mo2N_LATT)), HQw, dk=1e-3
) |> LatticeLab_bands_BandStructure;

BS_DFT  = band_structure_from_pw_xml_projwfc_up(
    band_xml_fn,
    projfn,
    Mo2N_LATT.UC.a[1,1],
    HSPi(reciprocal_basis(Mo2N_LATT)),
    [("Mo","4S"),("Mo","4P"),("Mo","4D"),("Mo","5S"),("N","2P")]
)
```

3.1.8 Fat bands with coloring

[continue with the previous example]

```
Mo2N_plot_settings = Dict(:line_colors=>["grey", "black"],
                          :lw=>[0.4,0.7],
                          :ref_levels=>[(-3.8617,"gray"),(0.0,"gray")],
                          :K_sep=>-1,
                          :range=>(-21.0,15.0),
                          :markerstrokealpha=>0.8,
                          :markerstrokewidth=>0.4,
                          :aspect_ratio=>0.4,
                          :figure_size=>(8,16),)
# Fig (a)
plot_bands("MoN2_DFT_black_Wann_red.pdf",
           [BS_DFT, BS_Wann],
           SIZE = [0, 0, 0, 0, 0],
           dpi  = 600,
           settings= merge(Mo2N_plot_settings,
                       Dict(:line_colors=>["black", "red"],
                            :lw=>[1.3,0.7],)))
# Fig (b)
plot_bands("MoN2_DFT_Wann_Mo_4S+4P.pdf",
           [BS_DFT, BS_Wann],
           SIZE = [120.0, 120.0, 0, 0, 0],
           dpi  = 600,
           settings= Mo2N_plot_settings)
# Fig (c)
plot_bands("MoN2_DFT_Wann_Mo_4D+5S.pdf",
           [BS_DFT, BS_Wann],
           SIZE = [0, 0, 120, 120, 0],
           dpi  = 600,
           settings = Mo2N_plot_settings)
# Fig (d)
plot_bands("MoN2_DFT_Wann_N_2P.pdf",
           [BS_DFT, BS_Wann],
           SIZE = [0, 0, 0, 0, 120],
           dpi  = 600,
           settings = Mo2N_plot_settings )
```

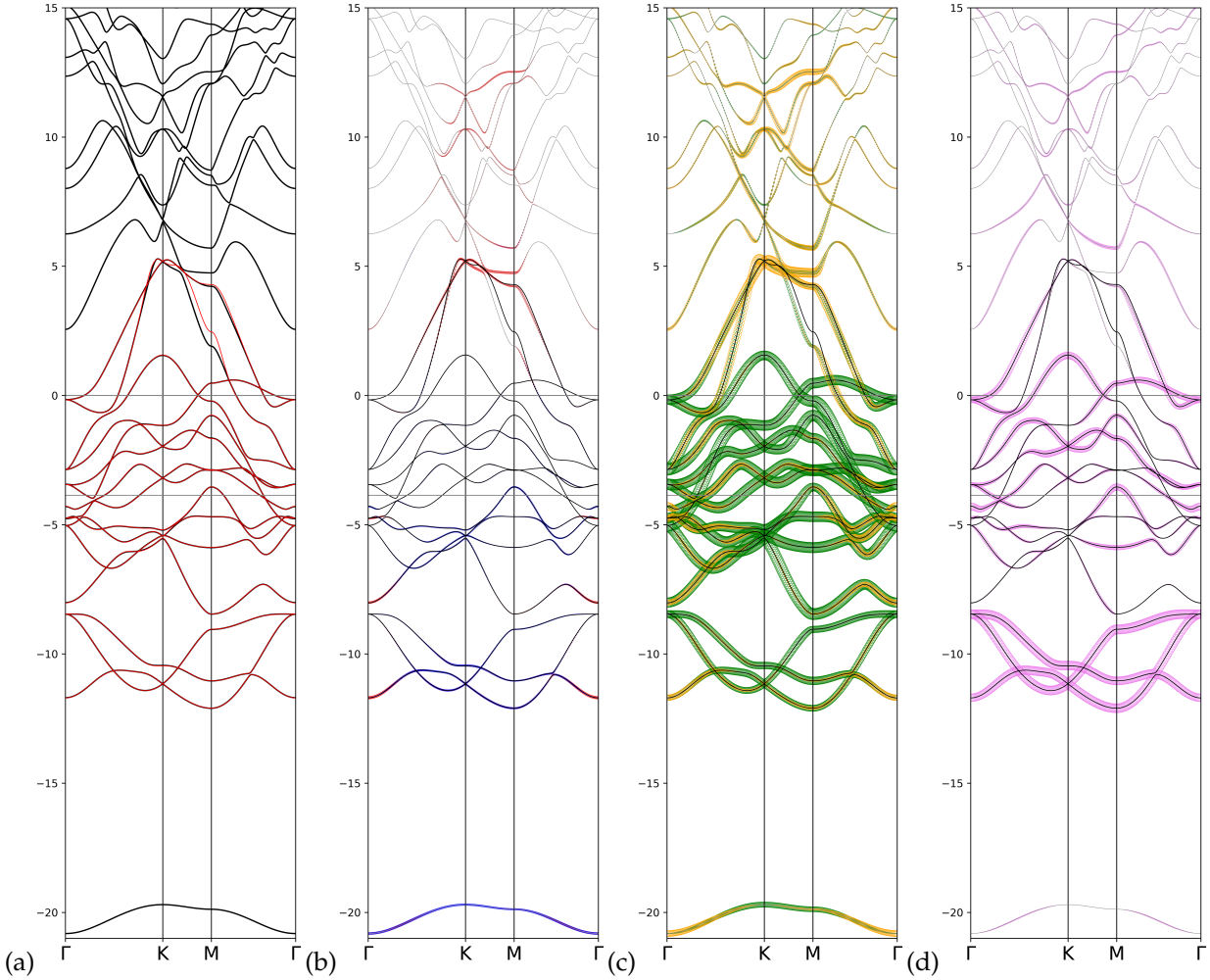


Figure 3.2: Results of the four calls to plot functions.