
chinook Documentation

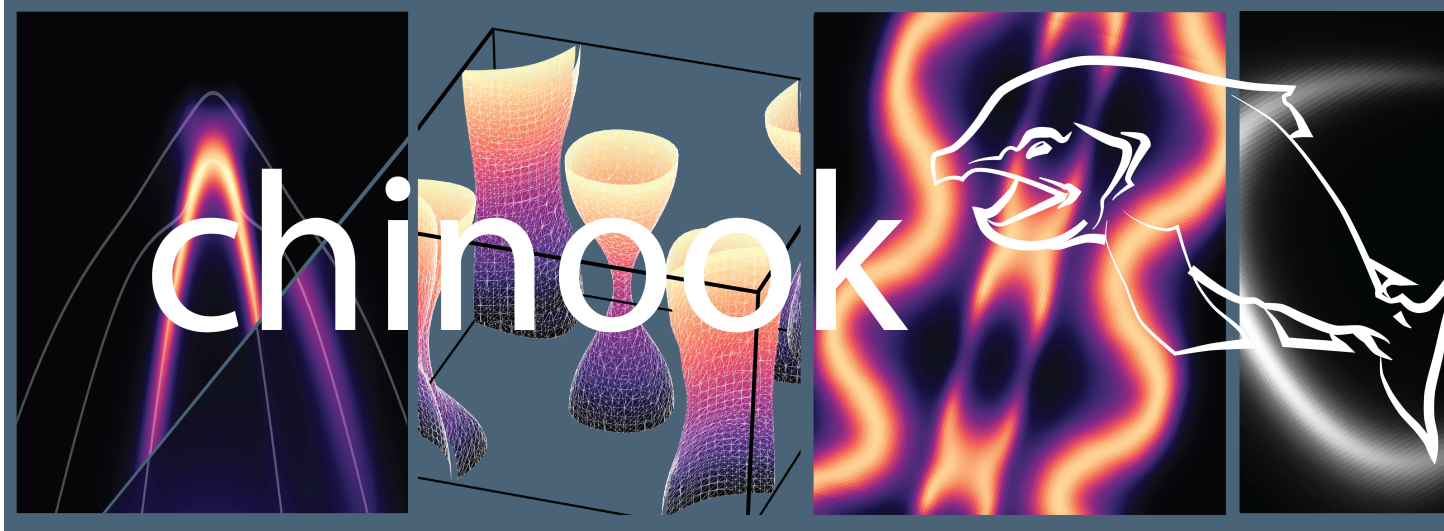
Release 1.0

Ryan P. Day

Jun 05, 2022

Contents:

1	Introduction	3
2	ARPES Simulation	7
3	Tight Binding	25
4	Model Diagnostics	43
5	Slab Calculation	59
6	Support Files	71
7	Input Arguments	77
8	Tutorial	85
9	License	95
10	Contact	97
11	Indices and tables	99
	Python Module Index	101
	Index	103



Welcome to the documentation for chinook, developed at the Quantum Matter Institute in Vancouver, BC at the University of British Columbia. chinook was designed with an aim towards providing a convenient framework in which to simulate angle-resolved photoemission spectroscopy (ARPES) experiments. In doing so, we have built a more general environment in which one can construct and study tight-binding and other effective single-particle Hamiltonians. In these pages, you will find comprehensive documentation for all methods available in the chinook package, in addition to a few instructive tutorials and some instructions for getting started. We hope you like what you find, and encourage you to get involved in this project by building new tools to extend the library here beyond its current form. For a detailed overview of our methodology, in addition to a few motivating examples, please see our recent paper in [npj Quantum Materials](#). The development of chinook was motivated by an absence of readily available tools to facilitate interpretation and design of new experiments. In this spirit, this software is offered for free; we request only that any publications which make use of chinook cite the following reference:

R.P. Day, B. Zwartsenberg, I.S. Elfimov, A. Damascelli, *Computational Framework chinook for Simulation of Angle-Resolved Photoelectron Spectroscopy*, npj Quantum Materials 4, 54 (2019) Thanks for stopping by!

1.1 Installation

The entire source code for *chinook* is available as a github repository, which can be accessed by visiting our [repo](#).

For most convenient access, *chinook* is registered and available for download via *pip*. Open a command line shell and execute

```
pip install chinook
```

We have avoided use of unconventional packages to run *chinook*, and the only required packages are [numpy](#), [matplotlib](#), and [scipy](#). Some high performance operation is available if [psutil](#) is also available.

For any publications which make use of *chinook*, please reference

R.P. Day, B. Zwartsenberg, I.S. Elfimov and A. Damascelli, *Computational framework chinook for angle-resolved photoemission spectroscopy*, npj Quantum Materials 4, 54 (2019)

1.2 Getting Started

The recommended structure for defining the necessary input for a *chinook* calculation can be found in the following `template` file. Entire calculations can be done in a single script, or as illustrated for the example in the [Tutorial](#) page, input and execution can be split across multiple python scripts.

At the core of any chinook calculation is the tight-binding model. The requisite information to construct such a model is an orbital basis and a Hamiltonian which effectively describes the Hilbert space spanned by this orbital basis.

In order to construct the essential objects in chinook, we make use of Python dictionaries, a hash-table representation which allows for an efficient, but more importantly human-readable, input format. If you open the `template.py` file in your chinook distribution, you will find the basic input for starting a calculation. A comprehensive overview of the required and optional arguments supported for the input dictionaries can be found on the [Input Arguments](#) page.

1.3 Lattice

The lattice is defined as a 3x3 array called `avec`, and is defined in units of Angstrom. The default provided in `template.py` is a unit-length simple cubic lattice. As an example here, we construct the unit cell of GaAs,

```
avec = np.array([[2.8162, 2.8162, 0], [0, 2.8162, 2.8162], [2.8162, 0, 2.8162]])
```

1.4 Basis

The basis in chinook is defined as a list of orbital objects.

```
basis_args = {'atoms':[0,1],
              'Z':{0:31,1:33},
              'pos':[np.array([0,0,0]),np.array([1.4081,1.4081,1.4081])],
              'orbs':[['40','41x','41y','41z'],['40','41x','41y','41z']]}
```

We see from this that the distinct atomic species (Ga and As) are indexed as 0 and 1 in the ‘atoms’ argument. Their atomic numbers *Z* are then given below in a dictionary format, with key-value pairs corresponding to the atom index, and the atomic number. The atom positions can then be given, in units of Angstrom. These are entered as numpy arrays, within a list which orders in the same way as atoms does. Finally, we define the orbitals at each of these basis sites which are to be included in our model. In this case, we include both *s* and all *px*, *py*, and *pz* orbitals for both Ga and As. Orbitals are labelled by the principal quantum number *n*, orbital angular momentum *l* and then a standard label. Details can be found in `chinook.orbital.py`. Note that each atom gets its own list of orbitals, rather than putting them all together in a single list. Finally, there are additional options, including rotations and spinful spinor bases which can be implemented. More details also in `chinook.orbital.py`.

1.5 Hamiltonian

The user is given a fair amount of flexibility in passing the Hamiltonian to chinook. For further details on Hamiltonian types and the way these are converted into a functioning tight-binding model in our framework, it is recommended to refer to the documentation under [Tight Binding](#). In the template, we are assuming that the user wishes to enter a Slater-Koster type Hamiltonian. This is a fairly compact representation for tight-binding with minimal parameters, requiring only onsite energies and a few hopping terms need to be defined. We can use the parameters given by, for example Harrison or Cohen, which gives us a dictionary which looks like so:

```
V_SK = {'040':-6.01, '041':0.19, '140':-4.79, '141':4.59,
        '014400S':-1.75, '014401S':-3.15, '014410S':-1.60,
        '014411S':2.59, '014411P':-0.94}
```

At first glance, the format of this dictionary may appear cryptic. We’ll break it down here. Each key:value pair indicates the orbitals and type of coupling, and the value the amplitude, in units of eV. The order is arbitrary, but we begin above by defining the on-site energies. These are labelled uniquely with key-strings formatted as ‘atom-n-l’. So ‘041’ indicates the onsite energy for the first atom in our basis (i.e. Ga), in the *n*=4, *l*=1 (i.e. *p*) shell. In other words, the onsite energy for the Ga 4*p* states. Similarly, ‘140’ corresponds to the As 4*s* states. Next, we define the various Slater-Koster type hoppings, such as *V*_{ss}, *V*_{sp}, *V*_{pp}, *V*_{ppp}. These have key-strings as ‘atom1-atom2-n1-n2-l1-l2-x’. The ‘x’ at the end refers to sigma (S), pi (P), or delta (D) bonding symmetry. For example, ‘014410S’ is the Ga 4*p* - As 4*s* sigma bonding strength, and ‘014411P’ the Ga 4*p* - As 4*p* pi bonding. An arbitrary bond will ultimately be a linear combination of these. Note that we do not include any terms with ‘00’ or ‘11’ at the beginning. Considering only nearest-neighbour hopping in this model, we account for only on-site and Ga-As hopping. For more information on Slater-Koster Hamiltonians and other tight-binding model formats, we refer you to [Tight Binding](#). Moving on,

with the `V_SK` dictionary defined, the user can combine this with the other pertinent descriptors for the hamiltonian arguments, as seen in the template:

```
hamiltonian_args = {'type': 'SK',
                    'V': V_SK,
                    'avec': avec,
                    'cutoff': 2.5,
                    'renorm': 1.0,
                    'offset': 0.0,
                    'tol': 1e-4}
```

In addition to denoting that we will use a Slater-Koster type Hamiltonian, we pass the dictionary of potentials ‘V’. In addition to the lattice vectors, we finally define a few numeric values, corresponding to the cutoff hopping lengthscale (in Angstrom), any overall renormalization factor for the bandstructure, an offset for the Fermi level, and a tolerance, or minimal Hamiltonian matrix element size we wish to consider—any smaller terms will be neglected. The cutoff is chosen here to specify that only nearest-neighbour hoppings will be considered (Ga-As bondlength is approximately 2.48 Å).

1.6 Tight-Binding Model

With these input arguments defined we can now actually build our tight-binding model and begin to test it. To do so, we first define the list of orbitals for our model:

```
basis = chinook.build_lib.gen_basis(basis_args)
```

This is then used with the *hamiltonian_args* defined above to build the tight-binding model:

```
TB = chinook.build_lib.gen_TB(basis, hamiltonian_args)
```

We now have a functioning tight binding model. However, to actually perform any work, we need to know where in momentum space we are interested in diagonalizing over. This is done using a similar argument structure as before.

1.7 Momentum

The momentum arguments get passed as a dictionary:

```
L, G, X = np.array([0.5, 0.5, 0.5]), np.array([0.0, 0.0, 0.0]), np.array([0.5, 0.5, 0.0])
momentum_args = {'type': 'F',
                  'avec': avec,
                  'grain': 100,
                  'pts': [L, G, X],
                  'labels': ['L', '$\\Gamma$', 'X']}
```

Here we are using *F* or fractional units for momentum space (as opposed to *A* absolute units of inverse Angstrom) to define our k-path. This requires also that we pass then the unit cell vectors. The *grain* sets the number of points between each high-symmetry point we want to evaluate at. The endpoints of interest are passed similar to what we did with the basis positions, as a list of numpy arrays, which I have pre-defined for tidier code. Finally, we have an option to provide labels for when we go ultimately to plot our bandstructure over this k-path. I can now set the k-path for my tight-binding model:

```
TB.Kobj = chinook.build_lib.gen_K(momentum_args)
```

And then diagonalize and plot the band structure.

```
TB.solve_H()
TB.plotting()
```

1.8 ARPES Calculation

Presumably, we're all here for the ARPES calculations. Once you have a tight-binding model you're happy with, you can proceed to initialize and execute an ARPES experiment. We do this with the following input

```
arpes_args = {'cube':{'X':[-0.5,0.5,100],
                      'Y':[-0.5,0.5,100],
                      'kz':0.0,
                      'E':[-3,0.1,1000]},
              'hv':21.2,
              'SE':['constant',0.001],
              'T':4.2,
              'pol':np.array([1,0,0]),
              'resolution':{'E':0.01,'k':0.01}}
```

This is sort of the most basic set of arguments we can define for an ARPES experiment, leaving most others as default. We have defined a *cube* of momentum and energy over which we are interested in evaluating the photoemission intensity. We set both the endpoints and grain for the momentum in-plane as well as the energy, given here as binding energy relative to the zero of our tight-binding model, which will be the Fermi level. A fixed out-of-plane momentum is chosen, and defined as *kz*. Along with this *cube*, we fix the photon energy for the experiment. With these two sets of parameters defined, the matrix elements can be calculated. As a result, all other arguments given here can be updated after evaluating the matrix elements, such that different parameter choices and their influence on the ARPES intensity can be surveyed very quickly and with little computational overhead. The first of these is the self-energy. Here, we the self-energy (i.e. *SE*) is taken to be purely imaginary, giving only the width as a constant 1 meV width for the peaks. Various other approximations are available, to which you are referred to the documentation of *chinook.ARPES_lib.py*, including Kramers-Kronig related Real and Imaginary parts of the self-energy. In addition, the Fermi function is evaluated here at 4.2 K to suppress intensity from unoccupied states. A polarization and both energy and momentum resolutions are also included. The units for the latter are in terms of eV and inverse Angstrom respectively, and are evaluated at FWHM. These parameters, along with our tight-binding model can then be used to seed an experiment

```
experiment = chinook.arpes_lib.experiment(TB,arpes_args)
experiment.datacube()
Imap = experiment.spectral()
```

In these three lines, we initialize the experiment, evaluate the matrix elements, and generate an ARPES intensity map. Please refer to the documentation for notes on further parameters available for these calculations, and methods for updating parameters following execution of *experiment.datacube()*.

ARPES Simulation

In addition to the core *ARPES_lib* library, several other scripts in the module are written with the express purpose of facilitating calculation of the ARPES intensity. All relevant docs are included below.

In setting up an ARPES calculation, one requires an existing model system, i.e. instance of the *TB_lib.TB_model* class. This includes all relevant information regarding the orbital basis and the model Hamiltonian. In addition to this, a number of experimental parameters should be specified. Similar to the input for defining an orbital basis and a Hamiltonian, we use python *dictionaries* to specify these details. An example input is shown here.

```
ARPES_dict = {'cube':{'X':[-1,1,200], 'Y':[-0.5,0.5,100], 'E':[-1,0.05,1000], 'kz':0},
              'hv':21.2,
              'pol':np.array([1,0,0]),
              'T':4.2,
              'SE':['fixed',0.02],
              'resolution':{'dE':0.005, 'dk':0.01}}
```

By passing this dictionary to the input statement

```
experiment = ARPES_lib.experiment(TB,ARPES_dict)
```

We initialize an ARPES experiment which will diagonalize the Hamiltonian over a 200x100 point mesh in the region $-1 \leq k_x \leq 1$ and $-0.5 \leq k_y \leq 0.5$. For states in the range of energy $-1 \leq E \leq 0.05$ eV of the Fermi level, we will explicitly compute the ARPES cross section when requested. The calculation will be done with photon energy for He-1 α at 21.2 eV, and a sample temperature of 4.2 K. This carries into the calculation in the form of a Fermi-Dirac distribution, which suppresses intensity from states with positive energies. The polarization of light is taken to be along the x direction, and is indicated by length-3 array associated with the keyword *pol*.

The *SE* key indicates the form of the self-energy to be imposed in evaluating the lineshape of the spectral features associated with each peak. Here we impose a fixed-linewidth of 20 meV on all states, to allow us to focus on the matrix elements alone, without further complications from energy-dependent lineshape broadening. As detailed in the *ARPES_lib.SE_gen()* below, more sophisticated options are available.

Finally, resolution is passed as well, with arguments for both energy and momentum resolution, expressed as full-width half maximum values, in units of electron volts and inverse Angstrom. Many more non-default options are available, including sample rotations, spin-projection (for spin-ARPES) and radial-integrals. See the documentation for *ARPES_lib.experiment* for further details.

Once a calculation of the matrix elements is completed, one is interested in plotting the associated intensity map. There are several options for this. First, the intensity map must be built using `ARPES_lib.experiment.spectral()`. Note that GUI tools do this automatically, without the user's input. The `ARPES_lib.experiment.spectral()` method will apply the matrix element associated with each state, to its spectral function, and sum over all states to produce a complete dataset. It generates a raw and resolution broadened intensity map. The *slice_select* option allows for plotting specific cuts in energy or momentum. Once a full dataset has been generated, this can be passed to `ARPES_lib.experiment.plot_intensity_map()` to quickly image a different cut. Alternatively, interactive GUI tools are available under *Matplotlib Plotter*, or if the user has Tkinter installed, `ARPES_lib.experiment.plot_gui()`.

2.1 Numerical Integration

For evaluation of radial integrals

$$B_{n,l}^{l'}(k) = (i)^{l'} \int dr R_{n,l}(r) r^3 j_{l'}(kr)$$

we use an adaptive integration algorithm which allows for precise and accurate evaluation of numeric integrals, regardless of local curvature. We do this by defining a partition of the integration domain which is recursively refined to sample regions of high curvature more densely. This is done until the integral converges to within a numerical tolerance.

The user is given some opportunity to specify details of the evaluation of radial integrals $B_{n,l}^{l'}(k)$ used in the calculations. Specifications can be passed to the ARPES calculation through the *ARPES_dict* argument passed to the `ARPES_lib.experiment` object.

`adaptive_int.general_Bnl_integrand(func, kn, lp)`

Standard form of executable integrand in the e.r approximation of the matrix element

args:

- **func**: executable function of position (float), in units of Angstrom
- **kn**: float, norm of the k vector (in inverse Angstrom)
- **lp**: int, final state angular momentum quantum number

return:

- executable function of float (position)

`adaptive_int.integrate(func, a, b, tol)`

Evaluate the integral of **func** over the domain covered by **a**, **b**. This begins by seeding the evaluation with a maximally coarse approximation to the integral.

args:

- **func**: executable
- **a**: float, start of interval
- **b**: float, end of interval
- **tol**: float, tolerance for convergence

return:

- **Q**: (complex) float, value of the integral

`adaptive_int.rect` (*func*, *a*, *b*)

Approximation to contribution of a finite domain to the integral, evaluated as a rough rectangle

args:

- **func**: executable to evaluate
- **a**: float, start of interval
- **b**: float, end of interval

return:

- **recsum**: (complex) float approximated area of the region under function between **a** and **b**

`adaptive_int.recursion` (*func*, *a*, *b*, *tol*, *currsum*)

Recursive integration algorithm—rect is used to approximate the integral under each half of the domain, with the domain further divided until result has converged

args:

- **func**: executable
- **a**: float, start of interval
- **b**: float, end of interval
- **tol**: float, tolerance for convergence
- **currsum**: (complex) float, current evaluation for the integral

return:

- recursive call to the function if not converged, otherwise the result as complex (or real) float

`radint_lib.define_radial_wavefunctions` (*rad_dict*, *basis*)

Define the executable radial wavefunctions for computation of the radial integrals

args:

- **rad_dict**: essential key is '*rad_type*', if not passed, assume Slater orbitals.

- **rad_dict['rad_type']**:

- '*slater*': default value, if '*rad_type*' is not passed,

Slater type orbitals assumed and evaluated for the integral

- '*rad_args*': dictionary of float, supplying optional final-state

phase shifts, accounting for scattering-type final states. keys of form 'a-n-l-lp'. Radial integrals will be accordingly multiplied

- '*hydrogenic*': similar in execution to '*slater*',

but uses Hydrogenic orbitals—more realistic for light-atoms

- '*rad_args*': dictionary of float, supplying optional final-state

phase shifts, accounting for scattering-type final states. keys of form 'a-n-l-lp'. Radial integrals will be accordingly multiplied

- *'grid'*: radial wavefunctions evaluated on a grid of

radial. Requires also another key_value pair:

- *'rad_args'*: dictionary of numpy arrays evaluating

the radial wavefunctions. Requires an *'r'* array, as well as *'a-n-l'* indicating 'atom-principal quantum number-orbital angular momentum'. Must pass such a grid for each orbital in the basis!

- *'exec'*: executable functions for each 'a-n-l' i.e.

'atom-principal quantum number-orbital angular momentum'. If executable is chosen, require also:

- *'rad_args'*, which will be a dictionary of

executables, labelled by the keys 'a-n-l'. These will be passed to the integral routine. Note that it is required that the executables are localized, i.e. vanishing for large radial.

- *'fixed'*: radial integrals taken to be constant float,

require dictionary:

- *'rad_args'* with keys 'a-n-l-lp', i.e.

'atom-principal quantum number-orbital angular momentum-final state angular momentum' and complex float values for the radial integrals.

- **basis**: list of orbital objects

return:

orbital_funcs: dictionary of executables

`radint_lib.fill_radint_dic(Eb, orbital_funcs, hv, W=0.0, phase_shifts=None, fixed=False)`

Function for computing dictionary of radial integrals. Can pass either an array of binding energies or a single binding energy as a float. In either case, returns a dictionary however the difference being that the key value pairs will have a value which is itself either a float, or an interpolation mesh over the range of the binding energy array. The output can then be used by either writing **Bdic['key']** or *****Bdic['key']****(valid float between endpoints of input array)

args:

- **Eb**: float or tuple indicating the extremal energies
- **orbital_funcs**: dictionary of executable orbital radial wavefunctions
- **fixed**: bool, if True, constant radial integral for each scattering

channel available: then the orbital_funcs dictionary already has the radial integral evaluated

- **hv**: float, photon energy of incident light.

kwargs:

- **W**: float, work function

- **phase_shifts**: dictionary for final state phase shifts, as an optional extension beyond pure- free electron final states. For now, float type.

return:

- **Brad**: dictionary of executable interpolation grids

`radint_lib.find_cutoff (func)`

Find a suitable cutoff lengthscale for the radial integration: Evaluate the function over a range of 20 Angstrom, with reasonable detail ($dr = 0.02$ Å). Find the maximum in this range. The cutoff tolerance is set to $1/1e4$ of the maximum value. Since this 'max' is actually a lower bound on the true maximum, this will only give us a more strict cutoff tolerance than is absolutely possible. With this point found, we then find all points which are within the tolerance of zero. The frequency of these points is then found. When the frequency is constant and 1 for all subsequent points, we have found the point of convergence. If the 'point of convergence' is the last point in the array, the radial wavefunction really isn't suitably localized and the user should not proceed without giving more consideration to the application of the LCAO approximation to such a function.

args:

- **func**: the integrand executable

return:

- float, cutoff distance for integration

`radint_lib.gen_const (val)`

Create executable function returning a constant value

args:

- **val**: constant value to return when executable function

return:

- lambda function with constant value

`radint_lib.gen_orb_labels (basis)`

Simple utility function for generating a dictionary of atom-n-l:[Z, orbital label] pairs, to establish which radial integrals need be computed.

args:

- **basis**: list of orbitals in basis

return:

- **orbitals**: dictionary of radial integral pairs

`radint_lib.make_radint_pointer (rad_dict, basis, Eb)`

Define executable radial integral functions, and store in a pointer-integer referenced array. This allows for fewer executions of the interpolation function in the event where several orbitals in the basis share the same a,n,l. Each of these gets 2 functions for $l \pm 1$, which are stored in the rows of the array **B_array**. The orbitals in the basis then are matched to these executables, with the corresponding executable row index saved in **B_pointers**.

Begin by defining the executable radial wavefunctions, then perform integration at several binding energies, finally returning an interpolation of these integrations.

args:

- **rad_dict**: dictionary of ARPES parameters: relevant keys are ‘hv’ (photon energy), ‘W’ (work function), and the `rad_type` (radial wavefunction type, as well as any relevant additional pars, c.f. `radint_lib.define_radial_wavefunctions`). Note: ‘`rad_type`’ is optional, (as is `rad_args`, depending on choice of radial wavefunction.)
- **basis**: list of orbitals in the basis
- **Eb**: tuple of 2 floats indicating the range of energy of interest (increasing order)

return:

- **B_array**: numpy array of Nx2 executable functions of float
- **B_pointers**: numpy array of integer indices matching orbital basis ordering to the functions in **B_array**

`radint_lib.radint_calc(k_norm, orbital_funcs, phase_shifts=None)`

Compute dictionary of radial integrals evaluated at a single **|k|** value for the whole basis. Will avoid redundant integrations by checking for the presence of an identical dictionary key. The integration is done as a simple adaptive integration algorithm, defined in the `adaptive_int` library.

args:

- **k_norm**: float, length of the k-vector
(as an argument for the spherical Bessel Function)
- **orbital_funcs**: dictionary, radial wavefunction executables

kwargs:

- **phase_shifts**: dictionary of phase shifts, to convey final state scattering

returns:

- **Bdict**: dictionary, key value pairs in form – ‘ATOM-N-L’:*Bval*

`radint_lib.radint_dict_to_arr(Bdict, basis)`

Take a dictionary of executables defined for different combinations of a,n,l and send them to an array, with a corresponding pointer array which can be used to dereference the relevant executable.

args:

- **Bdict**: dictionary of executables with ‘a-n-l’ keys
- **basis**: list of orbital objects

return:

- **Blist**: numpy array of the executables, organized by a-n-l, and l’ (size Nx2, where N is the length of the set of distinct a-n-l triplets)
- **pointers**: numpy array of length (basis), integer datatype indicating the related positions in the **Blist** array

2.2 ARPES Library

`ARPES_lib.G_dic()`

Initialize the gaunt coefficients associated with all possible transitions relevant

return:

- **Gdict:** dictionary with keys as a string representing (l,l',m,dm) “ll’mdm” and values complex float.

All unacceptable transitions set to zero.

`ARPES_lib.Gmat_make(lm, Gdictionary)`

Use the dictionary of relevant Gaunt coefficients to generate a small 2x3 array of float which carries the relevant Gaunt coefficients for a given initial state.

args:

- **lm:** tuple of 2 int, initial state orbital angular momentum and azimuthal angular momentum
- **Gdictionary:** pre-calculated dictionary of Gaunt coefficients, with key-values associated with “ll’mdm”

return:

- **mats:** numpy array of float 2x3

`ARPES_lib.all_Y(basis)`

Build L-M argument array input arguments for every combination of l,m in the basis. The idea is for a given k-point to have a single call to evaluate all spherical harmonics at once. The pointer array orb_point is a list of lists, where for each projection in the basis, the integer in the list indicates which row (first axis) of the Ylm array should be taken. This allows for very quick access to the l+/-1, m+/-1,0 Ylm evaluation required.

args:

- **basis:** list of orbital objects

return:

- **l_args:** numpy array of int, of shape `len(lm_inds),3,2`, with the latter two indicating the final state orbital angular momentum
- **m_args:** numpy array of int, of shape `len(lm_inds),3,2`, with the latter two indicating the final state azimuthal angular momentum
- **g_arr:** numpy array of float, shape `len(lm_inds),3,2`, providing the related Gaunt coefficients.
- **orb_point:** numpy array of int, matching the related sub-array of `l_args`, `m_args`, `g_arr` related to each orbital in basis

`ARPES_lib.con_ferm(ekbt)`

Typical values in the relevant domain for execution of the Fermi distribution will result in an overflow associated with 64-bit float. To circumvent, set fermi-function to zero when the argument of the exponential in the denominator is too large.

args:

- **ekbt:** float, (E-u)/kbT in terms of eV

return:

- **fermi:** float, evaluation of Fermi function.

`class ARPES_lib.experiment(TB, ARPES_dict)`

The experiment object is at the centre of the ARPES matrix element calculation. This object keeps track of the

experimental geometry as well as a local copy of the tight-binding model and its dependents. Such a copy is used to avoid corruption of these objects in the global space during a given run of the ARPES experiment.

args:

- **TB**: instance of a tight-binding model object
 - **ARPES_dict**: dictionary of relevant experimental parameters including
 - ‘*hv*’: float, photon energy (eV),
 - ‘*mfp*’: float, mean-free path (Angstrom),
 - ‘*resolution*’: dictionary for energy and momentum resolution:
 - * ‘*dE*’: float, energy resolution (FWHM eV),
 - * ‘*dk*’: float, momentum resolution (FWHM 1/Angstrom)
 - ‘*T*’: float, Temperature of sample (Kelvin)
 - ‘*cube*’: dictionary momentum and energy domain
- (‘*kz*’ as float, all others (‘*X*’ , ‘*Y*’ , ‘*E*’) are list or tuple of floats *Xo*,*Xf*,*dX*)

optional args:

In addition to the keys above, *ARPES_dict* can also be fed the following:

- ‘*spin*’: spin-ARPES measurement, list [± 1 , np.array([a,b,c])]

with the numpy array indicating the spin-projection direction (with respect to) experimental frame.

- ‘*rad_type*’: string, radial wavefunctions, c.f. *chinook.rad_int.py* for details
- ‘*threads*’: int, number of threads on which to calculate the matrix elements.

Requires very large calculation to see improvement over single core.

- ‘*slab*’: boolean, will truncate the eigenfunctions beyond the penetration depth (specifically 4x penetration depth), default is False
- ‘*angle*’: float, rotation of sample about normal emission i.e. z-axis (radian), default is 0.0
- ‘*W*’: float, work function (eV), default is 4.0

M_compute (*i*)

The core method called during matrix element computation.

args:

- **i**: integer, index and energy of state

return:

- **Mtmp**: numpy array (2x3) of complex float corresponding to the matrix element projection for $dm = -1, 0, 1$ (columns) and spin down or up (rows) for a given state in *k* and energy.

Mk_wrapper (*ilist*)

Wrapper function for use in multiprocessing, to run each of the processes as a serial matrix element calculation over a sublist of state indices.

args:

- **ilist**: list of int, all state indices for execution.

return:

- **Mk_out:** numpy array of complex float with shape (len(ilst), 2,3)

SE_gen ()

Self energy arguments are passed as a list, which supports mixed-datatype. The first entry in list is a string, indicating the type of self-energy, and the remaining entries are the self-energy.

args:

- **SE_args:** list, first entry can be 'func', 'poly', 'constant', or 'grid'

indicating an executable function, polynomial factors, constant, or a grid of values

return:

- SE, numpy array of complex float, with either shape of the datacube, or as a one dimensional array over energy only.

T_distribution ()

Compute the Fermi-distribution for a fixed temperature, over the domain of energy of interest

return:

- **fermi:** numpy array of float, same length as energy domain array defined by *cube[2]* attribute.

datacube (ARPES_dict=None, diagonalize=False)

This function computes the photoemission matrix elements. Given a kmesh to calculate the photoemission over, the mesh is reshaped to an nx3 array and the Hamiltonian diagonalized over this set of k points. The matrix elements are then calculated for each of these E-k points

kwargs:

- **ARPES_dict:** can optionally pass a dictionary of experimental parameters, to update those defined

in the initialization of the *experiment* object.

return:

- boolean, True if function finishes successfully.

diagonalize (diagonalize)

Diagonalize the Hamiltonian over the desired range of momentum, reshaping the band-energies into a 1-dimensional array. If the user has not selected a energy grain for calculation, automatically calculate this.

return: None, however *experiment* attributes *X*, *Y*, *ph*, *TB.Kobj*, *Eb*, *Ev*, *cube* are modified.

gen_all_pol ()

Rotate polarization vector, as it appears for each angle in the experiment. Assume that it only rotates with THETA_y (vertical cryostat), and that the polarization vector defined by the user relates to centre of THETA_x axis. Right now only handles zero vertical rotation (just tilt)

return:

- numpy array of len(expmt.cube[1]) x 3 complex float, rotated polarization vectors expressed in basis of spherical harmonics

plot_gui ()

Generate the Tkinter gui for exploring the experimental parameter-space associated with the present experiment.

args:

- **ARPES_dict**: dictionary of experimental parameters, c.f. the `__init__` function for details.

return:

- **Tk_win**: Tkinter window.

plot_intensity_map (*plot_map*, *slice_select*, *plot_bands=False*, *ax_img=None*, *colourmap=None*)

Plot a slice of the intensity map computed in *spectral*. The user selects either an array index along one of the axes, or the fixed value of interest, allowing either integer, or float selection.

args:

- **plot_map**: numpy array of shape (self.cube[0],self.cube[1],self.cube[2]) of float
- **slice_select**: list of either [int,int] or [str,float], corresponding to dimension, index or label, value. The former option takes dimensions 0,1,2 while the latter can handle 'x', 'kx', 'y', 'ky', 'energy', 'w', or 'e', and is not case-sensitive.
- **plot_bands**: boolean, option to overlay a constant-momentum cut with the dispersion calculated from tight-binding
- **ax_img**: matplotlib Axes, for option to plot onto existing Axes
- **colourmap**: matplotlib colourmap

return:

- **ax_img**: matplotlib axis object

rot_basis ()

Rotate the basis orbitals and their positions in the lab frame to be consistent with the experimental geometry

return:

- list of orbital objects, representing a rotated version of the original basis if the angle is finite. Otherwise, just return the original basis.

sarpes_projector ()

For use in spin-resolved ARPES experiments, project the computed matrix element values onto the desired spin-projection direction. In the event that the spin projection direction is not along the standard out-of-plane quantization axis, we rotate the matrix elements computed into the desired basis direction.

return:

- **spin_projected_Mk**: numpy array of complex float with same shape as *Mk*

serial_Mk (*indices*)

Run matrix element on a single thread, directly modifies the *Mk* attribute.

args:

- **indices**: list of all state indices for execution; restricting states in *cube_indx* to those within the desired window

smat_gen (*svector=None*)

Define the spin-projection matrix related to a spin-resolved ARPES experiment.

return:

- **Smat**: numpy array of 2x2 complex float corresponding to Pauli operator along the desired direction

spectral (*ARPES_dict=None, slice_select=None, add_map=False, plot_bands=False, ax=None, colourmap=None*)

Take the matrix elements and build a simulated ARPES spectrum. The user has several options here for the self-energy to be used, c.f. *SE_gen()* for details. Gaussian resolution broadening is the last operation performed, to be consistent with the practical experiment. *slice_select* instructs the method to also produce a plot of the designated slice through momentum or energy. If this is done, the function also returns the associated matplotlib.Axes object for further manipulation of the plot window.

kwargs:

- **ARPES_dict**: dictionary, experimental configuration. See *experiment.__init__* and *experiment.update_pars()*
- **slice_select**: tuple, of either (int,int) or (str,float) format. If (int,int), first is axis index (0,1,2 for x,y,E) and the second is the index of the array. More useful typically is (str,float) format, with str as 'x', 'kx', 'y', 'ky', 'E', 'w' and the float the value requested. It will find the index along this direction closest to the request. Note the strings are not case-sensitive.
- **add_map**: boolean, add intensity map to list of intensity maps. If true, a list of intensity objects is appended, otherwise, the intensity map is overwritten
- **plot_bands**: boolean, plot bandstructure from tight-binding over the intensity map
- **ax**: matplotlib Axes, only relevant if **slice_select**, option to pass existing Axes to plot onto

return:

- **I**: numpy array of float, raw intensity map.
- **Ig**: numpy array of float, resolution-broadened intensity map.
- **ax**: matplotlib Axes, for further modifications to plot only if **slice_select** True

thread_Mk (*N, indices*)

Run matrix element on *N* threads using multiprocessing functions, directly modifies the *Mk* attribute.

NOTE 21/2/2019 – this has not been optimized to show any measureable improvement over serial execution. May require a more clever way to do this to get a proper speedup.

args:

- **N**: int, number of threads
- **indices**: list of int, all state indices for execution; restricting states in *cube_indx* to those within the desired window.

truncate_model ()

For slab calculations, the number of basis states becomes a significant memory load, as well as a time bottleneck. In reality, an ARPES calculation only needs the small number of basis states near the surface. Then for slab-calculations, we can truncate the basis and eigenvectors used in the calculation to dramatically improve our capacity to perform such calculations. We keep all eigenvectors, but retain only the projection of the basis states within 2*the mean free path of the surface. The states associated with this projection are retained, while remainders are not.

return:

- **tmp_basis**: list, truncated subset of the basis' orbital objects
- **Evec**: numpy array of complex float corresponding to the truncated eigenvector array containing only the surface-projected wavefunctions

update_pars (*ARPES_dict*, *datacube=False*)

Several experimental parameters can be updated without re-calculating the ARPES intensity explicitly. Specifically here, we can update resolution in both energy and momentum, as well as temperature, spin-projection, self-energy function, and polarization.

args:

- **ARPES_dict**: dictionary, specifically containing
 - ‘*resolution*’: dictionary with ‘*E*’:float and ‘*k*’:float
 - ‘*T*’: float, temperature, a negative value will suppress the Fermi function
 - ‘*spin*’: list of [int, numpy array of 3 float] indicating projection and spin vector
 - ‘*SE*’: various types accepted, see *SE_gen* for details
 - ‘*pol*’: numpy array of 3 complex float, polarization of light

kwargs:

- **datacube**: bool, if updating in *spectral*, only the above can be changed. If instead, updating

at the start of *datacube*, can also pass:

- **hv**: float, photon energy, eV
- **ang**: float, sample orientation around normal, radians
- **rad_type**: string, radial integral type
- **rad_args**: various datatype, see *radint_lib* for details
- **kz**: float, out-of-plane momentum, inverse Angstrom
- **mfp**: float, mean-free path, Angstrom

write_1k (*filename*, *mat*)

Function for producing the textfiles associated with a 2 dimensional numpy array of float

args:

- **filename**: string indicating destination of file
- **mat**: numpy array of float, two dimensional

return:

- boolean, True

write_map (*_map*, *directory*)

Write the intensity maps to a series of text files in the indicated directory.

args:

- **_map**: numpy array of float to write to file
- **directory**: string, name of directory + the file-lead name

return:

- boolean, True

write_params (*Adict*, *parfile*)

Generate metadata text file associated with the saved map.

args:

- **Adict:** dictionary, ARPES_dict same as in above functions, containing relevant experimental parameters for use in saving the metadata associated with the related calculation.
- **parfile:** string, destination for the metadata

ARPES_lib.**find_mean_dE**(Eb)

Find the average spacing between adjacent points along the dispersion calculated.

args:

- **Eb:** numpy array of float, eigenvalues

return:

- **dE_mean:** float, average difference between consecutive eigenvalues.

ARPES_lib.**gen_SE_KK**(w, SE_args)

The total self-energy is computed using Kramers' Kronig relations:

The user can pass the self-energy in the form of either a callable function, a list of polynomial coefficients, or as a numpy array with shape Nx2 (with the first column an array of frequency values, and the second the values of a function). For the latter option, the user is responsible for ensuring that the function goes to zero at the tails of the domain. In the former two cases, the 'cut' parameter is used to impose an exponential cutoff near the edge of the domain to ensure this is the case. In all cases the input imaginary self-energy must be single-signed to ensure it is purely even function. It is forced to be negative in all cases to give a positive spectral function. With the input defined, along with the energy range of interest to the calculation, a MUCH larger domain (100x in the maximal extent of the energy region of interest) is defined wf. This is the domain over which we evaluate the Hilbert transform, which itself is carried out using: the `scipy.signal.hilbert()` function. This function acting on an array f : $H(f(x)) \rightarrow f(x) + i Hf(x)$. It relies on the FFT performed on the product of the $\text{sgn}(w)$ and $F(w)$ functions, and then IFFT back so that we can use this to extract the real part of the self energy, given only the input. args:

w – numpy array energy values for the spectral peaks used in the ARPES simulation
 SE_args – dictionary containing the 'imfunc' key value pair (values being either callable, list of polynomial prefactors (increasing order) or numpy array of energy and $\text{Im}(\text{SE})$ values)

– for the first two options, a 'cut' key value pair is also required to force the function to vanish at the boundary of the Hilbert transform integration window.

return: self energy as a numpy array of complex float. The indexing matches that of w, the spectral features to be plotted in the matrix element simulation.

ARPES_lib.**pol_2_sph**(pol)

return polarization vector in spherical harmonics – order being Y_11, Y_10, Y_1-1. If an array of polarization vectors is passed, use the einsum function to broadcast over all vectors.

args:

- **pol:** numpy array of 3 complex float, polarization vector in Cartesian coordinates (x,y,z)

return:

- numpy array of 3 complex float, transformed polarization vector.

ARPES_lib.**poly**(input_x, poly_args)

Recursive polynomial function.

args:

- **input_x:** float, int or numpy array of numeric type, input value(s) at which to evaluate the polynomial

- **poly_args**: list of coefficients, in INCREASING polynomial order i.e. $[a_0, a_1, a_2]$ for $y = a_0 + a_1 * x + a_2 * x ** 2$

return:

- recursive call to *poly*, if *poly_args* is reduced to a single value, return explicit evaluation of the function.

Same datatype as input, with int changed to float if *poly_args* are float, polynomial evaluated over domain of *input_x*

ARPES_lib.**progress_bar** (*N*, *Nmax*)

Utility function, generate string to print matrix element calculation progress.

args:

- **N**: int, number of iterations complete
- **Nmax**: int, total number of iterations to complete

return:

- **st**: string, progress status

ARPES_lib.**projection_map** (*basis*)

In order to improve efficiency, an array of orbital projections is generated, carrying all and each orbital projection for the elements of the model basis. As these do not in general have the same length, the second dimension of this array corresponds to the largest of the sets of projections associated with a given orbital. This will in practice remain a modest number of order 1, since at worst we assume f-orbitals, in which case the projection can be no larger than 7 long. So output will be at worst len(basis)x7 complex float

args:

- **basis**: list of orbital objects

return:

- **projarr**: numpy array of complex float

2.3 Intensity Maps

class intensity_map.**intensity_map** (*index*, *Imat*, *cube*, *kz*, *T*, *hv*, *pol*, *dE*, *dk*, *self_energy=None*, *spin=None*, *rot=0.0*, *notes=None*)

Class for organization and saving of data, as well as metadata related to a specific ARPES calculation.

copy ()

Copy-by-value of the intensity map object.

return:

- *intensity_map* object with identical attributes to self.

save_map (*directory*)

Save the intensity map: if 2D, just a single file, if 3D, each constant-energy slice is saved separately. Saved as .txt file

args:

- **directory**: string, directory for saving intensity map to

return:

- boolean

write_2D_Imat (*filename, index*)

Sub-function for producing the textfiles associated with a 2dimensional numpy array of float

args:

- **filename:** string, indicating destination of file
- **index:** int, energy index of map to save, if -1, then just a 2D map, and save the whole thing

return:

- boolean

write_meta (*destination*)

Write meta-data file for ARPES intensity calculation.

args:

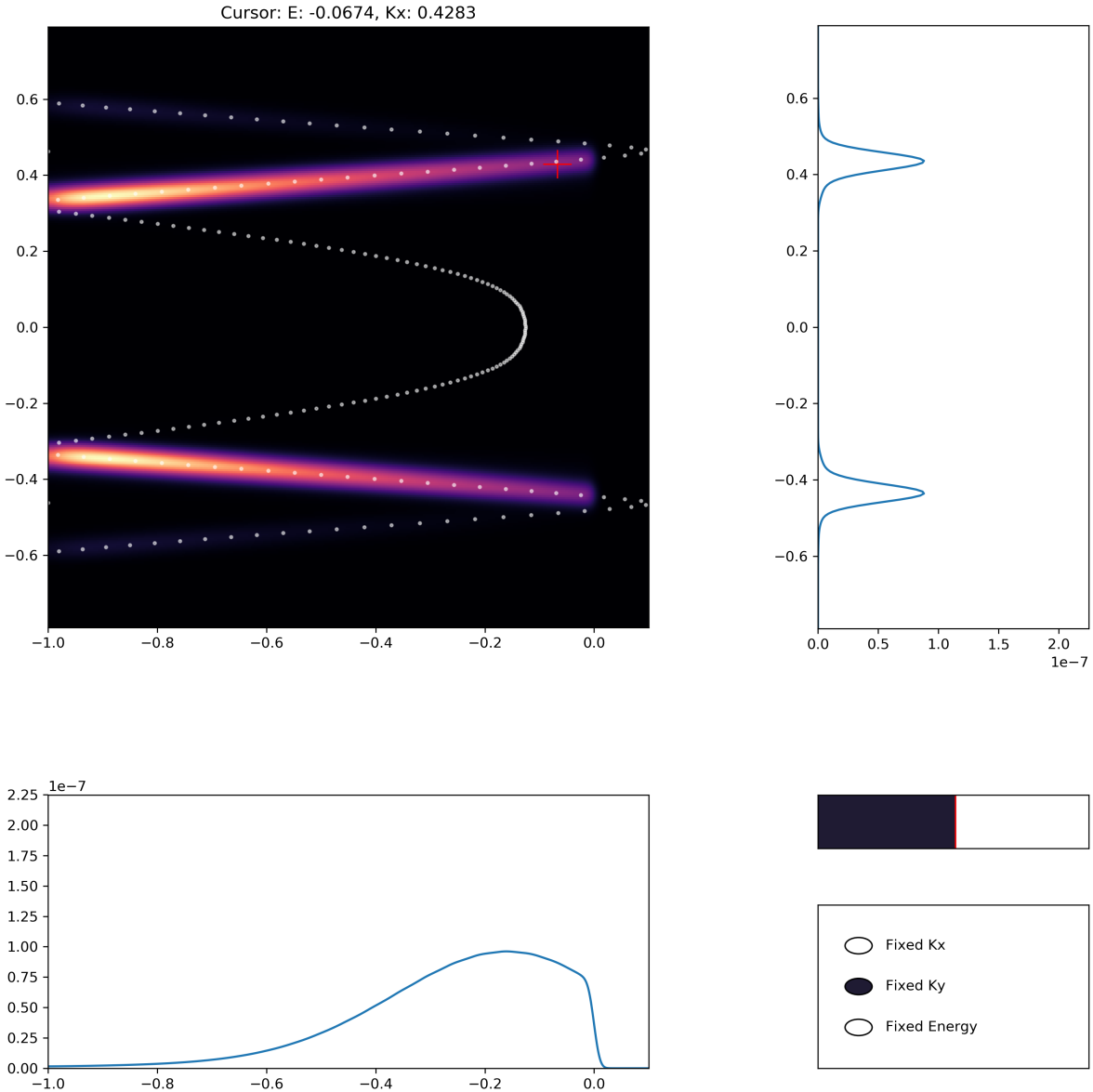
- **destination:** string, file-lead

return:

- boolean

2.4 Matplotlib Plotter

A built-in data-explorer is included in chinook, built using matplotlib to ensure cross platform stability. The figure below shows an example screen capture for a calculation on Sr_2IrO_4 . The user has the ability to scan through the momentum and energy axes of the dataset, and the cursor can be used actively to select momentum- and energy-distribution curves in the side and lower panels. A scatterplot of the bare dispersion, as computed from the Hamiltonian diagonalization is plotted overtop the intensity map.



Created on Thu Sep 12 14:53:36 2019

@author: rday

class matplotlib_plotter.interface(experiment)

This interactive tool is intended for exploring the dataset associated with an ARPES simulation using chinook. The user can scan through the datacube in each dimension. This uses matplotlib natively rather than alternative gui systems in python like Tkinter, which makes it a bit more robust across platforms.

bin_energy ()

Translate the exact energy value for the band peaks into the discrete binning of the intensity map, to allow for cursor queries to be processed.

return:

- **coarse_pts**: numpy array of float, same lengths as *self.state_coords*,

but sampled over a discrete grid.

find_cursor()

Find nearest point to the desired cursor position, as clicked by the user. The cursor event coordinates are compared against the peak positions from the tight-binding calculation, and a best choice within the plotted slice is selected.

args:

- **cursor:** tuple of 2 float, indicating the column and row of the event, in units of the data-set scaling (e.g. 1/Å or eV)

plot_img()

Update the plotted intensity map slice. The plotted bandstructure states are also displayed.

run_gui()

Execution of the matplotlib gui. The figure is initialized, along with all widgets and chosen datasets. The user has access to both the slice of ARPES data plotted, in addition to the orbital projection plotted in upper right panel.

2.5 Tilt Geometry

One can account for rotation of the experimental geometry during acquisition of a particular dataset by performing an inverse rotation on the incoming light vector. Similarly, spin projections can also be rotated into the laboratory frame to reflect the effect of a misaligned or rotated sample in a spin-ARPES experiment.

Created on Fri Dec 28 13:33:59 2018

@author: ryanday

tilt.ang_mesh(N, th, ph)

Generate a mesh over the indicated range of theta and phi, with N elements along each of the two directions

args:

- **N:** int or iterable of length 2 indicating # of points along *th*, and *ph* respectively
- **th:** iterable length 2 of float (endpoints of theta range)
- **ph:** iterable length 2 of float (endpoints of phi range)

return:

- numpy array of N_th x N_ph float, representing mesh of angular coordinates

tilt.gen_kpoints(ek, N, thx, thy, kz)

Generate a mesh of kpoints over a mesh of emission angles.

args:

- **ek:** float, kinetic energy, eV
- **N:** tuple of 2 int, number of points along each axis
- **thx:** tuple of 2 float, range of horizontal angles, radian
- **thy:** tuple of 2 float, range of vertical angles, radian
- **kz:** float, k-perpendicular of interest, inverse Angstrom

return:

- **k_array**: numpy array of $N[1] \times N[0]$ float, corresponding to mesh of in-plane momenta

`tilt.k_mesh` (*Tmesh, Pmesh, ek*)

Application of rotation to a normal-emission vector (i.e. (0,0,1) vector) Third column of a rotation matrix formed by product of rotation about vertical (ky), and rotation around kx axis c.f. Labbook 28 December, 2018

args:

- **Tmesh**: numpy array of float, output of *ang_mesh*
- **Pmesh**: numpy array of float, output of *ang_mesh*
- **ek**: float, kinetic energy in eV

return:

- **kvec**: numpy array of float, in-plane momentum array associated with angular emission coordinates

`tilt.k_parallel` (*ek*)

Convert kinetic energy in eV to inverse Angstrom

`tilt.plot_mesh` (*ek, N, th, ph*)

Plotting tool, plot all points in mesh, for an array of N angles, at a fixed kinetic energy.

args:

- **ek**: float, kinetic energy, eV
- **N**: tuple of 2 int, number of points along each axis
- **thx**: tuple of 2 float, range of horizontal angles, radian
- **thy**: tuple of 2 float, range of vertical angles, radian

`tilt.rot_vector` (*vector, th, ph*)

Rotation of vector by theta and phi angles, about the global y-axis by theta, followed by a rotation about the LOCAL x axis by phi. This is analogous to the rotation of a cryostat with a vertical-rotation axis (theta), and a sample-mount tilt angle (phi). NOTE: need to extend to include cryostats where the horizontal rotation axis is fixed, as opposed to the vertical axis—I have never seen such a system but it is of course totally possible.

args:

- **vector**: numpy array length 3 of float (vector to rotate)
- **th**: float, or numpy array of float – vertical rotation angle(s)
- **ph**: float, or numpy array of float – horizontal tilt angle(s)

return:

- numpy array of float, rotated vectors for all angles: shape 3 x len(ph) x len(th)

NOTE: will flatten any length-one dimensions

The user tight-binding model contains all relevant information regarding the orbital basis, the model Hamiltonian (in addition to eigenvalues/eigenvectors), as well as the momentum domain of interest. In addition, the tight-binding class contains all relevant methods required to extract this information.

The user has a reasonable amount of flexibility in the format which they would use to generate the model Hamiltonian. This is intended to accommodate various programs which can be used to generate a tight-binding Hamiltonian (for example Wannier90), as well as the different presentations used in publications (Slater-Koster V_{lmk} , t_{ij}), as well as alternative Hamiltonian types, such as low-energy effective models which do not adhere to the full translational symmetry required by Bloch's theorem. These latter models do however provide a highly useful and physically transparent parameterization of the system Hamiltonian for narrow regions of momentum space near high-symmetry points. For these reasons, there are 4 general categories of Hamiltonian inputs we accept. The first three are described as: *Slater-Koster*, *list*, and *text* input. The last is described generically as *executable*.

While in principal a tight-binding Hamiltonian can be passed in the acceptable form for any of the above, the last option also supports these low-energy theories described above.

3.1 Slater-Koster Models:

In their 1954 PRB paper, Slater and Koster presented a fairly simple framework for defining the tight-binding model associated with hoppings between localized atomic-like orbitals on neighbouring lattice sites. To define the overlap for a pair of basis states with orbital angular momentum l_1 and l_2 for an arbitrary lattice geometry, we require only $(\min(l_1, l_2) + 1)$ parameters. For example, for $l_1 = l_2 = 1$ we define $V_{pp\sigma}$, $V_{pp\pi}$. Intuitively, these parameters correspond to overlap integrals between the two orbitals when the lobes of the 'p' orbitals are aligned parallel to the connecting vector (σ) and aligned perpendicular to the connecting vector (π). One can often use the frequently published table of Slater-Koster parameters to then define general rules for how these two parameters should be combined for a specific lattice geometry to arrive at the hopping amplitude between each pair of lattice sites.

This table is however restrictive as it provides rules for only hoppings between non-distorted cubic harmonics. One can alternatively take advantage of the representation of the orbital states in the basis of spherical harmonics $Y_l^m(\Omega)$ to rotate an arbitrary pair of basis states into a common representation, and then rotate the frame of reference to align the bond-direction with a designated quantization axis: here the \hat{z} vector. A diagonal Hamiltonian matrix filled with the associated $V_{l_1 l_2 \gamma}$ can then be applied. The rotation and basis transformation can be undone, ultimately producing

a matrix of Hamiltonian parameters for the entire orbital shell along the designated bond vector. Mathematically, the procedure is represented by

$$\langle \psi | H | \phi \rangle = \langle \psi | U^\dagger R^{-1}(\theta, \phi) V_{SK} R(\theta, \phi) U | \phi \rangle$$

This formalism then allows for fast and user-friendly generation of a Hamiltonian over an arbitrary basis and geometry. Given only the $V_{l_1, l_2, \gamma}$ parameters and the lattice geometry, a full tight-binding Hamiltonian can be built.

The Slater-Koster parameters are passed in the form of a dictionary, with the keys taking the form of $a_1 a_2 n_1 n_2 l_1 l_2 \gamma$. For example, if I have two distinct atoms in my basis, where I have the Carbon 2p and Iron 3d e_g orbitals, the dictionary may look like shown here

```
V_SK = {'021':3.0, '13XY':0.0, '13ZR':-0.5,
'002211S':1.0, '002211P':-0.5,
'012312S':0.1, '012312P':0.6,
'113322S':0.05, '113322P':0.7, '113322D':-0.03}
```

In the first line I have written the on-site terms. While in a simple case I may have a single onsite energy for each orbital shell, here I have distinguished the onsite energy of the $d_{x^2-y^2}$ and $d_{3z^2-r^2}$ states. In the second-fourth lines, I have written the p-p, p-d, and d-d hopping amplitudes.

In many models, hopping beyond nearest neighbours are relevant, and will typically not have the same strength as the nearest neighbour terms. In these cases, we can pass a list of dictionaries to H_dict . For example

```
H_dict = {'type':'SK',
          'V':[SK_dict1, SK_dict2],
          'cutoff':[3.5, 5.0],
          ...}
```

To include these next-nearest neighbour terms, I specify a list of hopping dictionaries, in addition to a list of cutoff distances, indicating the range of connecting vectors each of the indicated dictionaries should apply to. For this case, for connecting vectors where $|R_{ij}| < 3.5$ we use SK_dict1 , whereas for $3.5 \leq |R_{ij}| < 5.0$ we use SK_dict2 .

3.2 Hamiltonian Construction:

From these matrices, we proceed to build a list of H_me objects. This class is the standard representation of the Hamiltonian in *chinook*. Each instance of H_me carries integers labelling the associated basis indices. For example $\langle \phi_3 | H | \phi_7 \rangle$ will be stored with $H_me.i = 3$, $H_me.j = 7$. We note here that the Hermiticity of the Hamiltonian allows one to explicitly define only the upper or lower diagonal of the Hamiltonian. Consequently, in *chinook*, we use only the *upper* diagonal, that is $i \leq j$.

In addition to basis indices, the H_me object carries the functional information of the matrix element in its $H_me.H$ attribute. This will be a list of connecting vectors and hopping strengths in the case of a standard tight-binding model, or a list of python executables otherwise.

In standard format then one might have

```
TB.mat_els[9].i = 3
TB.mat_els[9].j = 3
TB.mat_els[9].H = [[0.0, 0.0, 0.0, 2.0],
                  [1.5, 0.0, 0.0, 3.7],
                  [-1.5, 0.0, 0.0, 3.7]]
```

This is the 10-th element in our model's list $TB.mat_els$ of H_me objects. This H_me instance contains an on-site term of strength 2.0, and a cosine-like hopping term along the x -axis of strength 3.7 eV. A closer consideration of the H

attribute reveals the essential information. Each element of the list is a length-4 list of float, containing \vec{R}_{ij}^n and t_{ij}^n . Ultimately, the full matrix element will be expressed as

$$H_{ij}(\vec{k}) = \sum_n t_{ij}^n e^{i\vec{k} \cdot \vec{R}_{ij}^n}$$

For this reason, if one does not have access to a suitable Slater-Koster representation of the tight-binding model, we can bypass the methods described above, passing a list of prepared matrix elements directly to the generator of the *H_me* objects. To accommodate this, we then also accept Hamiltonian input in the form of *list*, where each element of the list is written as for example

```
Hlist[10] = [3,3,1.5,0.0,0.0,3.7]
```

The elements then correspond to basis index 1, basis index 2, connecting vector x, y, and z components, and finally the t_{ij} value. Similarly, a textfile of this form is also acceptable, with each line in the textfile being a comma-separated list of values

```
3,3,1.5,0.0,0.0,3.7
...
```

A list of *H_me* objects can then be built as above.

3.3 Executable Hamiltonians:

The *executable* type Hamiltonian will ultimately require a slightly modified form of the input, as we do not intend to express the matrix elements as a Fourier transform of the real-space Hamiltonian in the same way as above.

In this form, we should have

$$H_{ij}(\vec{k}) = \sum_n f_n(\vec{k})$$

This then requires a modified form of *H_me.H*. By contrast with the above,

```
TB.mat_els[9].H = [np.cos,my_hopping_func]
```

where *my_hopping_func* is a user-defined executable. It could be for example a 2nd order polynomial, or any other function of momentum. The essential point is that the executables *must* take an Nx3 array of float as their only argument. This allows for these Hamiltonians to fit seamlessly into the *chinook* framework.

Warning: Automated tools for building these Hamiltonians are currently in the process of being built. Proceed with caution, and notify the developers of any unexpected performance.

For advanced users who would like to take advantage of this functionality now,

```
H_dict = {'type':'exec',
          'exec':exec_list,
          ...}
```

where the item

```
exec_list = [(0,0),my_func1],
[(0,1),my_func2],
...]
```

The tuple of integers preceding the name of the executable corresponds to the basis indices. Here *my_func1* applies to the diagonal elements of my first basis member, and *my_func2* to the coupling of my first and second basis elements. As always, indexing in python is 0-based.

For the construction of the executable functions, we recommend the use of python's *lambda* functions. For example, to define a function which evaluates to $\alpha k_x^2 + \beta k_y^2$, one may define

```
def my_func_generator(alpha,beta):
    return lambda kvec: alpha*k[:,0]**2 + beta*k[:,1]**2

#Define specific parameters for my executable functions
a1 = 2.3
b1 = 0.7

a2 = 5.6
b2 = 0.9

#Define my executables
my_func1 = my_func_generator(a1,b1)
my_func2 = my_func_generator(a2,b2)

#Populate a list of executables
exec_list = [(0,0),my_func1],
[(0,1),my_func2]]
```

Please stay tuned for further developments which will facilitate more convenient construction of these Hamiltonians.

Below, we include the relevant documentation to the construction of tight-binding models.

3.4 Model Initialization

`build_lib.gen_K(Kdic)`

Generate k-path for TB model to be diagonalized along.

args:

- **Kdic:** dictionary for generation of kpath with:
 - ‘type’: string ‘A’ (absolute) or ‘F’ (fractional) units
 - ‘avec’: numpy array of 3x3 float lattice vectors
 - ‘pts’: list of len3 array indicating the high-symmetry points along the path of interest
 - ‘grain’: int, number of points between *each* element of ‘pts’

optional:

- ‘labels’:list of strings with same length as ‘pts’, giving plotting labels for the kpath

return:

Kobj: K-object including necessary attributes to be read by the **TB_model**

`build_lib.gen_TB(basis_dict, hamiltonian_dict, Kobj=None, slab_dict=None)`

Build a Tight-Binding Model using the user-input dictionaries

args:

- **basis_dict**: dictionary, including the *'bulk'* key value pair

generated by **gen_basis**

- **hamiltonian_dict**: dictionary,
 - *'spin'*: same dictionary as passed to **gen_basis**
 - *'type'*: string, Hamiltonian type–*'list'* (list of matrix elements), *'SK'* (Slater-Koster dictionaries, requires also a *'V'* and *'avec'* entry), *'txt'* (textfile, requires a *'filename'* key as well)
 - *'cutoff'*: float, cutoff hopping distance
 - *'renorm'*: optional float, renormalization factor default to 1.0
 - *'offset'*: optional float, offset of chemical potential, default to 0.0
 - *'tol'*: optional float, minimum matrix element tolerance, default to 1e-15
- **Kobj**: optional, standard K-object, as generated by **gen_K**
- **slab_dict**: dictionary for slab generation
 - *'avec'*: numpy array of 3x3 float, lattice vectors
 - *'miller'*: numpy array of 3 integers, indicating the Miller index of the surface normal in units of lattice vectors
 - *'fine'*: fine adjustment of the slab thickness, tuple of two numeric to get desired termination correct (for e.g. inversion symmetry)
 - *'thick'*: integer approximate number of unit cells in the slab (will not be exact, depending on the fine, and termination)
 - *'vac'*: int size of the vacuum buffer – must be larger than the largest hopping length to ensure no coupling of slabs
 - *'termination'*: tuple of 2 integers: atom indices which terminate the top and bottom of the slab

return:

TB_model: tight-binding object, as defined in **chinook.TB_lib.py**

build_lib.gen_basis (*basis*)

Generate a list of orbital objects as the input basis for a tight-binding model. User passes a basis dictionary, function returns a modified version of this same dictionary, with the list of orbitals now appended as the *'bulk'* entry

args:

- **basis**–dictionary with keys:
 - *'atoms'*: list of integer, indices for distinct atoms,
 - *'Z'*: dictionary of integer: *'atom'*:element (integer) pairs
 - *'orbs'*: list of lists of string, for each atom containing the orbital labels (usually in conventional nlxx format)),
 - *'pos'*: list of numpy arrays of length 3 float indicating

positions of the atoms in direct Angstrom units,

– optional keys:

- * *'orient'*: list, one entry for each atom, indicating a local rotation of the indicated atom, various formats accepted; for more details, c.f. **chinook.orbital.py**
- * *'spin'*: dictionary of spin information:
 - *'bool'*: boolean, double basis into spinor basis,
 - *'soc'*: boolean, include spin-orbit coupling
 - *'lam'*: dictionary of SOC constants, integer:float pairs for atoms in *'atoms'* list, and *lambda_SOC* in eV

return:

- **basis** dictionary, modified to include the **bulk** list of orbital objects

`build_lib.recur_product` (*elements*)

Utility function: Recursive evaluation of the product of all elements in a list

args:

- **elements**: list of numeric type

return:

- product of all elements of **elements**

3.5 Hamiltonian Library

`H_library.AFM_order` (*basis, dS, p_up, p_dn*)

Add antiferromagnetism to the tight-binding model, by adding a different on-site energy to orbitals of different spin character, on the designated sites.

args:

- **basis**: list, orbital objects
- **dS**: float, size of spin-splitting (eV)
- **p_up, p_dn**: numpy array of float indicating the orbital positions

for the AFM order

return:

- **h_AF**: list of matrix elements, as conventionally arranged `[[o1,o2,0,0,0,H12],...]`

`H_library.FM_order` (*basis, dS*)

Add ferromagnetism to the system. Take *dS* to assume that the splitting puts spin-up lower in energy by *dS*, and viceversa for spin-down. This directly modifies the *TB_model*'s **mat_els** attribute

args:

- **basis**: list, of orbital objects in basis
- **dS**: float, energy of the spin splitting (eV)

return:

- list of matrix elements `[[o1,o2,0,0,0,H12],...]`

`H_library.Lm(l)`

L- operator in the l, m_l basis, organized with $(0,0) = |l,l\rangle \dots (2l,2l) = |l,-l\rangle$

The nonzero elements are on the upper diagonal

arg:

- **l**: int orbital angular momentum

return:

- **M**: numpy array $(2l+1, 2l+1)$ of real float

`H_library.Lp(l)`

L+ operator in the l, m_l basis, organized with $(0,0) = |l,l\rangle \dots (2l,2l) = |l,-l\rangle$ The nonzero elements are on the upper diagonal

arg:

- **l**: int orbital angular momentum

return:

- **M**: numpy array $(2l+1, 2l+1)$ of real float

`H_library.Lz(l)`

Lz operator in the l, m_l basis

arg:

- **l**: int orbital angular momentum

return:

- numpy array $(2*l+1, 2*l+1)$

`H_library.SO(basis)`

Generate L.S matrix-elements for a given basis. This is generic to all l , except the normal_order, which is defined here up to and including the f electrons. Otherwise, this method is generic to any orbital angular momentum.

In the factors dictionary defined here indicates the weight of the different $L_i S_i$ terms. The keys are tuples of $(L+/-z, S+/-z)$ in a bit of a cryptic way: for L, $(0,1,2) \rightarrow (-1,0,1)$ and for S, $(-1,0,1) = S1-S2$ with $S1,2 = +/- 1$ here

L+,L-,Lz matrices are defined for each l shell in the basis, transformed into the basis of the tight-binding model. The nonzero terms will then just be used along with the spin and weighted by the factor value, and slotted into a `len(basis)xlen(basis)` matrix **HSO**

args:

- **basis**: list of orbital objects

return:

- **HSO**: list of matrix elements in standard format `[o1,o2,0,0,0,H12]`

`H_library.Vlist_gen(V, pair)`

Select the relevant hopping matrix elements to be used in defining the value of the Slater-Koster matrix elements for a given pair of orbitals. Handles situation where insufficient parameters have been passed to system.

args:

- **V**: dictionary of Slater-Koster hopping terms
- **pair**: tuple of int defining the orbitals to be paired, $(a1,a2,n1,n2,l1,l2)$

return:

- **Vvals**: numpy array of V_{llx} related to a given pairing, e.g. for s-p `np.array([Vsps,Vspp])`

`H_library.cluster_init` (*Vdict, cutoff, avec*)

Generate a cluster of neighbouring lattice points to use in defining the hopping paths—ensuring that it extends sufficiently far enough to capture even the largest hopping vectors. Also reforms the SK dictionary and cutoff lengths to be in list format. Returns an array of lattice points which go safely to the edge of the cutoff range.

args:

- **Vdict**: dictionary, or list of dictionaries of Slater Koster matrix elements
- **cutoff**: float, or list of float
- **avec**: numpy array of 3x3 float

return:

- **Vdict**: list of length 1 if a single dictionary passed, else unmodified
- **cutoff**: numpy array, append 0 to the beginning of the cutoff list, else leave it alone.
- **pts**: numpy array of lattice vector indices for a region of lattice points around the origin.

`H_library.index_ordering` (*basis*)

We use an universal ordering convention for defining the Slater-Koster matrices which may (and most likely will) not match the ordering chosen by the user. To account for this, we define a dictionary which gives the ordering, relative to the normal order convention defined here, associated with a given a-n-l shell at each site in the lattice basis.

args:

- **basis**: list of orbital objects

return:

- **indexing**: dictionary of key-value pairs (a,n,l,x,y,z):numpy.array([...])

`H_library.mat_els` (*Rij, SKmat, tol, i1, i2*)

Extract the pertinent, and non-zero elements of the Slater-Koster matrix and transform to the conventional form of Hamiltonian list entries (o1,o2,Rij0,Rij1,Rij2,H12(Rij))

args:

- **Rij**: numpy array of 3 float, relevant connecting vector
- **SKmat**: numpy array of float, matrix of hopping elements

for the coupling of two orbital shells

- **tol**: float, minimum hopping included in model
- **i1, i2**: int,int, proper index ordering for the relevant

instance of the orbital shells involved in hopping

return:

- **out**: list of Hamiltonian matrix elements, extracted from the ordered SKmat, in form [[o1,o2,x12,y12,z12,H12],...]

`H_library.mirror_SK` (*SK_in*)

Generate a list of values which is the input appended with its mirror reflection. The mirror boundary condition suppresses the duplicate of the last value. e.g. [0,1,2,3,4] -> [0,1,2,3,4,3,2,1,0], ['r','a','c','e','c','a','r'] -> ['r','a','c','e','c','a','r','a','r','a','c','e','c','a','r'] Intended here to take an array of Slater-Koster hopping terms and reflect about its last entry i.e. [Vsps,Vspp] -> [Vsps,Vspp,Vsps]

args:

- **SK_in**: iterable, of arbitrary length and data-type

return:

- list of values with same data-type as input

`H_library.on_site(basis, V, offset)`

On-site matrix element calculation. Try both anl and alabel formats, if neither is defined, default the onsite energy to 0.0 eV

args:

- **basis**: list of orbitals defining the tight-binding basis
- **V**: dictionary, Slater Koster terms
- **offset**: float, EF shift

return:

- **Ho**: list of Hamiltonian matrix elements

`H_library.region(num)`

Generate a symmetric grid of points in number of lattice vectors.

args:

- **num**: int, grid will have size $2*\text{num}+1$ in each direction

return:

- numpy array of size $((2*\text{num}+1)**3,3)$ with centre value of first entry of $(-\text{num}, -\text{num}, -\text{num}), \dots, (0,0,0), \dots, (\text{num}, \text{num}, \text{num})$

`H_library.sk_build(avec, basis, Vdict, cutoff, tol, renorm, offset)`

Build SK model from using D-matrices, rather than a list of SK terms from table. This can handle orbitals of arbitrary orbital angular momentum in principal, but right now implemented for up to and including f-electrons. NOTE: f-hoppings require thorough testing

args:

- **avec**: numpy array 3x3 float, lattice vectors
- **basis**: list of orbital objects
- **Vdict**: dictionary, or list of dictionaries, of Slater-Koster integrals/ on-site energies
- **cutoff**: float or list of float, indicating range where Vdict is applicable
- **tol**: float, threshold value below which hoppings are neglected
- **offset**: float, offset for Fermi level

return:

- **H_raw**: list of Hamiltonian matrix elements, in form $[o1, o2, x12, y12, z12, t12]$

`H_library.spin_double(H, lb)`

Duplicate the kinetic Hamiltonian terms to extend over the spin-duplicated orbitals, which are by construction in same order and appended to end of the original basis.

args:

- **H**: list, Hamiltonian matrix elements $[[o1, o2, x, y, z, H12], \dots]$
- **lb**: int, length of basis before spin duplication

return:

- **h2** modified copy of **H**, filled with kinetic terms for both spin species

`H_library.txt_build(filename, cutoff, renorm, offset, tol, Nonsite)`

Build Hamiltonian from textfile, input is of form $o1, o2, x12, y12, z12, t12$, output in form $[o1, o2, x12, y12, z12, t12]$.

To be explicit, each row of the textfile is used to generate a k-space Hamiltonian matrix element of the form:

$$H_{1,2}(k) = t_{1,2} e^{i(k_x x_{1,2} + k_y y_{1,2} + k_z z_{1,2})}$$

args:

- **filename**: string, name of file
- **cutoff**: float, maximum distance of hopping allowed, Angstrom
- **renorm**: float, renormalization of the bandstructure
- **offset**: float, energy offset of chemical potential, electron volts
- **tol**: float, minimum Hamiltonian matrix element amplitude
- **Nonsite**: int, number of basis states, use to apply **offset**

return:

- **Hlist**: the list of Hamiltonian matrix elements

3.6 Momentum Library

`klib.b_zone(a_vec, N, show=False)`

Generate a mesh of points over the Brillouin zone. Each of the cardinal axes are divided by the same number of points (so points are not necessarily evenly spaced along each axis).

args:

- **a_vec**: numpy array of size 3x3 float
- **N**: int mesh density

kwargs:

- **show**: boolean for optional plotting of the mesh points

return:

- **m_pts**: numpy array of mesh points (float), shape (len(m_pts),3)

`klib.bvectors(a_vec)`

Define the reciprocal lattice vectors corresponding to the direct lattice in real space

args:

- **a_vec**: numpy array of 3x3 float, lattice vectors

return:

- **b_vec**: numpy array of 3x3 float, reciprocal lattice vectors

`klib.kmesh(ang, X, Y, kz, Vo=None, hv=None, W=None)`

Take a mesh of kx and ky with fixed kz and generate a Nx3 array of points which rotates the mesh about the z axis by **ang**. N is the flattened shape of **X** and **Y**.

args:

- **ang**: float, angle of rotation
- **X**: numpy array of float, one coordinate of meshgrid
- **Y**: numpy array of float, second coordinate of meshgrid
- **kz**: float, third dimension of momentum path, fixed

kwargs:

- **Vo**: float, parameter necessary for inclusion of inner potential
- **hv**: float, photon energy, to be used if **Vo** also included, for evaluating kz

- **W**: float, work function

return:

- **k_arr**: numpy array of shape Nx3 float, rotated kpoint array.
- **ph**: numpy array of N float, angles of the in-plane momentum

points, before rotation.

`klib.kmesh_hv(ang, x, y, hv, Vo=None)`

Take a mesh of k_x and k_y with fixed k_z and generate a Nx3 array of points which rotates the mesh about the z axis by **ang**. N is the flattened shape of **X** and **Y**.

args:

- **ang**: float, angle of rotation
- **X**: numpy array of float, one coordinate of meshgrid
- **Y**: numpy array of float, second coordinate of meshgrid
- **kz**: float, third dimension of momentum path, fixed

kwargs:

- **Vo**: float, parameter necessary for inclusion of inner potential
- **hv**: float, photon energy, to be used if **Vo** also included, for evaluating k_z
- **W**: float, work function

return:

- **k_arr**: numpy array of shape Nx3 float, rotated kpoint array.
- **ph**: numpy array of N float, angles of the in-plane momentum

points, before rotation.

`class klib.kpath(pts, grain=None, labels=None)`

Momentum object, defining a path in reciprocal space, for use in defining the Hamiltonian at different points in the Brillouin zone. ***

points()

Use the endpoints of **kpath** defined in **kpath.pts** to create numpy array of len(3) float which cover the entire path, based on method by I.S. Elfimov.

return:

- **kpath.kpts**: numpy array of float, len(**kpath.pts**)(1+****kpath.grain****) by 3

`klib.kz_kpt(hv, kpt, W=0, V=0)`

Extract the k_z associated with a given in-plane momentum, photon energy, work function and inner potential

args:

- **hv**: float, photon energy in eV
- **kpt**: float, in plane momentum, inverse Angstrom
- **W**: float, work function in eV
- **V**: float, inner potential in eV

return:

- **kz**: float, out of plane momentum, inverse Angstrom

`klib.mesh_reduce (blatt, mesh, inds=False)`

Determine and select only k-points corresponding to the first Brillouin zone, by simply classifying points on the basis of whether or not the closest lattice point is the origin. By construction, the origin is index 13 of the `blatt`. If it is not, return error. Option to take only the indices of the mesh which we want, rather than the actual array points –this is relevant for tetrahedral interpolation methods

args:

- **blatt**: numpy array of len(27,3), nearest reciprocal lattice vector points
- **mesh**: numpy array of (N,3) float, defining a mesh of k points, before

being reduced to contain only the Brillouin zone.

kwargs:

- **inds**: option to pass a list of bool, indicating the indices one wants to keep, instead of autogenerating the mesh

return:

- **bz_pts**: numpy array of (M,3) float, Brillouin zone points

`klib.plt_pts (pts)`

Plot an array of points in 3D

args:

- **pts**: numpy array shape N x 3

`klib.raw_mesh (blatt, N)`

Define a mesh of points filling the region of k-space bounded by the set of reciprocal lattice points generated by *bvectors*. These will be further reduced by *mesh_reduce* to find points which are within the first-Brillouin zone

args:

- **blatt**: numpy array of 27x3 float
- **N**: int, or iterable of len 3, defines a coarse estimation

of number of k-points

return:

- **mesh**: numpy array of mesh points, size set roughly by N

`klib.region (num)`

Generate a symmetric grid of points in number of lattice vectors.

args:

- **num**: int, grid will have size 2 num+1 in each direction

return:

- numpy array of size ((2 num+1)**3,3) with centre value of first entry of (-num,-num,-num),..., (0,0,0),..., (num,num,num)

3.7 Orbital Objects

`orbital.fact (n)`

Recursive factorial function, works for any non-negative integer.

args:

- **n**: int, or integer-float

return:

- int, recursively evaluates the factorial of the initial input value.

class `orbital.orbital` (*atom, index, label, pos, Z, orient=[0.0], spin=1, lam=0.0, sigma=1.0, slab_index=None*)

The **orbital** object carries all essential details of the elements of the model Hamiltonian basis, for both generation of the tight-binding model, in addition to the evaluation of expectation values and ARPES intensity.

copy ()

Copy by value method for orbital object

return:

- **orbital_copy**: duplicate of **orbital** object

`orbital.rot_projection` (*l, proj, rotation*)

Go through a projection array, and apply the intended transformation to the Ylm projections in order. Define Euler angles in the z-y-z convention THIS WILL BE A COUNTERCLOCKWISE ROTATION ABOUT a vector BY angle gamma expressed in radians. Note that we always define spin in the lab-frame, so spin degrees of freedom are not rotated when we rotate the orbital degrees of freedom.

args:

- **l**: int,orbital angular momentum
- **proj**: numpy array of shape Nx4 of float, each element is [Re(projection),Im(projection),l,m]
- **rotation**: float, or list, defining rotation of orbital. If float, assume rotation about z-axis. If list, first element is a numpy array of len 3, indicating rotation vector, and second element is float, angle.

return:

- **proj**: numpy array of Mx4 float, as above, but modified, and may now include additional, or fewer elements than input *proj*.
- **Dmat**: numpy array of (2l+1)x(2l+1) complex float indicating the Wigner Big-D matrix associated with the rotation of this orbital shell about the intended axis.

`orbital.slab_basis_copy` (*basis, new_posns, new_inds*)

Copy elements of a slab basis into a new list of orbitals, with modified positions and index ordering.

args:

- **basis**: list or orbital objects
- **new_posns**: numpy array of len(basis)x3 float, new positions for orbital
- **new_inds**: numpy array of len(basis) int, new indices for orbitals

return:

- **new_basis**: list of duplicated orbitals following modification.

`orbital.sort_basis` (*basis, slab*)

Utility script for organizing an orbital basis that is out of sequence

args:

- **basis**: list of orbital objects
- **slab**: bool, True or False if this is for sorting a slab

return:

- **orb_basis**: list of sorted orbital objects (by orbital.index value)

`orbital.spin_double(basis, lamdict)`

Double the size of a basis to introduce spin to the problem. Go through the basis and create an identical copy with opposite spin and incremented index such that the orbital basis order puts spin down in the first N/2 orbitals, and spin up in the second N/2.

args:

- **basis**: list of orbital objects
- **lamdict**: dictionary of int:float pairs providing the spin-orbit coupling strength for the different inequivalent atoms in basis.

return:

- doubled basis carrying all required spin information

3.8 Slater Koster Library

`SlaterKoster.SK_cub(Ymats, l1, l2)`

In order to generate a set of independent Lambda functions for rapid generation of Hamiltonian matrix elements, one must nest the definition of the lambda functions within another function. In this way, we avoid cross-contamination of unrelated functions. The variables which are fixed for a given lambda function are the cubic -to- spherical harmonics (Ymat) transformations, and the orbital angular momentum of the relevant basis channels. The output lambda functions will be functions of the Euler-angles pertaining to the hopping path, as well as the potential matrix V, which will be passed as a numpy array (min(l1,l2)*2+1) long of float.

We follow the method described for rotated d-orbitals in the thesis of JM Carter from Toronto (HY Kee), where the Slater-Koster hopping matrix can be defined as the following operation:

1. Transform local orbital basis into spherical harmonics
2. Rotate the hopping path along the z-axis
3. Product with the diagonal SK-matrix
4. Rotate the path backwards
5. Rotate back into basis of local orbitals
6. Output matrix of hopping elements between all orbitals in the shell to fill Hamiltonian

args:

- **Ymats**: list of numpy arrays corresponding to the relevant

transformation from cubic to spherical harmonic basis

- **l1, l2**: int orbital angular momentum channels relevant

to a given hopping pair

return:

- lambda function for the SK-matrix between these orbital shells, for arbitrary hopping strength and direction.

SlaterKoster.**SK_full** (*basis*)

Generate a dictionary of lambda functions which take as keys the atom,orbital for both first and second element. Formatting is a1a2n1n2l1l2, same as for SK dictionary entries

args:

- **basis**: list of orbital objects composing the TB-basis

return:

- **SK_funcs**: a dictionary of hopping matrix functions

(lambda functions with args EA,EB,Ey,V as Euler angles and potential (V)) which can be executed for various hopping paths and potential strengths The keys of the dictionary will be organized similar to the way the SK parameters are passed, labelled by a1a2n1n2l1l2, which completely defines a given orbital-orbital coupling

SlaterKoster.**Vmat** (*l1, l2, V*)

For Slater-Koster matrix element generation, a potential matrix is sandwiched in between the two bond-rotating Dmatrices. It should be of the shape $2 \times l_1 + 1 \times 2 \times l_2 + 1$, and have the $V_{l,l}$, D terms along the ‘diagonal’ – a concept that is only well defined for a square matrix. For mismatched angular momentum channels, this turns into a diagonal square matrix of dimension $\min(2 \times l_1 + 1, 2 \times l_2 + 1)$ centred along the larger axis. For channels where the orbital angular momentum change involves a change in parity, the potential should change sign, as per Slater Koster’s original definition from 1954. This is taken care of automatically in the Wigner formalism I use here, no need to have exceptions

args:

- **l1, l2**: int orbital angular momentum of initial and final states
- **V**: numpy array of float – length should be $\min(l_1, l_2) \times 2 + 1$

return:

- **Vm**: numpy array of float, shape $2 \times l_1 + 1 \times 2 \times l_2 + 1$

3.9 Tight-Binding Library

class TB_lib.**H_me** (*i, j, executable=False*)

This class contains the relevant executables and data structure pertaining to generation of the Hamiltonian matrix elements for a single set of coupled basis orbitals. Its attributes include integer values **i, j** indicating the basis indices, and a list of hopping vectors/matrix element values for the Hamiltonian.

The method **H2Hk** provides an executable function of momentum to allow broadcasting of the Hamiltonian over a large array of momenta. Python’s flexible protocol for equivalency and passing variables by reference/value require definition of a copy operator which allows one to produce safely, a copy of the object rather than its coordinates in memory alone. ***

H2Hk ()

Transform the list of hopping elements into a Fourier-series expansion of the Hamiltonian. This is run during diagonalization for each matrix element index. If running a low-energy Hamiltonian, executable functions are simply summed for each basis index **i, j**, rather than computing a Fourier series. **x** is implicitly a numpy array of $N \times 3$: it is essential that the executable conform to this input type.

return:

- lambda function of a numpy array of float of length 3

append_H (*H, R0=0, R1=0, R2=0*)

Add a new hopping path to the coupling of the parent orbitals.

args:

- **H**: complex float, matrix element strength, or if self.exectype, should be an executable
- **R0, R1, R2**: float connecting vector in cartesian coordinate frame—this is the TOTAL vector, not the relevant lattice vectors only

return:

- directly modifies the Hamiltonian list for these matrix coordinates

clean_H()

Remove all duplicate instances of hopping elements in the matrix element list. This function is run automatically during slab generation.

The Hamiltonian list is not itself directly modified.

return:

- list of hopping vectors and associated Hamiltonian matrix element strengths

copy()

Copy by value of the **H_me** object

return:

- **H_copy**: duplicate **H_me** object

class `TB_lib.TB_model` (*basis, H_args, Kobj=None*)

The **TB_model** object carries the model basis as a list of **orbital** objects, as well as the model Hamiltonian, as a list of **H_me**. The orbital indices paired in each instance of **H_me** are stored in a dictionary under **ijpairs**

append_H (*new_elements*)

Add new terms to the Hamiltonian by hand. This directly modifies the list of Hamiltonian matrix element, self.mat_els of the TB object.

args:

- **new_elements**: list of Hamiltonian matrix elements, either a single element `[i,j,x_ij,y_ij,z_ij,H_ij(x,y,z)]` or as a list of such lists. Here i, j are the related orbital-indices.

build_ham (*H_args*)

Buld the Hamiltonian using functions from **chinook.H_library.py**

args:

- **H_args**: dictionary, containing all relevant information for defining the Hamiltonian list. For details, see **TB_model.__init__**.

return:

- sorted list of matrix element objects. These objects have

i,j attributes referencing the orbital basis indices, and a list of form $[R0,R1,R2,Re(H)+1.0jIm(H)]$

copy()

Copy by value of the **TB_model** object

return:

- **TB_copy**: duplicate of the **TB_model** object.

plot_unitcell (*ax=None*)

Utility script for visualizing the lattice and orbital basis. Distinct atoms are drawn in different colours

kwargs:

- **ax**: matplotlib Axes, for plotting on existing Axes

return:

- **ax**: matplotlib Axes, for further modifications to plot

plotting (*win_min=None, win_max=None, ax=None*)

Plotting routine for a tight-binding model evaluated over some path in k . If the model has not yet been diagonalized, it is done automatically before proceeding.

kwargs:

- **win_min, win_max**: float, vertical axis limits for plotting

in units of eV. If not passed, a reasonable choice is made which covers the entire eigenspectrum.

- **ax**: matplotlib Axes, for plotting on existing Axes

return:

- **ax**: matplotlib axes object

print_basis_summary()

Very basic print function for printing a summary of the orbital basis, including their label, atomic species, position and spin character.

solve_H (*Eonly=False*)

This function diagonalizes the Hamiltonian over an array of momentum vectors. It uses the **mat_el** objects to quickly define lambda functions of momentum, which are then filled into the array and diagonalized. According to <https://ntrs.nasa.gov/archive/nasa/casi.ntrs.nasa.gov/20050192421.pdf> SVD algorithms require memory of $2 * \text{order} * (4 * \text{order} + 1) \sim 8 * \text{order}^2$. The matrices are complex float, so this should be 16 bytes per entry: so $\text{len}(k) * (2048 * \text{order}^2)$. If the diagonalization is requesting more than 85% of the available memory, then split up the k -path into sequential diagonalizations.

return:

- **self.Eband**: numpy array of float, shape($\text{len}(\text{self.Kobj.kpts}), \text{len}(\text{self.basis})$),

eigenvalues

- **self.Evec**: numpy array of complex float, shape($\text{len}(\text{self.Kobj.kpts}), \text{len}(\text{self.basis}), \text{len}(\text{self.basis})$)

eigenvectors

unpack ()

Reduce a Hamiltonian object down to a list of matrix elements. Include the Hermitian conjugate terms

return:

- **Hlist**: list of Hamiltonian matrix elements

TB_lib.**atom_coords** (*basis*)

Define a dictionary organizing the distinct coordinates of instances of each atomic species in the basis

args:

- **basis**: list of orbital objects

return:

- ******dictionary with integer keys, numpy array of float values. atom:locations are encoded in this way

TB_lib.**cell_edges** (*avec*)

Define set of line segments which enclose the unit cell.

args:

- **avec**: numpy array of 3x3 float

return:

- **edges**: numpy array of 12 x 6, endpoints of the 12 edges of the unit cell parallelepiped

TB_lib.**gen_H_obj** (*htmp*, *executable=False*)

Take a list of Hamiltonian matrix elements in list format: [i,j,Rij[0],Rij[1],Rij[2],Hij(R)] and generate a list of **H_me** objects instead. This collects all related matrix elements for a given orbital-pair for convenient generation of the matrix Hamiltonians over an input array of momentum

args:

- **htmp**: list of numeric-type values (mixed integer[:2], float[2:5], complex-float[-1])

kwargs:

- **executable**: boolean, if True, we don't have a standard Fourier-type Hamiltonian, but perhaps a low-energy expansion. In this case, the htmp elements are

return:

- **Hlist**: list of Hamiltonian matrix element, **H_me** objects

chinook is designed with the capacity to do fairly extensive characterization of the tight-binding model being used. These scripts contain useful tools for understanding the model in more detail

4.1 Density of States

Several different approaches can be taken to the execution of density of states. One can follow [Blöchl](#), partitioning the Brillouin zone into a cubic mesh, with each cube further divided into a group of identical tetrahedra. This approach provides the ability to perform matrix diagonalization over a fairly small number of k-points, as one interpolates within the tetrahedra to construct the full density of states. This is executed using *dos_tetra* defined below.

Alternatively, one can perform diagonalization explicitly only at the nodes of a mesh defined over the Brillouin zone, and apply some broadened peak at the eigenvalues of the Hamiltonian at each point. Generally, for reasonably narrow Gaussian broadening this requires a fairly dense k-mesh. In practice, the large supercells where diagonalization becomes costly are also associated with much smaller Brillouin zones, allowing for a smaller k-space sampling. We find this method to perform better in most cases. This is executed using the *dos_broad* function defined below.

Related tools are also available to find the Fermi level, given a specified electronic occupation (*dos.find_EF* using gaussian, and *dos.EF_find* using tetrahedra).

```
dos.band_contribution (eigenvals, w_domain, volume)
```

Compute the contribution over a single tetrahedron, from a single band, to the density of states

args:

- **eigenvals**: numpy array of float, energy values at corners
- **w_domain**: numpy array of float, energy domain
- **volume**: int, number of tetrahedra in the total mesh

return:

- **DOS**: numpy array of float, same length as w_domain

```
dos.def_dE (Eband)
```

If energy broadening is not passed for density-of-states calculation, compute a reasonable value based on the energy between adjacent energies in the tight-binding calculation

args:

- **Eband**: numpy array of float, indicating band energies

return:

dE: float, energy broadening, as the smallest average energy spacing over all bands.

dos.**dos_broad**(*TB, NK, NE=None, dE=None, origin=array([0., 0., 0.])*)

Energy-broadened discrete density of states calculation. The Hamiltonian is diagonalized over the kmesh defined by NK and states are summed, as energy-broadened Gaussian peaks, rather than delta functions.

args:

- **TB**: tight-binding model object
- **NK**: int, or tuple of int, indicating number of k-points

kwargs:

- **NE**: int, number of energy bins for final output
- **dE**: float, energy broadening of peaks, eV
- **origin**: numpy array of 3 float, indicating the origin of the mesh to be used, relevant for example in kz-specific contributions to density of states

return:

- **DOS**: numpy array of float, density-of-states in states/eV
- **Elin**: numpy array of float, energy domain in eV

dos.**dos_func**(*energy, epars*)

Piecewise function for calculation of density of states

args:

- **energy**: numpy array of float (energy domain)
- **epars**: tuple of parameters: e[0],e[1],e[2],e[3],V_T,V_G being the ranked band energies for the tetrahedron,

as well as the volume of both the tetrahedron and the Brillouin zone, all float

return:

- numpy array of float giving DOS contribution from this tetrahedron

dos.**dos_tetra**(*TB, NE, NK*)

Generate a tetrahedra mesh of k-points which span the BZ with even distribution Diagonalize over this mesh and then compute the resulting density of states as prescribed in the above paper. The result is plotted, and DOS returned

args:

- **TB**: tight-binding model object
- **NE**: int, number of energy points
- **NK**: int or list of 3 int – number of k-points in mesh

return:

- **Elin**: linear energy array of float, spanning the range of the eigenspectrum
- **DOS**: numpy array of float, same length as Elin, density of states

dos.**error_function**(*x0, x, sigma*)

Integral over the gaussian function, evaluated from -infinity to x, using the scipy implementation of the error function

args:

- **x0**: float, centre of Gaussian, in eV

- **x**: numpy array of float, energy domain eV
- **sigma**: float, width of Gaussian, in eV

return:

- analytical form of integral

`dos.find_EF_broad_dos (TB, NK, occ, NE=None, dE=None, origin=array([0., 0., 0.]))`

Find the Fermi level of a model Hamiltonian, for a designated electronic occupation. Note this is evaluated at $T=0$, so EF is well-defined.

args:

- **TB**: tight-binding model object
- **NK**: int, or tuple of int, indicating number of k-points
- **occ**: float, desired electronic occupation

kwargs:

- **NE**: int, number of energy bins for final output
- **dE**: float, energy spacing of bins, in eV
- **origin**: numpy array of 3 float, indicating the origin of the mesh to be used, relevant for example in kz-specific contributions to density of states

return:

- **EF**: float, Fermi level in eV

`dos.find_EF_tetra_dos (TB, occ, dE, NK)`

Use the tetrahedron-integration method to establish the Fermi-level, for a given electron occupation.

args:

- **TB**: instance of tight-binding model object from *TB_lib*
- **occ**: float, desired electronic occupation
- **dE**: estimate of energy precision desired for evaluation of the Fermi-level (in eV)
- **NK**: int or iterable of 3 int, number of k points in mesh.

return:

EF: float, Fermi Energy for the desired occupation, to within dE of actual value.

`dos.gaussian (x0, x, sigma)`

Evaluate a normalized Gaussian function.

args:

- **x0**: float, centre of peak, in eV
- **x**: numpy array of float, energy domain in eV
- **sigma**: float, width of Gaussian, in eV

return:

- numpy array of float, gaussian evaluated.

`dos.n_func (energy, epars)`

Piecewise function for evaluating contribution of tetrahedra to electronic occupation number

args:

- **energy**: numpy array of float, energy domain

- **epars**: tuple of parameters: `e[0],e[1],e[2],e[3],V_T,V_G` being the ranked band energies for the tetrahedron,

as well as the volume of both the tetrahedron and the Brillouin zone, all float

return:

- numpy array of float, same length as **energy**, providing contribution of tetrahedra to the occupation function

`dos.n_tetra (TB, dE, NK, plot=True)`

This function, also from the algorithm of Blochl, gives the integrated DOS at every given energy (so from bottom of bandstructure up to its top. This makes for very convenient and precise evaluation of the Fermi level, given an electron number)

args:

- **TB**: tight-binding model object
- **dE**: float, energy spacing (meV)
- **NK**: int, iterable of 3 int. number of k-points in mesh
- **plot**: bool, to plot or not to plot the calculated array

return:

- **Elin**: linear energy array of float, spanning the range of the eigenspectrum
- **n_elect**: numpy array of float, same length as **Elin**, integrated DOS

at each energy, i.e. total number of electrons occupied at each energy

`dos.ne_broad_analytical (TB, NK, NE=None, dE=None, origin=array([0., 0., 0.]), plot=True)`

Analytical evaluation of the occupation function. Uses scipy's errorfunction executable to evaluate the analytical form of a Gaussian-broadened state's contribution to the total occupation, at each energy

args:

- **TB**: tight-binding model object
- **NK**: int, or tuple of int, indicating number of k-points

kwargs:

- **NE**: int, number of energy bins for final output
- **dE**: float, energy spacing of bins, in eV
- **origin**: numpy array of 3 float, indicating the origin of the mesh to be used, relevant for example in kz-specific contributions to density of states
- **plot**: boolean, default to True, if false, suppress plot output

return:

- **nE**: numpy array of float, occupied states
- **Elin**: numpy array of float, energy domain in eV

`dos.ne_broad_numerical (TB, NK, NE=None, dE=None, origin=array([0., 0., 0.]))`

Occupation function, as a numerical integral over the density of states function.

args:

- **TB**: tight-binding model object
- **NK**: int, or tuple of int, indicating number of k-points

kwargs:

- **NE**: int, number of energy bins for final output
- **dE**: float, energy spacing of bins, in eV

- **origin**: numpy array of 3 float, indicating the origin of the mesh to be used, relevant for example in kz-specific contributions to density of states
- return*:
- **ne**: numpy array of float, integrated density-of-states at each energy
 - **Elin**: numpy array of float, energy domain in eV
- ***

`dos.pdos_tetra (TB, NE, NK, proj)`

Partial density of states calculation. Follows same tetrahedra method, weighting the contribution of a given tetrahedra by the average projection onto the indicated user-defined projection. The average here taken as the sum over projection at the 4 vertices of the tetrahedra.

args:

- **TB**: tight-binding model object
- **NE**: int, number of energy bins
- **NK**: int, or iterable of 3 int, indicating the number of k-points

along each of the axes of the Brillouin zone

- **proj**: numpy array of float, 1D or 2D, c.f. *proj_mat*.

return:

- **Elin**: numpy array of float, with length **NE**, spanning the range of the tight-binding bandstructure
- **pDOS**: numpy array of float, len **NE**, projected density of states
- **DOS**: numpy array of float, len **NE**, full density of states

`dos.proj_avg (eivecs, proj_matrix)`

Calculate the expectation value of the projection operator, for each of the eigenvectors, at each of the vertices, and then sum over the vertices. We use *numpy.einsum* to perform matrix multiplication and contraction.

args:

- **eivecs**: numpy array of complex float, 4xNxM, with M number of eigenvectors, N basis dimension
- **proj_matrix**: numpy array of complex float, NxN in size

return:

- numpy array of M float, indicating the average projection over the 4 corners of the tetrahedron

`dos.proj_mat (proj, lenbasis)`

Define projection matrix for fast evaluation of the partial density of states weighting. As the projector here is diagonal, and represents a Hermitian matrix, it is by definition a real matrix operator.

args:

- **proj**: numpy array, either 1D (indices of projection), or 2D (indices of projection and weight of projection)
- **lenbasis**: int, size of the orbital basis

return:

- **projector**: numpy array of float, lenbasis x lenbasis

4.2 Fermi Surface

We use a modified version of the method of [marching tetrahedra](#) to find the Fermi surface within the reciprocal lattice. The standard definition of the reciprocal space mesh runs over the primitive parallelepiped defined by the reciprocal lattice vectors. Shifts to the lattice origin are provided as an option. The loci of the Fermi surface are found for each tetrahedra, and used to assemble a continuous set of triangular patches which ultimately construct the Fermi surface for each band which crosses the Fermi level.

`FS_tetra.EF_tetra` (*TB, NK, EF, degen=False, origin=None*)

Generate a tetrahedra mesh of k-points which span the BZ with even distribution Diagonalize over this mesh and then compute the resulting density of states as prescribed in the above paper.

args:

- **TB**: tight-binding model object
- **NK**: int or list,tuple of 3 int indicating number of k-points in mesh
- **EF**: float, Fermi energy, or energy of interest

kwargs:

- **deg**: bool, flag for whether the bands are two-fold degenerate, as for Kramers degeneracy
- **origin**: numpy array of 3 float, corresponding to the desired centre of the plotted Brillouin zone

return:

- **surfaces**: dictionary of dictionaries: Each key-value pair corresponds to a different band index. For each case, the value is a dictionary with key-value pairs:
 - *'pts'*: numpy array of Nx3 float, the N coordinates of EF crossing
 - *'tris'*: numpy array of Nx3 int, the triangulation of the surface

`FS_tetra.FS_generate` (*TB, Nk, EF, degen=False, origin=None, ax=None*)

Wrapper function for computing Fermi surface triangulation, and then plotting the result.

args:

- **TB**: tight-binding model object
- **Nk**: int, or tuple/list of 3 int, number of k-points in Brillouin zone mesh
- **EF**: float, Fermi energy, or constant energy level of interest

kwargs:

- **deg**: bool, flag for whether the bands are two-fold degenerate, as for Kramers degeneracy
- **origin**: numpy array of 3 float, corresponding to the desired centre of the plotted Brillouin zone
- **ax**: matplotlib Axes, option for plotting onto existing Axes

return:

- **surfaces**: dictionary of dictionaries: Each key-value pair corresponds to a different band index. For each case, the value is a dictionary with key-value pairs:
 - *'pts'*: numpy array of Nx3 float, the N coordinates of EF crossing
 - *'tris'*: numpy array of Nx3 int, the triangulation of the surface
- **ax**: matplotlib Axes, for further modification

`FS_tetra.fermi_surface_2D(TB, npts=100, kfix=(2, 0), energy=0, shift=array([0, 0, 0]), do_plot=True)`

Generate a 2D contour of the Fermi surface, projected into one of the 3 cardinal planes. User specifies which b-vector to be normal to, and its fixed value. The user also specifies the 'Fermi' energy, and can shift the centre of the plot away from the origin if desired.

args:

- **TB:** tight-binding model object
- **npts:** int or tuple of 2-int, number of kpoints in grid
- **kfix:** tuple of two numeric. First is b-vector index (0-base), second is the fixed value (float)
- **energy:** float, Fermi energy
- **shift:** numpy array of 3 float, shift vector, in units or b-vectors
- **do_plot:** boolean, option to suppress plot and only return the FS contours

returns:

- **ax:** if `do_plot`, then a figure is generated and the axes object returned
- **FS:** if not `do_plot`, then a dictionary of contours, with keys indicating the associated band index, and values being the arrays of K points is returned

`FS_tetra.get_kpts(TB, kfix, npts=100, shift=array([0, 0, 0]))`

Get k-grid for Brillouin zone sampling

args:

- **TB:** tight-binding model object
- **kfix:** tuple of two numeric. First is b-vector index (0-base), second is the fixed value (float)
- **npts:** int or tuple of 2-int, number of kpoints in grid
- **shift:** numpy array of 3 float, shift vector, in units or b-vectors

`FS_tetra.heron(vert)`

Heron's algorithm for calculation of triangle area, defined by only the vertices

args:

- **vert:** numpy array of 3x3 indicating vertices of triangle

return:

- float, area of triangle

`FS_tetra.sim_tri(vert)`

Take 4 vertices of a quadrilateral and split into two alternative triangulations of the corners. Return the vertices of the triangulation which has the more similar areas between the two triangles decomposed.

args:

- **vert:** (4 by 3 numpy array (or list) of float) in some coordinate frame

return:

- **tris[0], tris[1]:** two numpy arrays of size 3 by 3 float containing the coordinates of a triangulation

4.3 Operators

A number of tools are included for characterizing the expectation values of observables, as projected onto the eigenstates of the model Hamiltonian. The main function here is *O_path*, which will compute the expectation value of an indicated operator (represented by the user as a Hermitian numpy array of complex float with the same dimensions as the orbital basis). The resulting values are then displayed in the form of a colourmap applied to the bandstructure calculation along the desired path in momentum space. Several common operators are defined to facilitate these calculations, such as spin \hat{S}_i and $\langle \vec{L} \cdot \vec{S} \rangle$. In addition, *fatbs* uses *O_path* to produce a “fat bands” type spaghetti plot, where the colourscale reflects the orbital projection onto the bandstructure.

Created on Mon Mar 23 20:08:46 2020

@author: ryanday

`operator_library.FS(TB, ktuple, Ef, tol, ax=None)`

A simplified form of Fermi surface extraction, for proper calculation of this, *chinook.FS_tetra.py* is preferred. This finds all points in kmesh within a tolerance of the constant energy level.

args:

- **TB**: tight-binding model object
- **ktuple**: tuple of k limits, len (3), First and second should be iterable, define the limits and mesh of k for kx,ky, the third is constant, float for kz
- **Ef**: float, energy of interest, eV
- **tol**: float, energy tolerance, float
- **ax**: matplotlib Axes, option for plotting onto existing Axes

return:

- **pts**: numpy array of len(N) x 3 indicating x,y, band index
- **TB.Eband**: numpy array of float, energy spectrum
- **TB.Evec**: numpy array of complex float, eigenvectors
- **ax**: matplotlib Axes, for further user modification

`operator_library.LSmat(TB, axis=None)`

Generate an arbitrary L.S type matrix for a given basis. Uses same *Yproj* as the *HSO* in the *chinook.H_library*, but is otherwise different, as it supports projection onto specific axes, in addition to the full vector dot product operator.

Otherwise, the LiSi matrix is computed with i the axis index. To do this, a linear combination of L+S+,L-S-,L+S-,L-S+,LzSz terms are used to compute.

In the factors dictionary, the weight of these terms is defined. The keys are tuples of (L+/-/z,S+/-/z) in a bit of a cryptic way. For L, range (0,1,2) ->(-1,0,1) and for S range (-1,0,1) = S1-S2 with S1/2 = +/- 1 here

L+,L-,Lz matrices are defined for each l shell in the basis, transformed into the basis of cubic harmonics. The nonzero terms will then just be used along with the spin and weighted by the factor value, and slotted into a len(basis)xlen(basis) matrix HSO

args:

- **TB**: tight-binding object, as defined in TB_lib.py
- **axis**: axis for calculation as either 'x','y','z',None, or float (angle in the x-y plane)

return:

- **HSO**: (len(basis)xlen(basis)) numpy array of complex float

`operator_library.Ldots(TB, axis=None, ax=None, colourbar=True)`

Wrapper for `O_path` for computing L.S along a vector projection of interest, or none at all.

args:

- **TB**: tight-binding object

kwargs:

- **axis**: numpy array of 3 float, indicating axis, or None for full L.S
- **ax**: matplotlib.Axes object for plotting
- **colourbar**: bool, display colourbar on plot

return:

- **O**: numpy array of $N \times \text{len}(\text{basis})$ float, expectation value of operator

on each band over the kpath of TB.Kobj.

`operator_library.Lm(l)`

L- operator in the l, m_l basis, organized with $(0,0) = |l, l\rangle$, $(2*1, 2*1) = |l, -l\rangle$ The nonzero elements are on the upper diagonal

arg:

- **l**: int orbital angular momentum

return:

- **M**: numpy array $(2*1+1, 2*1+1)$ of real float

`operator_library.Lp(l)`

L+ operator in the l, m_l basis, organized with $(0,0) = |l, l\rangle$, $(2*1, 2*1) = |l, -l\rangle$ The nonzero elements are on the upper diagonal

arg:

- **l**: int orbital angular momentum

return:

- **M**: numpy array $(2*1+1, 2*1+1)$ of real float

`operator_library.Lz(l)`

Lz operator in the l, m_l basis

arg:

- **l**: int orbital angular momentum

return:

- numpy array $(2*1+1, 2*1+1)$

`operator_library.O_path(Operator, TB, Kobj=None, vlims=None, Elims=None, degen=False, plot=True, ax=None, colourbar=True, colourmap=None)`

Compute and plot the expectation value of an user-defined operator along a k-path Option of summing over degenerate bands (for e.g. fat bands) with degen boolean flag

args:

- **Operator**: matrix representation of the operator (numpy array $\text{len}(\text{basis}), \text{len}(\text{basis})$ of complex float)
- **TB**: Tight binding object from TB_lib

kwargs:

- **Kobj**: Momentum object, as defined in *chinook.klib.py*
- **vlims**: tuple of 2 float, limits of the colourscale for plotting,
if default value passed, will compute a reasonable range
- **Elims**: tuple of 2 float, limits of vertical scale for plotting

- **plot**: bool, default to True, plot, or not plot the result
- **degen**: bool, True if bands are degenerate, sum over adjacent bands
- **ax**: matplotlib Axes, option for plotting onto existing Axes
- **colourbar**: bool, plot colorbar on axes, default to True
- **colourmap**: matplotlib colourmap, i.e. LinearSegmentedColormap

return:

- **O_vals**: the numpy array of float, (len Kobj x len basis) expectation values
- **ax**: matplotlib Axes, allowing for user to further modify

`operator_library.O_surf (O, TB, ktuple, Ef, tol, vlims=(0, 0), ax=None)`

Compute and plot the expectation value of an user-defined operator over a surface of constant-binding energy

Option of summing over degenerate bands (for e.g. fat bands) with degen boolean flag

args:

- **O**: matrix representation of the operator (numpy array len(basis), len(basis) of complex float)
- **TB**: Tight binding object from *chinook.TB_lib.py*
- **ktuple: momentum range for mesh**: ktuple[0] = (x0,xn,n), ktuple[1]=(y0,yn,n), ktuple[2]=kz

kwargs:

- **vlims**: limits for the colourscale (optional argument), will choose

a reasonable set of limits if none passed by user

- **ax**: matplotlib Axes, option for plotting onto existing Axes

return:

- **pts**: the numpy array of expectation values, of shape Nx3, with first two dimensions the kx,ky coordinates of the point, and the third the expectation value.
- **ax**: matplotlib Axes, allowing for further user modifications

`operator_library.S_vec (LB, vec)`

Spin operator along an arbitrary direction can be written as $n.S = n_x S_x + n_y S_y + n_z S_z$

args:

- **LB**: int, length of basis
- **vec**: numpy array of 3 float, direction of spin projection

return:

- numpy array of complex float (LB by LB), spin operator matrix

`operator_library.Sz (TB, ax=None, colourbar=True)`

Wrapper for **O_path** for computing Sz along a vector projection of interest, or none at all.

args:

- **TB**: tight-binding object

kwargs:

- **ax**: matplotlib.Axes plotting object
- **colourbar**: bool, display colourbar on plot

return:

- **O**: numpy array of Nxlen(basis) float, expectation value of operator on each band over the kpath of TB.Kobj.

`operator_library.colourmaps()`

Plot utility, define a few colourmaps which scale to transparent at their zero values

`operator_library.degen_Ovals(Oper_exp, Energy)`

In the presence of degeneracy, we want to average over the evaluated orbital expectation values—numerically, the degenerate subspace can be arbitrarily diagonalized during `numpy.linalg.eigh`. All degeneracies are found, and the expectation values averaged.

args:

- **Oper_exp**: numpy array of float, operator expectations
- **Energy**: numpy array of float, energy eigenvalues.

`operator_library.fatbs(proj, TB, Kobj=None, vlims=None, Elims=None, degen=False, ax=None, colourbar=True, plot=True)`

Fat band projections. User denotes which orbital index projection is of interest. Projection passed either as an Nx1 or Nx2 array of float. If Nx2, first column is the indices of the desired orbitals, the second column is the weight. If Nx1, then the weights are all taken to be equal

args:

- **proj**: iterable of projections, to be passed as either a 1-dimensional with indices of projection only, OR, 2-dimensional, with the second column giving the amplitude of projection (for linear-combination projection)
- **TB**: tight-binding object

kwargs:

- **Kobj**: Momentum object, as defined in `chinook.klib.py`
- **vlims**: tuple of 2 float, limits of the colorscale for plotting, default to (0,1)
- **Elims**: tuple of 2 float, limits of vertical scale for plotting
- **plot**: bool, default to True, plot, or not plot the result
- **degen**: bool, True if bands are degenerate, sum over adjacent bands
- **ax**: matplotlib Axes, option for plotting onto existing Axes
- **colorbar**: bool, plot colorbar on axes, default to True

return:

- **Ovals**: numpy array of float, `len(Kobj.kpts)*len(TB.basis)`

`operator_library.is_numeric(a)`

Quick check if object is numeric

args:

- **a**: numeric, float/int

return:

- bool, if numeric True, else False

`operator_library.operator_projected_fermi_surface(TB, matrix, npts=100, kfix=(2, 0), energy=0, shift=array([0, 0, 0]), degen=True, fig=None, cmap=<matplotlib.colors.LinearSegmentedColormap object>, scale=20)`

Simple 2D-projected FS with operator expectation values plot over the FS contours.

args:

- **TB**: tight-binding object
- **matrix**: numpy array of complex float, operator matrix
- **npts**: number of k-points along axes of BZ

- **kfix**: fixed index of BZ. First value is projected reciprocal lattice vector (0,1,2), second is value (inverse Angstrom)
- **energy**: float, fixed value of energy (EF = 0)
- **shift**: numpy array of 3 float. Shift of centre of plot
- **degen**: boolean, average over degenerate bands
- **fig**: matplotlib figure to plot on top of
- **cmap**: colourmap
- **scale**: multiplier for scatterplot point sizes

`operator_library.surface_proj(basis, length)`

Operator for computing surface-projection of eigenstates. User passes the orbital basis and an extinction length (1/e) length for the 'projection onto surface'. The operator is diagonal with exponential suppression based on depth.

For use with SLAB geometry only

args:

- **basis**: list, orbital objects
- **cutoff**: float, cutoff length

return:

- **M**: numpy array of float, shape len(TB.basis) x len(TB.basis)

4.4 Orbital Visualization

`orbital_plotting.col_phase(vals)`

Define the phase of a complex number

args:

- **vals**: complex float, or numpy array of complex float

return:

- float, or numpy array of float of same shape as vals, from -pi to pi

`orbital_plotting.make_angle_mesh(n)`

Quick utility function for generating an angular mesh over spherical surface

args:

- **n**: int, number of divisions of the angular space

return:

- **th**: numpy array of 2n float from 0 to pi
- **ph**: numpy array of 4n float from 0 to 2pi

`orbital_plotting.rephase_wavefunctions(vecs, index=-1)`

The wavefunction at different k-points can choose an arbitrary phase, as can a subspace of degenerate eigenstates. As such, it is often advisable to choose a global phase definition when comparing several different vectors. The user here passes a set of vectors, and they are rephased. The user has the option of specifying which basis index they would like to set the phasing. It is essential however that the projection onto at least one basis element is non-zero over the entire set of vectors for this rephasing to work.

args:

- **vecs**: numpy array of complex float, ordered as rows:vector index, columns: basis index

kwargs:

- **index:** int, optional choice of basis phase selection

return:

- **rephase:** numpy array of complex float of same shape as *vecs*

class orbital_plotting.**wavefunction** (*basis, vector*)

This class acts to reorganize basis and wavefunction information in a more suitable data structure than the native orbital class, or the sake of plotting orbital wavefunctions. The relevant eigenvector can be redefined, so long as it represents a projection onto the same orbital basis set as defined previously.

args:

- **basis:** list of orbital objects
- **vector:** numpy array of complex float, eigenvector projected onto the basis orbitals

calc_Ylm (*th, ph*)

Calculate all spherical harmonics needed for present calculation

return:

- numpy array of complex float, of shape (len(self.harmonics),len(th))

find_centres ()

Create a Pointer array of basis indices and the centres of these basis orbitals.

return:

- **all_centres:** list of numpy array of length 3, indicating unique positions in the basis set
- **centre_pointers:** list of int, indicating the indices of position array, associated with the location of the related orbital in real space.

find_harmonics ()

Create a pointer array of basis indices and the associated spherical harmonics, as well as a more convenient vector form of the projections themselves, as lists of complex float

return:

- **all_lm:** list of int, l,m pairs of all spherical harmonics relevant to calculation
- **lm_pointers:** list of int, pointer indices relating each basis orbital projection to the lm pairs in *all_lm*
- **projectors:** list of arrays of complex float, providing the complex projection of basis onto the related spherical harmonics

plot_wavefunction (*vertices, triangulations, colours, plot_ax=None, cbar_ax=None*)

Plotting function, for visualizing orbitals.

args:

- **vertices:** numpy array of float, shape (len(centres), len(th)*len(ph), 3) locations of vertices
- **triangulations:** numpy array of int, indicating the vertices connecting each surface patch
- **colours:** numpy array of float, of shape (len(centres),len(triangles)) encoding the orbital phase for each surface patch of the plotting
- **plot_ax:** matplotlib Axes, for plotting on existing axes

- **cbar_ax**: matplotlib Axes, for use in drawing colourbar

return:

- **plots**: list of plotted surfaces
- **plot_ax**: matplotlib Axes, for further modifications

redefine_vector (*vector*)

Update vector definition

args:

- **vector**: numpy array of complex float, same length as self.vector

triangulate_wavefunction (*n*, *plotting=True*, *ax=None*)

Plot the wavefunction stored in the class attributes as self.vector as a projection over the basis of spherical harmonics. The radial wavefunctions are not explicitly included, in the event of multiple basis atom sites, the length scale is set by the mean interatomic distance. The wavefunction phase is encoded in the colourscale of the mesh plot. The user sets the smoothness of the orbital projection by the integer argument *n*

args:

- **n**: int, number of angles in the mesh: Theta from 0 to pi is divided 2n times, and Phi from 0 to 2pi is divided 4n times

kwargs:

- **plotting**: boolean, turn on/off to display plot
- **ax**: matplotlib Axes, for plotting on existing plot

return:

- **vertices**: numpy array of float, shape (len(centres), len(th)*len(ph), 3) locations of vertices
- **triangulations**: numpy array of int, indicating the vertices connecting each surface patch
- **colours**: numpy array of float, of shape (len(centres),len(triangles)) encoding the orbital phase for each surface patch of the plotting
- **ax**: matplotlib Axes, for further modifications

4.5 tetrahedra

Created on Tue Sep 4 16:27:40 2018

@author: rday

`tetrahedra.corners()`

Establish the shortest main diagonal of a cube of points, so as to establish the main diagonal for tetrahedral partitioning of the cube

return:

- **main**: tuple of 2 integers indicating the cube coordinates
- **cube**: numpy array of 8 corners (8x3) float

`tetrahedra.gen_mesh(avec, N)`

Generate a mesh of points in 3-dimensional momentum space over the first Brillouin zone. These are defined first in terms of reciprocal lattice vectors,

i.e. from 0->1 along each, and then are multiplied by the rec. latt. vectors themselves. Note that this implicitly provides a mesh which is not centred at zero, but has an origin at the rec. latt. vector (0,0,0)

args:

- **avec**: numpy array of 3x3 float, lattice vectors
- **N**: int, or tuple of 3 int, indicating the number of points along

each of the reciprocal lattice vectors

`tetrahedra.mesh_tetra(avec, N)`

An equivalent definition of a spanning grid over the Brillouin zone is just one which spans the reciprocal cell unit cell. Translational symmetry imposes that this partitioning is equivalent to the conventional definition of the Brillouin zone, with the very big advantage that we can define a rectilinear grid which spans this volume in a way which can not be done for most Bravais lattices in R3.

args:

- **avec**: numpy array of 3x3 float, lattice vectors
- **N**: int, or iterable of 3 int which define the density of the mesh

over the Brillouin zone.

return:

- **pts**: numpy array of Mx3 float, indicating the points in momentum space at the vertices of the mesh
- **mesh_tet**: numpy array of Lx4 int, indicating the L-tetrahedra which partition the grid

`tetrahedra.neighbours(point)`

For an unit cube, we can define the set of 3 nearest neighbours by performing the requisite modular sum along one of the three Cartesian axes. In this way, for an input point, we can extract its neighbours easily.

args:

- **point**: numpy array of 3 int, all either 0 or 1

return:

- numpy array of 3x3 int, indicating the neighbours of **point** on the unit cube.

`tetrahedra.not_point(point)`

Inverse of point, defined in an N-dimensional binary coordinate frame

args:

- **point**: int or numpy array of int between 0 and 1

return:

- numpy array of int, NOT gate applied to the binary vector point

`tetrahedra.propagate(i, Nr, Nc)`

Distribute the generic corner numbering convention defined for a cube at the origin to a cube starting at some arbitrary point in our grid. Excludes the edge points as starting points, so that all cubes are within the grid.

args:

- **i**: int, index of origin
- **Nr**: int, number of rows in grid
- **Nc**: int, number of columns in grid

return:

- ******numpy array of int, len 8 corresponding to the re-numbering of the corners of the cube.

`tetrahedra.tet_inds()`

Generate, for a single cube, the tetrahedral designations, for the following conventional numbering:

6 o — o 7 // |

4 o — o 5 o 3

| /

0 o — o 1

with 2 hidden from view (below 6, and behind the line-segment connecting 4-5). Here drawn with x along horizontal, z into plane, y vertical. Defining the real-index spacing between adjacent cubes in a larger array, we can apply this simple prescription to define the spanning tetrahedra over the larger k-mesh

return:

- **tetra_inds**: numpy array of integer (6x4), with each row containing the index of the 4 tetrahedral vertices. Together, for a set of neighbouring points on a grid, we divide into a set of covering tetrahedra which span the volume of the cube.

`tetrahedra.tetrahedra()`

Perform partitioning of a cube into tetrahedra. The indices can then be dotted with some basis vector set to put them into the proper coordinate frame.

return:

- **tetra**: numpy array of 6 x 4 x 3 int, indicating the corners of the 6 tetrahedra

Slab Calculation

chinook facilitates the generation of slab-models based around bulk Hamiltonians defined by the user. This functionality is in beta-testing mode for version 1.0. so please proceed with caution, and contact the developers if you have any concerns.

The setup for a slab-type calculation proceeds similarly to that for a bulk model. The call to build a tight-binding model passes an additional argument,

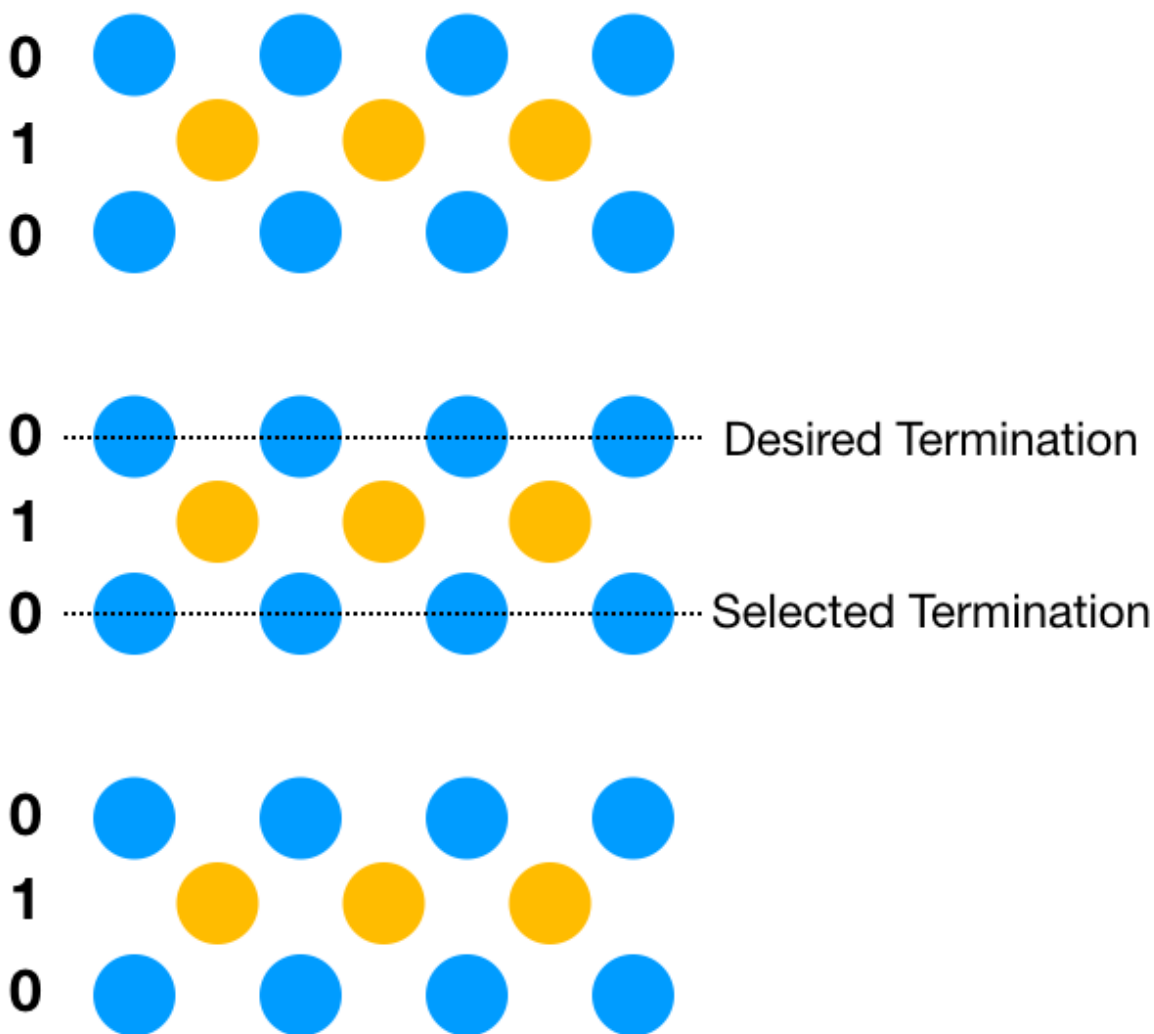
```
TB = chinook.build_lib.gen_TB(basis_dict,hamiltonian_dict,K_object,slab_dict)
```

The *slab_dict* formats as follows.

```
slab_dict = { 'avec':avec,
              'miller':np.array([0,0,1]),
              'thick':30,
              'vac':20,
              'termination':(0,0),
              'fine':(0,0)}
```

Here one passes the lattice vectors *avec*, along with the desired Miller index of the surface termination. A desired thickness of both the slab and the vacuum buffer are also required. These are in units of Angstrom. This defines a lower bound: to find a suitable slab which satisfies the desired atomic termination, the actual slab can often be larger. The *vac* should be at very least longer than the farthest hopping vector considered in the model. This doesn't add any computational overhead *after* the definition of the slab model is established, so a large value is not a serious bottleneck in subsequent calculations.

The *termination* tuple designates which inequivalent atoms in the basis one wants to have on the top and bottom surface. If a mixed-character surface (e.g. TiO₂ plane) is desired, either species can be selected. Finally the *fine* tuple allows user to adjust the position of the termination, which can be necessary in the event of incorrect software selection of surface. This can occur commonly in layered materials, for example below for a AB₂ material: while a real material will generally terminate at the bottom of the van-der Waals layer, either instance of atom 0 can satisfy the *slab_dict* input. This is indicated in the figure:



Fine allows the user to request a shift of designated thickness to force the program to select the proper surface termination. As *slab* is in beta development, some diagnostics of the generated slab should be conducted by the user before proceeding with more complex calculations.

Warning: Slab generation will rotate the global coordinate frame to place the surface normal along the \hat{z} direction, and one of the in-plane vectors along the \hat{x} direction. This may lead to unanticipated redirection of the high-symmetry points.

5.1 Slab Library

`slab.GCD(a, b)`

Basic greatest common denominator function. First find all divisors of each a and b . Then find the maximal common element of their divisors.

args:

- **a, b:** int

return:

- int, GCD of **a**, **b**

`slab.H_conj(h)`

Conjugate hopping path

args:

- **h**: list, input hopping path in format [i,j,x,y,z,Hij]

return:

- list, reversed hopping path, swapped indices, complex conjugate of the hopping strength

`slab.H_surf(surf_basis, avec, H_bulk, Rmat, lenbasis)`

Rewrite the bulk-Hamiltonian in terms of the surface unit cell, with its (most likely expanded) basis. The idea here is to organize all ‘duplicate’ orbitals, in terms of their various connecting vectors. Using modular arithmetic, we then create an organized dictionary which categorizes the hopping paths within the new unit cell according to the new basis index designation. For each element in the Hamiltonian then, we can do the same modular definition of the hopping vector, easily determining which orbital in our new basis this hopping path indeed corresponds to. We then make a new list, organizing corresponding to the new basis listing.

args:

- **surf_basis**: list of orbitals in the surface unit cell
- **avec**: numpy array 3x3 of float, surface unit cell vectors
- **H_bulk**: *H_me* object(defined in *chinook.TB_lib.py*), as

the bulk-Hamiltonian

- **Rmat**: 3x3 numpy array of float, rotation matrix

(pre-multiply vectors) for rotating the coordinate system from bulk to surface unit cell axes

- **lenbasis**: int, length of bulk basis

return:

- Hamiltonian object, written in the basis of the surface unit cell,

and its coordinate frame, rather than those of the bulk system

`slab.Hobj_to_dict(Hobj, basis)`

Associate a list of matrix elements with each orbital in the original basis. The hopping paths are given not as direct units, but as number of unit-vectors for each hopping path. So the actual hopping path will be:

`np.dot(H[2:5],svec)+TB.basis[j].pos-TB.basis[i].pos`

This facilitates determining easily which basis element we are dealing with. For the slab, the new supercell will be extended along the 001 direction. So to redefine the orbital indices for a given element, we just take `[i, len(basis)*(R_2)+j, (np.dot((R_0,R_1,R_2),svec)+pos[j]-pos[i]),H]` If the path goes into the vacuum buffer don’t add it to the new list!

args:

- **Hobj**: *H_me* object(defined in *chinook.TB_lib.py*), as

the bulk-Hamiltonian

- **basis**: list of *orbital* objects

return:

- **Hdict**: dictionary of hopping paths associated with a given orbital

index

`slab.LCM(a, b)`

Basic lowest-common multiplier for two values a,b. Based on idea that LCM is just the product of the two input, divided by their greatest common denominator.

args:

- **a, b:** int

return:

- int, LCM of **a** and **b**

`slab.LCM_3(a, b, c)`

For generating spanning vectors, require lowest common multiple of 3 integers, itself just the LCM of one of the numbers, and the LCM of the other two.

args:

- **a, b, c:** int

return:

int, LCM of the three numbers

`slab.abs_to_frac(avec, vec)`

Quick function for taking a row-ordered matrix of lattice vectors:

a_11 a_12 a_13 |

a_21 a_22 a_23 |

a_31 a_32 a_33 |

and using it to transform a vector, written in absolute units, to fractional units. Note this function can be used to broadcast over N vectors you would like to transform

args:

- **avec:** numpy array of 3x3 float lattice vectors, ordered by rows
- **vec:** numpy array of Nx3 float, vectors to be transformed to

fractional coordinates

return:

- Nx3 array of float, vectors translated into basis of lattice vectors

`slab.basal_plane(vvecs)`

Everything is most convenient if we redefine the basal plane of the surface normal to be oriented within a Cartesian plane. To do so, we take the v-vectors. We get the norm of v1,v2 and then find the cross product with the z-axis, as well as the angle between these two vectors. We can then rotate the surface normal onto the z-axis. In this way we conveniently re-orient the v1,v2 axes into the Cartesian x,y plane.

args:

- **vvecs:** numpy array 3x3 float

return:

- **vvec_prime:** numpy array 3x3 of float, rotated v vectors
- **Rmat:** numpy array of 3x3 float, rotation matrix to send original coordinate frame into the rotated coordinates.

`slab.build_slab_H(Hsurf, slab_basis, surf_basis, svec)`

Build a slab Hamiltonian, having already defined the surface-unit cell Hamiltonian and basis. Begin by creating a dictionary corresponding to the Hamiltonian matrix elements associated with the relevant surface unit cell

orbital which pairs with our slab orbital, and all its possible hoppings in the original surface unit cell. This dictionary conveniently redefines the hopping paths in units of lattice vectors between the relevant orbitals. In this way, we can easily relabel a matrix element by the slab_basis elements, and then translate the connecting vector in terms of the pertinent orbitals.

If the resulting element is from the lower diagonal, take its conjugate. Finally, only if the result is physical, i.e. corresponds to a hopping path contained in the slab, and not e.g. extending into the vacuum, should the matrix element be included in the new Hamiltonian. Finally, the new list Hnew is made into a Hamiltonian object, as always, and duplicates are removed.

args:

- **Hsurf**: H_{me} object (defined in *chinook.TB_lib.py*), as the bulk-Hamiltonian from the surface unit cell
- **slab_basis**: list of orbital objects, slab unit cell basis
- **surf_basis**: list of orbital objects, surface unit cell basis
- **svec**: numpy array of 3x3 float, surface unit cell lattice vectors

return:

- list of Hamiltonian matrix elements in [i,j,x,y,z,Hij] format

`slab.bulk_to_slab(slab_dict)`

Wrapper function for generating a slab tight-binding model, having established a bulk model.

args:

- **slab_dict**: dictionary containing all essential information

regarding the slab construction:

- **'miller'**: numpy array len 3 of int, miller indices
- **'TB'**: Tight-binding model corresponding to the bulk model
- **'fine'**: tuple of 2 float. Fine adjustment of the slab limits,

beyond the termination to precisely indicate the termination. units of Angstrom, relative to the bottom, and top surface generated

- **'thick'**: float, minimum thickness of the slab structure
- **'vac'**: float, minimum thickness of the slab vacuum buffer

to properly generate a surface with possible surface states

- **'termination'**: tuple of 2 int, specifying the basis indices

for the top and bottom of the slab structure

return:

- **slab_TB**: tight-binding TB object containing the slab basis
- **slab_ham**: Hamiltonian object, slab Hamiltonian
- **Rmat**: numpy array of 3x3 float, rotation matrix

`slab.divisors(a)`

Iterate through all integer divisors of integer input

args:

- **a**: int

return:

list of int, divisors of **a**

slab.**frac_inside** (*points, avec*)

Use fractional coordinates to determine whether a point is inside the new unit cell, or not. This is a very simple way of establishing this point, and circumvents many of the awkward rounding issues of the parallelepiped method I have used previously. Ultimately however, imprecision of the matrix multiplication and inversion result in some rounding error which must be corrected for. To do this, the fractional coordinates are rounded to the 4th digit. This leads to a smaller uncertainty by over an order to 10^3 than each rounding done on the direct coordinates.

args:

- **points**: numpy array of float (Nx4) indicating positions and basis indices of the points to consider
- **avec**: numpy array of 3x3 float, new lattice vectors

return:

- numpy array of Mx4 float, indicating positions and basis indices of the valid basis elements inside the new

unit cell.

slab.**frac_to_abs** (*avec, vec*)

Same as `abs_to_frac`, but in opposite direction, from fractional to absolute coordinates

args:

- **avec**: numpy array of 3x3 float, lattice vectors, row-ordered
- **vec**: numpy array of Nx3 float, input vectors

return:

- N x 3 array of float, vec in units of absolute coordinates (Angstrom)

slab.**gen_slab** (*basis, vn, mint, minb, term, fine=(0, 0)*)

Using the new basis defined for the surface unit cell, generate a slab of at least `mint` (minimum thickness), `minb` (minimum buffer) and terminated by orbital term. In principal the termination should be same on both top and bottom to avoid inversion symmetry breaking between the two lattice terminations. In certain cases, `mint`, `minb` may need to be tuned somewhat to get exactly the surface terminations you want.

args:

- **basis**: list of instances of orbital objects
- **vn**: numpy array of 3x3 float, surface unit cell lattice vectors
- **mint**: float, minimum thickness of the slab, in Angstrom
- **minb**: float, minimum thickness of the vacuum buffer, in Angstrom
- **term**: tuple of 2 int, termination of the slab tuple (`term[0]` = top termination, `term[1]` = bottom termination)
- **fine**: tuple of 2 float, fine adjustment of the termination to precisely specify terminating atoms

return:

- **avec**: numpy array of float 3x3, updated lattice vector for the SLAB unit cell
- **new_basis**: array of new orbital basis objects, with slab-index corresponding to the original basis indexing,

and primary index corresponding to the order within the new slab basis

slab.**gen_surface** (*avec, miller, basis*)

Construct the surface unit cell, to then be propagated along the 001 direction to form a slab

args:

- **avec**: numpy array of 3x3 float, lattice vectors for original unit cell
- **miller**: numpy array of 3 int, Miller indices indicating the surface orientation
- **basis**: list of orbital objects, orbital basis for the original lattice

return:

- **new_basis**: list of orbitals, surface unit cell orbital basis

- **vn_b**: numpy array of 3x3 float, the surface unit cell primitive lattice vectors
- **Rmat**: numpy array of 3x3 float, rotation matrix, to be used in post-multiplication order

`slab.iszero(a)`

Find where an iterable of numeric is zero, returns empty list if none found

args:

- **a**: numpy array of numeric

return:

- list of int, indices of iterable where value is zero

`slab.mod_dict(surf_basis, av_i)`

Define dictionary establishing connection between slab basis elements and the bulk Hamiltonian. The `slab_indices` relate to the bulk model, we can then compile a list of *slab* orbital pairs (along with their connecting vectors) which should be related to a given bulk model hopping. The hopping is expressed in terms of the number of surface lattice vectors, rather than direct units of Angstrom.

args:

- **surf_basis**: list of orbital objects, covering the slab model
- **av_i**: numpy array of 3x3 float, inverse of the lattice vector matrix

return:

- **cv_dict**: dictionary with key-value pairs of
`slab_index[i]-slab_index[j]:numpy.array([i,j,mod_vec]...)`

`slab.nonzero(a)`

Find where an iterable of numeric is non-zero, returns empty list if none found

args:

- **a**: numpy array of numeric

return:

- list of int, indices of iterable where value is non-zero

`slab.p_vecs(miller, avec)`

Produce the vectors *p*, as defined by Ceder, to be used in defining spanning vectors for plane normal to the Miller axis

args:

- **miller**: numpy array of len 3 float
- **avec**: numpy array of size 3x3 of float

return:

- **pvecs**: numpy array size 3x3 of float

`slab.par(avec)`

Definition of the parallelepiped, as well as a containing region within the Cartesian projection of this form which can then be used to guarantee correct definition of the new cell basis. The parallelepiped is generated, and then its extremal coordinates established, from which a containing parallelepiped is then defined.

args:

- **avec**: numpy array of 3x3 float

return:

- **vert**: numpy array 8x3 float vertices of parallelepiped
- **box_pts**: numpy array 8 x 3 float vertices of containing box

`slab.populate_box` (*box, basis, avec, R*)

Populate the bounding box with points from the original lattice basis. These represent candidate orbitals to populate the surface-projected unit cell.

args:

- **box**: numpy array of 8x3 float, vertices of corner of a box
- **basis**: list of orbital objects
- **avec**: numpy array of 3x3 float, lattice vectors
- **R**: numpy array of 3x3 float, rotation matrix

return:

- **basis_full**: list of Nx4 float, representing instances of orbitals copies, retaining only their position and their orbital basis index. These orbitals fill a container box larger than the region of interest.

`slab.populate_par` (*points, avec*)

Fill the box with basis points, keeping only those which reside in the new unit cell.

args:

- **points**: numpy array of Nx4 float ([:3] give position, [3] gives index)
- **avec**: numpy array of 3x3 float

return:

- **new_points**: Nx3 numpy array of float, coordinates of new orbitals
- **indices**: Nx1 numpy array of float, indices in original basis

`slab.region` (*num*)

Generate a symmetric grid of points in number of lattice vectors.

args:

- **num**: int, grid will have size 2 num+1 in each direction

return:

- numpy array of size ((2 num+1)^3,3) with centre value of first entry of (-num,-num,-num),..., (0,0,0),..., (num,num,num)

`slab.sorted_basis` (*pts, inds*)

Re-order the elements of the new basis, with preference to z-position followed by the original indexing

args:

- **pts**: numpy array of Nx3 float, orbital basis positions
- **inds**: numpy array of N int, indices of orbitals, from original basis

return:

- **labels_sorted**: numpy array of Nx4 float, [x,y,z,index], in order of increasing z, and index

`slab.unpack` (*Ham_obj*)

Reduce a Hamiltonian object down to a list of matrix elements. Include the Hermitian conjugate terms

args:

- **Ham_obj**: Hamiltonian object, c.f. *chinook.TB_lib.H_me*

return:

- **Hlist**: list of Hamiltonian matrix elements

`slab.v_vecs` (*miller, avec*)

Wrapper for functions used to determine the vectors used to define the new, surface unit cell.

args:

- **miller**: numpy array of 3 int, Miller indices for surface normal
- **avec**: numpy array of 3x3 float, Lattice vectors

return:

- **vvecs**: new surface unit cell vectors numpy array of 3x3 float

5.2 Surface Vector

`surface_vector.ang_v1v2(v1, v2)`

Find angle between two vectors:

args:

- **v1**: numpy array of 3 float
- **v2**: numpy array of 3 float

return:

- float, angle between the vectors

`surface_vector.are_parallel(v1, v2)`

Determine if two vectors are parallel:

args:

- **v1**: numpy array of 3 float
- **v2**: numpy array of 3 float

return:

- boolean, True if parallel to within 1e-5 radians

`surface_vector.are_same(v1, v2)`

Determine if two vectors are identical

args:

- **v1**: numpy array of 3 float
- **v2**: numpy array of 3 float

return:

- boolean, True if the two vectors are parallel and have same length, both to within 1e-5

`surface_vector.find_v3(v1, v2, avec, maxlen)`

Find the best out-of-plane surface unit cell vector. While we initialize with a fixed cutoff for maximum length, to avoid endless searching, we can slowly increase on each iteration until a good choice is possible.

args:

- **v1, v2**: numpy array of 3 float, in plane spanning vectors
- **avec**: numpy array of 3x3 float, bulk lattice vectors
- **maxlen**: float, max length tolerated for the vector we seek

return:

- **v3_choice**: the chosen unit cell vector

`surface_vector.initialize_search(v1, v2, avec)`

Seed search for v3 with the nearest-neighbouring Bravais lattice point which maximizes the projection out of plane of that spanned by v1 and v2.

args:

- **v1, v2:** numpy array of 3 float, the spanning vectors for plane
- **avec:** numpy array of 3x3 float

return:

- numpy array of 3 float, the nearby Bravais lattice point which maximizes the projection along the plane normal

`surface_vector.refine_search(v3i, v1, v2, avec, maxlen)`

Refine the search for the optimal v3-supercell lattice vector which both minimizes its length, while maximizing orthogonality with v1 and v2

args:

- **v3i:** numpy array of 3 float, initial guess for v3
- **v1:** numpy array of 3 float, in-plane supercell vector
- **v2:** numpy array of 3 float, in-plane supercell vector
- **avec:** numpy array of 3x3 float, bulk lattice vectors
- **maxlen:** float, upper limit on how long of a third vector we can reasonably tolerate. This becomes relevant for unusual Miller indices.

return:

- **v3_opt** list of numpy array of 3 float, list of viable options for the out of plane surface unit cell vector

`surface_vector.score(vlist, v1, v2, avec)`

To select the ideal out-of-plane surface unit cell vector, score the candidates based on both their length and their orthogonality with respect to the two in-plane spanning vectors. The lowest scoring candidate is selected as the ideal choice.

args:

- **vlist:** list of len 3 numpy array of float, choices for out-of-plane vector
- **v1, v2:** numpy array of 3 float, in plane spanning vectors
- **avec:** numpy array of 3x3 float, primitive unit cell vectors

return:

- numpy array of len 3, out of plane surface-projected lattice vector

`v3find.ang_v1v2(v1, v2)`

Find angle between two vectors, rounded to floating point precision.

args:

- **v1:** numpy array of N float
- **v2:** numpy array of N float

return:

- float, angle in radians

`v3find.are_parallel(v1, v2)`

Are two vectors parallel?

args:

- **v1:** numpy array of N float
- **v2:** numpy array of N float

return:

- boolean, True if parallel, to within 1e-5 radians, False otherwise

`v3find.are_same(v1, v2)`

Are two vectors identical, i.e. parallel and of same length, to within the precision of *are_parallel?*

args:

- **v1**: numpy array of N float
- **v2**: numpy array of N float

return:

- boolean, True if identical, False otherwise.

`v3find.find_v3(v1, v2, avec, maxlen)`

Wrapper function for finding the surface vector.

args:

- **v1**: numpy array of 3 float, a spanning vector of the plane
- **v2**: numpy array of 3 float, a spanning vector of the plane
- **avec**: numpy array of 3x3 float
- **maxlen**: float, longest accepted surface vector

return:

- numpy array of 3 float, surface vector choice

`v3find.initialize_search(v1, v2, avec)`

Seed search for v3 with the nearest-neighbouring Bravais lattice point which maximizes the projection out of plane of that spanned by v1 and v2

args:

- **v1**: numpy array of 3 float, a spanning vector of the plane
- **v2**: numpy array of 3 float, a spanning vector of the plane
- **avec**: numpy array of 3x3 float

return:

- numpy array of float, the nearby Bravais lattice point which maximizes
the projection along the plane normal

`v3find.refine_search(v3i, v1, v2, avec, maxlen)`

Refine the search for the optimal v3 which both minimalizes the length while maximizing orthogonality to v1 and v2

args:

- **v3i**: numpy array of 3 float, initial guess for the surface vector
- **v1**: numpy array of 3 float, a spanning vector of the plane
- **v2**: numpy array of 3 float, a spanning vector of the plane
- **avec**: numpy array of 3x3 float
- **maxlen**: float, longest vector accepted

return:

- **v3_opt**: list of numpy array of 3 float, options for surface vector

`v3find.score(vlist, v1, v2, avec)`

The possible surface vectors are scored based on their length and their orthogonality to the in-plane vectors.

args:

- **vlist**: list fo numpy array of 3 float, options for surface vector
- **v1**: numpy array of 3 float, a spanning vector of the plane
- **v2**: numpy array of 3 float, a spanning vector of the plane
- **avec**: numpy array of 3x3 float

return:

- numpy array of 3 float, the best scoring vector option

The following modules and their functions are used extensively throughout the chinook package.

6.1 Rotations

Created on Mon Oct 1 20:04:24 2018

@author: rday

Various functions relevant to rotations

`rotation_lib.Euler(rotation)`

Euler rotation angle generation, Z-Y-Z convention, as defined for a user-defined rotation matrix. Special case for $B = \pm Z \cdot \pi$ where conventional approach doesn't work due to division by zero, then Euler_A is zero and Euler_y is $\arctan(R_{10}, R_{00})$

args:

- **rotation**: numpy array of 3x3 float (rotation matrix)

OR tuple/list of vector and angle (numpy array of 3 float, float) respectively

return:

- **Euler_A, Euler_B, Euler_y**: float, Euler angles associated with

the given rotation.

`rotation_lib.Euler_to_R(Euler_A, Euler_B, Euler_y)`

Inverse of *Euler*, generate a rotation matrix from the Euler angles A,B,y with the same convention as in *Euler*.

args:

- **Euler_A, Euler_B, Euler_y**: float

return:

- numpy array of 3x3 float

`rotation_lib.Rodrigues_Rmat(nvec, theta)`

Following Rodrigues theorem for rotations, define a rotation matrix which corresponds to the rotation about a vector `nvec` by the angle `theta`, in radians. Works in pre-multiplication order (i.e. $v' = R.v$)

args:

- **nvec**: numpy array len 3 axis of rotation
- **theta**: float radian angle of rotation counter clockwise for $\theta > 0$

return:

- **Rmat**: numpy array 3x3 of float rotation matrix

`rotation_lib.rot_vector(Rmatrix)`

Inverse to *Rodrigues_Rmat*, take rotation matrix as input and return the angle-axis convention rotations corresponding to this rotation matrix.

args:

- **Rmatrix**: numpy array of 3x3 float, rotation matrix

return:

- **nvec**: numpy array of 3 float, rotation axis
- **theta**: float, rotation angle in float

`rotation_lib.rotate_v1v2(v1, v2)`

This generates the rotation matrix for PRE-multiplication rotation: or written another way, defining R s.t. $R.v1 = v2$. This rotation will rotate the vector **v1** onto the vector **v2**.

args:

- **v1**: numpy array len 3 of float, input vector
- **v2**: numpy array len 3 of float, vector to rotate into

return:

- **Rmat**: numpy array of 3x3 float, rotation matrix

6.2 Wigner Matrices

`wigner.Wd_denominator(j, m, mp, sp)`

Small function for computing the denominator in the s-summation, one step in defining the matrix elements of Wigner's small d-matrix

args:

- **j, m, mp**: integer (or half-integer) angular momentum

quantum numbers

- **s**: int, the index of the summation

return:

- int, product of factorials

`wigner.WignerD(l, Euler_A, Euler_B, Euler_y)`

Full matrix representation of Wigner's Big D matrix relating the rotation of states within the subspace of the angular momentum l by the Euler rotation $Z''(A)-Y'(B)-Z(y)$

args:

- **l**: int (or half integer) angular momentum
- **Euler_A, Euler_B, Euler_y**: float z-y-z Euler angles defining

the rotation
return:
 • **Dmat**: 2j+1 x 2j+1 numpy array of complex float

wigner.**big_D_element** (j, mp, m, Euler_A, Euler_B, Euler_y)

Combining Wigner's small d matrix with the other two rotations, this defines Wigner's big D matrix, which defines the projection of an angular momentum state onto the other azimuthal projections of this angular momentum. Wigner defined these matrices in the convention of a set of z-y-z Euler angles, passed here along with the relevant quantum numbers:

args:
 • **j, mp, m**: integer (half-integer) angular momentum quantum numbers
 • **Euler_A, Euler_B, Euler_y**: float z-y-z Euler angles defining

the rotation
return:
 • complex float corresponding to the [mp,m] matrix element

wigner.**fact** (N)

Recursive factorial function for non-negative integers.

args:
 • **N**: int, or int-like float
return:
 • factorial of N

wigner.**s_lims** (j, m, mp)

Limits for summation in definition of Wigner's little d-matrix

args:
 • **j**: int,(or half-integer) total angular momentum quantum number
 • **m**: int, (or half-integer) initial azimuthal angular momentum quantum number
 • **mp**: int, (or half-integer) final azimuthal angular momentum
 quantum number coupled to by rotation
return:
 • list of int, viable candidates which result in well-defined factorials in
 summation

wigner.**small_D** (j, mp, m, Euler_B)

Wigner's little d matrix, defined as $_j (-1)^{(mp-m+s)} \frac{2j+m-mp-2s}{s!(mp-m+s)!(j-mp-s)!} \cos(B/2) \sin(B/2)^{mp-m} /_s (j+m-s)!$

where the sum over s includes all integers for which the factorial arguments in the denominator are non-negative. The limits for this summation are defined by s_lims(j,m,mp).

args:
 • **j, mp, **m**** – integer (or half-integer) angular momentum
 quantum numbers for the orbital angular momentum, and its azimuthal projections which are related by the Wigner D matrix during the rotation
 • **Euler_B**: float, angle of rotation in radians, for the y-rotation
return:

- float representing the matrix element of Wigner’s small d-matrix
- ***

6.3 Spherical Harmonics

`Ylm.GramSchmidt(a, b)`

Simple orthogonalization of two vectors, returns orthonormalized vector

args:

- **a, b:** numpy array of same length

returns:

- **GS_a:** numpy array of same size, orthonormalized to the b vector

`Ylm.Y(l, m, theta, phi)`

Spherical harmonics, defined here up to $l = 4$. This allows for photoemission from initial states up to and including f-electrons (final states can be d- or g- like). Can be vectorized with `numpy.vectorize()` to allow array-like input

args:

- **l:** int orbital angular momentum, up to $l=4$ supported
- **m:** int, azimuthal angular momentum $|m| \leq l$
- **theta:** float, angle in spherical coordinates, radian measured from the z-axis $[0, \pi]$
- **phi:** float, angle in spherical coordinates, radian measured from the x-axis $[0, 2\pi]$

return:

- complex float, value of spherical harmonic evaluated at theta, phi

`Ylm.Yproj(basis)`

Define the unitary transformation rotating the basis of different inequivalent atoms in the basis to the basis of spherical harmonics for sake of defining L.S operator in basis of user

29/09/2018 added reference to the spin character ‘sp’ to handle rotated systems effectively

args:

- **basis:** list of orbital objects

return:

- dictionary of matrices for the different atoms and l-shells—keys are tuples of (atom, l)

`Ylm.binom(a, b)`

Binomial coefficient for ‘a choose b’

args:

- **a:** int, positive
- **b:** int, positive

return:

- float, binomial coefficient

`Ylm.fillin(M, l, Dmat=None)`

If only using a reduced subset of an orbital shell (for example, only t2g states in d-shell), need to fill in the rest of the projection matrix with some defaults

args:

- **M:** numpy array of $(2l+1) \times (2l+1)$ complex float
- **l:** int
- **Dmat:** numpy array of $(2l+1) \times (2l+1)$ complex float

return:

- **M**: numpy array of $(2l+1) \times (2l+1)$ complex float

`Ylm.gaunt(l, m, dl, dm)`

I prefer to avoid using the sympy library where possible, for speed reasons. These are the explicitly defined Gaunt coefficients required for dipole-allowed transitions ($dl = +/-1$) for arbitrary m, l and dm . These have been tested against the sympy package to confirm numerical accuracy for all l, m possible up to $l=5$. This function is equivalent, for the subset of dm, dl allowed to `sympy.physics.wigner.gaunt(1,1,1+dl,m,dm,-(m+dm))`

args:

- **l**: int orbital angular momentum quantum number
- **m**: int azimuthal angular momentum quantum number
- **dl**: int change in l ($+/-1$)
- **dm**: int change in azimuthal angular momentum ($-1,0,1$)

return:

- float Gaunt coefficient

`Ylm.laguerre(x, l, j)`

Laguerre polynomial of order l , degree j , evaluated over x

args:

- **x**: float or numpy array of float, input
- **l**: int, order of polynomial
- **j**: int, degree of polynomial

return:

- **laguerre_output**: float or numpy array of float, shape as input **x**

`Ylm.value_one(theta, phi)`

Flexible generation of the number 1.0, in either float or array format

args:

- **theta**: float or numpy array of float
- **phi**: float or numpy array of float

return:

- **out**: float or numpy array of float, evaluated to 1.0, of same shape and type as **theta, phi**

CHAPTER 7

Input Arguments

Most high-level functions in *chinook* operate on dictionaries as input format. This is done to support an efficient and high-density, yet largely human-readable input. Perhaps more importantly, this offers a high degree of flexibility in terms of required and optional keyword arguments. This comes however with a caveat that the full breadth of optional and required input arguments are not always conveniently accessible. This page then contains a comprehensive summary of the input dictionaries for a variety of functions.

7.1 Basis

Argument	Required	Datatype	Example	Notes
<code>atoms</code>	Yes	list of int	[0,0,1]	Unique elements are assigned distinct integers, consecutive
<code>Z</code>	Yes	dict of int:int	{0:77,1:8}	Atomic number for each atom from <i>atoms</i> above
<code>orbs</code>	Yes	list of lists of string	[[‘60’], [‘60’], [‘21x’,‘21y’]]	Elements define the orbital, one list for each atom, elements define the orbital definition. Structure of strings are: <i>principal quantum number n</i> , <i>orbital angular momentum l</i> , orbital label (s: N/A, p: x,y,z d:xz,yz,xy,XY,ZR)
<code>pos</code>	Yes	list of numpy arrays	[np.array([0.0,0.0,0.0]), np.array([1.2,0.0,0.0]), np.array([0.7,5.4,1.2])]	Positions of each atom in basis, each should be 3-float long, written in units of Angstrom
<code>spin</code>	No	dictionary	see below	Optional, for including spin-information, see below
<code>slab</code>	No	dictionary	see below	Optional, for generating a slab-superstructure, see below
<code>orient</code>	No	list	[np.pi/3,0,- np.pi/3]	One entry for each atom, indicating a local rotation of the indicated atom, various formats accepted. For more details, c.f. chinook.orbital.py

7.2 Spin

Argument	Required	Default type	Example	Notes
bool	Yes	boolean	True	Switch for activating spin-degree of freedom. Doubles basis size.
soc	No	boolean	True	Switch for activating spin-orbit coupling.
lam	Yes	dictionary	{0:0.15,1:0.0}	Required if soc:True, atomic-SOC strength
order	No	char	F	Spin-ordering, 'F' for ferro, 'A' for antiferro, None otherwise
dS	No	float	0.5	Required if order if 'F' or 'A'. Energy splitting between up and down in eV
p_up	No	numpy array of 3 float	np.array([0.0,0.0,0.0])	For AF, position of spin-up states, spin-up shifted down by dS on these sites.
p_dn	No	numpy array of 3 float	np.array([0.0,0.0,0.0])	For AF, position of spin-down states, spin-down shifted down by dS on these sites.

7.3 Hamiltonian

Argument	Required	Default	Example	Notes
type	Yes	string	SK	type of Hamiltonian. Any of 'SK' for Slater-Koster, 'txt' for textfile, 'list' for list of float, 'exec' for executable
cutoff	Yes	float or list of float	[1.5,3.6]	Cutoff hopping distance. Can be either a single float, or list of floats if different Hamiltonian arguments apply for different neighbour distances. Units of Angstroms.
renorm	Yes	float	1.0	overall bandwidth renormalization
offset	Yes	float	0.1	overall chemical potential shift in eV
tol	Yes	float	1e-4	Minimum strength term to include in eV
avec	No	numpy array of float	np.array([[1,0,0], [0,1,0], [0,0,1]])	Lattice vectors 3x3 float, required with SK.
spin	No	dictionary	See above	spin information, see above
V	No	dictionary	{ '020':0.0,'002200S':0.5 }	Slater Koster arguments, if applicable. See Slater-Koster on Tight-Binding page.
list	No	list	[[0,0,0,0,0,5],...]	List of Hamiltonian matrix elements, if passing in list format.
filename	No	string	my_hamiltonian.txt	Path to Hamiltonian textfile.
exec	No	list of methods	see Executable page	list of executable Python functions, see relevant page.

7.4 Momentum Path

Argument	Required	Default	Example	Notes
type	Yes	char	F	type of units, either fractional, i.e. units of reciprocal lattice units 'F', or absolute 'A' (i.e. 1/A)
pts	Yes	list of numpy array	<code>[np.array([0,0,0]), np.array([0.5,0.0,0])]</code>	Endpoints for the momentum path. An arbitrary number of points (>1) can be selected.
grain	Yes	int	200	Number of points between each endpoint in the list of 'pts'.
labels	No	list of str	<code>['\Gamma', 'M', 'X']</code>	Labels for each of 'pts', supports use of MathTex

7.5 Slab

Argument	Required	Default	Example	Notes
hkl	Yes	numpy array of int	<code>np.array([0,0,1])</code>	Miller index of surface
cells	Yes	float	100.0	Minimum thickness of slab, in Angstroms
buff	Yes	float	10.0	Minimum thickness of vacuum buffer layer.
term	Yes	tuple of two int	(1,1)	Atomic identities of atoms on top and bottom of surface, as defined in the basis arguments
fine	Yes	tuple of two float	(0.0,0.0)	Fine adjustments to surface termination. See slab page for details.

7.6 ARPES Experiment

Argument	Required	Default type	Example	Notes
cube	Yes	dictionary	{ 'X' : [-0.5,0.5,100], 'Y' : [-1,1,200], 'E' : [-2,0.1,1000], 'kz' : 0.0 }	Region of interest, X, and Y are momentum range, E is energy. Fixed kz only. For a single slice, can use e.g. 'X' : [0,0,1]. Can also pass angles (rad) using 'Tx', 'Ty' instead of 'X', 'Y'
pol	Yes	numpy array of float	np.array([1,0,0])	Polarization vector
hv	Yes	float	21.2	Photon energy, in eV
T	Yes	float	4.2	Temperature, for Fermi distribution. -1, or do not include 'T', to neglect thermal distribution.
resolution	No	dictionary	{ 'E' : 0.01, 'k' : 0.005 }	Energy, and momentum (FWHM) resolution, in eV, 1/Å.
SE	Yes	list	['constant', 0.05]	Self-Energy, see ARPES for further details
angle	No	float	1.59	Azimuthal rotation, in radians.
W	No	float	4.5	Work function, in eV
Vo	No	float	11.5	Inner potential in eV, for kz-dependence
slab	No	bool	True	Specify if using a slab geometry, will impose mean-free path. This also truncates the eigenvectors.
mfp	No	float	10.0	Electron mean-free path, sets penetration depth for slab-calculations.
spin	No	list	[1,np.array([0,0,1])]	Spin-ARPES, specify spin projection (+/- 1 for up/down) and axis as numpy array of 3 float.
threads	No	int	4	Split calculation across multiple parallel threads.
rad_type	No	string	See below	Radial integral type, See below for details.
rad_args	No	string	See below	Radial integral arguments, see below.

7.7 Radial Integrals

Argument	Datatype	Notes
rad_type	string	Type of initial radial state wavefunctions: 'slater' (default), 'hydrogenic', 'grid' (sample function on mesh), 'exec' for user-defined executable function, or 'fixed', where integrals are over-ridden for constant values.
rad_args		if 'rad_type' is 'grid', pass scaling of grid as a numpy array of float. If 'fixed', pass dictionary using 'a-n-l-lp' keys, float values (e.g. '0-2-1-0' for 2p to s transition).
phase_shift	dictionary	Final state phase shifts as first attempt at scattering final states. Same input type as for 'fixed' above, (e.g. '0-2-1-0' for 2p to s transition).

8.1 SBQMI Workshop

For use in the Student Seminar at SBQMI, we will use the following `input` and `run` files. The exercises contained there are also available as an interactive *jupyter* notebook.

8.2 Square Lattice

The following tutorial is also available as an interactive *jupyter* notebook, available for download with the [git repo](#), in the examples folder. Alternatively, it can be downloaded [here](#).

Before beginning, it is also worth noting that in the interest of maintaining an organized workspace, it can be beneficial to define a separate input script independent of the working experimental script. Examples of an `input file` and `experiment script` are available for this exercise.

In this tutorial, we work through a calculation of simulated ARPES spectra from scratch. While several input formats are accepted in chinook, here we use a Slater-Koster model for a square lattice.

8.3 Model Definition

The following modules will be needed for your work here.

```
import numpy as np
import chinook.build_lib as build_lib
import chinook.operator_library as operators
from chinook.ARPES_lib import experiment
```

We can begin by defining the lattice geometry, which we take here to be a square lattice. Throughout, feel free to try different parameters and geometries. Here, I've decided to make the in-plane dimensions smaller than the out of plane, and since I'm taking a 2-atom basis, I've expanded my unit cell accordingly.

```
a,c = 5.0,5.0
avec = np.array([[np.sqrt(0.5)*a,np.sqrt(0.5)*a,0],
[ np.sqrt(0.5)*a,-np.sqrt(0.5)*a,0],
[0,0,c]])
```

In *chinook*, functions which require many input arguments take dictionary-input, taking advantage of the readability and tidyness of this datatype. Specifically, orbital basis, Hamiltonian, ARPES configurations are all defined using dictionaries. The first relevant object will be the momentum path, used for defining a high-symmetry path to first explore our tight-binding model.

```
kpoints = np.array([[0.5,0.5,0.0],[0.0,0.0,0.0],[0.5,-0.5,0.0]])
labels = np.array(['$M_x$', '$\Gamma$', '$M_y$'])

kdict = {'type':'F',
'avec':avec,
'pts':kpoints,
'grain':200,
'labels':labels}

k_object = build_lib.gen_K(kdict)
```

Above, we get the first example of a dictionary-based set of arguments. We defined a k-path in fractional coordinate ('type':'F'). This requires the lattice vectors ('avec':avec) to define the path in units of inverse angstrom. The path itself is defined by 'points'. The 'grain' argument allows us to vary the number of steps between each segment of the k-path. Finally, we can pass an optional array of 'labels', for improving the readability of figures plotted.

We can now proceed to define our orbital basis. Since here I will use Antimony 5p states, we should include spin-orbit coupling, which is done with its own dictionary arguments. I define this on its own here, as I will want to re-use this dictionary for generating the Hamiltonian with spin-orbit coupling.

```
spin = {'bool':True, #include spin-degree of freedom: double the orbital basis
'soc':True, #include atomic spin-orbit coupling in the calculation of the_
↳Hamiltonian
'lam':{0:0.5}} #spin-orbit coupling strength in eV, require a value for each unique_
↳species in our basis

Sb1 = np.array([0.0,0.0,0.0])
Sb2 = np.array([np.sqrt(0.5)*a,0,0])

basis = {'atoms':[0,0], #two equivalent atoms in the basis, both labelled as species
↳#0
'Z':{0:51}, #We only have one atomic species, which is antimony #51 in the_
↳periodic table.
'orbs':[['51x','51y','51z'],['51x','51y','51z']], #each atom includes a full 5p_
↳basis in this model, written in n-l-xx format
'pos':[Sb1,Sb2], #positions of the atoms, in units of Angstrom
'spin':spin} #spin arguments.

basis_object = build_lib.gen_basis(basis)
```

We have defined the location of each atom in the orbital basis, and its full orbital content. Here we work with a full antimony 5p basis on each site. I have taken the second basis atom to sit in the centre of the ab plane, but you are free to modify this, as well as spin-orbit coupling strength, or even the atomic character.

At this point, we can define the Hamiltonian for our model. We will use a Slater Koster model for this, and we will assume nearest neighbour hoppings only. The options for the type of Hamiltonian are much more extensive, but the

Slater Koster framework is very amenable to tuning parameters, so is a nice option for this tutorial. For p-p hopping, Slater Koster requires only 3 parameters: the on-site energy ϵ_p , $V_{pp\sigma}$ and $V_{pp\pi}$.

```
Ep = 0.7
Vpps = 0.25
Vppp = -1.0
VSK = {'051':Ep, '005511S':Vpps, '005511P':Vppp}
cutoff = 0.72*a
```

In the following, I'm going to use a slightly more advanced model, where rather than consider nearest neighbour hoppings only, I will also include next nearest neighbour hoppings. To do this, I specify a Slater-Koster dictionary for on-site and nearest neighbours, and a second dictionary for next-nearest neighbours.

```
V1 = {'051':Ep, '005511S':Vpps, '005511P':Vppp}
V2 = {'005511S':Vpps/a, '005511P':Vppp/a}
VSK = [V1, V2]
```

These are distinguished by different cutoff-distances, with 0.8a sufficient for nearest neighbours, and 1.1a for next-nearest neighbours.

```
cutoff = [0.8*a, 1.1*a]

hamiltonian = {'type':'SK',          #Slater-Koster type Hamiltonian
               'V':VSK,              #dictionary (or list of dictionaries) of onsite and hopping_
               ↪potentials
               'avec':avec,          #lattice geometry
               'cutoff':cutoff,      #(cutoff or list of cutoffs) maximum length-scale for each_
               ↪hoppings specified by VSK
               'renorm':1.0,         #renormalize bandwidth of Hamiltonian
               'offset':0.0,         #offset the Fermi level
               'tol':1e-4,           #minimum amplitude for matrix element to be included in model.
               'spin':spin}          #spin arguments, as defined above
```

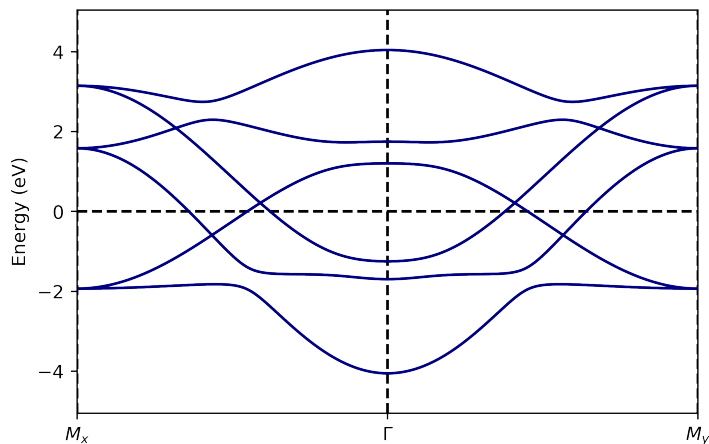
Finally, we can bring all this together to define a tight-binding model object.

```
TB = build_lib.gen_TB(basis_object, hamiltonian, k_object)
```

8.4 Model Characterization

With the model so defined, we can now diagonalize, and perform some diagnostics on the tight-binding model. We start by plotting the bandstructure over the k path.

```
TB.Kobj = k_object
TB.solve_H()
TB.plotting()
```



In the above model, we see the full bandstructure along high-symmetry directions for this model. Notably, the k_z dispersion is zero, as the c-lattice constant is larger than our cutoff-lengthscale. You can try adjusting the cutoff distance, or the c-lattice constant to introduce these hopping terms.

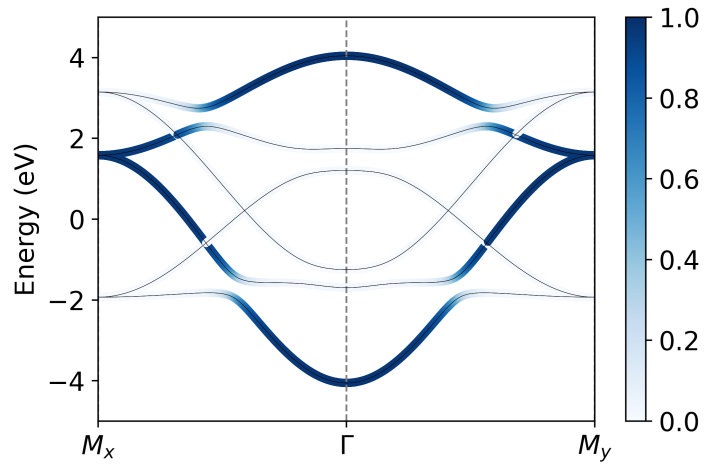
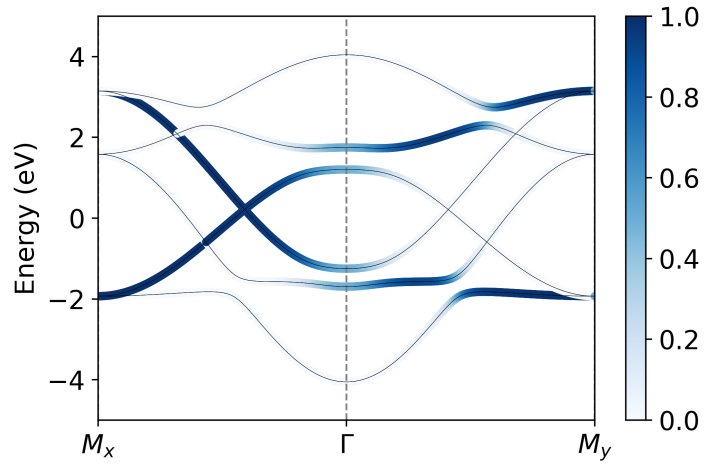
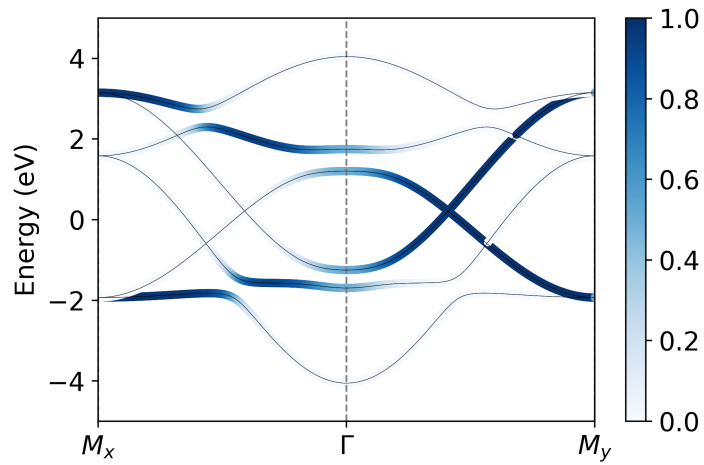
There are several ways in which we can characterize our tight-binding model before proceeding to calculation of the photoemission intensity. Here, we consider the orbital projection or ‘fat bands’ plot, which illustrates the orbital weight of different different basis states. Before doing this, it’s instructive to refer to the orbital basis ordering in our model.

```
TB.print_basis_summary()
```

```
>>> Index | Atom | Label | Spin | Position
=====
    0 |    0 | 51x   |   -0.5 | 0.000,0.000,0.000
    1 |    0 | 51y   |   -0.5 | 0.000,0.000,0.000
    2 |    0 | 51z   |   -0.5 | 0.000,0.000,0.000
    3 |    0 | 51x   |   -0.5 | 3.536,0.000,0.000
    4 |    0 | 51y   |   -0.5 | 3.536,0.000,0.000
    5 |    0 | 51z   |   -0.5 | 3.536,0.000,0.000
    6 |    0 | 51x   |    0.5 | 0.000,0.000,0.000
    7 |    0 | 51y   |    0.5 | 0.000,0.000,0.000
    8 |    0 | 51z   |    0.5 | 0.000,0.000,0.000
    9 |    0 | 51x   |    0.5 | 3.536,0.000,0.000
   10 |    0 | 51y   |    0.5 | 3.536,0.000,0.000
   11 |    0 | 51z   |    0.5 | 3.536,0.000,0.000
```

We want to plot the projection onto all p_x , p_y , and p_z orbitals. We see that orbitals the set $\{0,3,6,9\}$ are all p_x , $\{1,4,7,10\}$ are p_y and $\{2,5,8,11\}$ are p_z . To visualize these projections then:

```
px = operators.fatbs(proj=[0, 3, 6, 9], TB=TB, Elims=(-5, 5), degen=True)
py = operators.fatbs(proj=[1, 4, 7, 10], TB=TB, Elims=(-5, 5), degen=True)
pz = operators.fatbs(proj=[2, 5, 8, 11], TB=TB, Elims=(-5, 5), degen=True)
#The degen flag averages over degenerate states. All states are at least two-fold
↳degenerate,
#so this flag should certainly be on here.
```

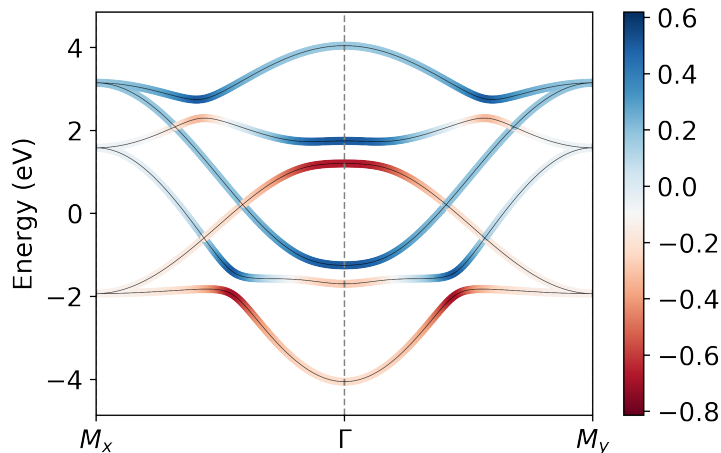


From this, we see that the bandwidth of the p_z band is largest, over 2 eV larger than that of the in-plane $p_{x,y}$ states. Furthermore, plotting along the k_x and k_y directions, we see clearly the strong momentum dependence of the orbital

character, which relaxes only near the avoided crossings where SOC mixes the orbital character.

In addition to orbital projections, we can also compute the expectation value of any Hermitian operator, and plot its value over each of the bands in a similar way. In this model, we have included spin-orbit coupling, and so we can plot $\langle \vec{L} \cdot \vec{S} \rangle$ as an example.

```
LdS_matrix = operators.LSmat(TB)
LdS = operators.O_path(LdS_matrix, TB, degen=True)
```



While here I have made use of the built-in function `operators.LSmat` to generate my operator matrix, one can pass in principal any valid Hermitian operator defined over the dimension of the orbital basis: in this case we require a 12 x 12 Hermitian matrix.

8.5 ARPES Calculation

While a number of other tools are available for model diagnostics, we'll proceed now to consideration of ARPES intensity simulation. Similar to the inputs defined above, we make use of Python dictionaries to organize the input arguments. Above, we saw that the Brillouin zone extends over a square of side-length 1.256 \AA^{-1} . We will begin by looking at the Fermi surface of our material.

```
arpes = {'cube':{'X':[-0.628,0.628,300], 'Y':[-0.628,0.628,300], 'E':[-0.05,0.05,
→50], 'kz':0.0}, #domain of interest
        'hv':100, #photon energy, in eV
        'T':10, #temperature, in K
        'pol':np.array([1,0,-1]), #polarization vector
        'SE':['constant',0.02], #self-energy, assume for now to be a constant_
→20 meV for simplicity
        'resolution':{'E':0.02, 'k':0.02}} #resolution

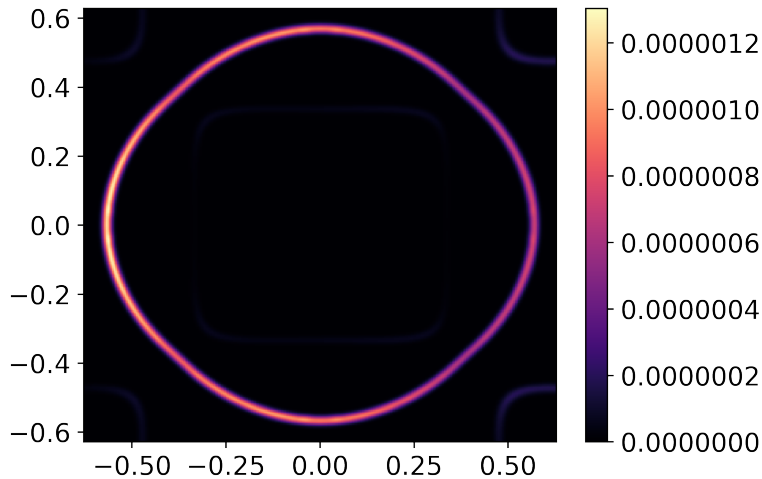
arpes_experiment = experiment(TB,arpes) #initialize experiment object
arpes_experiment.datacube() #execute calculation of matrix elements
```

Once the matrix elements have been calculated, the spectrum can be plot

The first thing we plot here is a constant energy contour, at the Fermi level

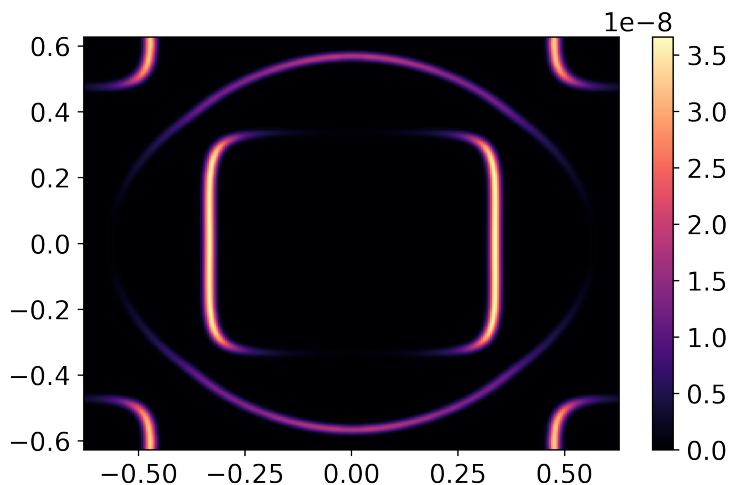
```
I, Ig, ax = arpes_experiment.spectral(slice_select=('w', 0.0))
```

The *spectral* function produces 2 output by default, corresponding to the raw, and resolution-broadened intensity maps. Generally speaking, the output is a 3-dimensional array of float, with the first two axes being momentum, and the last energy.



In consideration of the bandstructure above, we expect 3 Fermi-surface sheets, so why do we only see two? This is an explicit consequence of the matrix-element effects. If we consider the orbital-projected plots above, the large spin-orbit coupling strongly mixes the orbital character over much of the Fermi-surface. However, p_z character, which we are most strongly sensitive to with this polarization, is vanishing on the inner-most Fermi surface sheet. This readily explains its absence here. By however rotating polarization to be strictly in-plane, we can recover this sheet.

```
arpes['pol'] = np.array([0, 1, 0])
_ = arpes_experiment.spectral(arpes, slice_select=('w', 0))
```



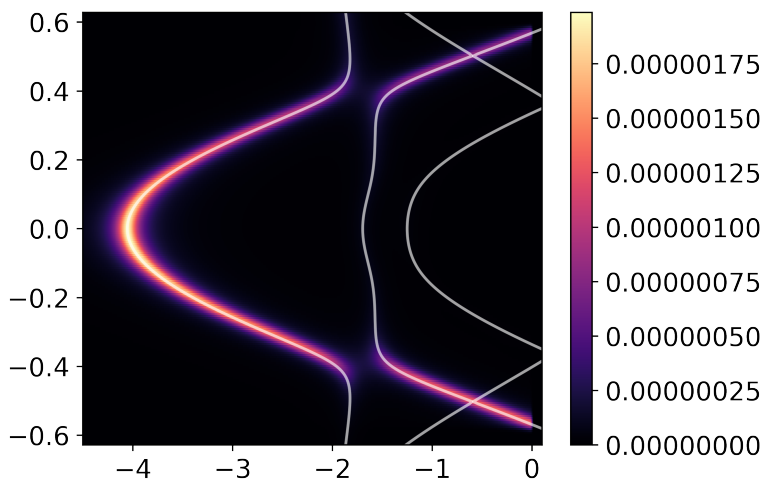
An easy way to quickly navigate a calculation over a 2-dimensional region of momentum space is with the `matplotlib_plotter.interface()`.

```
import chinook.matplotlib_plotter
interface = chinook.matplotlib_plotter.interface(arpes_experiment)
```

This will activate an interactive window, where constant k_x , k_y , and E can be selected, and all slices can be explored.

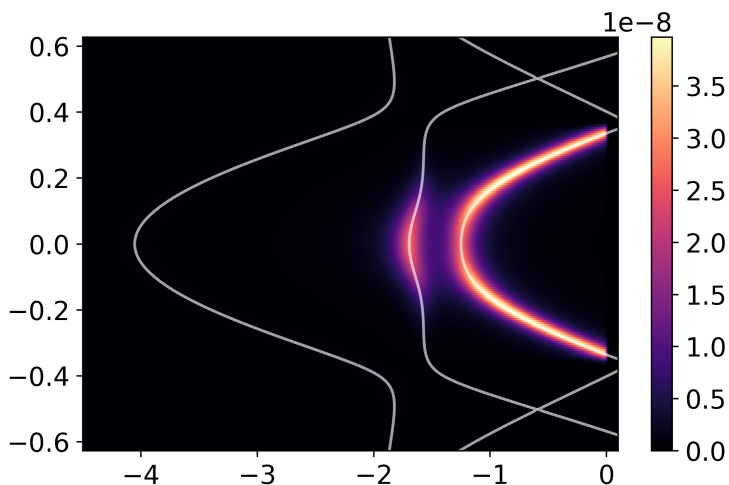
We can explore the evolution of the orbital character and ARPES intensity over a broader range of momentum by computing the matrix elements over a wider energy window. Below, we do so for a fixed in-plane momentum

```
arpes['cube'] = {'X':[-0.628,0.628,200], 'Y':[0,0,1], 'kz':0, 'E':[-4.5,0.1,1000]}
arpes['SE'] = ['constant',0.1]
arpes['pol'] = np.array([1,0,-1])
arpes_experiment = experiment(TB,arpes)
arpes_experiment.datacube()
_ = arpes_experiment.spectral(slice_select=('y',0),plot_bands=True)
```



Another perspective on the connection to orbital character can be gained by again, changing the polarization:

```
arpes['pol'] = np.array([0,1,0])
_ = arpes_experiment.spectral(arpes,slice_select=('y',0),plot_bands=True)
```



While some bands are suppressed as a result of orbital and polarization symmetry, others still are suppressed due

photoelectron interference. Our choice of a two-site unit cell is somewhat artificial, as there is no translational-symmetry breaking potential which prevents us from using a smaller one-atom unit cell. The folded bands in the reduced Brillouin zone of our excessively large unit cell can be described as anti-bonding states. To demonstrate this, we can perform a similar calculation over a wider domain of momentum, covering a full cut through the full Brillouin zone of the one-atom unit cell:

```
arpes['cube'] = {'X':[-1.256,1.256,300], 'Y':[0,0,1], 'kz':0, 'E':[-5,0.2,1000]}
arpes_experiment = experiment(TB,arpes)
arpes_experiment.datacube()
I,Ig,ax = arpes_experiment.spectral(slice_select=('y',0),plot_bands=True)
```

In this tutorial, we have seen how to define and characterize a tight-binding model using chinook. In addition to this, we have performed a few calculations of the photoemission intensity associated with this model, and seen a few ways in which the photoemission intensity is affected by the nature of both our system of interest and the experimental geometry. Feel free to make changes to the model here and experiment with different parameters to get a better sense of how to use some of these tools. You're now ready to begin building your own models and simulating ARPES intensity for your material of choice. A template input file is provided [here](#).

@author: rday

MIT License

Copyright (c) 2018 Ryan Patrick Day

Permission is hereby granted, free of charge, to any person obtaining a copy of this software and associated documentation files (the “Software”), to deal in the Software without restriction, including without limitation the rights to use, copy, modify, merge, publish, distribute, sublicense, and/or sell copies of the Software, and to permit persons to whom the Software is furnished to do so, subject to the following conditions:

The above copyright notice and this permission notice shall be included in all copies or substantial portions of the Software.

THE SOFTWARE IS PROVIDED “AS IS”, WITHOUT WARRANTY OF ANY KIND, EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT. IN NO EVENT SHALL THE AUTHORS OR COPYRIGHT HOLDERS BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER LIABILITY, WHETHER IN AN ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING FROM, OUT OF OR IN CONNECTION WITH THE SOFTWARE OR THE USE OR OTHER DEALINGS IN THE SOFTWARE.

CHAPTER 10

Contact

Questions? Please contact Ryan Day at rpday7_at_gmail_dot_com.

CHAPTER 11

Indices and tables

- `genindex`
- `modindex`
- `search`

b

`build_lib`, [28](#)

d

`dos`, [43](#)

f

`FS_tetra`, [48](#)

h

`H_library`, [30](#)

k

`klib`, [34](#)

o

`operator_library`, [50](#)

`orbital`, [36](#)

`orbital_plotting`, [54](#)

r

`rotation_lib`, [71](#)

s

`slab`, [60](#)

`SlaterKoster`, [38](#)

`surface_vector`, [67](#)

t

`TB_lib`, [39](#)

`tetrahedra`, [56](#)

v

`v3find`, [68](#)

w

`wigner`, [72](#)

y

`Ylm`, [74](#)

A

abs_to_frac() (in module slab), 62
 adaptive_int (module), 8
 AFM_order() (in module H_library), 30
 all_Y() (in module ARPES_lib), 13
 ang_mesh() (in module tilt), 23
 ang_v1v2() (in module surface_vector), 67
 ang_v1v2() (in module v3find), 68
 append_H() (TB_lib.H_me method), 39
 append_H() (TB_lib.TB_model method), 40
 are_parallel() (in module surface_vector), 67
 are_parallel() (in module v3find), 68
 are_same() (in module surface_vector), 67
 are_same() (in module v3find), 69
 ARPES_lib (module), 13
 atom_coords() (in module TB_lib), 42

B

b_zone() (in module klib), 34
 band_contribution() (in module dos), 43
 basal_plane() (in module slab), 62
 big_D_element() (in module wigner), 73
 bin_energy() (matplotlib_plotter.interface method), 22
 binom() (in module Ylm), 74
 build_ham() (TB_lib.TB_model method), 40
 build_lib (module), 28
 build_slab_H() (in module slab), 62
 bulk_to_slab() (in module slab), 63
 bvectors() (in module klib), 34

C

calc_Ylm() (orbital_plotting.wavefunction method), 55
 cell_edges() (in module TB_lib), 42
 clean_H() (TB_lib.H_me method), 40
 cluster_init() (in module H_library), 31
 col_phase() (in module orbital_plotting), 54
 colourmaps() (in module operator_library), 52

con_ferm() (in module ARPES_lib), 13
 copy() (intensity_map.intensity_map method), 20
 copy() (orbital.orbital method), 37
 copy() (TB_lib.H_me method), 40
 copy() (TB_lib.TB_model method), 41
 corners() (in module tetrahedra), 56

D

datacube() (ARPES_lib.experiment method), 15
 def_dE() (in module dos), 43
 define_radial_wavefunctions() (in module radint_lib), 9
 degen_Ovals() (in module operator_library), 53
 diagonalize() (ARPES_lib.experiment method), 15
 divisors() (in module slab), 63
 dos (module), 43
 dos_broad() (in module dos), 44
 dos_func() (in module dos), 44
 dos_tetra() (in module dos), 44

E

EF_tetra() (in module FS_tetra), 48
 error_function() (in module dos), 44
 Euler() (in module rotation_lib), 71
 Euler_to_R() (in module rotation_lib), 71
 experiment (class in ARPES_lib), 13

F

fact() (in module orbital), 36
 fact() (in module wigner), 73
 fatbs() (in module operator_library), 53
 fermi_surface_2D() (in module FS_tetra), 49
 fill_radint_dic() (in module radint_lib), 10
 fillin() (in module Ylm), 74
 find_centres() (orbital_plotting.wavefunction method), 55
 find_cursor() (matplotlib_plotter.interface method), 22
 find_cutoff() (in module radint_lib), 11

find_EF_broad_dos() (in module dos), 45
 find_EF_tetra_dos() (in module dos), 45
 find_harmonics() (orbital_plotting.wavefunction method), 55
 find_mean_dE() (in module ARPES_lib), 19
 find_v3() (in module surface_vector), 67
 find_v3() (in module v3find), 69
 FM_order() (in module H_library), 30
 frac_inside() (in module slab), 63
 frac_to_abs() (in module slab), 64
 FS() (in module operator_library), 50
 FS_generate() (in module FS_tetra), 48
 FS_tetra (module), 48

G

G_dic() (in module ARPES_lib), 13
 gaunt() (in module Ylm), 75
 gaussian() (in module dos), 45
 GCD() (in module slab), 60
 gen_all_pol() (ARPES_lib.experiment method), 15
 gen_basis() (in module build_lib), 29
 gen_const() (in module radint_lib), 11
 gen_H_obj() (in module TB_lib), 42
 gen_K() (in module build_lib), 28
 gen_kpoints() (in module tilt), 23
 gen_mesh() (in module tetrahedra), 57
 gen_orb_labels() (in module radint_lib), 11
 gen_SE_KK() (in module ARPES_lib), 19
 gen_slab() (in module slab), 64
 gen_surface() (in module slab), 64
 gen_TB() (in module build_lib), 28
 general_Bnl_integrand() (in module adaptive_int), 8
 get_kpts() (in module FS_tetra), 49
 Gmat_make() (in module ARPES_lib), 13
 GramSchmidt() (in module Ylm), 74

H

H2Hk() (TB_lib.H_me method), 39
 H_conj() (in module slab), 61
 H_library (module), 30
 H_me (class in TB_lib), 39
 H_surf() (in module slab), 61
 heron() (in module FS_tetra), 49
 Hobj_to_dict() (in module slab), 61

I

index_ordering() (in module H_library), 32
 initialize_search() (in module surface_vector), 67
 initialize_search() (in module v3find), 69
 integrate() (in module adaptive_int), 8
 intensity_map (class in intensity_map), 20
 intensity_map (module), 20

interface (class in matplotlib_plotter), 22
 is_numeric() (in module operator_library), 53
 iszero() (in module slab), 65

K

k_mesh() (in module tilt), 24
 k_parallel() (in module tilt), 24
 klib (module), 34
 kmesh() (in module klib), 34
 kmesh_hv() (in module klib), 35
 kpath (class in klib), 35
 kz_kpt() (in module klib), 35

L

laguerre() (in module Ylm), 75
 LCM() (in module slab), 62
 LCM_3() (in module slab), 62
 LdotS() (in module operator_library), 50
 Lm() (in module H_library), 31
 Lm() (in module operator_library), 51
 Lp() (in module H_library), 31
 Lp() (in module operator_library), 51
 LSmat() (in module operator_library), 50
 Lz() (in module H_library), 31
 Lz() (in module operator_library), 51

M

M_compute() (ARPES_lib.experiment method), 14
 make_angle_mesh() (in module orbital_plotting), 54
 make_radint_pointer() (in module radint_lib), 11
 mat_els() (in module H_library), 32
 matplotlib_plotter (module), 22
 mesh_reduce() (in module klib), 36
 mesh_tetra() (in module tetrahedra), 57
 mirror_SK() (in module H_library), 32
 Mk_wrapper() (ARPES_lib.experiment method), 14
 mod_dict() (in module slab), 65

N

n_func() (in module dos), 45
 n_tetra() (in module dos), 46
 ne_broad_analytical() (in module dos), 46
 ne_broad_numerical() (in module dos), 46
 neighbours() (in module tetrahedra), 57
 nonzero() (in module slab), 65
 not_point() (in module tetrahedra), 57

O

O_path() (in module operator_library), 51
 O_surf() (in module operator_library), 52
 on_site() (in module H_library), 33
 operator_library (module), 50

`operator_projected_fermi_surface()` (in module `operator_library`), 53
`orbital` (class in `orbital`), 37
`orbital` (module), 36
`orbital_plotting` (module), 54

P

`p_vecs()` (in module `slab`), 65
`par()` (in module `slab`), 65
`pdos_tetra()` (in module `dos`), 47
`plot_gui()` (`ARPES_lib.experiment` method), 15
`plot_img()` (`matplotlib_plotter.interface` method), 23
`plot_intensity_map()` (`ARPES_lib.experiment` method), 16
`plot_mesh()` (in module `tilt`), 24
`plot_unitcell()` (`TB_lib.TB_model` method), 41
`plot_wavefunction()` (or-
`bital_plotting.wavefunction` method), 55
`plotting()` (`TB_lib.TB_model` method), 41
`plt_pts()` (in module `klib`), 36
`points()` (`klib.kpath` method), 35
`pol_2_sph()` (in module `ARPES_lib`), 19
`poly()` (in module `ARPES_lib`), 19
`populate_box()` (in module `slab`), 65
`populate_par()` (in module `slab`), 66
`print_basis_summary()` (`TB_lib.TB_model` method), 41
`progress_bar()` (in module `ARPES_lib`), 20
`proj_avg()` (in module `dos`), 47
`proj_mat()` (in module `dos`), 47
`projection_map()` (in module `ARPES_lib`), 20
`propagate()` (in module `tetrahedra`), 57

R

`radint_calc()` (in module `radint_lib`), 12
`radint_dict_to_arr()` (in module `radint_lib`), 12
`radint_lib` (module), 9
`raw_mesh()` (in module `klib`), 36
`rect()` (in module `adaptive_int`), 8
`recur_product()` (in module `build_lib`), 30
`recursion()` (in module `adaptive_int`), 9
`redefine_vector()` (`orbital_plotting.wavefunction` method), 56
`refine_search()` (in module `surface_vector`), 68
`refine_search()` (in module `v3find`), 69
`region()` (in module `H_library`), 33
`region()` (in module `klib`), 36
`region()` (in module `slab`), 66
`rephase_wavefunctions()` (in module or-
`bital_plotting`), 54
`Rodrigues_Rmat()` (in module `rotation_lib`), 71
`rot_basis()` (`ARPES_lib.experiment` method), 16
`rot_projection()` (in module `orbital`), 37
`rot_vector()` (in module `rotation_lib`), 72

`rot_vector()` (in module `tilt`), 24
`rotate_v1v2()` (in module `rotation_lib`), 72
`rotation_lib` (module), 71
`run_gui()` (`matplotlib_plotter.interface` method), 23

S

`s_lims()` (in module `wigner`), 73
`S_vec()` (in module `operator_library`), 52
`sarpes_projector()` (`ARPES_lib.experiment` method), 16
`save_map()` (`intensity_map.intensity_map` method), 20
`score()` (in module `surface_vector`), 68
`score()` (in module `v3find`), 69
`SE_gen()` (`ARPES_lib.experiment` method), 15
`serial_Mk()` (`ARPES_lib.experiment` method), 16
`sim_tri()` (in module `FS_tetra`), 49
`sk_build()` (in module `H_library`), 33
`SK_cub()` (in module `SlaterKoster`), 38
`SK_full()` (in module `SlaterKoster`), 38
`slab` (module), 60
`slab_basis_copy()` (in module `orbital`), 37
`SlaterKoster` (module), 38
`small_D()` (in module `wigner`), 73
`smat_gen()` (`ARPES_lib.experiment` method), 16
`SO()` (in module `H_library`), 31
`solve_H()` (`TB_lib.TB_model` method), 41
`sort_basis()` (in module `orbital`), 37
`sorted_basis()` (in module `slab`), 66
`spectral()` (`ARPES_lib.experiment` method), 17
`spin_double()` (in module `H_library`), 33
`spin_double()` (in module `orbital`), 38
`surface_proj()` (in module `operator_library`), 54
`surface_vector` (module), 67
`Sz()` (in module `operator_library`), 52

T

`T_distribution()` (`ARPES_lib.experiment` method), 15
`TB_lib` (module), 39
`TB_model` (class in `TB_lib`), 40
`tet_inds()` (in module `tetrahedra`), 58
`tetrahedra` (module), 56
`tetrahedra()` (in module `tetrahedra`), 58
`thread_Mk()` (`ARPES_lib.experiment` method), 17
`tilt` (module), 23
`triangulate_wavefunction()` (or-
`bital_plotting.wavefunction` method), 56
`truncate_model()` (`ARPES_lib.experiment` method), 17
`txt_build()` (in module `H_library`), 33

U

`unpack()` (in module `slab`), 66
`unpack()` (`TB_lib.TB_model` method), 42

`update_pars()` (*ARPES_lib.experiment method*), 18

V

`v3find(module)`, 68

`v_vecs()` (*in module slab*), 66

`value_one()` (*in module Ylm*), 75

`Vlist_gen()` (*in module H_library*), 31

`Vmat()` (*in module SlaterKoster*), 39

W

`wavefunction` (*class in orbital_plotting*), 55

`Wd_denominator()` (*in module wigner*), 72

`wigner(module)`, 72

`WignerD()` (*in module wigner*), 72

`write_2D_Imat()` (*intensity_map.intensity_map method*), 21

`write_Ik()` (*ARPES_lib.experiment method*), 18

`write_map()` (*ARPES_lib.experiment method*), 18

`write_meta()` (*intensity_map.intensity_map method*), 21

`write_params()` (*ARPES_lib.experiment method*), 18

Y

`Y()` (*in module Ylm*), 74

`Ylm(module)`, 74

`Yproj()` (*in module Ylm*), 74