



# **pyfplo Documentation**

*Release 22.00-62*

**Klaus Koepernik**

**Jul 05, 2022**



# CONTENTS

<b>1</b>	<b>Introduction</b>	<b>1</b>
1.1	General . . . . .	1
1.2	Installation . . . . .	1
<b>2</b>	<b>Modules</b>	<b>3</b>
2.1	pyfplo.fedit . . . . .	3
2.1.1	Functions . . . . .	3
2.1.2	Fedit . . . . .	3
2.2	pyfplo.fploio . . . . .	15
2.2.1	INParser . . . . .	16
2.2.2	PObj . . . . .	17
2.2.3	FPLOInput . . . . .	20
2.2.4	OutGrep . . . . .	22
2.2.5	Basis . . . . .	24
2.2.6	BasDef . . . . .	25
2.2.7	BasDefSection . . . . .	25
2.2.8	MultiOrbital . . . . .	26
2.2.9	Data . . . . .	27
2.3	pyfplo.common . . . . .	27
2.3.1	BandFileContext . . . . .	27
2.3.2	BandPlot . . . . .	29
2.3.3	BandHeader . . . . .	32
2.3.4	BandWeights . . . . .	33
2.3.5	WeightDefinitions . . . . .	34
2.3.6	WeightDefinition . . . . .	35
2.3.7	OptionSet . . . . .	36
2.3.8	Site . . . . .	37
2.3.9	Watch . . . . .	37
2.3.10	Version . . . . .	39
2.3.11	Vlevel . . . . .	39
2.3.12	Constants . . . . .	40
2.4	pyfplo.slabify . . . . .	40
2.4.1	Slabify . . . . .	41
2.4.2	BoxMesh . . . . .	56
2.4.3	EnergyContour . . . . .	59
2.4.4	FermiSurfaceOptions . . . . .	60
2.4.5	DensPlotContext . . . . .	63
2.4.6	GreenOptions . . . . .	63
2.4.7	WeylPoint . . . . .	64
2.4.8	BfieldConfig . . . . .	65
2.4.9	WFSymOp . . . . .	66
2.4.10	BerryCurvatureData . . . . .	67
2.4.11	Site . . . . .	67
2.4.12	Data . . . . .	67

2.5	pyfplo.wanniertools . . . . .	68
2.5.1	WanDefCreator . . . . .	69
2.5.2	SingleOrbitalWandef . . . . .	70
2.5.3	MultipleOrbitalWandef . . . . .	71
2.5.4	Wandef . . . . .	71
2.5.5	Contrib . . . . .	73
<b>3</b>	<b>Examples</b>	<b>75</b>
3.1	A basic tutorial . . . . .	75
3.1.1	The bulk band structure . . . . .	75
3.1.2	The bulk Fermi surface . . . . .	78
3.1.3	Fermi surface cuts . . . . .	79
3.1.4	Bulk projected bands . . . . .	81
3.1.5	Finite slab with 10 unit cells . . . . .	83
3.1.6	Finite slab with 10 unit cells (doubled in-plane cell) . . . . .	86
3.1.7	Finite slab with 10 unit cells (doubled in-plane cell), one atom removed . . . . .	88
3.1.8	Finite slab with 10 unit cells (doubled in-plane cell), 3 atoms removed . . . . .	91
3.1.9	Semi infinite slab . . . . .	96
3.1.10	Semi infinite slab, doubled planar cell . . . . .	99
3.2	2D topological insulator . . . . .	101
3.2.1	The topological phase . . . . .	101
3.2.2	The trivial insulator phase . . . . .	107
3.3	3D topological insulator . . . . .	108
3.4	Weyl semi metals . . . . .	117
3.5	FEDIT scripting examples . . . . .	130
3.5.1	Set bandplot points . . . . .	130
3.5.2	Simple fedit example . . . . .	131
3.5.3	BCC Iron . . . . .	133
3.5.4	BCC and FCC Iron . . . . .	137
3.5.5	Set Extended basis . . . . .	142
3.5.6	BCC Iron, extended basis . . . . .	144
3.5.7	mBJ XC-potential . . . . .	148
3.6	FPLOIO examples . . . . .	151
3.6.1	Reading =.in files . . . . .	152
3.6.2	=.files to json . . . . .	153
3.6.3	Reading cif-files . . . . .	155
3.6.4	Write =.in with low level routines . . . . .	158
3.6.5	Write =.in with mid level routines . . . . .	159
3.6.6	Extract default basis into =.basdef . . . . .	161
3.6.7	User defined basis (in =.basdef) . . . . .	163
3.6.8	Extract =.basdef from output file . . . . .	168
3.6.9	Grep results . . . . .	170
	<b>Bibliography</b>	<b>175</b>
	<b>Python Module Index</b>	<b>177</b>
	<b>Index</b>	<b>179</b>

## INTRODUCTION

### 1.1 General

The python extension **pyfplo** exposes some of the functionality of the FPLO package to python. Among the main features are input management (write and read of `.in`), the easy creation of FPLO band/bandweight type files, the python version of `faddwei` and the `slabify` module, which allows to map Wannier function models onto larger unit cells, finite slabs and semi infinite slabs. Band structure and spectral densities can be calculated for these super structures. The Berry curvature for Weyl points can be calculated.

There are examples under `.../FPLO22.00-62/DOC/pyfplo/`.

**Attention:** Many data objects which are returned by the `pyfplo` classes are copies of the underlying c++ objects, which means that:

```
bp=sla.BandPlot()
bp.points.append(...)
```

will modify the temporary object returned by `pyfplo.common.BandPlot.points` (page 32) and then discard it. In general, the only supported operation is assignement:

```
l=[]
l.append(['$~G', [0,0,0]])
l.append(...)
l.append(...)

bp=sla.BandPlot()

bp.points=l
```

### 1.2 Installation

**Prerequisites:** `numpy` must be installed on your system. You need to know how to link to `lapack`, if you want to use it or you use our own copy (which might not as effective).

**Prepare Make:** As with the other **fplo** make files we need some preliminary setup. This is done by `install/Makefile` together with the default setup procedure.

**Make:** Go to `PYTHON` under your `fplo` source tree and run `make` for a `python2` package and `make python3` for a `python3` package. Note that the example scripts need to be run via `python3 script.py` in the latter case. Answer all questions. As a first trial just hit enter, unless a number is required and see if the installation works. Later try to refine the installation, if needed. The result should be a new directory called `pyfplo`. If you want you can copy the whole `pyfplo` directory somewhere else.

**Setup:** The standard installation procedure of python often requires admin rights, which is inconvenient. We decided to let `pyfplo` reside inside the FPLO source tree. To use it we need to set environment variables. If one copies `pyfplo` somewhere else, one need to adapt `pyfplopath` in the code below. This way

different versions can be managed. Use export statements in your batch scripts or local shell environment. (The actual syntax for exporting environment variables depends on the shell you use. There are most certainly already examples in your local shell configuration file.) Add

```
# Define the pyfplo path here. The last directory in this path
# must be the one under which pyfplo resides.

pyfplopath=.../FPLO/FPLO22.00-62/PYTHON

# Now export the environment variables.
export PYTHONPATH=$pyfplopath:$PYTHONPATH

# uncomment this if you get dynamic link issues
#export LD_LIBRARY_PATH=$pyfplopath/pyfplo:$LD_LIBRARY_PATH
```

to your script or `.bashrc` (or similar config files) where `.../FPLO/FPLO22.00-62/PYTHON` is replaced by the actual directory your `pyfplo` sits in. It is important that these variables are set before the python script is executed.

*If more than one pyfplo versions exist (which is often the case) it is absolutely necessary to either set the environment via `.bashrc` (and the like) or in job-queue scripts or to wrap `pyfplo` into a shell script which sets `PYTHONPATH` (and perhaps `LD_LIBRARY_PATH`) first and then executes the python script. If `LD_LIBRARY_PATH` is need both variables need to be set correctly otherwise python will import modules from one version and the library from another, which obviously is not good.*

There is yet another possibility, which might work. You could explicitly specify the `pyfplo` version path in the beginning of the script **before** you import any `pyfplo` modules:

```
import sys
sys.path.insert(0, "PATH_UNDER_WHICH_TO_FIND_pyfplo");
```

where `PATH_UNDER_WHICH_TO_FIND_pyfplo` is a path under which your desired `pyfplo` version sits. This method requires to modify your python scripts, but has its own charm.

You might want to find out, which version was used. Do this:

```
import pyfplo.fedit as fedit
print '\npyfplo version=: {0}\nfrom: {1}\n'.format(fedit.version, fedit.__file__
↪)
```

## 2.1 pyfplo.fedit

- *Functions* (page 3)
- *Fedit* (page 3)

The **fplo** editor **fedit** has a pipe mode, which is documented elsewhere. Read the documentation in **fedit** help screens and in the **fplo** manual (`FPLO.../DOC/MANUAL/doc.pdf`). This module allows to use **fedit** pipe mode from python scripts. It is recommended that this class is used to manipulate `=.in` files. The low level classes in module `pyfplo.fploio` (page 15) can be used as well, but do not ensure data consistency. E.g. one can set *relativistic.type* to a certain number (if one knows, which one) but the corresponding *relativistic.descriptor* would not be set automatically. Consequently the resulting `=.in` file would be correct from **fplo** point of view but a human will be misled when perusing the file.

Checkout the examples delivered with **fplo**.

### 2.1.1 Functions

**fploExecutable** (*suffix=""*)

**Parameters** *suffix* (*str*) – for convenience, *suffix* is appended to the exec name

Return the name of the fplo executable as *str*, which fits the pyfplo version. This is an educated guess. There is no guarantee that it is correct. If you compiled with a non-default build branch, this function does not return the name including the build branch suffix but just the generic name. Use *suffix* to select the correct executable or just write it out explicitly anyway.

### 2.1.2 Fedit

**class Fedit** (*recreate=False*)

Use this class as an interface for **fedit** pipe mode. There are general routines and specialized routines for a lot of possible input. Altogether arbitrary input can be created. To find out valid search strings or hotkeys consult **fedit** itself.

*recreate*: please consult *resetPipeInput* (page 4).

Most routines take keyword arguments. One only needs to specify the keyword explicitly if a selected subset of the arguments is used. Not all arguments need be given. If all are used the keywords can be left out (however the order of arguments does).

Many options are `bool`-like. In these cases the possible values are `True`, `1`, `'t'`, `'+'` or `False`, `0`, `'f'`, `'-'`.

Arguments like *defs* in *lsdau* (page 9) can be either a list of items, whose syntax is explained in each functions documentation, or an empty list `[]`, which basically resets *defs* to an empty list. In other words

there are no explicit counters for lists-like input. The number of list items in *defs* and the like is the counter. Have a look at the **fedit** LSDA+U submenu. There you have under CORRELATED STATES the item “(N)umber of definitions :” followed by the list of Wyckoff positions, states and FO... To set this counter to zero use *defs=[]*. If *recreate* or *resetPipeInput* (*recreate=True*) is used properly this is however rarely needed.

Illustrative example:

```
import pyfplo.fedit as fedit
fed=fedit.Fedit(recreate=True) # if `True`, reset to default input
                                # (except symmetry)
fed.relactivistic('s') # set to scalar relativistic
# or
fed.relactivistic('scal') # is the same
# or using the full explicit hotkey sequence
# note that the individual hotkey-value pairs are separated by newline
# (\n)
fed.addToPipeInput('@r@\n!scal!\n@x@')
# which is short for
fed.addToPipeInput('@r@') # enter relativistic submenu
fed.addToPipeInput('!scal!') # use search string
fed.addToPipeInput('@x@') # leave submenu
fed.pipeFedit(prot=True) # do the actual piping and show details
```

The simplest (old-style) way to create input is the following:

```
import pyfplo.fedit as fedit
fed=fed.Fedit()
fed.pipeFedit('''
# reset to default
@ e@
# vxc
@v@
!96!
@x@

#k-mesh
@k@ 16,,
#done
@q@
''',prot=False)
```

or with variable options:

```
import pyfplo.fedit as fedit
fed=fed.Fedit()
fed.pipeFedit('''
# reset to default
@ e@
# vxc
@v@
!{1}!
@x@

#k-mesh
@k@ {0},,
#done
@q@
'''.format(16,'Zunger'),prot=False)
```

**resetPipeInput** (*recreate=False*)

Reset all pipe input. If *recreate* is *True* the default-input- creation flag is set, which triggers non-symmetry input to become default after symmetry input. This way we start from a default input,

which avoids a buildup of unwanted input from former manipulations.

#### **addToPipeInput** (*txt*)

Add any hotkey sequence to the fedit pipe input. This routine can be used for arbitrary sequences and in fact is underlying all the specialized routines. Example:

```
# All in one go. (Note the use of newline \n)
fed.addToPipeInput('@r@\n!scal!\n@x@')
# which is short for
fed.addToPipeInput('@r@') # enter relativistic submenu
fed.addToPipeInput('!scal!') # use search string
fed.addToPipeInput('@x@') # leave submenu
```

#### **pipeFedit** (*text=None, prot=None*)

Execute the pipe mode of **fedit**. Use *text* if not *None*. Note that *text* must end in @q@ for success. If *text* is *None*, use the internal pipeinput collected so far. If *prot* is *True*, show verbose output about what is going on internally. See examples in *Fedit* (page 3).

#### **symmetry** (*compound=None, units=None, type=None, latcon=None, angles=None, atoms=None, spacegroup=None, setting=None, globalaxes=None, generators=None*)

This function (like most) can be called with different key words, also sequentially, as in:

```
fed.symmetry(latcon=[4,5,6])
fed.symmetry(spacegroup=221)
```

or in a single call:

```
fed.symmetry(latcon=[4,5,6],
             spacegroup=221)
```

*units* can be a search string or a hotkey.

*type* (structure type) can be a search string or a hotkey.

*spacegroup* (int or str) is the group number

*setting* (str) is the setting. Consult **fedit** itself for proper values.

Example:

```
fed.symmetry(spacegroup='74', setting='a-cb')
```

*latcon* can be missing or a list or string containing 3 lattice constants.

*angles* can be missing or a list or string of 3 axis angles.

*atoms* can be [[*element*, *position*], ...] where

*element* (str) is the element

*position* (list or str) is a list or a string of the positions x,y,z coordinates. The comma repeater can be used. in the same way as in **fedit** itself.

Example:

```
fed.symmetry(atoms=[['Sm' , ['0.123', '3/4', ', ']]]
#or
fed.symmetry(atoms=[['Sm' , [0.123, 3./4, ', ']]]
#or
fed.symmetry(atoms=[['Sm' , '0.123 3/4, ']]]
```

Note that in python 3/4 equals 0 (integer arithmetic) while 3./4=0.75.

*globalaxes* can be [*active*, *xaxis*, *zaxis*] where

*active* (bool-like) says, if the axes are active.

*xaxis* and *zaxis* (list or str) are a list or string of 3 numbers.

Example:

```
fed.symmetry(globalaxes=[True, [1, 1, 0], [-1, 1, 2]])  
# or  
fed.symmetry(globalaxes=[1, "1 1 0", "-1 1 2"])
```

*generators* (*list*) is a list or string of *int* and represents the list of valid subgroup generators. Use the **fed**it symmetry-information submenu in the symmetry menu to find out the desired numbers.

**options** (*oplist=None*)

*oplist* can be [[OPTION, True\_or\_False], ...] or simply [OPTION, True\_or\_False] for a single option, where

OPTION is one of fedits options-menus options.

True\_or\_False can also be 0 or 1

Examples:

```
fed.options([[ 'FULLBZ', 1], [ 'CALC_DOS', False]])  
# or  
fed.options([ 'FULLBZ', 1])  
fed.options([ 'CALC_DOS', False])
```

**spin** (*spin=None, initialspinsplit=None, initialspin=None, fsm=None*)

*spin* can be missing or 1 or 2.

*initialspinsplit* can be missing or True or False or equivalent values.

*initialspin* can be [[sort, spinmoment], ...] where

sort (*int*) is a sort/Wyckoff position number and

spinmoment (*float*) is the initial spin moment

*fsm* can be missing or a list: [True\_or\_False, fsmmoment]

Example:

```
fed.spin(initialspin=[[1, 2], [2, 0], [3, 0]])  
fed.spin(spin=2, initialspinsplit=True)  
  
fed.spin(fsm=[True, 2.4])
```

**xcoptions** (*cmBJ=None*)

*cmBJ* can be None or a float

Example:

```
fed.xcoptions(cmBJ=None)  
fed.xcoptions(cmBJ=1.1)
```

**relativistic** (*mode=None, quantaxis=None*)

*mode* can be a search string or a hotkey.

*quantaxis* (*list* or *str*): e.g. [1, 1, 0] or '1 1 0'

Example:

```
fed.relativistic('scal')  
# or  
fed.relativistic('s')  
  
# set full relativistic and quantization axis  
fed.relativistic('full', quantaxis=[1, 1, 0])
```

**bzintegration** (*nxyz=[12, 12, 12], temperature=None, method=None, mpparam=None*)

*nxyz* (list or str): a list or string with 3 int.

*temperature* (float or str): temp for molecules.

*method* (str): can be a hotkey ('T' or 'M') or a search string (e.g. 'tet' or 'pax'). Note that 'meth' will select the tetrahedron method because 'meth' matches 'method'

*mpparam*: can be a list [width, order] (width in Hartree)

Example:

```
# nxyz is the first argument to bzintegration. Hence
# we can leave out the keyword
fed.bzintegration([ 6, 9, 12])
# or we do it explicitly and use strings
fed.bzintegration(nxyz=' 6 9 12')
# choose method
fed.bzintegration(method='pax', mpparam=[0.005, 1])

# or all together
fed.bzintegration([ 6, 9, 12], method='pax', mpparam=[0.005, 1])
```

**optics** (*active=False, bounds=None, Ne=None, jointdos=None, stopafter=None*)

*active* (bool-like): is optics active?

*bounds*: can be a list with the content [lbound, ubound] where lbound... are float

*Ne* (int or str): number of energy points

*jointdos* (bool-like): see **fedit** help screens.

*stopafter* (bool-like): obvious.

Example:

```
fed.optics(True, bounds=[0, 10], Ne=2000, stopafter=True)
```

**charges** (*vca=None, charges=None*)

*vca*: can be a list: [active, Zlist] where

active is bool-like

Zlist is [] or [[sort, Z], ...] where

sort (int): is a Wyckoff position number

Z (float or str): is the nuclear charge

*charges* can be [mode, ionicity] where

mode (str): is a search string ('none', 'jel', 'mol') or a hotkey (1, 2, 3)

ionicity (float or str): is obvious

Example:

```
fed.charges(vca=True, [
    [1, 25.2],
    [2, 27.9]]
])
```

**vxc** (*version=None, xfactor=None*)

*version* (str): a hotkey or search string (e.g. '5' or 'Ernzerhof 96')

*xfactor* (float or str): xc-field scaling

Example:

```
fed.vxc(version=5)
```

**verbosity** (*level=None*)

*level* (str): a hotkey or search string (e.g. '2' or 'basic')

Example:

```
fed.verbosity(level='basic')
```

**ti** (*z2=None, homooffset=None, forcewanniercenters=None, integrationintervals=None, kyparamintervals=None*)

*z2* (bool-like): switch on/off the calculation of Z2 invariants.

*homooffset* (int): The code will determine the likely highest occupied valence band (assuming a real gap exists). Then it calculates the invariants for a set of bands from -offset to +offset around this band.

*forcewanniercenters* (bool-like): the Wannier center algo will additionally be executed for centro symmetric systems.

*integrationintervals* (int): The Wannier centers are functions of a paramter (ky-parameter), which are obtained by “integrating” along a second direction. Here you specify the number of integration intervals.

*kyparamintervals* (int): This determines the number of intervals along the ky-parameter axis.

Example:

```
fed.ti(z2=True,homooffset=4,forcewanniercenters=False,
integrationintervals=40,kyparamintervals=100)
```

**finuc** (*mode=None*)

Finite nucleus settings.

*mode* (str): hotkey or search string (e.g. 3 or 'linear')

Example:

```
fed.finuc('linear')
```

**opc** (*active=True, functional=None, defs=None*)

Orbital polarization correction.

*active* (bool-like): active or not?

*functional* (str): hotkey or search string (e.g. 1 or 'spin dependend')

*defs* (list): may be [] or [[sort, state-label], ...] where

sort (int) is a wyckoff position number and

state-label (str) is something like '3d' or '4f'

Example:

```
fed.opc(True,'spin independend',[[3,'4f'],[7,'4f']])
```

**bandplot** (*active=True, points=None, ksteps=None, weights=None, transaxis=None, xaxis=None, zaxis=None, bwdeffile=None, ldosites=None, interval=None, restrictbands=None, idos=None, ndos=None, coeff=None*)

Bandplot options.

*active* (bool-like): is bandplot active?

*weights* (bool-like): is bandweights active?

*points* (list): can be [] or [ [label, [kx, ky, kz]] , ... ] alternatively the point can be written [label, "kx ky kz"]

*kstep* (int): steps between sym-points  
*bwdef*file (str): file name for band weight definitions (= .bwdef)  
*transaxis* (bool-like): transform axes?  
*xaxis* (list or str): xaxis for band weights  
*zaxis* (list or str): xaxis for band weights  
*ldossites* (list): list of ldos sites  
*idos* (bool-like): plot idos?  
*ndos* (bool-like): plot netdos?  
*interval* (list): dos-energy interval: [ne, lbound, ubound] where  
     ne (int) is the number of energy points  
     lbound (float) is the lower energy bound  
     ubound (float) is the upper energy bound  
*coeff* (bool-like): trigger output of +coeff  
*restrictbands* (bool-like): restrict bands to window

Example:

```
...
# let's do it in two steps: general settings
fed.bandplot(active=True,ldossites=[2,17],
             weights=True,transaxis=True,xaxis='1 1 0',zaxis=[-1,1,0],
             restrictbands=True,
             interval=[2000,-2,3])
# high symmetry points
fed.bandplot(points=[
    ['$~G', [0,0,0]],
    ['X', [1,0,0]],
    ['M', [1,1,0]],
    ])

```

**lsdau** (*active=True, functional=None, projection=None, defs=None, tol=None, occumode=None*)  
 LSDA+U settings.

*active* (bool-like): is LSDA+U active?  
*functional* (str): hotkey or search string (e.g. 2 or 'atomic limit')  
*projection* (str): hotkey or search string (e.g. 4 or 'gross')  
*tol* (float): the inner occu-matrix loop convergence tolerance  
*occumode* (str): hotkey or search string (e.g. 2 or 'fixed')  
*defs* (list): can be [] or [[sort, state-label, [f0, f2, f4, f6]], ...] or [[sort, state-label, [], [U, J, F4oF2, F6oF2]], ...]. The second option with an empty F-list is for convenience. The list elements are  
     sort (int) is a sort  
     state-label (str) is e.g. '3d' or '4f'  
     f0, f2, f4, f6 (float) are obvious  
     U, J, F4oF2, F6oF2 (float) are U, J, F4/F2 and F6/F2. If not needed, J, F4oF2 and F6oF2 can be left out. Also, if F4oF2 and/or F6oF2 are not given but needed, default values are used. If only U is given all Fi, i>0 are zero and hence J is zero:

- s: only U is needed
- p: only U and J make sense
- d: U, J and F4oF2 (default 5.5/8.5)
- f: U, J, F4oF2 and F6oF2 (default 2./3 and 1./2)

Example:

```
# explicit F2,F4, two sorts
fed.lsdau(True, functional='atomic', projection='gross',
, defs=[
    [3, '3d', [8, 9, 5, 0]],
    [5, '3d', [8, 9, 5, 0]],
])
# implicit F2, F4, explicit U and J, two sorts
fed.lsdau(True, functional='atomic', projection='gross',
, defs=[
    [3, '3d', [], [8, 1]],
    [5, '3d', [], [8, 1]],
])
```

**gridoutput** (*active=None, stopafter=None, defs=None*)

Define the grid output settings.

*active* (bool-like): do we want it?

*stopafter* (bool-like): obvious

*defs* (list): can be [] or a list of dicts one for each grid definition, with the following structure:

```
[{ # first grid definition
  'basis': basis,
  'dir1': list_or_str,
  'dir2': list_or_str,
  'dir3': list_or_str,
  'origin': list_or_str,
  'subdiv': list_or_str,
  'includeperiodicpoints': True_or_False
  'createopendx': True_or_False
  'scfenergywindow': list_or_str_of_two_float
  'quantities': quantdict,
  'outputdata': outputdatadict,
  'kresolved': kresolvedlist,
  'eresolved': eresolvedlist},
{ # second grid definition
  ...},
...
]
```

where not all dict keys need to be present! The individual values are

list\_or\_str: a list or string of 3 numbers. In case of subdiv these must be int.

basis (str): 'prim' or 'conv' or 'cart'

quantdict (dict): {quantityname: True|False, ...}

quantityname (str): 'dens', 'pot', 'scf', 'coul', 'xc', 'ewald'

outputdatadict (dict): {outputdataname: True|False, ...}

outputdataname (str): index, points, 'total', 'spin', 'up', 'down', 'comp'

kresolvedlist (list): [] or [kresdict, ...]

```

kresdict (dict):  {'name':some_name,'point':list_or_str,
                  'bands': list_of_int,'ewindow': [lbound, uband]}
eresolvedlist (list): [] or [eresdict, ...]
eresdict (dict): {'name':some_name,'defs': defslst}
defslst (list): [[emin,emax,de,'up'|'do'|'bo'],...]

```

Example:

```

fed.gridoutput (True, stopafter=True, defs=[
{
'basis':'prim','dir1':[1,1,0],'dir2':[-1,1,0],'dir3':'0 0 1',
'origin':[-0.5,-0.5,'2/3'],'subdiv':'10 20 30',
'quantities':{'pot':True,'dens':True},
'outputdata':{'points':True,'total':True,'spin':True},
'kresolved':[
{'name':'Gamma','point':[0,0,0],'ewindow':[-1.1,2.4]},
{'name':'X','point':[1,0,0],'bands':'23 24 25'}
],
'eresolved':[
{'name':'r1','defs':[-7,9,0.01,'b']},
{'name':'r2','defs':[-1,1,0.01,'u']},
{'name':'r2','defs':[-1,1,0.01,'d']},
]
}
])

```

**coreoccupation** (*defs=None*)

Define the core occupation. (Experimental option)

```

defs (list): [] or [ [sort, state-label, [occ, occ, ...], [occ, occ, ...]] , ...
] where

```

sort (int) is a sort number

state-label (str) is e.g. '4f'

end where the last two lists (for spin up and down) are the occupation numbers for the  $2*1+1$  orbitals. Normally the occus should all be the same per spin channel. One only gives as many occupation numbers as indicated by the shell degeneracy ( $2*1+1$ ). Example:

```

fed.coreoccupation (defs=[
[2, '2p', [1, 1, 1], [0.2, 0.2, 0.2]]
])

```

A fancy example to avoid typing: we set all orbitals to 4/14 in order to get a homogenous non-polarized 4f shell with 4 electrons. For illustration we also add some polarized example for p-states:

```

fed.coreoccupation (defs=[
[2, '4f', ['4/14']*7, ['4/14']*7],
[3, '3p', [1]*3, ['2/5']*3],
])

```

**dhvaise** (*active=None, isovalue=None, subdiv=None, bisections=None, upbands=None, downbands=None, wanhamfile=None, deleteoldfiles=None, needbandweights=None, ad-dweifile=None, shifts=None*)

Settings for the iso-branch of the dHvA module.

*active* (bool-like): activate the preparation active.

*isovalue* (float or str): obvious

*subdiv* (list or str): a list or string of 3 int.

*bisections* (int): obvious

*upbands* (list): [] or a list or string of band numbers  
*downbands* (list): [] or a list or string of band numbers  
*wanhamfile* (str): a wannier hamiltonian file name (+hamdata).  
*addweifile* (str): a =.addwei type file  
*deleteoldfiles* (bool-like): obvious  
*needbandweights* (bool-like): obvious  
*shifts* (list): [] or [[weight-index, factor], ...]

Example:

```
fed.dhvaiso(active=True, isovalue=-0.1, subdiv='10,', bisections='4')
# special shift input (see FPLO.../DOC/MANUAL/doc.pdf)
fed.dhvaiso(shifts=[ [76,0.02], [77,-0.02] ])
```

**dhva** (*fields=None, nangles=None, anglerange=None, nplanes=None, planerange=None, bands=None, areachainthreshold=None, orbitsearchoffset=None, orbitssampleportion=None, arearadiusfactor=None, smoothnessthreshold=None, outputoptions=None, debugoptions=None*)

Settings for the dHvA-branch of the dHvA module.

*fields* (list): can be [] or [[label, [bx, by, bz]], ...] alternatively a field can be written as [label, "bx by bz"]

*nangles* (int): number of angles

*anglerange* (list): can be [from, to], where from and to are int.

*nplanes* (int): number of planes

*planerange* (list): can be [from, to], where from and to are int

*bands* (list): can be [bandnumbers, parts, spins] where

bandnumbers (list or str) must be [] or a list or string of int

parts (list or str) must be [] or a list or string of int

spins (str) must be 'up' or 'down' or 'both' or shorter 'u', 'd' or 'b'

*areachainthreshold* (float): see FPLO.../DOC/MANUAL/doc.pdf

*orbitsearchoffset* (float): see FPLO.../DOC/MANUAL/doc.pdf

*orbitssampleportion* (float): see FPLO.../DOC/MANUAL/doc.pdf

*arearadiusfactor* (float): see FPLO.../DOC/MANUAL/doc.pdf

*smoothnessthreshold* (float): see FPLO.../DOC/MANUAL/doc.pdf

*outputoptions* (list): can be [[OPTION, True|False], ...] see fedit screens for OPTION

*debugoptions* (list): can be [[OPTION, True|False], ...] see fedit screens for OPTION

Example:

```
#first we set the fields (note, we could do all of it in one call)
fed.dhva(fields=[
    ['[010]', [0, 1, 0]],
    ['[100]', [1, 0, 0]],
    ['[111]', '1 1 1']
])

# focus on particular bands, all parts and spins.
fed.dhva(bands=[[52, 53], [], 'b'])
```

(continues on next page)

(continued from previous page)

```
# increase accuracy
fed.dhva(nangles=50,nplanes=200,noewalddcor=True)
```

**basis** (*version=None, extensionlevel=None, add3d=None, addf=None, core4f=None, core4fNoValenceF=None, multicore=None, multisemicore=None*)

Here some standard basis settings can be defined. If you want to switch to an extended basis in a pre-existing `=.in` use:

```
fed=fedit.Fedit(recreate=False)
fed.basis(extensionlevel=2,add3d=True,addf=True)
fed.pipeFedit(prot=True)
```

If you want to switch to an extended basis in a pre-existing `=.in` and make sure that the rest of the basis is default use:

```
fed=fedit.Fedit(recreate=False)
fed.basis(extensionlevel=2,add3d=True,addf=True,
          multicore=[],multisemicore=[],
          core4f=[],core4fNoValenceF=[])
fed.pipeFedit(prot=True)
```

*version* (int or str): hotkey or search string of basis version

*extensionlevel* (int): the level of extended basis,

- default: `extensionlevel=1`,
- single->double, double->triple ... : `extensionlevel=2`
- double->triple, triple->quadruple ... : `extensionlevel=3`

*add3d* (bool-like): if `True` add 3d if no d-orbital in default basis. Applies to H and He.

*addf* (bool-like): if `True` add an f-orbital if none exists in the valence section. If the semicore section contains a 4f-state a 5f-valence orbital is added. In other words this option only considers the valence section.

*core4f* (list of str and/or int): a list of element names and/or sort numbers for which to shift an existing valence 4f-orbital into the core. The list can be a mix of names and sorts. To delete existing `core4f` definitions use:

```
core4f=[]
```

To leave this setting alone use:

```
core4f=None
```

*core4fNoValenceF* (list of str and/or int): a list of element names and/or sort numbers for which to shift an existing valence 4f-orbital into the core and remove remaining f-polarization orbitals from the valence section completely. The list can be a mix of names and sorts. To delete existing `core4f` definitions use:

```
core4fNoValenceF=[]
```

To leave this setting alone use:

```
core4fNoValenceF=None
```

*multicore* (list-like: [ [Q0,S0],[Q1,S1],... ]): the length the list defines the multiplicity of all core orbitals. The individual list members need to be lists of two float, the first is the Q-parameter and the second the S-parameter. Example:

```
multicore=[[0,0],[2,10]] # double core with Q[0]=0, S[0]=0, Q[1]=2, S[1]=10
```

To delete an existing multicore definition use:

```
multicore=[]
```

To leave this setting alone use:

```
multicore=None
```

*multisemicore* (list-like: [ [Q0,S0],[Q1,S1],...]): the length the list defines the multiplicity of all semicore orbitals. The individual list members need to be lists of two float, the first is the Q-parameter and the second the S-parameter. Example:

```
multisemicore=[[0,0],[2,10]] # double core with Q[0]=0, S[0]=0, Q[1]=2, S[1]=10
```

To delete an existing multisemicore definition use:

```
multisemicore=[]
```

To leave this setting alone use:

```
multisemicore=None
```

**numerics** (*maxL=None, thci\_nr=None, thci\_angmin=None, thci\_angmax=None, thci\_use\_symmetry=None*)

A subset of numerics menu options.

*maxL* (int): cutoff for angular momentum expansion in the potential.

*thci\_nr* (int): number of radial mesh mesh points for three center integrals.

*thci\_angmin* (int): smallest angular mesh type (for small radii) for three center integrals.

*thci\_angmax* (int): largest angular mesh type (for large radii) for three center integrals.

*thci\_use\_symmetry* (bool-like): use only irreducible mesh.

**iteration** (*n=30, acc=None, eacc=None, method=None, mixing=None, progress=None, subdim=None, occuratio=None, criterion=None*)

*n* (int): number of iterations

*acc* (float or str): accuracy of density

*eacc* (float or str): accuracy of etot

*method* (str): can be missing or a search string or a hotkey (e.g. 'lciterat')

*mixing* can be a float

*progress* can be a float

*subdim* (int or str): maximum subspace dimension

*occuratio* (float or str): occu-mat/density ratio

*criterion* (str): hotkey (e.g. 2 for 'density'). Note that the search strings are not very usefull here, due to their similarity (historical bad design).

Example:

```
fed.iterations(n=100,method='lciterat',mixing=0.1)
```

**forces** (*forcemode=None, n=None, acc=None, subdim=None, vary=None, noewaldcor=None, eachstep=None*)

*forcemode* (str): can be 'no', 'opt', 'single'

*n* (int): number of force iterations

*acc* (float or str): accuracy criterion in eV/Ang

*subdim* (int or str): maximum subspace dimension

*vary* (list of int or string of int): a list/string of sorts/Wyckoff position numbers to be varied. Empty list means 'vary all'.

*noewaldcor* (bool-like): disable the expensive (marginally more accurate) nonspherical Ewald correction for forces.

*eachstep* (bool-like): calculate forces in each step.

Example:

```
fed.forces('opt', acc=1e-2, vary=[5, 9])
```

## 2.2 pyfplo.fploio

- *INParser* (page 16)
- *PObj* (page 17)
- *FPLOInput* (page 20)
- *OutGrep* (page 22)
- *Basis* (page 24)
- *BasDef* (page 25)
- *BasDefSection* (page 25)
- *MultiOrbital* (page 26)
- *Data* (page 27)

This module contains the parser interface to read data from `=.in` type files. It can also be used to change data in the file. WARNING: it is better to use `pyfplo.fedit` (page 3) for changing input. The `fedit` interface is kept reasonably clean, while the naming conventions in `=.in` which is accessed directly by the parser is at times chaotic. Furthermore, there are semantic dependencies in the file, as e.g. descriptors which are for readability only. If the user changes the underlying value but not the descriptor the idea of having descriptors is lost. Furthermore, there are data which depend on the symmetry. `fedit` takes care of all of that.

Please note, that the underlying data representation is in `str` format. This allows to use `'1/3'` instead of `0.33333...` for real values. Hence, the natural return value of the data is `str`. If however an explicit `int` or `float` representation is wanted and if the type of the specific data conforms to these types an `int` value is returned by `PObj.L` (page 19) and a `float` by `PObj.D` (page 20).

Logicals are handled as `'f'` or `'t'` internally. Hence, `PObj.S` (page 20) returns these values. However, if returned by `PObj.L` (page 19), `True` and `False` are represented by `1` or `0` respectively. When writing a logical via `PObj.S` (page 20) `'t'`, `'T'` and `'+'` are considered to be `True` any other string as `False`. When writing a logical via `PObj.L` (page 19) `0` and any nonzero `int` represent `False` and `True`, respectively.

Character variables defined as `'char some_char='x';` in the `=.` file (which are not used currently) are returned as `str` by `PObj.S` (page 20) and as `int` by `PObj.L` (page 19). The latter interpretation is machine dependent and not recommended. Similarly, the variable can be written in both ways.

**fploExecutable** (*suffix*="")

**Parameters** *suffix* (*str*) – for convenience, *suffix* is appended to the exec name

Return the name of the fplo executable as `str`, which fits the pyfplo version. This is an educated guess. There is no guarantee that it is correct. If you compiled with a non-default build branch, this function does not return the name including the build branch suffix but just the generic name. Use *suffix* to select the correct executable or just write it out explicitly anyway.

## 2.2.1 INParser

### class INParser

This class is a low-level class for reading the `=.in` files. For manipulating `=.in` use the *pyfplo.fedit* (page 3) class. This is safer, since it ensures data (semantics) consistency.

To see how to use some of the low-level routines consult *Reading =.in files* (page 152), *Write =.in with low level routines* (page 158) and *=.files to json* (page 153).

Create an instance of the FPLO parser for files of type `=.in` in the following way:

```
import pyfplo.fploio as fploio
p=fploio.INParser()
```

#### parseFile (filename)

**Parameters** `filename` (*str*) – the filename of an existing file in FPLO `=.-format`

This opens and reads the file given in *filename*. Safe use:

```
import os
import pyfplo.fploio as fploio
p=fploio.INParser()
if not os.path.exists('=.in'):
    raise RuntimeError('cannot find the file')
try:
    p.parseFile("=.in")
except RuntimeError, ex:
    print(ex)
    pass # do whatever to handle the error or quit the program
```

#### writeFile (filename)

**Parameters** `filename` (*str*) – the filename of the new file in FPLO `=.-format`

Write the parser content to a file called *filename*.

#### \_\_call\_\_()

Return the root of the data tree. The returned object is an instance of class *PObj* (page 17), which can be used to access individual data.

#### varExists (varname)

**Parameters** `varname` (*str*) – the name of the queried variable

**Returns** existence of variable definition

**Return type** `bool`

Check if a variable definition exists in the parse table. Return `True` if it does. This function tests the struct array definition not the actual elements. Example:

```
d=p() # p is a INParser object and d is parser root
d2=d('wyckoff_positions')# d2 is node wyckoff_positions (struct array)
print d2[0].varExists('element') # -> True
print d.varExists('wyckoff_positions.element') # -> False but
print p.varExists('wyckoff_positions.element') # -> True
#also
print d.varExists('wyckoff_positions[1].element') # -> True
print d('wyckoff_positions')[1].varExists('element') # -> True
```

Compare this to `PObj.varExists` (page 17).

## 2.2.2 PObj

### class PObj

This class is a low-level class for reading the `=.in` files. For manipulating `=.in` use the `pyfpllo.fedit` (page 3) class. This is safer, since it ensures data (semantics) consistency.

To see how to use some of the low-level routines consult *Reading =.in files* (page 152), *Write =.in with low level routines* (page 158) and *=.files to json* (page 153).

Usually `PObj` (page 17) instances are returned by other methods. One does not create a `PObj` (page 17) by itself.

#### fullName()

Return the full node name. This includes element access (e.g. `nkxyz[3]`).

#### varExists(varname)

**Parameters** `varname` (*str*) – the name of the queried variable

**Returns** existence of variable definition

**Return type** `bool`

Check if a variable exists under the current node. Return `True` if it does. This function tests struct array elements in detail. Example:

```
d=p() # p is a INParser object and d is parser root
d2=d('wyckoff_positions') # d2 is node wyckoff_positions (struct array)
print d2[0].varExists('element') # -> True
print d.varExists('wyckoff_positions.element') # -> False but
print p.varExists('wyckoff_positions.element') # -> True
#also
print d.varExists('wyckoff_positions[1].element') # -> True
print d('wyckoff_positions')[1].varExists('element') # -> True
```

Compare this to `INParser.varExists` (page 16).

#### size(dim=1)

**Parameters** `dim` (*int*) – dimension

**Returns** size of the dimension

**Return type** `int`

If `PObj` (page 17) is array-like return its size‘

For struct arrays and rank-one arrays `size()` returns the array size.

For rank>1 arrays `size(dim)` returns the size of the *dim*-th dimension.

#### sizes()

**Returns** a list of all array sizes, such that `rank==len(n.sizes())`

**Return type** list of `int`

#### resize(size)

**Parameters** `size` (*int* or *list*) – can be a scalar `int` or a list of `int` in case of multi-dimensional arrays.

If `PObj` (page 17) is an array or struct array, resize it. Note, that for multidimensional arrays only the last dimension can be resized in the moment. But `=.in` does not use multidimensional arrays. (`=.dens` does though, which however should not be touched.)

#### \_\_call\_\_(name)

**Parameters** `name` (*str*) – name of the node

Return a new *PObj* (page 17) which references the node called *name* under the current *PObj* sparse tree node:

```
d=p() # parse tree root
ds=d('spin')
print ds('mspin').L
```

Compound names are possible:

```
d=p() # parse tree root
print d('spin.mspin').L
```

**\_\_getitem\_\_** (\*args)

If the node pointed to by *PObj* (page 17) represents a 1d array type the operator `[i]` returns the *PObj* (page 17), which represents the *i*-th element. For flag arrays the operator `[flagname]` returns the *PObj* (page 17), which represents this flag (if it exists). Use *S* (page 20) to print the full flag name and use *L* (page 19) to read or set the flags value. If *PObj* (page 17) represents a multidimensional array (up to 5d) the operator `[i, j, ...]` returns the corresponding *PObj* (page 17) of the element.

*args* can be:

*index1, index2, ...*: between one and five `int` indices:

```
dw2=d('wyckoff_positions')[2]
```

*flagname*: a single `str` which names a flag in a flag array:

```
d('options')['FULLBZ'].L=True
```

**name** ()

Return the node name. This excludes element access (e.g. `nkxyz[3]`).

**first** ()

**Returns** first node on the current level. See *=.files to json* (page 153)

**Return type** *PObj* (page 17)

**next** ()

**Returns** the next node on the current level. See *=.files to json* (page 153)

**Return type** *PObj* (page 17)

**hasNext** ()

**Returns** True if next node exists. See *=.files to json* (page 153)

**Return type** `bool`

**isScalar** ()

**Returns** True if node is a scalar. See *=.files to json* (page 153)

**Return type** `bool`

**isArray** ()

**Returns** True if node is an array. See *=.files to json* (page 153)

**Return type** `bool`

**isStruct** ()

**Returns** True if node is a struct. See *=.files to json* (page 153)

**Return type** `bool`

**isStructArray ()**

**Returns** True if node is a struct array. See [=.files to json](#) (page 153)

**Return type** bool

**isInt ()**

**Returns** True if node is a scalar int. See [=.files to json](#) (page 153)

**Return type** bool

**isReal ()**

**Returns** True if node is a scalar int. See [=.files to json](#) (page 153)

**Return type** bool

**isLogical ()**

**Returns** True if node is a scalar int. See [=.files to json](#) (page 153)

**Return type** bool

**isString ()**

**Returns** True if node is a scalar int. See [=.files to json](#) (page 153)

**Return type** bool

**isChar ()**

**Returns** True if node is a scalar int. See [=.files to json](#) (page 153)

**Return type** bool

**isFlag ()**

**Returns** True if node is a scalar int. See [=.files to json](#) (page 153)

**Return type** bool

**\_\_str\_\_ ()**

return printable representation. You do not need to call this explicitly. An object *obj* with this function provides usefull info when printed:

```
print(obj)
```

**listS**

If *PObj* (page 17) refers to a 1d array, return it's elements as a list of `str` or assign a list of `str` to the array elements. On assignment to a fixed size array the list must have the proper length On assignment to a variable size array, the array is resized accordingly. One can assign a list of `str` to integer and real arrays as long as the list's elements represent the correct type:

```
d('wyckoff_positions')[1]('tau').listD=[1./2,2./3,1./4]
d('wyckoff_positions')[1]('tau').listS=['1/2','2/3','1/4']
```

Note, the dot in `2./3`, which is needed to have `float` division while in the string version it is not allowed.

**listD**

If *PObj* (page 17) refers to a real 1d array, return the elements as a list of `float` or assign the list (see *listS* (page 19)).

**listL**

If *PObj* (page 17) refers to an integer 1d array, return is elements as a list of `int` or assign the list (see *listS* (page 19)).

**L**

If the current node refers to an integer type, return it as `int`. If it refers to a flag, 1 is returned if the flag is switched on, 0 otherwise. Assignment is possible:

```
d('spin.mspin').L=2
```

**D**

If the current node refers to a real type, return it as `float`. Assignment is possible:

```
d('wyckoff_positions')[0]('tau')[1].D=5.4
```

**S**

The current *PObj* (page 17) instance sits at a certain node of the data tree, depending on the history of calls, which created it.

If the current *PObj* (page 17) instance refers to a node, which represents a single value (scalar) the value is returned as `str` representation.

If *PObj* (page 17) refers to a flag the flagname followed by (+) or (-) is returned.

A new `str` value can be assigned to *S*. The user must ensure the correctness of the data, e.g. that a `str` representing an underlying `float` is actually a valid `float`.

Let us assume there is a real `tolerance=1e-12; in=.in`. Then one can set this value as:

```
# d is assumed to be the PObj under which the node tolerance sits.
d('tolerance').D=1e-13
# or
d('tolerance').S='1e-13'
```

## 2.2.3 FPLOInput

**class** `FPLOInput` (*fname=None*)

*FPLOInput* (page 20) is used to manage the manipulation of `=.in` files. It creates new files or reads existing files. The symmetry update functionality of **fed**it is hidden in this class. This is a low level interface. It is **strongly recommended** to use *pyfplo.fedit* (page 3) for manipulating data instead.

An example of proper usage is found in *Write =.in with mid level routines* (page 159).

**Parameters** `fname` – the name of an existing `=.in` file or `None` or nothing

Create a fresh *FPLOInput* (page 20) object via:

```
import pyfplo.fploio as fploio
...
fio=fploio.FPLOInput()
```

Create a fresh *FPLOInput* (page 20) object and read a file and update the version, if needed, or create fresh input if the file does not exist via:

```
import pyfplo.fploio as fploio
...
fio=fploio.FPLOInput('=.in')
```

which is equivalent to the following code:

```
import os
import pyfplo.fploio as fploio
...
fio=fploio.FPLOInput()
if os.path.exists('=.in'):
    fio.parseInFile(True, '=.in')
else:
    fio.createNewFileContent() # make fresh content
    ... # or handle this case differently
```

**parseInFile** (*forceupdate*, *fname*='.in', *newifnonexistent*=False)

Read the `=.` file called *fname* into an internal parse tree and optional update version or create fresh content. If something goes wrong an exception is raised.

#### Parameters

- **forceupdate** (*int*) – If `True`, convert older version files into new ones
- **fname** (*str*) – `=.in`-type file name
- **newifnonexistent** (*int*) – If `True` and the file called *fname* does not exist, create fresh content in the parse tree (does not create a file).

**symmetryUpdate** ()

After changing symmetry settings call this to update the rest of the data. A message is returned.

**resetNonSymmetrySections** ()

Reset all data in the internal parse table except for the symmetry section. This is used to reset all non symmetry data to default values.

**writeFile** (*name*)

**Parameters** **name** (*str*) – filename of `=.in`-type file

Write the parser content to the file called *fname*.

**structureFromCIFFile** (*ciffilename*, *wyckofftolerance*=1e-06, *symblockindex*=0, *symoptionindex*=0, *determinesyymetry*=False, *keeprotation*=False)

Read the cif-file called *ciffilename* and import the structure data into the parse tree.

To see how to use this consult [Reading cif-files](#) (page 155).

*wyckofftolerance* is used to convert approximations like 0.3333 into 1/3. If the round off error (as in 0.3333) is smaller than *wyckofftolerance* all numbers which are approximate fractionals  $n/d$ ,  $d$  in [1,12] are replaced by the fractional. This is especially necessary for hexagonal structures where an approximation like 0.3333 for 1/3 leads to trouble (doubling of atoms, missing symmetry operations and such).

Some cif files contain more than one data block. If several blocks are present and if several of these contain symmetry information *symblockindex* selects the corresponding symmetry block. By default *symblockindex* is 0, which selects the first such block. The available blocks are written to the output. An example could produce the following output:

```
-----
Blocks contained in cif file 'cg049782osi20040706_123518.cif':
-----
symmetry block No.   name
-----
                    global
0                   cu2as2o7
1                   cuasbeta
                    profile
-----
```

There are four blocks of which two contain structure/symmetry information. Possible values for *symblockindex* are 0 or 1.

In principle it is possible that a symmetry containing block has contradicting group information. Often the cif files contains a Hall-symbol, the xyz-operation symbol table and the space group number (or combinations of these). The space group number is not a good indicator, since it does not encode settings. Hence, we analyse the Hall- and xyz-symbols to determine the group. Only if both are missing we rely on the space group number. If, however, Hall- and xyz-symbols lead to different groups (rare but possible), we can select, which of the two to use. The possible symmetry options are written to the output and *symoptionindex* selects the corresponding option. By default *symoptionindex* is 0, which selects the first such option. The output of symmetry options could look like this:

```
Symmetry information for datablock: data_1
Space group number: 227
Name-Hall          : F 4d 2 3 -1d
xyz-symbol         : gives hallsymbol -F 4vw 2vw 3

Hall symbol: OK
xyz symbols: OK but not equivalent

The following symmetry options are available:
 0 'Hall symbol'
 1 'xyz-symbols'
```

You can see that the xyz symbols give a different setting than the Hall symbol. Hence, we have to chose *symoptionindex* from 0 or 1. You also understand that 227 is not a complete description.

If *determinesyymetry* is True the cif data are symmetry analysed before creating `=.in`.

If *determinesyymetry* is True and *keeprotation* is True the orientation of the cell is retained during symmetry analysis.

A possible way of importing a cif file works like this:

```
import pyfplo.fploio as fploio
import pyfplo.fedit as fedit

fio=fploio.FPLOInput('=.in')
fio.structureFromCIFFile('data.cif',wyckofftolerance=1e-4,
    symblockindex=0,symoptionindex=0)
fio.writeFile('=.in') # now we have created or update =.in from the cif_
↪file

fed=fedit.Fedit(recreate=True) # reset default input (except symmetry)
fed.iteration(n=100) # ... and more settings
fed.pipeFedit() # apply the settings
```

#### **reset()**

Reset the internal parse tree to nothing. The resulting *FPLOInput* (page 20) object is like a newly created one.

#### **createNewFileContent()**

Create a completely new default file in the internal parse tree.

#### **parser()**

**Returns** the underlying *INParser* (page 16)

**Return type** *INParser* (page 16)

## 2.2.4 OutGrep

**class OutGrep** (*outfilename='out', dir='.'*)

OutGrep helps to grep results from *fplo* output files directly into python variables.

#### **Parameters**

- **outfilename** (*str*) – the name of the *fplo* output file
- **dir** (*str*) – the directory in which it sits

This reads the whole output file, so that using *grep* (page 23) in a loop is more efficient.

- Example: grep last total energy and gap:

```
import pyfplo.fploio as fploio
...
og=fploio.OutGrep('out')
print('etot=',og.grep('EE')[-1],', gap=',og.grep('gap')[-1])
```

- Example: show iteration progress:

```
import os
import pyfplo.fploio as fploio
...
og=fploio.OutGrep('out')
with open('res','w') as fh:
    res=og.grep('it')
    for i,r in enumerate(res):
        fh.write('{} {}\n'.format(i,r))
os.system('xftp res')
```

- Example: grep all atom spins (as a function of iteration) and write them to file res:

```
import os
import pyfplo.fploio as fploio
og=fploio.OutGrep('out')
si=og.sites()

with open('res','w') as fh:
    for i,s in enumerate(si):
        site=i+1
        res=og.grep('SSat',site)
        fh.write('# {}{}\n'.format(s.element,site))
        for it,r in enumerate(res):
            fh.write('{} {}\n'.format(it,r))
        fh.write('\n')
os.system('xftp res')
```

More exmples in *Extract =.basdef from output file* (page 168)

## modes

**Type** dict of *mode:long-name*

class variable of all available *OutGrep* (page 22) modes as in `grepfplo -h`

```
for k in fploio.OutGrep.modes.keys():
    print('mode: {0:15s} : {1}'.format(k,fploio.OutGrep.modes[k]))
```

## sites()

return a list of *pyfplo.common.Site* (page 37), which contains info for sites in output file.

## grep(mode, site=1, orbital=0)

### Parameters

- **mode** (*str*) – one of the modes defined as keys in dict *OutGrep.modes* (page 23)
- **site** (*int*) – some modes need a site number (one-based, as in *fplo* output)
- **orbital** (*int*) – some modes (population analysis modes) need an orbital number. *orbital* is the one-based number of the wanted orbital in the order as printed in the population analysis. If *orbital* is out of range (e.g. 0) the total site population number is printed. For the mode *N\_gros* there is one more number than for *N\_net* and *S\_...*, which is the number of excess electrons of the site.

return a list of `str` of mode-dependent results for all iterations found in the output file. Some modes return a single result since it is not iteration dependent. If you need `float` results, convert like `float(og.grep('EE')[-1])` or `list(map(float, og.grep('EE')))`.

See, examples under *OutGrep* (page 22).

## 2.2.5 Basis

**class Basis** (*version=None, elementsoratomicnumbers=None, basdeffile=None*)

*Basis* (page 24) gives a low-level access to the basis definition. The basic operations are:

- Create a default basis for all sorts via input: `basis-version + list of elements/atomic numbers`.
- Optionally, read basis definitions from `=.basdef` (overwrites the default).
- Modify the basis thusly obtained.
- Write file `=.basdef`.

Consequently, at minimum one needs to know the list of elements. See examples *Extract default basis into =.basdef* (page 161) and *User defined basis (in =.basdef)* (page 163). If a `=.in` already exists one can obtain this list via:

```
import pyfplo.common as com
import pyfplo.fploio as fploio
p=fploio.INParser()
p.parseFile('=.in')
d=p() ('wyckoff_positions')
elements=[d[i] ('element') .S for i in range(d.size())]
# optionally:
atomicnumbers=list(map(lambda x: com.c_elements.index(x), elements))
print(elements)
print(atomicnumbers)
```

To create an FPLO9 default basis in `=.basdef` do this:

```
import pyfplo.fploio as fploio
...
b=fploio.Basis('default FPLO9 basis', elements)
# modify b if needed, then
b.writeFile('=.basdef')
```

To read and modify `=.basdef` do this:

```
import pyfplo.fploio as fploio
...
b=fploio.Basis('default FPLO9 basis', elements, basdeffile='=.basdef')
# modify b if needed, then
b.writeFile('=.basdef')
```

---

**Note:** Note, that you always need to provide the default version and the element/atomic number list, because some adjustments take place internally based on the elements.

---

### Parameters

- **version** (`int` or `str`) – `ID` (`int`) or unique search string (case sensitive) into version table. The version table is obtained via the class variable `pyfplo.fploio.Basis.versions` (page 25), which is a list of available basis versions, where each list member has two members: a unique `ID` (`int`) and a `str`, which can be used for searching.
- **elementsoratomicnumbers** (list of `str` and/or `int`) – list of element names or atomic numbers for each sort

- **basdeffile** (*str* or *None*) – read (and overwrite default) from file *basdeffile* if not *None*!

**versions**

class variable of all available Basis versions

**Type** list of 2-lists of structure [ID,versionname]

**writeFile** (*filename*='=.basdef')

**Parameters** **filename** (*str*) – the basis definitions file name. For use with *fplo* this needs to be =.basdef.

Write the basis definitions to file *filename*.

**\_\_getitem\_\_** ()

*basis*[*i*] returns basis (*BasDef* (page 25)) of sort *i*

## 2.2.6 BasDef

**class BasDef**

*BasDef* (page 25) contains the basis of a particular atom. This class is return by *pyfplo.fploio.Basis.\_\_getitem\_\_* (page 25) and cannot be instantiated otherwise.

**\_\_str\_\_** ()

return printable representation. You do not need to call this explicitly. An object *obj* with this function provides usefull info when printed:

```
print(obj)
```

**core**

**semicore**

**valence**

## 2.2.7 BasDefSection

**class BasDefSection**

**append** (*nl*, *Q=None*, *P=None*, *S=None*)

**Parameters**

- **nl** (*str*) – multiorbital name, e.g. '3d'
- **Q** (float or list of float-s) –
- **P** (float or list of float-s) –
- **S** (float or list of float-s) –

append a double 4f-orbital like D4f Q=(q1, q2) P=(p1, p2) via:

```
basdef.valence.append('4f', Q=[q1, q2], P=[p1, p2])
```

or a single 4f-orbital like s4f Q=(q1) P=(p1) via:

```
basdef.valence.append('4f', Q=q1, P=p1)
```

**remove** (*i*)

**Parameters** **i** (*int*) –

`__len__()`

`len(b)` returns the size of the Basis section (number of multi-orbitals).

`__getitem__()`

`BasDefSection[i]` returns the *i*-th MultiOrbital

`__str__()`

return printable representation. You do not need to call this explicitly. An object *obj* with this function provides usefull info when printed:

```
print(obj)
```

## 2.2.8 MultiOrbital

**class MultiOrbital**

This class is return by `pyfplo.fploio.BasDefSection.__getitem__` (page 26) and such

`append(Q=0.0, P=1.0, S=0.0)`

**Parameters**

- **Q** (*float*) –
- **P** (*float*) –
- **S** (*float*) –

`removeLast()`

`removeFirst()`

`qns(mult)`

**Parameters** `mult` (*int*) –

**Returns**

**Return type** `str`

`Q(mult)`

**Parameters** `mult` (*int*) –

**Returns**

**Return type** `float`

`S(mult)`

**Parameters** `mult` (*int*) –

**Returns**

**Return type** `float`

`P(mult)`

**Parameters** `mult` (*int*) –

**Returns**

**Return type** `float`

`set(mult, Q=None, P=None, S=None)`

**Parameters**

- **mult** (*int*) –
- **Q** (*float* or *None*) –
- **P** (*float* or *None*) –

- **S** (float or None) –

`__str__()`

return printable representation. You do not need to call this explicitly. An object *obj* with this function provides usefull info when printed:

```
print(obj)
```

**name**

**multiplicity**

## 2.2.9 Data

For convenience:

**version**

copy of `pyfplo.common.version` (page 40)

**Version**

copy of `pyfplo.common.Version` (page 39)

**c\_elements**

copy of `pyfplo.common.c_elements` (page 40)

## 2.3 pyfplo.common

- *BandFileContext* (page 27)
- *BandPlot* (page 29)
- *BandHeader* (page 32)
- *BandWeights* (page 33)
- *WeightDefinitions* (page 34)
- *WeightDefinition* (page 35)
- *OptionSet* (page 36)
- *Site* (page 37)
- *Watch* (page 37)
- *Version* (page 39)
- *Vlevel* (page 39)
- *Constants* (page 40)

This module contains a collection of usefull objects related to FPLO band/bandweights routines. You can easily write these files and read them into `numpy.ndarrays` for further processing. Have a look at `.../FPLO22.00-62/DOC/pyfplo/Examples/bandplot/model.py` for better understanding.

### 2.3.1 BandFileContext

**class BandFileContext**

This class wraps data to easily manage the creation of FPLO band/bandweight files. This class cannot be instantiated directly. It only is produced and returned via a call to `BandPlot.openBandFile()` (page 30)

Example:

```
# A simple band structure plotting using low level routines
# based on slabify.
import pyfplo.slabify as sla
import numpy as np
import numpy.linalg as LA

s=sla.Slabify()
s.dirname='.'
s.object='3d'
hamdata='+hamdata'
s.prepare(hamdata)

bp=sla.BandPlot()
bp.points=[
    ['$~G', [0, 0, 0]],
    ['X', [0.5, 0, 0]]
]
bp.ndiv=100
bp.calculateBandPlotMesh(s.dirname)
dists=bp.kdists
kpts=bp.kpnts

with bp.openBandFile(s.dirname+'/+b', s.nspin, len(kpts), \
                    progress='bandplot') as fb:

    # now fb is an instance of `BandFileContext`

    for ik, k in enumerate(kpts):
        for ms in range(s.nspin):
            Hk=s.hamAtKPoint(k*s.kscale, ms)
            (EV, C)=LA.eigh(Hk)
            fb.write(ms, dists[ik], k, EV)
```

**close()**

Explicitly close the file. Usefull, if multiple files are used in the same loop, in which case the `with`-statement is not usefull. The underlying file gets closed when this object gets garbage collected (after its scope is exited). For cleanliness it is a good measure to always close files.

Method1:

```
with bp.openBandFile(...) as f:
    doseomthing with f
# here f is closed
```

Method2:

```
f=bp.openBandFile(...):
do something with f
f.close()
# here f is closed
```

**write** (*ispin*, *dk*, *k*, *energies*, *weights=None*)

**Parameters**

- **ispin** (*int*) – spin number (NSP or FREL: 0, SP: 0 or 1)
- **dk** (*float*) – scalar path length along k-path
- **k** (*3-vector of float*) – k-point

- **energies** (*sequence (list, tuple, ...)*) – sequence of energies for all bands at this k-point
- **weights** – all weights for all bands at this k-point

Write the energies for spin component *ispin* and k-vector *k* and scalar path-length variable *dk* to the file. *ispin* must be 0 if there is only one spin. It must loop over [0,1] if there are two spins. See the *nspin* argument in `BandPlot.openBandFile()` (page 30). The spin loop must be inside the k-loop!

If a weight file is written (*weightlabels* argument in `BandPlot.openBandFile()` (page 30)) a matrix like argument (e.g. a `numpy.ndarray`) must be given as last argument to `write()` (page 28) whose rows correspond to the weights and the columns to the bands.

## 2.3.2 BandPlot

### class BandPlot

This is a helper class for data which control the path through the BZ for routines creating band structure or energy distribution cut plots. After setting the data (*points* (page 32) and *ndiv* (page 31)) `calculateBandPlotMesh()` (page 29) must be called before using `BandPlot` (page 29). Alternatively `readBandPlotMesh()` (page 30) can be called to import from `=.kp`.

This class is used by certain methods of `pyfplo.slabify.Slabify` (page 41).

Usage:

```
import pyfplo.slabify as sla
import pyfplo.common as com
s=sla.Slabify()

bp=com.BandPlot()
bp.points=[ ['$~G', [0,0,0]], ... ]
bp.ndiv=100
bp.calculateBandPlotMesh(s.dirname)

s.calculateBandStructure(bp)
```

This class also allows to read +band type files for further processing:

```
bp=com.BandPlot()
[bh,kdists,kptns,erg]=bp.readBands('+band')
print 'bandheader +b: nkp={0} nband={1} nspin={2}'\
      .format(bh.nkp,bh.nband,bh.nspin)
print 'kdists2 shape :',kdists2.shape
print 'kptns2 shape :',kptns2.shape
print 'erg shape :',erg.shape
# now do something with the information
```

Create a new `BandPlot` (page 29) instance via:

```
import pyfplo.slabify as sla
bp=sla.BandPlot()
```

or:

```
import pyfplo.common as com
bp=com.BandPlot()
```

`calculateBandPlotMesh` (*pointsfileoutputdir*)

Parameters `pointsfileoutputdir` (*str*) – directory in which to create +points

Finalize the bandplot definition. This actually calculates the k-points for the BZ-path from the input settings (*points* (page 32) and *ndiv* (page 31)). *pointsfileoutputdir* usually should be *Slabify.dirname* (page 54):

```
import pyfplo.slabify as sla
s=sla.Slabify()
...
bp.points=[...]
bp.calculateBandPlotMesh(s.dirname)
...
```

It serves to put the file `+points` in the right place.

**readBandPlotMesh** (*kpfilename*)

**Parameters** *kpfilename* (*str*) – name of an `xfplo =.kp`-type file

Read file `=.kp` (from `xfplo`) for mesh definition. *BandPlot* (page 29) restrictions (see *setOutputRestrictions()* (page 31)) are still applied and restrictions inside `=.kp` are ignored. IMPORTANT: do not use `=.in...` files from the Slabify output in context of visualizing the Fermi surface with `xfplo`. It will not work! Use the `=.in` files from the underlying FPLO calculation instead.”

**openBandFile** (*filename*, *nspin*, *nkpts*, *weightlabels=None*, *progress=None*)

**Parameters**

- **filename** (*str*) – the name of the `+band`-type file
- **nspin** (*int*) – number of spins
- **nkpts** (*int*) – number of k-points
- **weightlabels** – a list of labels (*str*) or `None`
- **progress** – a progress message (*str*) or `None`

**Returns** band file context

**Return type** *BandFileContext* (page 27)

Low level routine. Return an object of type *BandFileContext* (page 27) for creation of FPLO band files.

The returned object will open the file and organizes the proper file format. Its *BandFileContext.write()* (page 28) method can be used to write the actual data. If the object gets deleted (automatic if the scope is left) the file gets closed. The *BandFileContext.close()* (page 28) method can be called explicitly.

The best way to use it is in a `with`-statement. Then it is closed automatically after the `with`-block is exited:

```
with bp.openBandFile(...) as f:
    for ...:
        f.write(...)
pass # now the file is closed.
```

If multiple files are written at the same time one can do the following:

```
f1=bp.openBandFile(filename1,...)
f2=bp.openBandFile(filename2,...)
for ...:
    f1.write(data1,...)
    f2.write(data2,...)
f1.close()
f2.close()
pass # now the files are closed.
```

*nspin* is the number of effective spins:

not full relativistic:

- *nspin*=1 for non spin polarized
- *nspin*=2 for spin polarized

full relativistic:

- *nspin*=1

*nkpts* is the number of k-points.

If *weightlabels* is given and is a list of weight labels the resulting file will be a bandweight file.

If *progress* is set to a string a progress message is written in subsequent calls to `BandFileContext.write()` (page 28).

see help of `BandFileContext` (page 27).

**readBands** (*filename*)

**Parameters** *filename* (*str*) – name of a +band-type band structure file

**Returns** (*bh*, *kdists*, *kpnts*, *erg*), see below

**Return type** tuple

Read the bandfile (NOT bandweight file) called *filename* and return (*bh*, *kdists*, *kpnts*, *erg*) where

*bh* is an instance of `BandHeader` (page 32)

*kdists* is a 1d `numpy.ndarray` containing the k-path variable.

*kpnts* is a C-ordered 2d `numpy.ndarray` containing the k-vectors with dimension [*nkp*, 3].

*erg* is a 3d `numpy.ndarray` of energies, which is C-ordered, meaning that the last dimension is the innermost dimension. The three dimensions are `BandHeader.nspin` (page 32), `BandHeader.nband` (page 32) and `BandHeader.nkp` (page 32).

**on** ()

Activate bandstructure creation.

**off** ()

Deactivate bandstructure creation.

**setOutputRestrictions** (*active*, *ewindow*=[-20, 20], *offset*=[0, 0])

**Parameters**

- **active** (*int*) – restrictions are active
- **ewindow** (*sequence (list, tuple, ..)*) – a float list: [*emin*, *emax*]
- **offset** (*sequence (list, tuple, ..)*) – an int list: [*lower*, *upper*]

Convenience function: To reduce the size of the resulting files output restrictions can be set. If *active* (equivalent to `outputpartoccubands` (page 32)) is `True`, *ewindow* determines the energy interval of bands which are considered for output. This is a relatively crude algorithm, which usually only checks the band energies at the first point of the path. If more bands are needed it is often easier to additionally define a lower and upper band index *offset* (`partoccuoffset` (page 32)) which widens the interval of considered band indices. Positive numbers for both lower and upper offset make the interval wider. Negative numbers make it narrower. Be careful with these restrictions you might remove bands from the output without realizing it!

**active**

Make band structure routines active.

**ndiv**

Maximum number of points between two high symmetry points along the path in the Brillouin zone. The actual number in a path segment is adjusted such that the individual points are placed as equidistant as possible.

**lowerdepthdatalimit**

Limit the weight data written to files to the layers with `depth<=lowerDepthDataLimit` measured from the lower end of the finite (two-sided) slab. Beware that the default for both (lower/upper) limits is 1.0e30. So, if only one side's `datalimit` is set the other side's limit likely will still be big enough to enable all layers. Simply put, define both limits unless it is a semislab.

**upperdepthdatalimit**

Limit the weight data written to files to the layers with `depth<=upperDepthDataLimit` measured from the upper end of the finite (two-sided) slab or semi-finite (one-sided) semislab. Beware that the default for both (lower/upper) limits is 1.0e30. So, if only one side's `datalimit` is set the other side's limit likely will be big enough to enable all layers. Simply put, define both limits unless it is a semislab. For semislabs the *upperdepthdatalimit* determines which layers are used for the spectral density.

**outputpartoccubands**

See *setOutputRestrictions()* (page 31)

**partoccuoffset**

See *setOutputRestrictions()* (page 31)

**ewindow**

See *setOutputRestrictions()* (page 31)

**points**

A list of high symmetry points. Example:

```
bp.points=[
    ['$~G' , [0,0,0] ],
    ['X' , [0.5,0,0] ],
    ... ]
print bp.points
```

**kpnts**

Return a `numpy.ndarray` of the k-points along the path. The first index runs over the k-points. The number of points in each segment is proportional to the length of the interval. The max number is *ndiv* (page 31).

**kdists**

Return a `numpy.ndarray` of the scalar path-length variable along the path.

## 2.3.3 BandHeader

**class BandHeader**

The class is returned by *BandPlot.readBands* (page 31) or *BandWeights.readBandWeights* (page 34) and contains information about the header information in **fplo** +band or +bweights like files.

**\_\_str\_\_()**

return printable representation. You do not need to call this explicitly. An object *obj* with this function provides usefull info when printed:

```
print(obj)
```

**nkp**

The number of k-points.

**nband**

The number of bands.

**nspin**

The number of spins (full-relativistic – 1, otherwise – 1 or 2).

**norb**

The number of weights (if present)

**ilower**

The index of the lowest band present in the data.

**iupper**

The index of the highest band present in the data.

**labels**

A list of labels.

## 2.3.4 BandWeights

### class `BandWeights` (*infile*)

Use this to sum up weights contained in a bandweights file or to read the content of a weights file into numpy arrays. The resulting weights will be written to another bandweights file.

Example:

```
import pyfplo.common as com
wds=com.WeightDefinitions()
# add a new state (a single sum of certain weights)
w=wds.add(name="all")
# add labels
w.addLabels(labels=["Al (001) 3s+0", "Al (001) 3p-1", "Al (001) 3p+0",
                  "Al (001) 3p+1"], fac=1)
# or add via orbital info, if the bandweights file contains
# default weight labels.
w.addAtoms(atom='Al', sites=[1], orbitals=['3s', '3p'], fac=1)

print wds # want see what we did

# read a bandweights file (more general file example)
bw=com.BandWeights("+bweights")
# add weights and write to +bwsum
bw.addWeights(wds, '+bwsum')
```

For further information see: [WeightDefinitions](#) (page 34) and [WeightDefinition](#) (page 35)

`BandWeights(infile)` creates a new `BandWeights` (page 33) object with the input bandweights file name set to the string *infile*.

### `header()`

**Returns** header information

**Return type** `BandHeader` (page 32)

Return the header information of the underlying file.

### `addWeights(weightdefs, outfile, ewindow=[], vlevel=100)`

#### Parameters

- **weightdefs** (`WeightDefinitions` (page 34)) – the definitions for the resulting weights.
- **outfile** (*str*) – the name of the output file with resulting weights.
- **ewindow** (*sequence (list, tuple, ..)*) – optional, list of two `float`: [emin, emax]: the energy interval to which to restrict the output bands
- **vlevel** (*int*) – optional: verbosity level.

Take state definitions for resulting weights from the *weightdefs* (instance of *WeightDefinitions* (page 34)) which is similar to the definitions in the file `=.addwei` for `faddwei...` and create a new weights file *outfile*, with the weights added up according to the *weightdefs*.

Restrict number of bands in *outfile* according to the energywindow *ewindow*.

#### **readBandWeights ()**

**Returns** (*bh*, *kdists*, *erg*, *wei*), see below

**Return type** tuple

Read the bandweights file given as argument to *BandWeights* (page 33) and return (*bh*, *kdists*, *erg*, *wei*) where

*bh* is an instance of *BandHeader* (page 32)

*kdists* is a 1d `numpy.ndarray` containing the k-path variable.

*erg* is a 3d `numpy.ndarray` of energies, which is C-ordered, meaning that the last dimension is the innermost dimension. The three dimensions are *BandHeader.nspin* (page 32), *BandHeader.nband* (page 32) and *BandHeader.nkp* (page 32).

*wei* is a 4d `numpy.ndarray` of energies, which is C-ordered, meaning that the last dimension is the innermost dimension. The four dimensions are *BandHeader.nspin* (page 32), *BandHeader.nband* (page 32), *BandHeader.norb* (page 33) and *BandHeader.nkp* (page 32).

This function does not return the list of k-vectors, since these are not part of the weights file. If you need them and have a corresponding band file you could get them from there via *BandPlot.readBands* (page 31).

## 2.3.5 WeightDefinitions

### **class WeightDefinitions**

Helper class to collect input for *BandWeights.addWeights()* (page 33) or for usage in some *Slabify* (page 41) routines. This is the python equivalent of `faddwei...`

Example:

```
import pyfplo.common as com
wds=com.WeightDefinitions()
# get a new state
w=wds.add(name='all')
# add labels
w.addLabels(labels=['Al (001) 3s+0', 'Al (001) 3p-1', 'Al (001) 3p+0', 'Al (001) 3p+1'],
→fac=1)
# or add via orbital info
w.addAtoms(atom='Al', sites=[1], orbitals=['3s', '3p'], fac=1)

print wds
bw=com.BandWeights('+bweights')
bw.addWeights(wds, '+bwsum')

#fictitious more efficient example

wds=com.WeightDefinitions()
w=wds.add('plaquette')\
.addAtoms('Cu', [9, 10], ['3d'])\
.addAtoms('O', [1, 2], ['2p'])
```

For further information see *WeightDefinition* (page 35) and *BandWeights* (page 33)

*WeightDefinitions()* (page 34) creates a new *WeightDefinitions* (page 34) object.

`add (name=)`

**Parameters** `name (str)` – name of the new weight sum (*WeightDefinition* (page 35))

**Returns** a new weight definition

**Return type** *WeightDefinition* (page 35)

Add a new state definition (weight sum) called *name* to the collection of states and return an instance of *WeightDefinition* (page 35), which can be used to add specific input weights.

`__str__ ()`

return printable representation. You do not need to call this explicitly. An object *obj* with this function provides usefull info when printed:

```
print (obj)
```

## 2.3.6 WeightDefinition

**class WeightDefinition**

This collects information for a single weight (state) definition. This object cannot be created directly. Rather it is returned by *WeightDefinitions.add ()* (page 34).

`addLabels (labels, fac=1)`

**Parameters**

- **labels** (*list of str*) – a list of existing weight labels (as found in the input bandweights file header)
- **fac** (*float*) – the weights enter the weight sum with this factor

**Returns** current instance of *WeightDefinition*

**Return type** *WeightDefinition* (page 35)

This routine adds the labels to the current *WeightDefinition* (page 35) and returns the current instance of *WeightDefinition* (page 35), in order to allow constructions like:

```
w.addLabels (...) \
.addAtoms (...)
```

`addAtoms (atom, sites, orbitals, fac=1, spin=)`

**Parameters**

- **atom** (*str*) – element name
- **sites** (*sequence (list, tuple, ..)*) – list of site numbers (*int*)
- **orbitals** (*list of str*) – list of orbitals (e.g. ‘3d’ or ‘3d+1’ or ‘3d5/2-1/2’ or ‘all’)
- **fac** (*float*) – the weights enter the weight sum with this factor.
- **spin** (*str*) – ‘up’, ‘dn’, or ‘both’

**Returns** current instance of *WeightDefinition*

**Return type** *WeightDefinition* (page 35)

Defines a set of single orbital weights to be added to this state. Note, that in the full relativistic case *spin* must be specified if the orbitals in the input weight file refer to pseudo non-relativistic symmetries (e.g. ‘Pt(001)5d+1 up’).

It returns the current instance of *WeightDefinition* (page 35), in order to allow constructions like:

```
w.addAtoms(...)\
.addAtoms(...)\
.addLabels(...)
```

`__str__()`

return printable representation. You do not need to call this explicitly. An object *obj* with this function provides usefull info when printed:

```
print(obj)
```

## 2.3.7 OptionSet

### class OptionSet

A collection of options for debugging output. This class cannot be instantiated directly. It only is returned from objects, which have an *OptionSet* (page 36) member variable (see *Slabify.options* (page 55)). Example usage:

```
s=sla.Slabify()
op=s.options # a possible way to get an OptionSet object

print op # print the option list including their values
print op.names # print the available option names

for n in op.names: # python loop for option print
    print n,op[n]

for n in op.names: # python loop to set all options
    if n.startswith('prep'):
        op[n]=True

# or let us suppose there is an option called prep_pairs

op['prep_pairs']=True
```

`__getitem__(self, n)`

**Parameters** *n* (str) – option name

Return the value of the option *n* as int:

```
op=s.options # just an example
print op['some_option_name']
```

`__setitem__(self, n, value)`

**Parameters**

- *n* (str) – option name
- **value** (int or bool) – the options value (on or off)

Set the value of the option called *n*:

```
op=s.options # just an example
op['some_option_name']=True
```

`__str__()`

return printable representation. You do not need to call this explicitly. An object *obj* with this function provides usefull info when printed:

```
print(obj)
```

**names**

Return a list of available options:

```
s=sla.Slabify()
op=s.options # just an example
print op.names
```

## 2.3.8 Site

**class Site**

A list of instances of this class is returned by `pyfplo.slabify.Slabify.layerSites` (page 42) or `pyfplo.fploio.OutGrep.sites` (page 23).

**\_\_str\_\_()**

return printable representation. You do not need to call this explicitly. An object *obj* with this function provides usefull info when printed:

```
print(obj)
```

**element**

the atom name

**Type** str

**type**

the atom's type

**Type** int

**sort**

the atom's sort (Wyckoff position)

**Type** int

**tau**

the position of the atom

**Type** 3-vector

## 2.3.9 Watch

**class Watch** (*name*, *decimals*=2)

Measure time with this or print progress reports.

Example:

```
import pyfplo.common as com

nk=100
wa=com.Watch('task').setProgress(nk,0.02)
for ik in range(nk):
    wa.printProgress(ik)
    do_something()
    if ik%50 == 0: print(wa.status('50 steps done'))

print(wa.status('total'))
```

**Parameters**

- **name** (*str*) – a name for the task
- **decimals** (*int*) – how many decimals of time to print

**stop ()**

**Returns**

self to allow call chaining as in

```
Watch.stop(...).setSomethingElse(...)
```

**Return type** *Watch* (page 37)

Halt the clock.

**go ()**

**Returns**

self to allow call chaining as in

```
Watch.go(...).setSomethingElse(...)
```

**Return type** *Watch* (page 37)

Let it run/continue.

**reset ()**

**Returns**

self to allow call chaining as in

```
Watch.reset(...).setSomethingElse(...)
```

**Return type** *Watch* (page 37)

Reset the clock to zero.

**status** (*txt='total', prefix=", suffix="*)

**Parameters**

- **txt** (*str*) – some additional information
- **prefix** (*str*) – some additional information
- **suffix** (*str*) – some additional information

**Returns** **status** – a printable version of the elapsed time

**Return type** *str*

Return a printable string of the time which elapsed while the clock was running (not stopped).

**setProgress** (*steps, delta=0.1*)

**Parameters**

- **steps** (*int*) – total number of steps in the task to be progress reported on
- **delta** (*float*) – print progress information if this amount of time of the estimated total has passed. `delta=0.1` means every 10%.

**Returns**

self to allow call chaining as in

```
Watch.setProgress(...).setSomethingElse(...)
```

**Return type** *Watch* (page 37)

Set the parameters of a progress report.

**printProgress** (*istep, prefix='V', text=", suffix="*)

**Parameters**

- **istep** (*int*) – the current step of the running task

- **prefix** (*str*) – some additional info
- **text** (*str*) – some additional info
- **suffix** (*str*) – some additional info

Print the progress with an estimate of the total time needed to complete the task.

### 2.3.10 Version

#### class Version

This class manages the version numbers of pyfplo. The easiest use is:

```
import pyfplo.common as com
...
print 'pyfplo version ',com.version
# one can protect scripts in the following way:
if com.version!='22.00': raise RuntimeError('pyfplo version is incorrect.')
```

*version* is also bound as a module variable in `pyfplo`, `pyfplo.slabify` (page 41) and `pyfplo.fploio` (page 15) such that the example above could read:

```
import pyfplo.slabify as sla
...
print 'pyfplo version '+str(sla.version)
# one can protect scripts in the following way:
if sla.version!='22.00': raise RuntimeError('pyfplo version is incorrect.')
```

**release** ()

**Returns** release number

**Return type** str

**mainVersion** ()

**Returns** main version number

**Return type** str

**\_\_eq\_\_** ()

Compare the main version (for code sanity purposes). Example:

```
if com.version!='22.00': raise RuntimeError('pyfplo version is incorrect.')
```

**\_\_ne\_\_** ()

Compare the main version (for code sanity purposes). Example:

```
if com.version!='22.00': raise RuntimeError('pyfplo version is incorrect.')
```

**\_\_str\_\_** ()

return printable representation. You do not need to call this explicitly. An object *obj* with this function provides usefull info when printed:

```
print (obj)
```

### 2.3.11 Vlevel

#### class Vlevel

This class merely defines the verbosity level constants (Silent, Info,..., All). You can use any int where ever a vlevel is needed as an argument. For an arbitrary int N as argument the actual vlevel is set to the largest constant (defined below), which is  $\leq N$  or to 0 (Silent) if  $\text{int} < 0$ ; One can use the constants:

```
print com.Vlevel.All
```

**Silent** = 0

**Info** = 100

**More** = 200

**Many** = 300

**All** = 1000

### 2.3.12 Constants

A collection of physical constants.

**c\_abtoang**

Bohr radii / Angstroem

**c\_hatoev**

Hartree/eV

**c\_speed\_of\_light\_mpers**

speed of light in m/s

**c\_hbar Js**

Planck constant/2pi in Js

**c\_me\_kg**

electron mass in Kg

**c\_angstroem\_m**

angstroem in m

**c\_echarge\_C**

electron charge in C

**c\_elements**

all elements

Predefined:

**version**

instance of *Version* (page 39)

## 2.4 pyfplo.slabify

- *Slabify* (page 41)
- *BoxMesh* (page 56)
- *EnergyContour* (page 59)
- *FermiSurfaceOptions* (page 60)
- *DensPlotContext* (page 63)
- *GreenOptions* (page 63)
- *WeylPoint* (page 64)
- *BfieldConfig* (page 65)
- *WFSymOp* (page 66)

- [BerryCurvatureData](#) (page 67)
- [Site](#) (page 67)
- [Data](#) (page 67)

This is a collection of routines to map Wannier Hamiltonians onto larger structures, create Fermisurfaces, cuts, spectral densities and more. Some objects which are needed are defined in `pyfplo.common` (page 27) but are also accessible from this module via aliases of the same name.

To get help on all of those objects use:

```
import pyfplo.common as com
help(com)
```

or if `pydoc` is installed: `pydoc pyfplo.common`

Please have a look at the [Examples](#) (page 75).

To better understand the structure manipulation performed consult [Structure Manipulation Algorithm](#) (page 85).

## 2.4.1 Slabify

### `class Slabify`

The main object to access all procedures. There are low level procedures to extract the tight-binding Hamiltonian directly as well as high level routines, which perform various tasks.

---

**Note:** do not use `=.in_...` files from [Slabify](#) (page 41) output in context of `xfplo` fermi surfaces.

---

Here is how [Slabify](#) (page 41) works.

The structure is taken from the Wannier Hamiltonian file, which usually is called `+hamdata`. It also contains the Hamiltonian matrix elements and optionally spin operator matrix elements.

There are several options to change the structure into something new. After the structure setup the Hamiltonian matrix elements are mapped onto this new structure.

Three structure types are available (see [object](#) (page 55)):

'3d' (bulk), 'slab' (finite slab) and 'semislab' (semi infinite slab).

Depending in the structure type various operations can be performed to obtain the new structure. The resulting structure after each particular operation is saved in `=.in_...` files in the local result directory (stored in [Slabify.dirname](#) (page 54), default 'slabifyres'). The operations available for the different structure types are listed next:

- 3d/slab/semislab
  - enlargement** Use the matrix in [enlarge](#) (page 55) to construct a larger 3d lattice basis.
- slab/semislab:
  - layering** Define [zaxis](#) (page 55) to determine the direction perpendicular to the surface. The 3d cell will be transformed such that the a- and b-axis are perpendicular to [zaxis](#) (page 55). Note, that the c-axis might be inclined towards the [zaxis](#) after this step.
  - anchoring** Define [anchor](#) (page 55) in relative z-coordinates of the layered cell. At this position the 3d-solid is cut (vacuum being inserted). Load the `=.in_...` files into `xfplo` to orient yourself and to find a proper anchor.
- slab
  - cutting layers** After anchoring the third (z) coordinate is in absolute units. Define [cutlayersat](#) (page 55) as the lowest and highest z-coordinate of the slab. Everything outside will be removed. If the interval is inverted cutting will not be applied.

**cutting atoms** Supply a list of atoms (sites) in *cutatoms* (page 55) to be removed from the result of cutting layers.

(see *Structure Manipulation Algorithm* (page 85))

---

**Note:** All data members can be set by simple assignment. A returned data member is returned as copy:

```
s=sla.Slabify()
a=s.zaxis # a copy of s.zaxis
s.zaxis=[1,1,0] # set s.zaxis
```

---

**Note:** Do not use `.in_...` files from the Slabify output in context of the **xfplo** fermi surfaces.

---

**hamdataCell()**

**Returns cell** – The individual vectors are the columns of the returned matrix.

**Return type** 3x3 `numpy.ndarray`

Return the primitive unit cell vectors of the underlying data (see *prepare* (page 42)).

**hamdataCCell()**

**Returns cell** – The individual vectors are the columns of the returned matrix.

**Return type** 3x3 `numpy.ndarray`

Return the conventional unit cell vectors of the underlying data (see *prepare* (page 42)). For rhombohedral lattices the conventional lattice is the trigonal (hexagonal) cell if the spacegroup setting is hexagonal or the same as the primitive rhombohedral cell if the spacegroup setting is rhombohedral.

**hamdataRCell()**

**Returns cell** – The individual vectors are the columns of the returned matrix.

**Return type** 3x3 `numpy.ndarray`

Return the primitive reciprocal unit cell vectors of the underlying data (see *prepare* (page 42)). The vectors are in units of *kscale* (page 56).

For better understanding the following code does the same thing:

```
# assume s is a Slabify instance, LA is numpy.linalg, np is numpy
A=s.hamdataCell()
# Next the reciprocal cell is the inverse-transpose of A times 2*Pi
# Use internal scale though
G= LA.inv(A.T) * (2*np.pi) / s.kscale
# now G is equal to s.hamdataRCell()
```

**layerCell()**

**Returns cell** – The individual vectors are the columns of the returned matrix.

**Return type** 3x3 `numpy.ndarray`

Return the primitive unit cell vectors of the primary layer.

**layerSites()**

**Returns sites**

**Return type** a list of *Sites* (page 67)

Return a copy of the list of *Sites* (page 67) of the primary layer.

**prepare** (*hamdatafilename*)

**Parameters** `hamdatafilename` (*str*) – name of (WF) Hamiltonian data file (usually +hamdata)

Setup structure and map hopping data.

**printStructureSettings** ()

Print a summary of all structure settings.

**calculateBandStructure** (*bandplot*, *suffix=""*)

**Parameters**

- **bandplot** (`BandPlot` (page 29)) – see help of `pyfplo.common.BandPlot` (page 29)
- **suffix** (*str*) – a file suffix for convenience

Calculate the bandstructure for 'slab' and '3d' along the path defined by *bandplot* and produce corresponding files with a file *suffix* appended.

**calculateBulkProjectedEDC** (*bandplot*, *energymesh*, *zaxis=[0, 0, 1]*, *nz=30*, *kzs=None*, *suffix=""*, *query=False*)

**Parameters**

- **bandplot** (`BandPlot` (page 29)) – see help of `pyfplo.common.BandPlot` (page 29)
- **energymesh** (`EnergyContour` (page 59)) – see help for `EnergyContour` (page 59)
- **zaxis** (*3-vector of float*) – a vector in cartesian coordinates, which describes the projection direction along which the spectral density is k-integrated. This direction must be parallel to a reciprocal lattice vector, since we need periodicity in the projection direction in order to define a proper integration interval.
- **nz** (*int*) – the number of integration intervals
- **kzs** (sequence of *float*) – specify this instead of *nz* to build your own non-uniform integration mesh. You could for instance have a fine mesh in some region and a course one in the rest of the integration interval.
- **suffix** (*str*) – a suffix to alter the file name.
- **query** (*bool*) – if this is True the function returns a float without calculating much, indicating the length of the integration interval in units of *kyscale* (page 56). Use this to build your own non-uniform integration net. Call `calculateBulkProjectedEDC` (page 43) with *kzs* set to a list or numpy ndarray of kz-values in the desired interval. You can do what you want. The queried interval length just indicates the periodicity in the *zaxis* direction in said units.

**Returns** `kzlength` – The length of the integration interval along the projection direction

**Return type** `float`

Calculate bulk projected band energy distribution curves. This is the 1d-integral of the spectral density over a single period in k-space along a direction, indicated by *zaxis*. The *bandplot* input defines a set of high-symmetry points, which should be in a 2d-projection plane perpendicular to the *zaxis*. (The latter restriction is not really needed.)

The integration interval is from 0 to some number, which is determined by inspecting the *zaxis* and the lattice. If a user defined mesh is given in *kzs* the integration goes from  $k+zaxis*kzs[0]$  to  $k+zaxis*kzs[\text{len}(kzs)-1]$ , where *k* is a k-point along the path spanned by the high symmetry points in *bandplot*.

**calculateBulkProjectedFS** (*fso*, *zaxis=[0, 0, 1]*, *nz=30*, *kzs=None*, *suffix=""*, *query=False*)

**Parameters**

- **fso** (*FermiSurfaceOptions* (page 60)) – see help for *FermiSurfaceOptions* (page 60)
- **zaxis** (*3-vector of float*) – a vector in cartesian coordinates, which describes the projection direction along which the spectral density is k-integrated. This direction must be parallel to a reciprocal lattice vector, since we need periodicity in the projection direction in order to define a proper integration interval.
- **nz** (*int*) – the number of integration intervals
- **kzs** (sequence of *float*) – specify this instead of *nz* to build your own non-uniform integration mesh. You could for instance have a fine mesh in some region and a course one in the rest of the integration interval.
- **suffix** (*str*) – a suffix to alter the file name.
- **query** (*bool*) – if this is True the function returns without calculating much with a float indicating the length of the integration interval in units of *kscale* (page 56). Use this to build your own non-uniform integration net. Call *calculateBulkProjectedFS* (page 43) with *kzs* set to a list or numpy ndarray of kz-values in the desired interval. You can do what you want. The queried interval length just indicates the periodicity in the *zaxis* direction in said units.

**Returns** **kzlength** – The length of the integration interval along the projection direction

**Return type** *float*

Calculate bulk projected Fermi surface projections. This is the 1d-integral of the spectral density over a single period in k-space along a direction, indicated by *zaxis*. The *fso* input defines a 2d mesh in a plane perpendicular to the projection axis, a Fermi energy and an imaginary energy part. The integration interval is from 0 to some number, which is determined by inspecting the *zaxis* and the lattice. If a user defined mesh is given in *kzs* the integration goes from  $k+zaxis*kzs[0]$  to  $k+zaxis*kzs[len(kzs)-1]$ , where *k* is a k-point in the 2d mesh (from *fso*).

**calculateFermiSurfaceCuts** (*fso*, *wds=None*, *bandplot=None*, *suffix=""*, *forcerecalculation=False*)

#### Parameters

- **fso** (*FermiSurfaceOptions* (page 60)) – see help for *FermiSurfaceOptions* (page 60)
- **wds** (*WeightDefinitions* (page 34)) – see help for *pyfplo.common.WeightDefinitions* (page 34)
- **bandplot** (*BandPlot* (page 29)) – see help for *pyfplo.common.BandPlot* (page 29) Only *lowerdepthdatalimit* and *upperdepthdatalimit* and *bandplot* restrictions (*pyfplo.common.BandPlot.setOutputRestrictions* (page 31)) are used
- **suffix** (*str*) – a string which is appended the the end of the output file +*cuts\_spin1* (and +*cuts\_spin2*) *basename*.
- **forcerecalculation** (*bool*) – If a re-calculation is wanted use this argument or follow the file deletion hint of the program output.

Construct Fermi surface cuts. The cuts are calculated in two steps.

**diagonalization** For each point of the grid defined by *FermiSurfaceOptions* (page 60) the Hamiltonian is diagonalized. The results are written to files +*cut\_band\_sf* and +*cut\_bweights\_sf*. This step is costly.

**iso line determination** In this step the files mentioned above are read back in and from it the iso-lines corresponding to the current Fermi energy (*FermiSurfaceOptions.fermienergy* (page 62)) are determined. The results are written to +*cuts\_spin1* (and +*cuts\_spin2*), *suffix* is appended to these files *basename*. One can change

the fermi energy or the weights definitions and re-run the iso line step without diagonalization by just running `calculateFermiSurfaceCuts` (page 44) again with `forcerecalculation=False`

`calculateEDC` (*bandplot*, *energymesh*, *penetrationdepth*=-1.0, *greenoptions*=None, *suffix*="")

#### Parameters

- **bandplot** (`BandPlot` (page 29)) – see help for `pyfplo.common.BandPlot` (page 29)
- **energymesh** (`EnergyContour` (page 59)) – see help for `EnergyContour` (page 59)
- **penetrationdepth** (*float*) – define to which depth the spectral density is collected. *penetrationdepth* is measured from the atom closest to the vacuum. When interpreting output messages: orbital indices are increasing with the z-coordinate of the atom:
  - Positive values mean depth in `+hamdata` length units (usually Bohr radii).
  - Negative values mean depth in number of blocks. A block is the cell in `=.in_final_PLlayer`
- **greenoptions** (`GreenOptions` (page 63)) – see help for `GreenOptions` (page 63)
- **suffix** (*str*) – the output file suffix

Calculate energy distribution curves (EDC) along the path defined by *bandplot*. These are k,energy resolved surface spectral densities.

`calculateFermiSurfaceSpectralDensity` (*fso*, *penetrationdepth*=-1.0, *greenoptions*=None, *suffix*="")

#### Parameters

- **fso** (`FermiSurfaceOptions` (page 60)) – see help for `FermiSurfaceOptions` (page 60)
- **penetrationdepth** (*float*) – define to which depth the spectral density is collected. *penetrationdepth* is measured from the atom closest to the vacuum. When interpreting output messages: orbital indices are increasing with the z-coordinate of the atom.
  - Positive values mean depth in `+hamdata` length units (usually Bohr radii).
  - Negative values mean depth in number of blocks. A block is the cell in `=.in_final_PLlayer`
- **greenoptions** (`GreenOptions` (page 63)) – see help for `GreenOptions` (page 63)
- **suffix** (*str*) – the output file suffix

Calculate the surface Fermi surface (k,k resolved spectral density) for a particular Fermi energy.

`orbitalIndicesByDepth` (*lowerdepthdatalimit*=1e+30, *upperdepthdatalimit*=1e+30)

**Returns** list of orbital indices

**Return type** `numpy.ndarray`

Return a list of orbital (WF) indices for orbitals which belong to layers of maximum depth *lowerdepthdatalimit* and *upperdepthdatalimit* measured from the lower (upper) end of the slab.

`orbitalNames` (*orbitalindices*=None)

**Parameters** **orbitalindices** (sequence of `int`) – a sequence of valid orbital indices as e.g. returned by `orbitalIndicesByDepth` (page 45).

**Returns** list of orbital names

**Return type** list

Return a list of orbitalnames. Optionally, give a list of orbital indices to narrow down the returned list.

### Example

```
import pyfplo.slabify as sla

s=sla.Slabify()
...
s.prepare('+hamdata')

# This is a list of all orbitals upto a depth of 5 Bohr radii
# at the upper side of a slab.
upper=s.orbitalNames(s.orbitalIndicesByDepth(-1,5))
# or simply
upper=s.orbitalNamesByDepth(-1,5)

# This is a list of all orbitals upto a depth of 5 Bohr radii
#at the lower side of a slab.
lower=s.orbitalNames(s.orbitalIndicesByDepth(5,-1))
#
# Here comes the rest of the orbitals.
# (A nice python trick to get the rest list)
rest=list(set(s.orbitalNames())-set(upper)-set(lower))
# this can also be done with index lists
iupper=s.orbitalIndicesByDepth(-1,5)
ilower=s.orbitalIndicesByDepth(10,-1)
iall=s.orbitalIndicesByDepth()
irest=list(set(iall)-set(iupper)-set(ilower))
print 'the orbital indices lists:',ilower,irest,iupper
upper=s.orbitalNames(iupper)
lower=s.orbitalNames(ilower)
rest=s.orbitalNames(irest)
print 'the orbital lists:',lower,rest,upper
```

**orbitalNamesByDepth** (*lowerdepthdatalimit=1e+30, upperdepthdatalimit=1e+30*)

**Returns** list of orbital names

**Return type** list

Convenience function which is equivalent to:

```
orbitalNames(orbitalIndicesByDepth(lowerdepthdatalimit,
↪lowerdepthdatalimit))
```

**orbitalIndicesBySite** (*isite*)

**Parameters** *isite* (*int*) – the site number

**Returns** list of orbital indices

**Return type** numpy.ndarray

Return a list of orbital (WF) indices for orbitals which belong to site number *isite*.

**wannierCenterMatrix** ()

**Returns** a list of 3 matrices each containing the cartesian component of the Wannier centers (site vectors) in the diagonal.

**Return type** list of 3 numpy.ndarray

If  $\vec{s}$  is the site in the unit cell on which the Wannier function  $w_{\vec{R}\vec{s}n}$  sits, `wannierCenterMatrix` (page 46) returns  $\vec{M}_{\vec{s}'n',\vec{s}n} = \delta_{\vec{s}',\vec{s}}\delta_{n',n}\vec{s}$  where  $n', n$  are orbital indices.

`calculateBerryCurvatureOnBox` (*box*, *homo*, *suffix*=", *fullF*=False, *toldegen*=1e-10)

#### Parameters

- **box** (*BoxMesh* (page 56)) – see help for *BoxMesh* (page 56)
- **homo** (*int*) – The berry curvature is calculated for a set of bands. This sets the band number of the highest band included in this set.
- **suffix** (*str*) – the output file suffix
- **fullF** (*int*) – if the position operator (basis connection) is contained in +hamdata the full Berry curvature is calculated if *fullF* is True
- **toldegen** (*float*) – levels closer than this tolerance are considered to be a degenerate subspace in the non-Abelian correction to the Berry curvature

Output of the Berry curvature on a box mesh (optionally integrated over the third box direction). The non-Abelian expression is used for degenerate subspaces and a symmetry-restoring correction in periodic gauge is applied. The curvature corresponds to  $A^k = -i \langle u | \nabla u \rangle$ .

Now, a correction for the non-Abelian Berry curvature is applied for degenerate subspaces. Which levels are considered degenerate is controlled by *toldegen*. If it is too small, spurious results can be obtained. If it is very large, some of the actual curvature will be missing. *toldegen* should be smaller than the smallest physical gap in the band structure.

See `berryCurvature` (page 52) for details about the Berry curvature calculation.

`calculateChernNumberInSphere` (*center*, *radius*=0.1, *nsubdiv*=20, *homo*=1, *suffix*=", *nradius*=1, *fullF*=False, *toldegen*=1e-10)

#### Parameters

- **center** (*3-vector of float*) – the sphere center
- **radius** (*float*) – the sphere radius
- **nsubdiv** (*int*) – the mesh subdivision level
- **homo** (*int*) – The berry curvature is calculated for a set of bands. This sets the band number of the highest band included in this set
- **suffix** (*str*) – the output file suffix
- **nradius** (*int*) – if given and >1 include a radial-mesh in the output of the Berry curvature. The result is volumetric instead of 2d-on-sphere data.
- **fullF** (*int*) – if the position operator (basis connection) is contained in +hamdata the full Berry curvature is calculated if *fullF* is True
- **toldegen** (*float*) – levels closer than this tolerance are considered to be a degenerate subspace in the non-Abelian correction to the Berry curvature

Calculate the Chern number within a sphere. If you suspect the presence of a Weyl point first it is good to find its position. Then a small sphere can be put around the Weyl point position. The integral of the Berry curvature over this sphere gives the associated Chern number (Chirality). The output shows the Chern numbers for the set of the lowest n bands if it is larger than some tolerance, where n runs over all bands. This means that it is not the band wise Chern number but the Chern number for various sets of occupied bands with corresponding homo. If the shown number is far from integer try a finer *nsubdiv*. The number will converge to an integer eventually. Usually these spheres have to be rather small, especially if the Berry curvature is very structured. Note, that center and radius are in units of *k*scale (page 56).

Now, a correction for the non-Abelian Berry curvature is applied for degenerate subspaces. Which levels are considered degenerate is controlled by *toldegen*. If it is too small, spurious results can be

obtained. If it is very large, some of the actual curvature will be missing. *toldegen* should be smaller than the smallest physical gap in the band structure.

See *berryCurvature* (page 52) for details about the Berry curvature calculation.

**calculateZ2Invariant** (*gamma0*, *gamma1*, *gamma2*, *Nint*, *Nky*, *homos*, *efhomo=0*, *xmesh=None*, *ymesh=None*, *suffix=""*, *gauge='periodic'*)

#### Parameters

- **gamma0** (*3-vector of float*) – marks the center of the 2D plane in units of *kyscale* (page 56)
- **gamma1** (*3-vector of float*) – in units of *kyscale* (page 56) The integration direction is  $\Gamma_0 \rightarrow \Gamma_1$
- **gamma2** (*3-vector of float*) – in units of *kyscale* (page 56) The ky-parameter of the Wannier centers runs along  $\Gamma_0 \rightarrow \Gamma_2$ .
- **Nint** (*int*) – gives the subdivision for the integration direction
- **Nky** (*int*) – the subdivision of the ky-parameter direction.
- **homos** (*list or single int*) – the band indices, which form the highest occupied bands.
- **efhomo** (*int*) – One can specify the band below the Fermi energy as *efhomo* which is only used in output for orientation.
- **xmesh** (*sequence of float*) – if given it must contain a grid in [-1,1], which will be used as integration grid instead of the default equidistant one. If given *Nint* is ignored.
- **ymesh** (*sequence of float*) – if given it must contain a grid in [0,1], which will be used as ky-parameter grid instead of the default equidistant one. If given *Nky* is ignored. If the first point equals 0 and/or the last equals 1, the points are slightly shifted inwards to avoid particular problem cases.
- **suffix** (*str*) – a convenience suffix to be appended to the created files.
- **gauge** (*str*) – Can be 'periodic' (default) or 'relative'. The relative gauge seems to preserve the symmetry of the Wannier center curves, while the periodic does not. In pyfplo version  $\leq 18.00$  the periodic gauge was implemented. Formally, the relative gauge should be correct. This option was not tested a lot. The topological invariants should NOT depend on the gauge choice though, unless there is no gap. It seems that the Wannier center curves spread out more evenly in relative gauge, which makes the automatic index determination less reliable. So, visual checks should always be performed.

Calculate the Z2 invariant for a plane spanned by the TRIM points  $\Gamma_0$ ,  $\Gamma_1$  and  $\Gamma_2$  via Wannier centers.

All TRIM points must be actual points in the BZ (not directions). The plane spanned by the TRIMS shall only contain  $\Gamma_0$  in the center and the other TRIMS at the corners and mid-edges. No other TRIMS shall fall inside the planar cell. It is necessary that the whole plane is gapped for the results to make sense. The TRIM points are always given with respect to the default 3d cell as defined by the data in *+hamdata*. In other words if you use *enlarge* (page 55) the TRIM points must not be given with respect to the enlarged cell! See *calculate3dTIInvariants* (page 49) for more explanations.

Example with *xmesh/ymesh*:

```
import numpy as np
...
s=sla.Slabify()
...
Gp=s.hamdataRCell()
G=[0,0,0]
Z=Gp.dot(np.array([0,0,1])/2.)
```

(continues on next page)

(continued from previous page)

```

X=Gp.dot(np.array([1,0,0])/2.)
...
Nint=20
Nky=200
xmesh=map(lambda x: np.sign(x)*x**2,np.linspace(-1,1,Nint))
if False: # or
    xmesh=map(lambda x: x**3,np.linspace(-1,1,Nint))

ymesh=np.append(np.linspace(0,0.2,int(0.7*Nky),endpoint=False),
                np.linspace(0.2,1.,int(0.3*Nky)))
if False: # or
    ymesh=map(lambda x: np.sign(x)*x**2,np.linspace(0,1,Nky))

s.calculateZ2Invariant(G,X,Z,Nint,Nky,[14,16,18,20],xmesh,ymesh)

```

**calculate3dTIInvariants** (*Nint*, *Nky*, *homos*, *efhomo*=0, *gauge*='periodic')

#### Parameters

- **Nint** (*int*) – the number of intervals along the integration direction.
- **Nky** (*int*) – the number of intervals along the ky-parameter direction.
- **homos** (*list* or *single int*) – a list or a single homo can be given. A homo is the number of the highest band, assumed to be occupied. To find out the band numbers (*homos*) first make a 3d band calculation using [calculateBandStructure](#) (page 43) and load the resulting `+band...` into `xfbp`: right click on a desired band and the set numbers and band numbers of the bands nearby are displayed. The band number is the important one, not the set number!
- **efhomo** (*int*) – One can specify the band below the Fermi energy as *efhomo* which is only used in output for orientation.
- **gauge** (*str*) – Can be 'periodic' (default) or 'relative'. The relative gauge seems to preserve the symmetry of the Wannier center curves, while the periodic does not. In pyfplo version  $\leq 18.00$  the periodic gauge was implemented. Formally, the relative gauge should be correct. This option was not tested a lot. The topological invariants should NOT depend on the gauge choice though, unless there is no gap. It seems that the Wannier center curves spread out more evenly in relative gauge, which makes the automatic index determination less reliable. So, visual checks should always be performed.

Use Wannier centers to calculate the Z2 invariants for a 3d TI.

For the whole thing to make sense the bulk band structure must be gapped above the band numbered by the *homos* throughout the whole BZ. The output will be a set of files, which have to be inspected by the user to decide how many Wannier centers cross a chosen horizontal reference line. There is also an automatic algorithm to determine the invariants (printed to output) This is, however, not always correct: a sufficiently large number of integration (*Nint*) and parameter points (*Nky*) are needed to obtain a valid result. Especially, if the wannier center curves vary fast in some parameter regions *Nky* must be sufficiently high for the automatic algorithm for the determination of the Z2 invariants to be correct. The reason are many small gaps and/or highly fluctuating Berry curvature.

The programm creates data files `+Z2_homo..._...` containing the Wannier centers where the last suffix indicates in which plane we are: `_z0` is a (1/2 1/2 0) plane through the origin in primitive reciprocal basis, while `_x1`, `_y1` and `_z1` denote (0 1/2 1/2), (1/2 0 1/2) and (0 1/2 1/2) planes through (1/2 0 0), (0 1/2 0) and (0 0 1/2) as also printed to the output. If the invariants for `z0` and `z1` differ it is a strong TI. The `x1`, `y1`, `z1` invariants give the three weak indices  $\nu_{1,2,3}$ . Note, that the weak indices depend on the chosen planes.

Additionally files `+zgap_homo..._...` containing a reference line which follows the largest gap [A. A. Soluyanov, PRB 83, 235401 (2011)] are created together with convenience files `Z2_3dTI_homo*.cmd` which load all data into `xfbp`:

```
xfbp Z2_3dTI_homo16.cmd
```

The reference line is printed with blue weights if the number of centers crossed so far is even and in red weights if the number is odd. If the last data point is odd the invariant is non-trivial. This algorithm only works for a sufficiently large grid although it is much better for smaller number of points when humans might not yet see how the centers are connected. Since this algo does not always work as wished we modified it such that some of the larger gaps are followed in separate curves (only one is shown in `Z2_3dTI_homo...cmd`). At the end we call the invariant odd if a majority of these gap-following curves indicate odd-ness. The reliability of the results are printed in the output table. Afterall, it is always better to check how the results converge with the grid spacing.

**hamAtKPoint** (*kpoint*, *ms*, *gauge*='relative', *opindices*=None, *makedhk*=False, *makesigma*=False, *makexcfield*=False, *makebasisconnection*=False, *makewfsymops*=False)

### Parameters

- **kpoint** (*3-vector of float*) – The k-point in absolute cartesian coordinates.
- **ms** (*int*) – The spin component. Full relativistic: ms=0 else: ms in [0, nspin-1]
- **gauge** (*str*) – Can be 'relative' (default) or 'periodic' or 'forcerelative'. There are two possible phase choices for the underlying Bloch sums. The default is the relative-distance gauge  $H_{s's}^k = \sum_R e^{ik(R+s-s')} \langle w_{0s'} | H | w_{Rs} \rangle$  The second is the periodic gauge which is needed if derivatives with respect to k are required:  $H_{s's}^k = \sum_R e^{ikR} \langle w_{0s'} | H | w_{Rs} \rangle$  If the velocity matrix is requested (*makedhk*=True) the gauge will be set to 'periodic' automatically unless the gauge is set to 'forcerelative' (this is new and allows to overwrite the old behaviour).
- **opindices** (sequence of *int*) – A list of operation indices can be given to restrict the list of returned *WFSymOp* (page 66). See also *makewfsymops*. These indices are unique identifiers independent of the structure and are printed when *prepare* (page 42)-ing the structure.
- **makedhk** (*int*) – Return dH/dk in the wannier basis additionally to H. The return value will be a tuple (H, dhk, ...) if this option is True. The actual type of dhk is *BerryCurvatureData* (page 67).
- **makesigma** (*int*) – Return the 3 matrices  $\langle \sigma_{x,y,z} \rangle$  in the Wannier basis additionally to H. The return value will be a tuple (H,...,sigma,...) if this option is True.
- **makexcfield** (*int*) – Return the 3 matrices  $\mu_B \langle B_{x,y,z}^{xc} \rangle$  in the Wannier basis in eV additionally to H. The return value will be a tuple (H,...,xcfield,...) if this option is True. (Only for full relativistic.) In a spin polarized calculation “ $H - (B[0].dot(S[0])+B[1].dot(S[1])+B[2].dot(S[2]))$ ” will approximately be the non-polarized Hamiltonian, where H, S and B are returned by *hamAtKPoint* (page 50) with options *makesigma* and *makexcfield*. This is approximate since to obtain B we need to factor the xc-Zeeman term into the spin matrices and the field matrices. The FPLO basis is not complete and the Wannier basis even less. An illustrative example is given in `./Examples/slabify/Fe/SP/slabify/xcfield`.
- **makebasisconnection** (*int*) – Return, additionally to H, the 6 matrices  $\vec{A}^{w,k} = -i \langle u_w | \nabla u_w \rangle$  and  $\nabla \times \vec{A}^{w,k}$  in the Wannier basis, which are the Berry connection and Berry curvature of the WF basis itself: The relation to the position operator is given in periodic gauge by  $-\vec{A}_{mn}^{b,k} = \langle \vec{r} \rangle_{mn}^k$  and in relative gauge by  $-\vec{A}_{mn}^k = \langle \vec{r} \rangle_{mn}^k - \delta_{mn} \delta_{ts} \vec{s}$  where *s* and *t* are Wannier centers (sites) and  $\langle \vec{r} \rangle_{mn}^k = \sum_R e^{ik(R+\lambda(s-t))} \langle w_{0t} | r | w_{Rs} \rangle$  where  $\lambda = 0$  in the periodic gauge and  $\lambda = 1$  in the relative gauge. The return value will be a tuple (H, ..., basisconnection, ...) if this option is True. The first 3 components of *basisconnection* are the basis Berry connection and the last 3 the basis Berry curvature. To convert the connection in relative gauge to the position operator add the matrices returned

by `wannierCenterMatrix` (page 46). See argument `gauge` above to remember why `gauge=forcerelative` needs to be used when `relative` is required. Examples for the use of this argument are found in `./Examples/slabify/Fe/SP/slabify/3dR` and `./Examples/slabify/Fe/SP/slabify/AHC`.

- **makewfsymops** (*int*) – If `True` return a list of `WFSymOp` (page 66) instances, which describe the symmetry transformations of the Hamiltonian and the Wannier function Bloch sums. The return value will be a tuple `(H,...,wfsymops)` if this option is `True`.

**Returns** `(H,...)` – If all `make...` options are `False` it simply returns the Hamiltonian `H` (no tuple) at `kpoint` for spin `ms`. If any of the `make...` options are `True` a tuple is returned containing `H` and all additionally requested operators in the order defined by the `make...`-arguments in the `argument list` (page 50) above. The order of the keyword arguments in an actual function call does not matter for the order of return values. If the requested operator is a vector operator it is returned as a list of `numpy.ndarrays`. The derivative `dH/dk` is in units of `eV*aB`.

**Return type** `numpy.ndarray` or a tuple of return values

### Example

For convenience you can use `kyscale` (page 56) as in:

```
s=sla.Slabify()
k=[1,0,0]
ms=0
H=hamAtKPoint(k*s.kyscale,ms) # or to get dH/dk
(H,dHk)=hamAtKPoint(k*s.kyscale,ms,makedhk=True)
```

**diagonalize** (*h*, *dhk*=None, *makef*=False, *basisconnection*=None, *toldegen*=1e-10)

#### Parameters

- **h** (*square complex matrix*) – the Hamiltonian
- **dhk** (*BerryCurvatureData* (page 67)) – behaves as a list of 3 complex `numpy.ndarrays` which contain `dH/dk` as returned by `hamAtKPoint` (page 50)
- **makef** (*int*) – if `True` return band wise Berry curvature
- **basisconnection** (*list of 6 complex numpy.ndarrays*) – basisconnection and curvature as returned by `hamAtKPoint` (page 50)
- **toldegen** (*float*) – levels closer than this tolerance are considered to be a degenerate subspace in the non-Abelian correction to the Berry curvature

**Returns** `(E,C,...)` – The first tuple value is a `numpy.ndarray` of the energies, the second a `numpy.ndarray` of eigenvectors `C[:,i]` is the *i*-th eigenvector. If `makef` is `True` the third tuple elements is a `numpy.ndarray` containing the Berry curvature where `F[:,n]` is the band-*n* Berry curvature (a 3-vector). The dimension of the Hamiltonian is `nvdim` (page 54).

**Return type** tuple of return values

Diagonalize the Hamiltonian *h* and return the eigenvalues and eigenvectors. If `makef` is `True`, `dhk` must be given. Then additionally the Berry curvatures for all bands is returned. If `basisconnection` is also supplied the full Berry curvature is returned. See `berryCurvature` (page 52).

`dhk` must be obtained from `hamAtKPoint` (page 50). New in version 19: Note, that in the periodic gauge (used for *h* and `dhk`) a correction for the position operator is used when calculating the Berry curvature. If `gauge=forcerelative` was used in `hamAtKPoint` (page 50) this correction is zero. The curvature corresponds to  $A^k = -i \langle u | \nabla u \rangle$ .

Now, a correction for the non-Abelian Berry curvature is applied for degenerate subspaces. Which levels are considered degenerate is controlled by *toldegen*. If it is too small, spurious results can be obtained. If it is very large, some of the actual curvature will be missing. *toldegen* should be smaller than the smallest physical gap in the band structure.

Example:

```
k=np.array([0.5,0,0])
(Hk,dHk)=s.hamAtKPoint(k*s.kscale,ms,makedhk=True)

(E,CC,F)=s.diagonalize(Hk,dhk=dHk,makef=True,
                       toldegen=1.0e-9)
print 'Berry curvature of homo',homo,' : ',F[:,homo]
```

**diagonalizeUnitary** (*U*, *evdegentol*=1e-08)

**Parameters**

- **U** (*square complex matrix*) – the unitary matrix
- **evdegentol** (*float*) – eigenvalues with a distance less than this are considered degenerate.

**Returns** (**E,Z**) – *E* are the complex eigenvalues and *Z* the eigenvectors.

**Return type** tuple of return values

For unitary matrices LAPACK does not return orthogonal eigenvectors for degenerate subspaces. This method calls LAPACK for diagonalization and corrects the eigenvectors afterwards.

**coDiagonalize** (*E*, *C*, *Dk*, *evtol*=1e-08, *check*=False)

**Parameters**

- **E** (sequence of *float*) – the Hamiltonian eigenvalues
- **C** (*square complex matrix*) – the Hamiltonian eigenvectors
- **Dk** (*square complex matrix*) – the Bloch sum symmetry representation matrix
- **evtol** (*float*) – tolerance to determine degenerate subspaces of *E*
- **check** (*int*) – if `True` perform paranoia checks

**Returns** (**EU,CZ**) – *EU* are the complex symmetry eigenvalues and *CZ* the transformed eigenvectors which diagonalize the Hamiltonian and the symmetry.

**Return type** tuple of return values

Given Hamiltonian eigenvalues *E* and eigenvectors *C* such that  $HC = CE$  and a representation matrix *Dk* for the Wannier function Bloch sums with symmetry properties  $H = D^+HD$  determine  $U = C^+DC$ .

Then diagonalize *U*:  $U = ZE_UZ^+$  in each degenerate subspace of *E* and form the transformed eigenvectors  $C' = CZ$ . Now  $C'$  diagonalizes *H* and *U*. The eigenvalues  $E_U$  of *U* and the transformed eigenvectors  $C'$  are returned.

*evtol* is used to determine which Hamiltonian eigenvalues  $E[i]$  are degenerate.

The symmetry representation matrices are return by *hamAtKPoint* (page 50).

Note, that operations which are combined with time reversal do not really make sense in this context, so do not use this routine if the operation is combined time reversal.

**berryCurvature** (*E*, *C*, *dhk*, *subspace*=None, *basisconnection*=None, *toldegen*=1e-10, *returnde-tails*=False)

**Parameters**

- **E** (sequence of *float*) – the Hamiltonian eigenvalues
- **C** (*square complex matrix*) – the Hamiltonian eigenvectors

- **dhk** (*BerryCurvatureData* (page 67)) – *BerryCurvatureData* (page 67), behaves as a list of 3 complex `numpy.ndarrays` which contain  $H/dk$  as returned by *hamAtKPoint* (page 50)
- **subspace** (sequence of `int`) – a list of zeros and ones, which describes the wanted subspace or `None` for the whole space.
- **basisconnection** (list of 6 complex `numpy.ndarrays`) – basisconnection and curvature as returned by *hamAtKPoint* (page 50)
- **toldegen** (*float*) – levels closer than this tolerance are considered to be a degenerate subspace in the non-Abelian correction to the Berry curvature
- **returndetails** (`bool`) – if `True` details of the curvature contributions are returned

**Returns (F,divergenttermsdetected)** – *F* is a `numpy.ndarray` of shape=(3, *nvdim*) and contains the band wise Berry curvature. *divergenttermsdetected* is now always `False`. It is still there to not break older code. Sorry. If *basisconnection* is given and *returndetails*==`True` the returned tuple is (F, *Fdetails*, *divergenttermsdetected*), where *Fdetails* is a dict of the various terms *F* is made of. The dict values are again `numpy.ndarray` of the same shape as *F*.

#### Return type tuple

Calculate the Berry curvature from the Hamiltonian eigenvalues *E* and eigenvectors *C* and form  $dH/dk$  returned from *hamAtKPoint* (page 50). If *basisconnection* is given (as returned from *hamAtKPoint* (page 50)) the full curvature is calculated (Wang2006) not only the D-D part. If *returndetails*==`True` the separate terms *F* is made off are returned in a dict.

If *subspace* is `None` the Berry curvature is returned for all bands. If it is a list of size `len(E)` the Berry curvature is calculated from the subspace for which *subspace* contains a nonzero value, preferably 1. This can be used to make plots of the k-resolved mirror Chern number.

If *basisconnection* is not provided, the resulting Berry curvature will contain a correction for the position operator in the periodic gauge. In the relative gauge (*gauge*=*forcerelative*) this correction is zero.

The curvature corresponds to  $A^k = -i \langle u | \nabla u \rangle$ .

Now, a correction for the non-Abelian Berry curvature is applied for degenerate subspaces. Which levels are considered degenerate is controlled by *toldegen*. If it is too small, spurious results can be obtained. If it is very large, some of the actual curvature will be missing. *toldegen* should be smaller than the smallest physical gap in the band structure.

FPL0/...DOC/pyfplo/Examples/slabify/TCI/SnTe contains an example.

**findWeylPoints** (*box*, *homos*, *tol*=0.001)

#### Parameters

- **box** (*BoxMesh* (page 56)) – a definition of the box and its mesh.
- **homos** (list or single `int`) – the band indices, which form the highest occupied bands. If a list is given WPs for all these highest occupied bands are searched.
- **tol** (*float*) – This sets the finest bi-section of the refinement part of the algorithm. What is a useful value depends on the actual structuring of the Berry curvature.

This method tries to automatically find Weyl points using a modification of the algorithm described in [Takahiro Fukui et.al. J. Phys. Soc. Jpn. 74, 1674 (2005)] The user defines a *box* which spans a grid. On all grid points the Hamiltonian is diagonalized. For each micro cell of the grid the Berry curvature is integrated over the surface of the cell. This gives the chirality of the WP if the box contains a WP. For all cells, which have a non zero result subsequent bisections is performed until the box size falls below *tol*. Finally, all resulting boxes with non-zero chirality are written to the file *weylpoints.py*. This file can be loaded into a script for further processing (e.g. to use *calculateChernNumberInSphere* (page 47) to make sure that is actually is a Weyl point.)

Note: that the algorithm does not always find all Weyl points due to several reasons.

- The *box* grid must be fine enough such that the Berry curvature variation is properly sampled by the micro cells.
- The origin of the *box* matters if e.g. the WPs are pinned to symmetry planes. The WPs ideally sit well within a micro cell. Since, one can not always know this in advance it is probably good to try two different origins. One at say (000) and another at  $-\Delta_k/2$ , where  $\Delta_k$  is the body diagonal of a micro cell.
- A micro cell might contain two WPs of opposite chirality, which would result in zero total chirality and hence these WPs are missed. The grid must be fine enough to avoid this.

Often symmetries can be used to complete the list of found WPs.

Warning: the algorithm can produce false positives. This is why it is always a good idea to check the validity of the results via [calculateChernNumberInSphere](#) (page 47).

(also see [the Weyl semi metal example](#) (page 117))

**calculateMirrorChernNumbers** (*homo*, *nint*, *atneqk=False*, *showevs=False*, *evtol=1e-08*, *fullF=False*, *toldegen=1e-10*)

#### Parameters

- **homo** (*int*) – the highest occupied band
- **nint** (*int*) – the number of subdivision in the first k direction of the mirror plane
- **atneqk** (*int*) – if `True` also calculate for plane at nonzero out-off plane position
- **showevs** (*int*) – if `True` show mirror eigenvalues
- **evtol** (*float*) – energies eigenvalues which are closer than this tolerance are considered degenerate when determining mirror subspaces
- **fullF** (*int*) – if the position operator (basis connection) is contained in `+hamdata` the full Berry curvature is calculated if *fullF* is `True`
- **toldegen** (*float*) – levels closer than this tolerance are considered to be a degenerate subspace in the non-Abelian correction to the Berry curvature

For each symmorphic mirror operation the mirror chern number for the mirror plane in k-space is calculated. The result will depend on the density of the integration mesh which is controlled by *nint*. The mirror eigenvalues are determined in the subspaces of degenerate energies. This increases the accuracy of the diagonalization of the unitary transformation. The Hamiltonian eigenvectors are transformed to belong to mirror eigenvalue subspaces from which the Berry curvature is determined. *toldegen* controls the application of the non-Abelian correction to the Berry curvature (see [berryCurvature](#) (page 52)).

The mirror planes are determined automatically, which leads to a 2d reciprocal basis in the plane and an out off plane vector an integer multiple of which describes the position of all mirror planes. The mirror chern number is calculated for the plane through the origin and for the closest plane.

See [berryCurvature](#) (page 52) for details about the Berry curvature calculation.

FPLO/...DOC/pyfplo/Examples/slabify/TCI/SnTe contains an example.

#### dirname

Local result directory. You can also set it

```
s=sla.Slabify()
s.dirname='.' # now output lands in the current directory
```

**Type** str

#### nvdim

Return dimension of WF Hamiltonian.

**Type** int

**nspin**

Return number of spins (always 1 for full-relativistic).

**Type** int

**object**

The type of structure-object to be created

Values: '3d', 'slab' or 'semislab'

**Type** str

**enlarge**

3x3 list or numpy.ndarray: A 3x3 integer matrix  $U$  for enlarging the cell. The rows of  $U$  represent the three new vectors. This matrix produces a new unit cell  $A'$  out of the old  $A$  via  $A' = U \cdot A$

where we assume that  $A$  is a column vector of the tree lattice vectors.  $A = \begin{pmatrix} \mathbf{a}_1 \\ \mathbf{a}_2 \\ \mathbf{a}_3 \end{pmatrix}$  (see *Structure Manipulation Algorithm* (page 85))

*Manipulation Algorithm* (page 85))

**zaxis**

The axis perpendicular to the surface (in cartesian coordinates) (see *Structure Manipulation Algorithm* (page 85))

**Type** 3-vector

**numberoflayers**

Number of layers to be constructed from the layered cell. For 'semislab's the smallest possible number should be chosen. This number is determined by the maximal inter-atom overlap in the Hamiltonian data. It is best to start with 1 and then increase it according to the program output (it will complain if needed). (see *Structure Manipulation Algorithm* (page 85))

**Type** int

**anchor**

at which relative z-position in the layered unit cell do we cut the solid.

**Type** float

**Type** for 'slab' and 'semislab'

**cutlayersat**

Currently only for 'slab'! Lower and upper absolute z-coordinate at which to cut a finite slab. If `cutlayersat[1]<cutlayersat[0]` it is not applied. (see *Structure Manipulation Algorithm* (page 85))

**Type** list of 2 float

**cutatoms**

Currently only for 'slab'! A list of atoms to be removed in the final step. Note that the site numbers refer to the cell after *cutlayersat* (page 55) was applied. Use `xfplo` on the resulting `.in` file to find the site numbers. (see *Structure Manipulation Algorithm* (page 85))

**Type** list of int

**options**

A set of options (for debugging), see `pyfplo.common.OptionSet` (page 36)

**Example**

```
import pyfplo.slabify as sla
s=sla.Slabify()
op=s.options
```

(continues on next page)

```
for n in op.names:
    print n, 'is set to ', op[n]
```

Type *OptionSet* (page 36)

#### **kyscale**

The common factor used for the k-points. Usually k-points are given as  $2\pi/a * (k_x, k_y, k_z)$  where  $a$  is obtained from `+hamdata` or more precisely from the conventional cell. One can set *kyscale* (page 56) to any value which is convenient.

Note: For rhombohedral lattices the conventional lattice is the trigonal (hexagonal) cell if the spacegroup setting is hexagonal or the same as the primitive rhombohedral cell if the spacegroup setting is rhombohedral. Consequently *kyscale* (page 56) depends on the setting for rhombohedral lattices.

Type `float`

#### **bfield**

the optional B-field configuration.

Type *BfieldConfig* (page 65)

#### **hasxcfield**

if `True` the Bxc-data are available in `+hamdata`

#### **hassigma**

if `True`  $\langle \sigma_{x,y,z} \rangle$  is available in `+hamdata`

#### **hasbasisconnection**

if `True` basis connection and curvature are available in `+hamdata`

## 2.4.2 BoxMesh

### **class BoxMesh**

Define a box mesh in 3 dimensions. For a planar mesh set `nz=1` and `zinterval=[0, 0]`.

**setBox** (*xaxis*=[1, 0, 0], *yaxis*=[0, 1, 0], *zaxis*=[0, 0, 1], *origin*=[0, 0, 0])

Returns `self` – for call chaining.

Return type *BoxMesh* (page 56)

Convenience function: Define the box. See *origin* (page 58), *xaxis* (page 58), *yaxis* (page 58), *zaxis* (page 58) for docs.

**setMesh** (*nx*=100, *ny*=100, *nz*=100, *xinterval*=[-0.5, 0.5], *yinterval*=[-0.5, 0.5], *zinterval*=[-0.5, 0.5])

Returns `self` – for call chaining.

Return type *BoxMesh* (page 56)

Convenience function: Define the mesh subdivisions of the box. See *nx* (page 58), *ny* (page 58), *nz* (page 58), *xinterval* (page 58), *yinterval* (page 58) and *zinterval* (page 58) for docs.

**xmesh** ()

Returns `xmesh` – the mesh coordinates along the *xaxis* (page 58)

Return type `numpy.ndarray`

**ymesh** ()

Returns `ymesh` – the mesh coordinates along the *yaxis* (page 58)

Return type `numpy.ndarray`

**zmesh** ()

**Returns** `zmesh` – the mesh coordinates along the `zaxis` (page 58)

**Return type** `numpy.ndarray`

**mesh** (`scale`, `xfirst=False`)

**Parameters**

- **scale** (`float`) – all points are multiplied with `scale`
- **xfirst** (`int`) – if `True` invert the loop order (make the `x`-loop the outer loop).

**Returns** `mesh` – The flat list of all vectors of the mesh.

**Return type** `numpy.ndarray`

Return the box mesh in absolute coordinates as a single flattend continous `numpy.ndarray` of vectors. This routine actually first calculates the mesh. Hence it is strongly advised to use a local copy as in:

```
kpnts=box.mesh(scale, False)
someFunctionCall(..., kpnts, ...)
```

When creating the mesh the innermost loop is `x`, then `y`, the outer loop is `z`. If `xfirst` is `True` the loop order is inverted. All points are scaled by `scale`.

The mesh is calculated as:

```
mesh[ind]=(xaxis*xinterval[i]+
           yaxis*yinterval[j]+
           zaxis*zinterval[k]+origin)*scale`
```

where `i`, `j` and `k` run over all possible interval values and `ind` runs over all mesh indices depending on the loop order.

**relToAbs** (`*args`)

**Parameters** `args` (3 `float` or list or `numpy.ndarray`) – can be a 3-vector or three coordinates.

**Returns** `point` – the mesh point in absolute coordinates.

**Return type** `numpy.ndarray`

Take a point (`x`, `y`, `z`) in relative mesh coordinates and return the absolute coordinates. Relative means relative with respect to the normalized axes not the intervals:

```
abs=origin+x*xaxis+y*yaxis+z*zaxis
```

**absToRel** (`*args`)

**Parameters** `args` (3 `float` or list or `numpy.ndarray`) – can be a 3-vector or three coordinates.

**Returns** `point` – the mesh point in relative coordinates.

**Return type** `numpy.ndarray`

Take a point in absolute coordinates and return the relative mesh coordinates (`x`, `y`, `z`). Relative means relative with respect to the normalized axes not the intervals:

```
abs=origin+x*xaxis+y*yaxis+z*zaxis
```

**xyzFromIndex** (`ind`, `xfirst=False`)

**Parameters**

- **ind** (`int`) – an index into the continous flattend mesh array of vectors.
- **xfirst** (`int`) – if `True` invert the loop order (make the `x`-loop the outer loop).

**Returns** `xyz` – a list of 3 indices such that the mesh position `xyz[0]`, `xyz[1]`, `xyz[2]` has point `mesh[ind]`

**Return type** `numpy.ndarray` of 3 `int`

Given an index into the flattened mesh array return the individual indices of the three box axis: `ix`, `iy`, `iz`. The order of loops matter (see [mesh](#) (page 57)).

`xfirst=False: ind=ix+nx*(iy+ny*iz)`

`xfirst=True: ind=iz+nz*(iy+ny*ix)`

**\_\_str\_\_()**

return printable representation. You do not need to call this explicitly. An object *obj* with this function provides usefull info when printed:

```
print(obj)
```

**origin**

`numpy.ndarray`: The origin(shift) of the box mesh given with respect to the global coordinate system, in some units, which usually are `pyfplo.slabify.Slabify.kscale` (page 56). (See the *scale* parameter in [mesh](#) (page 57))

**xinterval**

`numpy.ndarray`: Interval along the first (x) axis of the box as defined by [xaxis](#) (page 58). The units often are `pyfplo.slabify.Slabify.kscale` (page 56), which makes it a uniform scale no matter the orientation of the box.

**yinterval**

`numpy.ndarray`: Interval along the first (y) axis of the box as defined by [yaxis](#) (page 58). The units often are `pyfplo.slabify.Slabify.kscale` (page 56), which makes it a uniform scale no matter the orientation of the box.

**zinterval**

`numpy.ndarray`: Interval along the first (z) axis of the box as defined by [zaxis](#) (page 58). The units often are `pyfplo.slabify.Slabify.kscale` (page 56), which makes it a uniform scale no matter the orientation of the box.

**nx**

number of points in the first (x) direction.

**Type** `int`

**ny**

number of points in the second (y) direction.

**Type** `int`

**nz**

number of points into the thid (z) direction, which also is integrated over for planar projections

**Type** `int`

**xaxis**

the first (x) axis spanning the box. It will be automatically normalized.

**Type** `numpy.ndarray`

**yaxis**

the second (y) axis spanning the box. It will be automatically normalized.

**Type** `numpy.ndarray`

**zaxis**

the third (z) axis spanning the box. It will be automatically normalized. This is integrated over.

**Type** `numpy.ndarray`

### 2.4.3 EnergyContour

**class EnergyContour** (*ne=100, e0=-1, e1=1, ime='auto'*)

Create a complex energy contour for energy distribution curves (EDC) Currently, only equidistant and parallel to real axis. Usage:

```
import pyfplo.slabify as sla
ec=sla.EnergyContour(100,-1.5,2.5,0.001)
```

or set individual properties:

```
import pyfplo.slabify as sla
ec=sla.EnergyContour()
ec.ne=100
...
```

or set individual mesh:

```
import pyfplo.slabify as sla
ec=sla.EnergyContour()
ec.setMesh([1,2,3,4+.1j])
```

#### Parameters

- **ne** (*int*) – see *ne* (page 59)
- **e0** (*float*) – see *e0* (page 59)
- **e1** (*float*) – see *e1* (page 59)
- **ime** ('auto' or *float*) – see *ime* (page 59)

Set an equidistant energy contour paralell to the real axis. See the individual properties for further docs.

**mesh** ()

**Returns** **mesh** – the energy mesh

**Return type** `numpy.ndarray`

**setMesh** (*m*)

**Parameters** **m** (*list or numpy.ndarray of complex*) – a list of energy points in the complex plane

Set a hand crafted mesh. DO NOT set any other variable after this.

**\_\_str\_\_** ()

return printable representation. You do not need to call this explicitly. An object *obj* with this function provides usefull info when printed:

```
print(obj)
```

**ne**

number of energy points

**Type** `int`

**e0**

real part of starting point

**Type** `float`

**e1**

real part of end point

**Type** `float`

**ime**

imaginary part of the contour. *ime* can be a real number or the string 'auto' in which case *ime* is set to  $(e1-e0)/ne$ . Of course you have to define *e0*, *e1* and *ne* first or simply use the constructor.

**Type** float or str

## 2.4.4 FermiSurfaceOptions

**class FermiSurfaceOptions**

Fermisurface controls.

Define the 2d k-mesh on which to calculate.

For '3d'/'slab' this mesh is used as a basis for interpolation to obtain the iso-lines. For 'semislab' it defines the pixel mesh on which to calculate the spectral density.

Note, that the slabify structure manipulation routines transform the original cell in such a way that the orientation of the resulting cell with respect to the original global cartesian system is unaltered. The k-mesh, is given in this global cartesian system. So a slab with *zaxis*=[1, 0, 0] has a surface BZ in the 010/001-plane and the axis must be set accordingly. Always try with a small mesh first to get your bearings. Also for spectral densities (pixelized pictures) in *semislab* mode the k-plane axis must be orthogonal!

**setPlane** (*xaxis*=[1, 0, 0], *yaxis*=[0, 1, 0], *origin*=[0, 0, 0])

**Parameters**

- **xaxis** (3-vector of float) – see *xaxis* (page 62)
- **yaxis** (3-vector of float) – see *yaxis* (page 63)
- **origin** (3-vector of float) – see *origin* (page 62)

Convenience function: Define the k-plane.

**setMesh** (*nx*=100, *xinterval*=[-0.5, 0.5], *ny*=100, *yinterval*=[-0.5, 0.5])

**Parameters**

- **nx** (*int*) – see *nx* (page 62)
- **xinterval** (2-vector of float) – see *xinterval* (page 62)
- **ny** (*int*) – see *ny* (page 62)
- **yinterval** (2-vector of float) – see *yinterval* (page 62)

Convenience function: Define the mesh subdivisions of the k-plane.

**on** ()

Activate fermi-surface related routines.

**off** ()

Deactivate fermi-surface related routines.

**xmesh** ()

**Returns** **xmesh** – the mesh coordinates along the xaxis

**Return type** numpy.ndarray

**ymesh** ()

**Returns** **ymesh** – the mesh coordinates along the yaxis

**Return type** numpy.ndarray

**mesh** (*scale*)

**Parameters** **scale** (*float*) – all points are multiplied with *scale*

**Returns** **mesh** – The flat list of all vectors of the mesh. x is running first.

**Return type** `numpy.ndarray`

Return the box mesh in absolute coordinates as a single flattened continuous `numpy.ndarray` of vectors. This routine actually first calculates the mesh. Hence it is strongly advised to use a local copy as in:

```
kpnts=fso.mesh(scale)
someFunctionCall(...,kpnts,...)
```

When creating the mesh the innermost loop is `y`, the outer loop is `x`. All points are scaled by `scale`.

The mesh is calculated as:

```
mesh[ind]=(xaxis*xmesh()[i]+yaxis*ymesh()[j]+origin)*scale`
```

where `i, j` interval values and `ind` runs over all mesh indices.

**openDensPlotFile** (`filename`, `ms`, `plotorigin=[0, 0]`, `plotxaxis=[1, 0]`, `plotyaxis=[0, 1]`, `progress=None`)

#### Parameters

- **filename** (`str`) – the name of the `xynz`-type file
- **ms** (`int`) – The spin component. Full relativistic: `ms=0` else: `ms` in `[0, nspin-1]`
- **plotorigin** (`2-vector of float`) – a vector describing the origin in the plotting plane
- **plotxaxis** (`2-vector of float`) – a vector describing the x-axis in the plotting plane
- **plotyaxis** (`2-vector of float`) – a vector describing the y-axis in the plotting plane
- **progress** – a progress message (`str`) or `None`

**Returns** band file context

**Return type** `DensPlotContext` (page 63)

Low level routine. Return an object of type `DensPlotContext` (page 63) for creation of FPLO `xynz` density plot files.

The returned object will open the file and organizes the proper file format. Its `DensPlotContext.write()` (page 63) method can be used to write the actual data. If the object gets deleted (automatic if the scope is left) the file gets closed. The `DensPlotContext.close()` (page 63) method can be called explicitly.

The best way to use it is in a `with`-statement. Then it is closed automatically after the `with`-block is exited:

```
with fso.openDensPlotFile(...) as f:
    for ...:
        f.write(...)
pass # now the file is closed.
```

If multiple files are written at the same time one can do the following:

```
f1=fso.openDensPlotFile(filename1,...)
f2=fso.openDensPlotFile(filename2,...)
for ...:
    f1.write(data1,...)
    f2.write(data2,...)
f1.close()
f2.close()
pass # now the files are closed.
```

Usually `FermiSurfaceOptions` (page 60) defines a plane in k-space on which pixelated data are calculated. To plot them, we need to map this onto the plotting x,y-plane. If the x,y-axes of `FermiSurfaceOptions` (page 60) are not orthogonal one wants to give two axes for the same angle as arguments to `openDensPlotFile` (page 61) to orient the data in the plotting plane. The origin in the plotting plane can also be given. The length of `plotxaxis` and `plotyaxis` determines the length scale when plotted with `xfbp`.

If `progress` is set to a string a progress message is written in subsequent calls to `DensPlotContext.write()` (page 63).

see help of `DensPlotContext` (page 63).

An example can be found in `./Examples/slabify/densplot` and `./Examples/slabify/Fe/SP/slabify/AHC/curvature.py`.

**\_\_str\_\_()**

return printable representation. You do not need to call this explicitly. An object `obj` with this function provides useful info when printed:

```
print(obj)
```

**active**

if True calculate the 3d/slab Fermi surface features.

**Type** bool

**nx**

number of points into the first (x) direction.

**Type** int

**ny**

number of points into the first (y) direction.

**Type** int

**xinterval**

interval along the first (x) axis of the BZ-plane as defined by the in-plane vector `xaxis` (page 62). The units are `pyfplo.slabify.Slabify.kscale` (page 56), which makes it a uniform scale no matter the orientation of the k-plane.

**Type** numpy.ndarray

**yinterval**

interval along the first (y) axis of the BZ-plane as defined by the in-plane vector `yaxis` (page 63). The units are `pyfplo.slabify.Slabify.kscale` (page 56), which makes it a uniform scale no matter the orientation of the k-plane.

**Type** numpy.ndarray

**fermienergy**

at which Fermi energy? Units like in `+hamdata`, which by default is eV.

**Type** float

**fermienergyim**

a finite imaginary part for the energy is needed for `semislab` spectral functions (same units as `fermienergy` (page 62)). The smaller this value the higher the k-point density must be. Often a good value is `interval-length/max(Nx, Ny)`

**Type** float

**origin**

the origin of the k-plane in units of `pyfplo.slabify.Slabify.kscale` (page 56)

**Type** numpy.ndarray

**xaxis**

the first (x) axis spanning the k-plane. It will be automatically normalized. For semislabs the axes must be orthogonal

**Type** 3-vector of float

**yaxis**

the first (y) axis spanning the k-plane. It will be automatically normalized. For semislabs the axes must be orthogonal

**Type** 3-vector of float

## 2.4.5 DensPlotContext

### class DensPlotContext

This class wraps data to easily manage the creation of density plot files. This class cannot be instantiated directly. It only is produced and returned via a call to `FermiSurfaceOptions.openDensPlotFile()` (page 61) An example can be found in `./Examples/slabify/densplot` and `./Examples/slabify/Fe/SP/slabify/AHC/curvature.py`.

**close()**

Explicitly close the file. Useful, if multiple files are used in the same loop, in which case the `with`-statement is not useful. The underlying file gets closed when this object gets garbage collected (after its scope is exited). For cleanliness it is a good measure to always close files.

Method1:

```
with fso.openDensPlotFile(...) as f:
    do something with f
# here f is closed
```

Method2:

```
f=fso.openDensPlotFile(...):
do something with f
f.close()
# here f is closed
```

**write(components)**

**Parameters** **components** (*sequence (list, tuple, ..)*) – sequence of values for all density data components at this k-point. There can be more than one value per k-point, e.g. three components of a vector field.

Write the density data components for the current k-vector to the file.

## 2.4.6 GreenOptions

### class GreenOptions (nsigiter=30, sigitertol=0.001, sigitermethod='accel')

Settings which control the self energy (sigma) calculation.

**Parameters**

- **nsigiter** (*int*) – see `nsigiter` (page 64)
- **sigitertol** (*float*) – see `sigitertol` (page 64)
- **sigitermethod** (*str*) – see `sigitermethod` (page 64)

Create a GreenOptions object with the settings given in the argument.

**\_\_str\_\_()**

return printable representation. You do not need to call this explicitly. An object *obj* with this function provides useful info when printed:

```
print(obj)
```

**nsigiter**

maximum number of sigma (self energy) iteration loops.

**Type** int

**sigitermethod**

sigma iteration method, 'accel' (default) or ''

**Type** str

**sigitertol**

sigma iteration tolerance.

**Type** float

## 2.4.7 WeylPoint

**class WeylPoint** (*k*, *axis1*=[1.0, 0.0, 0.0], *axis2*=[0.0, 1.0, 0.0], *axis3*=[0.0, 0.0, 1.0], *chirality*=1.0, *radius*=0.1, *energy*=0.0, *homo*=1, *spin*=1)

Collect information about a Weyl point. This is for easier book keeping.

for documentation consult the individual properties.

**\_\_str\_\_** ()

return printable representation. You do not need to call this explicitly. An object *obj* with this function provides usefull info when printed:

```
print(obj)
```

**k**

k is the WP position. Usually it is given in units *Slabify.kscale* (page 56)

**Type** 3-vector

**axis1**

the first axis to span a volume around the WP or for bandplot purposes. On being set it will be normalized.

**Type** 3-vector

**axis2**

the second axis to span a volume around the WP or for bandplot purposes. On being set it will be normalized.

**Type** 3-vector

**axis3**

the third axis to span a volume around the WP or for bandplot purposes. On being set it will be normalized.

**Type** 3-vector

**chirality**

Store the chirality

**Type** float

**radius**

a radius within which a monopole was found.

**Type** float

**energy**

float at which energy does the WP sit.

**homo**

The highest occupied band.

**Type** int

**spin**

always 1).

**Type** int

**Type** the spin (1=up, 2=down, relativistic

## 2.4.8 BfieldConfig

### class BfieldConfig

The class allows to add model spin-only magnetic fields. to the Hamiltonian. This class cannot be instantiated directly. It only is returned from objects, which have an *BfieldConfig* (page 65) member variable (see *Slabify.bfield* (page 56)). Example usage:

```
s=sla.Slabify()
bf=s.bfield
bf.setGlobalField([0,0,1.3])
#or
s=sla.Slabify()
s.bfield.setGlobalField([0,0,1.3])
```

For an example see `./Examples/slabify/Fe/NSP/slabify/addBfield`.

#### setGlobalField(*B*)

**Parameters** *B* (3-vector of float) – the magnetic field

Set constant global spin-only magnetic field. The last field set in the script (global or local) will win.

#### setLocalFields (*listofcomponents*)

**Parameters** *listofcomponents* (list of pairs of a projector (list) and field 3-vector) – list of pairs of a projector and field vector:

```
[
 [ [0,0,0,1,1,1,1,0,0,0], [0, 1.2, 0] ],
 [ [1,1,1,0,0,0,0,0,0,0], [0,-1.2,0] ],
 ...
]
```

where the first internal list is a projector *P* and the second a field *B*. The projector must have the dimension of the Hamiltonian. It usually is a list of zeros and ones. The Hamiltonian is modified according to:

$$H = H_0 + \sum_i P_i (\vec{\sigma} \cdot \vec{B}_i) P_i$$

which means that the blocks where *P* is nonzero get a Zeeman term added. There can be several *P*, *B* pairs as indicated by the subscript *i*. You can e.g. use *Slabify.orbitalNames()* (page 45) and python map to create the projectors.

Set constant local spin-only magnetic fields. The last field set in the script (global or local) will win.

#### \_\_str\_\_()

return printable representation. You do not need to call this explicitly. An object *obj* with this function provides usefull info when printed:

```
print(obj)
```

## 2.4.9 WFSymOp

### class WFSymOp

A list of instances of this class is returned by `hamAtKPoint` (page 50) if options are set accordingly. It provides information on the symmetry transformations of the Hamiltonian and Wannier function Bloch sums.

There is a script in `FPLO/...DOC/pyfplo/Examples/slabify/symmetryops` which after adjusting the path should run through without errors. It demonstrates the symmetry operations and also prints the eigenvalues of operations. Use this as a starting point for your own work.

#### `__str__()`

return printable representation. You do not need to call this explicitly. An object `obj` with this function provides usefull info when printed:

```
print(obj)
```

#### symbol

A symbolic representation of the symmetry operation. If in doubt refer to `alpha` (page 66) and `tau` (page 66) for the exact meaning.

#### index

This is a unique index, which is used to identify particular operations. A list of such indices can be used in a call to `Slabify.hamAtKPoint` (page 50) to request a subset of symmetries. The indices are unique identifiers independent of the structure and are printed when `Slabify.prepare` (page 42)-ing the structure.

#### isinlittlegroup

This is `True` if the operation is in the little group of the k-point for which `Slabify.hamAtKPoint` (page 50) was called.

#### alpha

This is the (improper) rotational part (3x3-matrix) of the symmetry operation. It acts on a real space vector (or k-point) as  $\vec{r}' = \alpha\vec{r}$ . Together with `tau` (page 66) it forms the seitz symbol of the space group operation which acts like  $\{\alpha | \tau\}\vec{r} = \alpha\vec{r} + \tau$ . `alpha` (page 66) and `tau` (page 66) are given in absolute cartesian coordinates.

#### tau

This is the fractional translational part of the space group operation. It describes the location of the operation in space and possible glide and screw components. If it is zero the operation is definitely symmorphic but if it is non-zero it can be symmorphic (operation does not sit at (000)) or non-symmorphic (is a glide/screw). See `alpha` (page 66).

#### timerev

For full relativistic magnetic calculations some space group operations of the input space group are invalid. FPLO reduces the symmetry to a subset which leaves the magnetic field unchanged. These are all operations which leave the field axis invariant. Among these there can be operations which invert the axis. For these an additional time reversal puts the field back into its original direction. Such combined operations have `timerev==True`. Note, that these operations do not act like normal space group representations. If a normal space group operation acts like  $H^{\vec{k}} = D^{\vec{k}+} H^{\alpha\vec{k}} D^{\vec{k}}$ , time reversed ops act like  $H^{\vec{k}} = \left( D^{\vec{k}+} H^{-\alpha\vec{k}} D^{\vec{k}} \right)^*$  where  $D^{\vec{k}}$  is a product of the time reversal representation matrix and the space group representation matrix in the Wannier orbital space.

Note: that for non-magnetic full relativistic calculations time reversal is also a symmetry. The corresponding representation matrix is to found among the list of `WFSymOp` (page 66) instances returned by `Slabify.hamAtKPoint` (page 50). For this operation the complex conjugation also has to be applied.

#### isimproper

If `True` this operation is improper; a rotation times inversion.

#### Dk

This is the representation matrix of  $\hat{g} = \{\alpha | \tau\}$  (or pure time reversal  $\Theta$  if present or a combined

operation  $\Theta\hat{g}$ ) in the space of the Wannier orbitals at a given k-point. If the operation is in the little group of the k-point the Hamiltonian fulfills  $H^{\vec{k}} = D^{\vec{k}+} H^{\vec{k}} D^{\vec{k}}$  otherwise  $H^{\vec{k}} = D^{\vec{k}+} H^{\alpha\vec{k}} D^{\vec{k}}$ . If the operation is combined with time reversal an additional complex conjugation must be applied. If time reversal is in the little group we get  $H^{\vec{k}} = (D^{\vec{k}+} H^{\vec{k}} D^{\vec{k}})^*$  otherwise  $H^{\vec{k}} = (D^{\vec{k}+} H^{-\alpha\vec{k}} D^{\vec{k}})^*$ . Also see *timerev* (page 66).

The Wannier orbital Bloch sums transform as  $\hat{g}w^{\vec{k}} = w^{\alpha\vec{k}} D^{\vec{k}}$  where  $w$  is the row vector of all WFs and the multiplication with  $D^{\vec{k}}$  a matrix multiplication. For time reversed operations this reads  $\Theta\hat{g}w^{\vec{k}} = w^{-\alpha\vec{k}} D^{\vec{k}} K_0$  where  $K_0$  indicates complex conjugation of everything which stands right of this operator as in  $\Theta\hat{g}w^{\vec{k}} C^{\vec{k}} = w^{-\alpha\vec{k}} D^{\vec{k}} C^{\vec{k}*} K_0$

The matrix  $D^{\vec{k}}$  contains the fractional translational part *tau* (page 66). For additional translations by lattice vectors additional phase factors  $\exp(-i\alpha\vec{k} \cdot \vec{R})$  must be added. However, such a situation usually does not occur. For time reversed operations the minus sign becomes a plus sign.

Note: that the chosen gauge of the Hamiltonian affects the periodicity of the Hamiltonian and WFs in k-space. In particular these objects are only k-periodic for the periodic gauge. In the relative gauge additional phase factors would occur.

#### **equivalentSites**

The site number `gisa=equivalentSites[isa]` is related to `isa` by this operation.

### 2.4.10 BerryCurvatureData

#### **class BerryCurvatureData**

This helper class is returned by *Slabify.hamAtKPoint* (page 50) if option `makedhk==True`. It contains the k-gradient of the Hamiltonian matrix and some additional data which are needed to calculate the Berry curvature with proper symmetry for periodic gauge.

#### **\_\_getitem\_\_()**

For compatibility with older pyfplo versions this object behaves a bit like a `list` of 3 complex `numpy.ndarrays` in that it can be indexed. The returned `numpy.ndarrays` are copies of the underlying data. So you cannot modify the *BerryCurvatureData* (page 67) data.

### 2.4.11 Site

#### **Site**

### 2.4.12 Data

For convenience:

#### **version**

copy of *pyfplo.common.version* (page 40)

#### **Version**

copy of *pyfplo.common.Version* (page 39)

#### **c\_elements**

copy of *pyfplo.common.c\_elements* (page 40)

#### **BandPlot**

copy of *pyfplo.common.BandPlot* (page 29)

#### **BandWeights**

copy of *pyfplo.common.BandWeights* (page 33)

#### **WeightDefinition**

copy of *pyfplo.common.WeightDefinition* (page 35)

**WeightDefinitions**copy of *pyfplo.common.WeightDefinitions* (page 34)**BandFileContext**copy of *pyfplo.common.BandFileContext* (page 27)**Vlevel**copy of *pyfplo.common.Vlevel* (page 39)**OptionSet**copy of *pyfplo.common.OptionSet* (page 36)**BandHeader**copy of *pyfplo.common.BandHeader* (page 32)

## 2.5 pyfplo.wanniertools

- *WanDefCreator* (page 69)
- *SingleOrbitalWandef* (page 70)
- *MultipleOrbitalWandef* (page 71)
- *Wandef* (page 71)
- *Contrib* (page 73)

This module provides some simple functions, which can be used to create `=.wandef` files. It is convenient in the context of *pyfplo.slabify* (page 41), where one usually needs to construct minimum basis models. For multi-contrib Wannier functions consult *Wandef* (page 71) and *Contrib* (page 73) and the example MgB2 in `DOC/WannierFunctions/examples/MgB2/bond-centered`. Example for simple Wannier functions:

```
wdc=wt.WanDefCreator(rcutoff=25,wftol=0.001,coeffformat='bin',
                    wfgriddirections=[[2,0,0],[0,2,0],[0,0,2]],
                    wfgridsubdiv=[1,1,1],savespininfo=True,
                    savebfield=True,savepositionoperator=True,
                    gradorder=1,
                    keeprunning=True,opendxinterface=False,
                    wfinrealspace=False,wfcoeffstats=True,
                    hamtstats=True,printT=True)

emin=-5
emax= 3
delower=1
deupper=1

# Add all wandefs for all 3d orbitals for both spins
# for Fe site 1 and 2.

wdc.add(wt.MultipleOrbitalWandef('Fe',[1,2],['3db'],
                                emin=emin,emax=emax,
                                delower=delower,deupper=deupper))

# Add all wandefs for all 2p orbitals for both spins
# for O site 7, 8 and 9.

wdc.add(wt.MultipleOrbitalWandef('O',[7,8,9],['2pb'],
                                emin=emin,emax=emax,
                                delower=delower,deupper=deupper))

wdc.writeFile('=.wandef')
```

## 2.5.1 WanDefCreator

```
class WanDefCreator (rcutoff=20, wftol=0.01, coeffformat='bin', wfgridbasis='conv', wfgridsubdiv=[1, 1, 1], wfgriddirections=[[1, 0, 0], [0, 1, 0], [0, 0, 1]], wfgridorigin=None, savespininfo=False, savebfield=False, savepositionoperator=False, gradorder=1, keeprunning=True, opendxinterface=False, wfinrealspace=True, wfcoeffstats=True, hamtstats=True, printT=True, automode=None)
```

The WanDefCreator object defines the content of the file `= .wandef`. You can add a single-**contrib wandef** via [SingleOrbitalWandef](#) (page 70) and several such **wandefs** via [MultipleOrbitalWandef](#) (page 71). More complex multi-**contrib wandefs** are added via [Wandef](#) (page 71) which in turn uses [Contrib](#) (page 73) to add single **contribs** to it.

### Parameters

- **rcutoff** (*float*) – Where in Bohr radii do we cutoff the hopping matrix elements to construct the tight-binding representation.
- **wftol** (*float*) – Hoppings below this threshold are not used.
- **coeffformat** (*str*) – Can be 'bin' or something else. If set to 'bin' the binary `+wancoeffbin` will be created and used.
- **wfgridbasis** (*str*) – Can be 'conv', 'prim' or 'cart'
- **wfgridsubdiv** (*list or numpy.ndarray of 3 int*) – The subdivision of the real-space grid on which to plot the Wannier functions.
- **wfgriddirections** (*list or numpy.ndarray of 3 vectors*) – The three directions, which span the real-space grid.
- **wfgridorigin** (*list or numpy.ndarray of 3 float or None*) – The origin of the real-space grid. Please consult `FPL0../DOC/WannierFunctions/wan_user.pdf` for more information.
- **savespininfo** (*bool*) – If True additional matrices (spin operators) are saved into `+wancoeff` for use by [slabify](#) (page 41). This option only works with full-relativistic calculations.
- **savebfield** (*bool*) – If True additional matrices (exchange field) are saved into `+wancoeff` for use by [slabify](#) (page 41). This option only works with full-relativistic calculations.
- **savepositionoperator** (*bool*) – If True additional matrices (position operator) are saved into `+wancoeff` for use by [slabify](#) (page 41).
- **gradorder** (*int*) – For the position operator a numerical gradient in k-space is needed. The gradient formula can have order 1,3,5 and 7. 1 or 3 are most likely best. The position operator converges very slowly with the number of SCF k-points. Often, `gradorder=3` helps to improve the accuracy.
- **opendxinterface** (*bool*) – If True additionally the old-style opendx interface files will be written (`WF.net`, `WF.cfg`, `opendxWF.dx`).
- **keeprunning** (*bool*) – If True fplo waits after the Wannier function calculation to make a quick re-run possible. If False fplo stops.
- **wfinrealspace** (*bool*) – If True the real space representation of the WFs will be calculated and written to the files `wfdata001 ...`. These files can be loaded by `xfplo =.in wfdata002 wfdata005`. Additionally the WF statistics files `+WF . . .` are written if `wfcoeffstats` is True.
- **wfcoeffstats** (*bool*) – Triggers the output of the files `+WFstat . . .`, which contain the size of the orbital contributions to a WF as a function of distance between WF

and orbital, which serves as a measure for localization. This option is inactive if `wfin_realspace` is `False`. It also triggers the output of `+WF_coefficients`, which gives a detailed account of the orbital contributions to the WFs.

- **hamtstats** (*bool*) – Triggers the output of the files `+T_...`, which contain the size of the hoppings around a WF as a function of distance, which serves as a measure for localization.
- **printT** (*bool*) – Triggers the output of `T=...` lines on standard output. Nowadays `+hamdata` is much more useful.
- **automode** (None or 'valence' or 'all') – If this is 'valence' (or 'all') no wandefs need to be specified. Instead the code creates wandefs for all valence (or really all) orbitals. If some semi core states overlap the valence energy region semi cores are added to the wandefs until we get a clear gap below the lowest band considered. This option leads to a slow down, but it is useful for automatic situations. Note, that a larger `rcutoff` might be needed in this case. For convenience the file `makewandeffromauto.py` is created after the file `+wancoff` was processed in the WF creation run. This file is overwritten. To modify it copy it.

An example with automatic creation is:

```
import pyfplo.wanniertools as wt

if __name__ == '__main__':
    wdc=wt.WanDefCreator(rcutoff=25,wftol=0.001,coeffformat='bin',
                        wfgriddirections=[[2,0,0],[0,2,0],[0,0,2]],
                        wfgridsubdiv=[1,1,1],savespininfo=True,
                        savebfield=True,savepositionoperator=True,
                        gradorder=1,
                        wfinrealspace=True,wfcoeffstats=True,hamtstats=True,
                        printT=True,automode='val')

    wdc.writeFile('=.wandef')
```

**add** (*wandef*)

Add a *SingleOrbitalWandef* (page 70), *MultipleOrbitalWandef* (page 71) or *Wandef* (page 71) to the definitions:

```
import pyfplo.wanniertools as wt
wdc=wt.WanDefCreator(rcutoff=30,wftol=0.001)
wdc.add(MultipleOrbitalWandef('Al',[1,4],['3s','3p'],
                             emin=-13,emax=-5,
                             delower=1,deupper=10))
wdc.writeFile()
```

**writeFile** (*filename='=.wandef'*)

Write the content into the file called *filename*.

## 2.5.2 SingleOrbitalWandef

**class SingleOrbitalWandef** (*el, site=1, orb='1s+0', emin=-1, emax=1, de=1, delower=1, deupper=1, fewind=[-100, 100, 1, 1, 0], lbands=None, ubands=None, xaxis=[1, 0, 0], zaxis=[0, 0, 1], fac=1.0, onoff=1, sxaxis='qua', szaxis=None*)

This class defines a single wandef entry with a single contrib. To collect multiple wandefs into a single python statement use *MultipleOrbitalWandef* (page 71).

Ignore *fewind* for now. The rest should be clear. Read the Wannier function documentation.

**Parameters**

- **el** (*str*) – An element name.

- **site** (*int*) – The site number.
- **fac** (*float or complex*) – The factor of the contrib. Makes no sense in this context, but is needed if you use *Wandef* (page 71) and *Contrib* (page 73).
- **emin, emax, de, delower, deupper** (*float*) – The energy window settings; all values can be single numbers or a list of two numbers (for spin up and spin down).
- **lbands, ubands** (*int or list or None*) – Optional. Either a single int (for both spins) or list of two int (spin up and down), specifying the lower and upper most band index to take into the projector. This narrows the energy window to a definite set of bands. which might be usefull if there is a separation of bands.
- **orb** (*str*) – can be something like
  - 5d : all 5d+m orbitals
  - 5d-1: only 5d-1

For full relativistic a 'u', 'd' or 'b' can follow the orbital name, specifying spin up,down or both.

  - 5d b : all 5d+m orbitals for both spins
  - 5d-1 b: only 5d-1, both spins

Alternatively, a spherical spinor basis can be chosen.

  - 5d5/2: all 5d5/2+mu/2 orbitals
  - 5d5/2-3/2: only this orbital
- **xaxis, zaxis** (*list or numpy.ndarray of 3 float*) – The local coordinate system in which the orbital is defined
- **sxaxis, szaxis** (*str or list or numpy.ndarray of 3 float*) – For full relativistic: the local coordinate system in which the spin eigenstates are defined. If it is a *str* is must be 'global', 'local', or 'quant', which can be abbreviated as 'glo', 'loc', or 'qua'. If it is vector *szaxis* must also be given.
  - 'glo': uses the global cartesian system
  - 'loc': uses the system defined by *xaxis* and *zaxis*
  - 'qua': uses the spin-quantization axis as defined in **fedit**.

### 2.5.3 MultipleOrbitalWandef

```
class MultipleOrbitalWandef(el, sites, orbs, emin=-1, emax=1, de=1, delower=1, deupper=1, fewind=[-100, 100, 1, 1, 0], lbands=None, ubands=None, xaxis=[1, 0, 0], zaxis=[0, 0, 1], fac=1.0, onoff=1, sxaxis='qua', szaxis=None)
```

Define multiple single-contrib wandefs.

#### Parameters

- **sites** (*list*) – a list of sites
- **orbs** (*list*) – a list of orbitals (see *SingleOrbitalWandef* (page 70))

For other parameters consult *SingleOrbitalWandef* (page 70).

### 2.5.4 Wandef

```
class Wandef(name, emin=-1, emax=1, de=1, delower=1, deupper=1, fewind=[-100, 100, 1, 1, 0], lbands=None, ubands=None, onoff=1, contribs=None)
```

This class defines a single wandef which can have multiple *Contrib* (page 73).

Example:

```
#!/usr/bin/env python

import sys
import numpy as np
import pyfplo.wanniertools as wt

# =====
def work():

    # operations
    s=np.sqrt(3.)/2.
    C6=np.matrix([[0.5,-s,0],[s,0.5,0],[0,0,1]],dtype='float')
    C3=C6.dot(C6)

    emin=-13
    emax=-10
    delower=1
    deupper=20

    xaxis=np.matrix([1,0,0],dtype='float')

    wdc=wt.WanDefCreator(rcutoff=15,wftol=0.001,coeffformat='bin',
                        wfgridbasis='conv',wfgridsubdiv=[30,30,30],
                        wfgriddirections=[[2,0,0],[0,2,0],[0,0,2]],
                        wfgridorigin=None,savespininfo=False,
                        savebfield=False,savepositionoperator=False,
                        gradorder=1,
                        keeprunning=True,opendxinterface=False,
                        wfinrealspace=True,wfcoeffstats=True,
                        hamtstats=True,printT=False)

    for isa in range(2,4):
        xaxisp=xaxis
        if isa == 3:
            xaxisp=-xaxis
        for i in range(1,4):
            wdc.add(wt.WanDef(
                name='Bs sp{0} at {1}'.format(i,isa),
                emin=emin,emax=emax,de=1.0,delower=delower,deupper=deupper)
                .addContrib(site=isa,orb='2s+0',fac=1.,xaxis=xaxis)
                .addContrib(site=isa,orb='2p+1',fac=np.sqrt(2),xaxis=xaxisp)
            )
            xaxisp=xaxisp.dot(C3)

    wdc.writeFile()
    return

# =====
#
# =====
if __name__ == '__main__':
    work()
```

### Parameters

- **name** (*str*) – The name of the Wannier function.
- **emin, emax, de, delower, deupper** (*float*) – The energy window settings; all values can be single numbers or a list of two numbers (for spin up and spin down).

- **lbands, ubands** (*int or list or None*) – Optional. Either a single int (for both spins) or list of two int (spin up and down), specifying the lower and upper most band index to take into the projector. This narrows the energy window to a definite set of bands. which might be usefull if there is a separation of bands.
- **contribs** (*list of Contrib*) – A list of instances of *Contrib* (page 73). Alternatively, *Contribs* can be added via *addContrib* (page 73).

Ignore *fewind* for now.

**addContrib** (*site, orb='1s+0', fac=1.0, difvec=[0.0, 0.0, 0.0], xaxis=[1.0, 0.0, 0.0], zaxis=[0.0, 0.0, 1.0], sxaxis='qua', szaxis=None*)

Add a single contrib. For meaning of the parameters consult *SingleOrbitalWandef* (page 70).

#### Returns

self for call chaining as in:

```
Wandef(...)\
    .addContrib(...)\
    .addContrib(...)
```

**Return type** *Wandef* (page 71)

## 2.5.5 Contrib

**class Contrib** (*site, orb='1s+0', fac=1.0, difvec=[0.0, 0.0, 0.0], xaxis=[1.0, 0.0, 0.0], zaxis=[0.0, 0.0, 1.0], sxaxis='qua', szaxis=None*)

A contrib for a *Wandef* (page 71).

#### Parameters

- **site** (*int*) – The site number.
- **orb** (*str*) – can be something like
  - 5d+2: real harmonics
  - 5d+2 up: real harmonics spin up (full relativistic)
  - 5d+2 dn: real harmonics spins down (full relativistic)
 Alternatively, a spherical spinor basis can be chosen.
  - 5d5/2-3/2: j=5/2 mu=-3/2
- **fac** (*float or complex*) – can be complex as in 0.577+0.5j
- **difvec** (*list or numpy.ndarray of 3 float*) – The connection vector from the Wannier center to this contrib's site.
- **xaxis, zaxis** (*list or numpy.ndarray of 3 float*) – The local coordinate system in which the orbital is defined
- **sxaxis, szaxis** (*str or list or numpy.ndarray of 3 float*) – For full relativistic: the local coordinate system in which the spin eigenstates are defined. If it is a *str* is must be 'global', 'local', or 'quant', which can be abbreviated as 'glo', 'loc', or 'qua'. If it is vector *szaxis* must also be given.
  - 'glo': uses the global cartesian system
  - 'loc': uses the system defined by *xaxis* and *zaxis*
  - 'qua': uses the spin-quantization axis as defined in **fedit**.



## 3.1 A basic tutorial

- *The bulk band structure* (page 75)
- *The bulk Fermi surface* (page 78)
- *Fermi surface cuts* (page 79)
- *Bulk projected bands* (page 81)
- *Finite slab with 10 unit cells* (page 83)
- *Finite slab with 10 unit cells (doubled in-plane cell)* (page 86)
- *Finite slab with 10 unit cells (doubled in-plane cell), one atom removed* (page 88)
- *Finite slab with 10 unit cells (doubled in-plane cell), 3 atoms removed* (page 91)
- *Semi infinite slab* (page 96)
- *Semi infinite slab, doubled planar cell* (page 99)

---

**Note:** You need to use the newer **xfbp/xfplo** version, which comes with `pyfplo` in order for the `cmd` scripts to work properly.

---

This example tries to explain how `pyfplo.slabify` (page 41) works in detail. The tutorial files are in `FPLO.../DOC/pyfplo/Examples/slabify/model` where `FPLO...` stands for your version's `FPLO` directory, e.g. `FPLO22.00-62`.

We use a hand written Hamiltonian file (`+hamdata`) containing some model data. Usually this is created by the Wannier function module of **fplo**. The model defines a single orbital tight binding model on a cubic lattice.

### 3.1.1 The bulk band structure

In a first step we plot the 3d bulk band structure. The python script `3d/slabify.py` is shown in the following

```
1  #!/usr/bin/env python
2
3  from __future__ import print_function
4  import sys
5
6  # If your pyfplo is not found you could also
7  # explicitly specify the pyfplo version path:
8  #sys.path.insert(0, "/home/magru/FPLO/FPLO22.00-62/PYTHON/doc");
9
```

(continues on next page)

(continued from previous page)

```

10 import numpy as np
11 import pyfplo.slabify as sla
12
13
14 print( '\npyfplo version=: {0}\nfrom: {1}\n'.format(sla.version,sla.__file__)
15 # protect against wrong version
16 #if fedit.version!='22.00': raise RuntimeError('pyfplo version is incorrect.')
17
18
19 # =====
20 #
21 # =====
22
23 def work():
24
25     hamdata='../+hamdata'
26
27     s=sla.Slabify()
28
29
30     s.object='3d'
31     s.printStructureSettings()
32
33     s.prepare(hamdata)
34
35     bp=sla.BandPlot()
36     bp.points=[
37         ['$~G', [0,0,0]],
38         ['X', [0.5,0,0]],
39         ['M', [0.5,0.5,0]],
40         ['$~G', [0,0,0]],
41         ['Z', [0,0,0.5]],
42         ['R', [0.5,0,0.5]],
43         ['L', [0.5,0.5,0.5]],
44         ['Z', [0,0,0.5]],
45     ]
46     bp.calculateBandPlotMesh(s.dirname)
47
48     s.calculateBandStructure(bp,suffix='_my_suffix');
49
50
51
52
53
54 # =====
55 #
56 # =====
57
58
59 if __name__ == '__main__':
60
61     work()
62

```

Line 8 can be uncommented and edited to make python search for pyfplo in a particular location (also see [Setup](#) (page 1)). Line 14 shows which pyfplo version was loaded and from where. If you uncomment line 16, the script is showing an error message if the pyfplo version does not match a particular version.

Please run the script (from 3d/README.rst):

```
# run the script as in
```

(continues on next page)

(continued from previous page)

```
./slabify.py | tee out
xftp bands.cmd

# and have a look at the output and into slabifyres/

#alternatively run

python ./slabify.py | tee out
xftp bands.cmd
```

Now for the actual script. Line 25 defines a convenient variable pointing to the Hamiltonian file. Line 27 makes `s` an instance of `slabify.Slabify` (page 41). In line 30 we tell it that we want a 3d object and since we don't set any other options it will be the simple cubic cell as defined in `+hamdata`. Line 33 now reads the Hamiltonian data and sets up the structure. After this step there will be several files in the output directory `slabifyres/` called `=.in_...`:

**=in\_step\_1\_3d\_enlarged:** This is the cell after the first structure manipulation step, which is the application of the `enlarge` (page 55) matrix.

**=in\_final\_PLlayer:** I the final result.

Of course in our case the two are identical, since we did not give any structure options except for `s.object=3d`. Furthermore, there are bandstructure files. Ignore `+sweights_sf_my_suffix` for now.

On with the code: Line 35 make `bp` an instance of `common.BandPlot` (page 29) while lines 36.. set the high symmetry points. Note, that the units are `slabify.Slabify.kscale` (page 56). One can set `kscale` to something else after the call to `s.prepare(hamdata)`. Line 46 sets up `bp` and writes the file `+points` to the directory defined in `slabify.Slabify.dirname` (page 54). If this variable is set by the user, it must be before the call to `slabify.Slabify.prepare` (page 42). Finally, in line 48 the band structure and band weights are calculated. The `suffix` argument helps to give the files custom made names.

Finally, if you successfully installed `xftp` and executed `xftp bands.cmd` you saw the simple bulk band structure (That is what you should see. (page 77)).

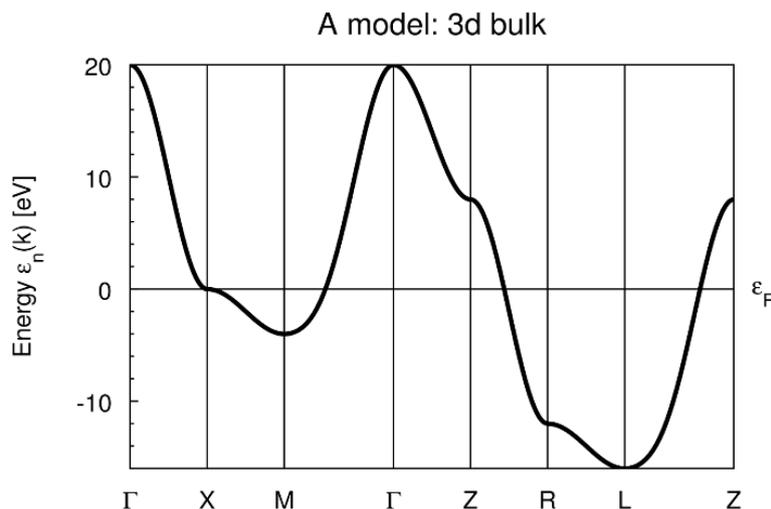


Fig. 3.1: The 3d band structure of the model.

It is strongly encouraged to also study the `.cmd` scripts to learn tricks. If you want to use other software for plotting you can use the `pyfplo.common.BandPlot.readBands` (page 31) and `pyfplo.common.BandWeights.readBandWeights` (page 34) methods to deal with the data as you please.

### 3.1.2 The bulk Fermi surface

Next we create a bulk Fermi surface of the model. We created an appropriate `=.in` with symmetry P1 and a simple cubic lattice with correct lattice constants (as in `+hamdata`). Next, we opened `xfplo` in fermi surface mode, defined a mesh, exported it, saved the settings in `=.xef`, stopped before the automatic `fplo` run and quit the program. Now, we got `=.kp` and `=.xef`.

Please change into `FS/` and run the script (from `FS/README.rst`):

```
# Here we created an appropriate =.in with the correct
# lattice. The we used xfplo -fs to setup and export a k-mesh to =.kp.
# We use slabify to calculate the Fermisurface corresponding to the
# model data in ../+hamdata.
#
# run

./slabify.py | tee out
xfplo =.xef

# to see the result.
```

You should see something like this. (page 78)

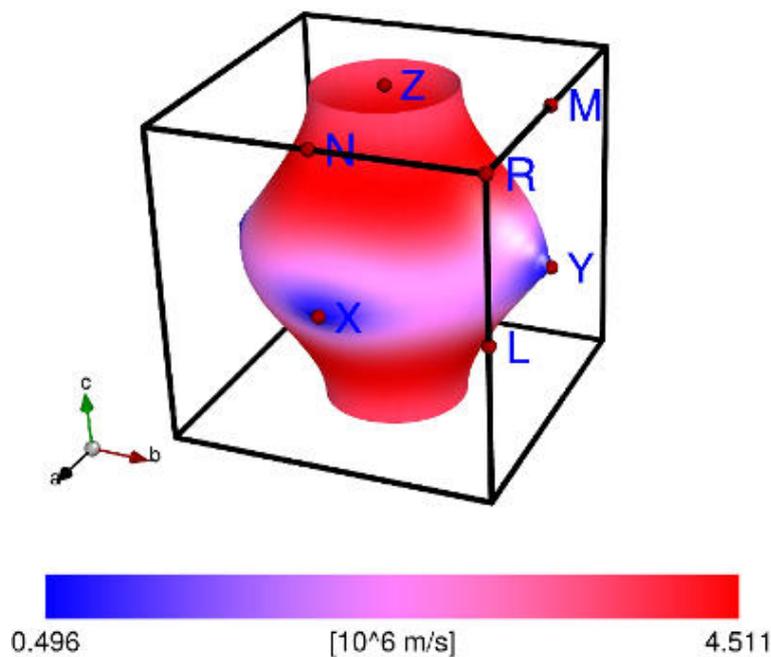


Fig. 3.2: The 3d Fermi surface extracted via slabify.

Let's explain the script `FS/slabify.py` now.

```
1  #! /usr/bin/env python
2
3  from __future__ import print_function
4  import sys
5
6  # If your pyfplo is not found you could also
7  # explicitly specify the pyfplo version path:
8  #sys.path.insert(0, "/home/magru/FPLO/FPLO22.00-62/PYTHON/doc");
9
10 import numpy as np
11 import pyfplo.slabify as sla
```

(continues on next page)

(continued from previous page)

```

12
13
14 print( '\npyfplo version=: {0}\nfrom: {1}\n'.format(sla.version,sla.__file__))
15 # protect against wrong version
16 #if fedit.version!='22.00': raise RuntimeError('pyfplo version is incorrect.')
17
18
19 # =====
20 #
21 # =====
22
23 def work():
24
25     hamdata='../+hamdata'
26
27
28
29     s=sla.Slabify()
30     # set output directory to current directory
31     s.dirname='.'
32
33
34     s.object='3d'
35     s.printStructureSettings()
36
37     s.prepare(hamdata)
38
39
40
41     bp=sla.BandPlot()
42     bp.readBandPlotMesh('../=.kp')
43
44
45     s.calculateBandStructure(bp);
46
47
48 # =====
49 #
50 # =====
51
52
53 if __name__ == '__main__':
54
55     work()
56

```

In line 31 we explicitly set `slabify.Slabify.dirname` (page 54) to the local directory. This will put everything into the same directory where `slabify.py` was executed. The setup continues as in *The bulk band structure* (page 75) until line 42 where instead of setting a path through high symmetry points the k-point file from `xfplo` is used.

### 3.1.3 Fermi surface cuts

There is an option to create Fermi surface cuts.

Please change into `cuts/` and run the script (from `cuts/README.rst`):

```

#run
./slabify.py | tee out

```

(continues on next page)

(continued from previous page)

```

xftp cuts.cmd
#have a look at slabifyres/ and cuts.png

```

You should see something like this. (page 80)

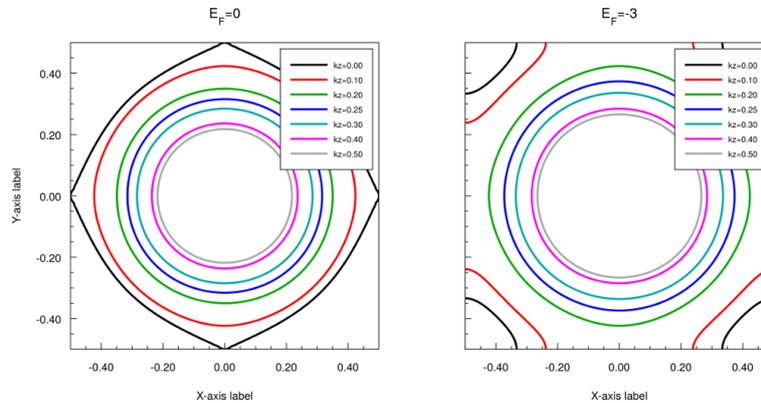


Fig. 3.3: Cuts through the 3d Fermi surface of the model.

Let's explain the script `cuts/slabify.py` now.

```

1  #!/usr/bin/env python
2  from __future__ import print_function
3  import sys
4
5  # If your pyfplo is not found you could also
6  # explicitly specify the pyfplo version path:
7  #sys.path.insert(0, "/home/magru/FPLO/FPLO22.00-62/PYTHON/doc");
8
9  import numpy as np
10 import pyfplo.slabify as sla
11
12
13 print( '\npyfplo version=: {0}\nfrom: {1}\n'.format(sla.version,sla.__file__)
14 # protect against wrong version
15 #if fedit.version!='22.00': raise RuntimeError('pyfplo version is incorrect.')
16
17
18 # =====
19 #
20 # =====
21
22 def work():
23
24     hamdata='../+hamdata'
25
26     s=sla.Slabify()
27
28
29     s.object='3d'
30     s.printStructureSettings()
31
32     s.prepare(hamdata)
33
34
35     fso=sla.FermiSurfaceOptions()

```

(continues on next page)

(continued from previous page)

```

36     fso.setMesh(100, [-0.5, 0.5], 100, [-0.5, 0.5])
37
38     n=10
39     for ikz in range(0, n+1):
40         kz=ikz*0.5/n
41         fso.setPlane([1, 0, 0], [0, 1, 0], [0, 0, kz])
42         fso.fermienergy=0
43         suffix='_kz={0:05.2f}'.format(kz)
44         # we need to set forcerecalculation=True since
45         # the k-plane changes with every loop
46         s.calculateFermiSurfaceCuts(fso, suffix=suffix, forcerecalculation=True);
47
48
49         fso.fermienergy=-3
50         suffix='_kz={0:05.2f}_ef={1:05.2f}'.format(kz, fso.fermienergy)
51         # Now we do NOT need to set forcerecalculation=True since
52         # only the Fermi energy changed.
53         # The actual example is so fast that you do not see the impact.
54         s.calculateFermiSurfaceCuts(fso, suffix=suffix, forcerecalculation=False);
55
56
57
58
59 # =====
60 #
61 # =====
62
63
64 if __name__ == '__main__':
65
66     work()
67

```

In lines 35-36 we set up a `slabify.FermiSurfaceOptions` (page 60) instance with default axes and define a 100x100 2d mesh which stretches from -0.5 to 0.5 in units of `slabify.Slabify.kscale` (page 56) which happens to be  $2\pi/a$  by default.

Next a loop is done over various `kz`-values. In line 41 the plane (axes and origin) is specified explicitly. The origin `[0, 0, kz]` puts this plane parallel to `kx`, `ky` through the `kz`-value. Line 42 sets the Fermi energy and line 43 a filename suffix. In line 46 `Slabify.calculateFermiSurfaceCuts` (page 44) is called with the flag `forcerecalculation=True`, which is explained under the link above. After this line the files

```

+cut_band_sf,
+cut_bweights_sf and
+cuts_kz=00.00.spin1_kz=...

```

are created in `slabifyres/`.

Line 49 sets a different Fermi energy and line 54 recalculates the files `+cuts_kz=00.00.spin1_kz=...` (iso lines) but not the other files because of `forcerecalculation=False`. In a real example with lots of orbitals this avoids the expensive diagonalization step. There are also `+cutswithweights` files if the `wds` option is specified.

### 3.1.4 Bulk projected bands

One can calculate the bulk projected bands (BPB) as energy distribution curves (EDC) or Fermi surface projections.

Please change into `bpb/` and run the script (from `bpb/README.rst`):

```
#run

./slabify.py | tee out
xfbp bpbdc.cmd
xfbp bpbfs.cmd

#have a look at slabifyres/ and *.png
```

You should see something like this. (page 82)

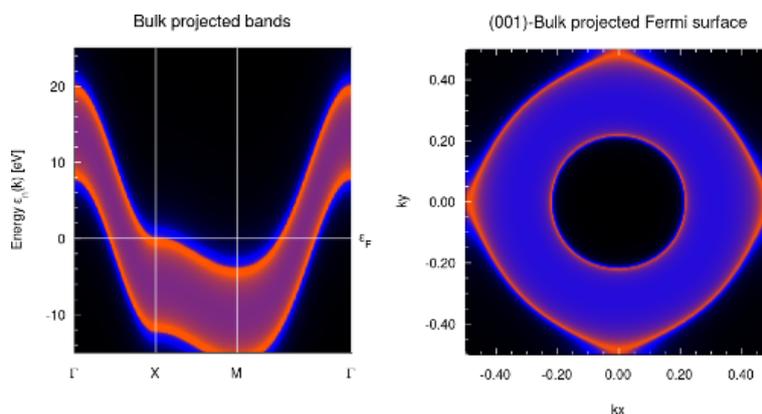


Fig. 3.4: The bulk projected bands of the model.

Let's explain the script bpb/slabify.py now.

```
1  #!/usr/bin/env python
2  from __future__ import print_function
3  import sys
4
5  # If your pyfplo is not found you could also
6  # explicitly specify the pyfplo version path:
7  #sys.path.insert(0, "/home/magru/FPLO/FPLO22.00-62/PYTHON/doc");
8
9  import numpy as np
10 import pyfplo.slabify as sla
11
12
13 print( '\npyfplo version:= {0}\nfrom: {1}\n'.format(sla.version, sla.__file__)
14 # protect against wrong version
15 #if fedit.version!='22.00': raise RuntimeError('pyfplo version is incorrect.')
16
17
18 # =====
19 #
20 # =====
21
22 def work():
23
24     hamdata='../+hamdata'
25
26     s=sla.Slabify()
27
28
29     s.object='3d'
30     s.printStructureSettings()
31
```

(continues on next page)

(continued from previous page)

```

32     s.prepare(hamdata)
33
34     bp=sla.BandPlot()
35     bp.points=[
36         ['$~G', [0,0,0]],
37         ['X', [0.5,0,0]],
38         ['M', [0.5,0.5,0]],
39         ['$~G', [0,0,0]],
40     ]
41     bp.ndiv=100
42     bp.calculateBandPlotMesh(s.dirname)
43
44     Ne=200
45     Nkz=100
46
47     ec=sla.EnergyContour(Ne,-15,25)
48     print(ec)
49     s.calculateBulkProjectedEDC(bp,ec,[0,0,1],nz=Nkz)
50
51
52     Nk=200
53     fso=sla.FermiSurfaceOptions()
54     fso.setMesh(Nk,[-0.5,0.5],Nk,[-0.5,0.5])
55     fso.setPlane([1,0,0],[0,1,0],[0,0,0])
56     fso.fermienergyim=2./Nk*10
57
58     # now make projected fermi surfaces
59     s.calculateBulkProjectedFS(fso,zaxis=[0,0,1],nz=Nkz)
60
61
62
63     # =====
64     #
65     # =====
66
67
68     if __name__ == '__main__':
69
70         work()
71

```

In line 34-42 we define a path through the 2d BZ and set the maximum number of subdivisions for the path segments to 150. In line 47 we define an *EnergyContour* (page 59) for the EDC. In line 49 the bulk projected EDC is calculated (*Slabify.calculateBulkProjectedEDC* (page 43)) by defining a projection axis *zaxis* (page 43) and setting the number of integration intervals for the projection. The code finds out which is the shortest period in the projection direction and integrates over this period via linear interpolation. A complex representation of the delta functions is used, which necessitates an imaginary part in *EnergyContour* (automatic in our case). The result is written to `slabifyres/+bpb_fs.spin1`.

In lines 52-59 we set up a 2d mesh in a plane perpendicular to the projection axis, set an imaginary energy part (needs some experimenting for good results) and do the calculation (*Slabify.calculateBulkProjectedFS* (page 43)). The result is written to `'slabifyres/+bpfs_sf.spin1'`.

### 3.1.5 Finite slab with 10 unit cells

We construct a finite slab with 10 unit cells. Please change into `slab10/` and run the script (from `slab10/README.rst`):

```
# run the script as in

./slabify.py | tee out
xftp weigths.cmd

# and have a look at the output and into slabifyres/

#alternatively run

python ./slabify.py | tee out
xftp weigths.cmd
```

You should see something like this. (page 84)

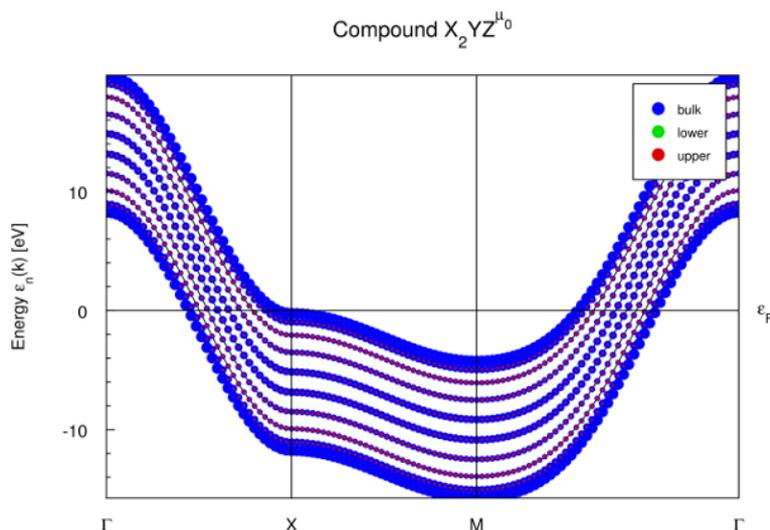


Fig. 3.5: The 10-cell finite slab band structure.

Let's explain the script slab10/slabify.py now.

```
1  #!/usr/bin/env python
2  from __future__ import print_function
3  import sys
4
5  # If your pyfplo is not found you could also
6  # explicitly specify the pyfplo version path:
7  #sys.path.insert(0, "/home/magru/FPLO/FPLO22.00-62/PYTHON/doc");
8
9  import numpy as np
10 import pyfplo.slabify as sla
11
12
13 print( '\npyfplo version=: {0}\nfrom: {1}\n'.format(sla.version,sla.__file__))
14 # protect against wrong version
15 #if fedit.version!='22.00': raise RuntimeError('pyfplo version is incorrect.')
16
17
18 # =====
19 #
20 # =====
21
22 def work():
23
24
```

(continues on next page)

(continued from previous page)

```

25 hamdata='../+hamdata'
26
27 s=sla.Slabify()
28
29
30 s.object='slab'
31 s.anchor=-0.001
32 s.numberoflayers=10
33 s.printStructureSettings()
34
35 s.prepare(hamdata)
36
37 bp=sla.BandPlot()
38 bp.points=[
39     ['$~G', [0, 0, 0]],
40     ['X', [0.5, 0, 0]],
41     ['M', [0.5, 0.5, 0]],
42     ['$~G', [0, 0, 0]],
43 ]
44 bp.calculateBandPlotMesh(s.dirname)
45
46 s.calculateBandStructure(bp);
47
48
49 # all orbitals up to 10 length units at the upper surface
50 upper=s.orbitalNamesByDepth(-1,10)
51 # all orbitals up to 10 length units at the lower surface
52 lower=s.orbitalNamesByDepth(10,-1)
53 # a nice python trick to get the rest list
54 rest=list(set(s.orbitalNames())-set(upper)-set(lower))
55
56
57 wds=sla.WeightDefinitions()
58 wds.add('bulk').addLabels(rest)
59 wds.add('lower').addLabels(lower)
60 wds.add('upper').addLabels(upper)
61
62 bw=sla.BandWeights(s.dirname+'/+bweights_sf')
63 bw.addWeights(wds,s.dirname+'/+bwsum_sf')
64
65
66
67
68
69 # =====
70 #
71 # =====
72
73
74 if __name__ == '__main__':
75
76     work()
77

```

This time we make a slab structure (free standing x,y-periodic slab) in line 30. The *zaxis* (page 55) is not set and hence has its default value  $[0, 0, 1]$ . Line 32 sets *numberoflayers* (page 55). We set the *anchor* (page 55) to slightly below 0 in line 31. What happens now:

- The default 3d unit cell is enlarged to produce the “enlarged 3d” unit cell in `in_step_1_3d_enlarged` (identical to the default here).
- The enlarged 3d unit cell will be transformed into a 3d unit cell which has the a and b axis perpendicular to

the `zaxis`. (the `c` axis it not necessarily parallel to the `z` axis) This creates the elementary building block for `'slabs'/'semislabs'`.

- Then this block is repeated `numberoflayers` of times in `c`-direction to get the 3d “layered cell” (`= .in_step_2_3d_layered`).
- Next, the “layered cell” will be anchored, which means that the `z`-periodic 3d “layered cell” will be shifted such that `anchor` (page 55) becomes the `z=0`-plane. After this the cell is cut at `z=0` and `z=1`, which results in a free standing slab of the length of one “layered cell” unit cell. The result is found in `= .in_step_3_slab_anchored` For `'semislab'`s the resulting cell is used as surface block (layer) such that the upper boundary (largest `z`) represents the surface to the vaccum. Internally this block is repeated indefinitely at the lower boundary of the unit cell to form a semi-infinite slab.

The next option is only needed for `'slab'`s.

- (Not used here) After this, `cutlayersat` (page 55) is applied to cut away some layers at the top and bottom of the slab.

The next option is currently only used for `'slab'`s, which means that we cannot remove selected atoms for `'semislab'`s yet.

- (Not used here) Finally, all atoms in the list `cutatoms` (page 55) are removed from the slab. The number in this list must be taken from the site numbers in `= .in_step_4_slab_cutlayers`. Use **xfplo** and point the mouse at the atoms: the status bar will show the site number as in `A1(S3,W3,T3) . . . S3` means site 3. The result after this step is `= .in_final_PLayer`

We setup the high symmetry points an calculate bands and weights up to line 46.

What comes next is adding all orbitals at the two sides and in the middle to get band weights to color the band structure with. In the current slab this is not very interesting. But it becomes interesting later. Line 51 defines a list of orbitals which sit up to 10 length units (usually Bohr radii) deep at the upper side of the slab. Note the `-1` in the first argument. This is explained in *orbitalNamesByDepth* (page 46) and *orbitalIndicesByDepth* (page 45). It just takes orbitals at the lower side which lie below the lower vaccum boundary, which is no orbitals. The same is done for the `lower` variable. The line 55 shows a neat trick to calculate the rest (which is not orbitals in the upper or lower set of orbitals).

Line 58-61 defines three added weights named `'bulk'`, `'lower'` and `'upper'` and line 63-64 read the bandweights file and write the added weights to `+bwsum_sf`.

Now, we can color the band weights according to their “surface character” which is shown in *The 10-cell finite slab band structure*. (page 84). Nothing interesting to see here. (But later!)

### 3.1.6 Finite slab with 10 unit cells (doubled in-plane cell)

We construct a finite slab with 10 unit cells, doubled in the `a, b`-plane.

Please change into `slab10x2/` and run the script (from `slab10x2/README.rst`):

```
# run the script as in
./slabify.py | tee out
xfbp weigths.cmd

# and have a look at the output and into slabifyres/

#alternatively run
python ./slabify.py | tee out
xfbp weigths.cmd
```

*You should see something like this.* (page 87)

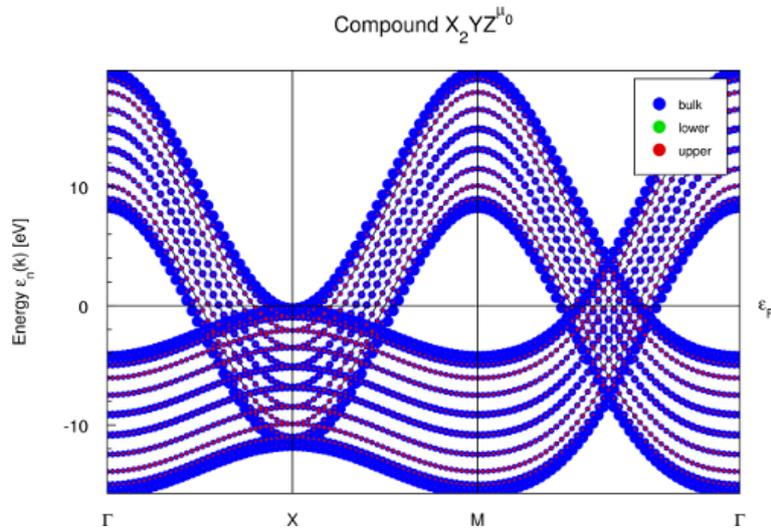


Fig. 3.6: The 10-cell slab with doubled in-plane unit cell.

Let's explain the script `slab10x2/slabify.py` now.

```

1  #!/usr/bin/env python
2  from __future__ import print_function
3  import sys
4
5  # If your pyfplo is not found you could also
6  # explicitly specify the pyfplo version path:
7  #sys.path.insert(0, "/home/magru/FPLO/FPLO22.00-62/PYTHON/doc");
8
9  import numpy as np
10 import pyfplo.slabify as sla
11
12
13 print( '\npyfplo version=: {0}\nfrom: {1}\n'.format(sla.version,sla.__file__)
14 # protect against wrong version
15 #if fedit.version!='22.00': raise RuntimeError('pyfplo version is incorrect.')
16
17
18 # =====
19 #
20 # =====
21
22 def work():
23
24
25     hamdata='../+hamdata'
26
27     s=sla.Slabify()
28
29
30     s.object='slab'
31     s.enlarge=[ [1,1,0], [-1,1,0], [0,0,1]]
32     s.anchor=-0.001
33     s.numberoflayers=10
34     s.printStructureSettings()
35
36     s.prepare(hamdata)
37
38     bp=sla.BandPlot()
39     bp.points=[

```

(continues on next page)

```

40     ['$~G', [0,0,0]],
41     ['X', [0.5,0,0]],
42     ['M', [0.5,0.5,0]],
43     ['$~G', [0,0,0]],
44     ]
45     bp.calculateBandPlotMesh(s.dirname)
46
47     s.calculateBandStructure(bp);
48
49
50
51     # all orbital indices up to 10 length units at the upper surface
52     iupper=s.orbitalIndicesByDepth(-1,10)
53     # all orbital indices up to 10 length units at the lower surface
54     ilower=s.orbitalIndicesByDepth(10,-1)
55     # get all orbital indices
56     iall=s.orbitalIndicesByDepth()
57     # a nice python trick to get the rest list
58     irest=list(set(iall)-set(iupper)-set(ilower))
59
60     print( 'the orbital indices lists:',ilower,irest,iupper)
61
62
63
64
65     wds=sla.WeightDefinitions()
66     wds.add('bulk').addLabels(s.orbitalNames(irest))
67     wds.add('lower').addLabels(s.orbitalNames(ilower))
68     wds.add('upper').addLabels(s.orbitalNames(iupper))
69
70     bw=sla.BandWeights(s.dirname+'/+bweights_sf')
71     bw.addWeights(wds,s.dirname+'/+bwsum_sf')
72
73
74
75
76     # =====
77     #
78     # =====
79
80
81     if __name__ == '__main__':
82
83         work()
84

```

We now used an *enlarge* (page 55) matrix to double the 3d unit cell (Line 31). In contrast to the previous example we now use another route to get the needed orbital sets (Line 52-60). This also alters the line 66-68. Whatever method you use depends on what you want to do. The orbital-name based approach allows specific filtering, which was shown elsewhere.

This example was not very interesting either. Let's go on.

### 3.1.7 Finite slab with 10 unit cells (doubled in-plane cell), one atom removed

We construct a finite slab with 10 unit cells, doubled in the a, b-plane with one atom removed at the upper side.

Please change into `slab10x2mod1/` and run the script (from `slab10x2mod1/README.rst`):

```
# run the script as in

./slabify.py | tee out
xftp weights.cmd
xftp sweights.cmd

# and have a look at the output and into slabifyres/

#alternatively run

python ./slabify.py | tee out
xftp weights.cmd
xftp sweights.cmd
```

You should see something like this. (page 89)

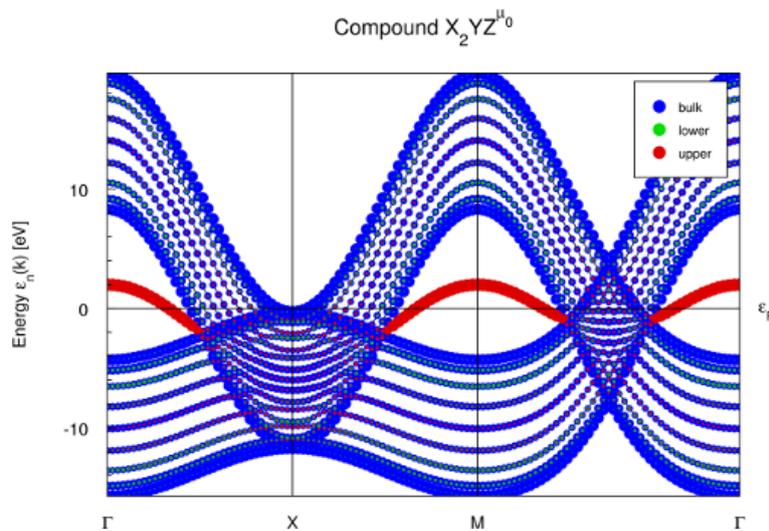


Fig. 3.7: 10-cell slab, double in-plane cell, one atom removed.

Now, that is interesting. We see a surface state stemming from the removed atom at the upper boundary.

Let's explain the script `slab10x2mod1/slabify.py` now.

```
1  #!/usr/bin/env python
2  from __future__ import print_function
3  import sys
4
5  # If your pyfplo is not found you could also
6  # explicitly specify the pyfplo version path:
7  #sys.path.insert(0, "/home/magru/FPLO/FPLO22.00-62/PYTHON/doc");
8
9  import numpy as np
10 import pyfplo.slabify as sla
11
12
13 print( '\npyfplo version=: {0}\nfrom: {1}\n'.format(sla.version,sla.__file__)
14 # protect against wrong version
15 #if fedit.version!='22.00': raise RuntimeError('pyfplo version is incorrect.')
16
17
18 # =====
19 #
20 # =====
```

(continues on next page)

```

21
22 def work():
23
24
25     hamdata='../+hamdata'
26
27     s=sla.Slabify()
28
29
30     s.object='slab'
31     s.enlarge=[ [1,1,0], [-1,1,0], [0,0,1]]
32     s.anchor=-0.001
33     s.numberoflayers=10
34     s.cutatoms=[20]
35     s.printStructureSettings()
36
37     s.prepare(hamdata)
38
39     bp=sla.BandPlot()
40     bp.points=[
41         ['$~G', [0,0,0]],
42         ['X', [0.5,0,0]],
43         ['M', [0.5,0.5,0]],
44         ['$~G', [0,0,0]],
45     ]
46     bp.calculateBandPlotMesh(s.dirname)
47
48     s.calculateBandStructure(bp);
49
50
51
52
53
54     # all orbitals up to 10 length units at the upper surface
55     upper=s.orbitalNamesByDepth(-1,10)
56     # all orbitals up to 10 length units at the lower surface
57     lower=s.orbitalNamesByDepth(10,-1)
58     # a nice python trick to get the rest list
59     rest=list(set(s.orbitalNames())-set(upper)-set(lower))
60
61
62     wds=sla.WeightDefinitions()
63     wds.add('bulk').addLabels(rest)
64     wds.add('lower').addLabels(lower)
65     wds.add('upper').addLabels(upper)
66
67     bw=sla.BandWeights(s.dirname+'/+bweights_sf')
68     bw.addWeights(wds,s.dirname+'/+bwsum_sf')
69
70
71     # now we first create BandWeights
72     bw=sla.BandWeights(s.dirname+'/+sweights_sf')
73     # such that we can read its header and get the labels
74     labels=bw.header().labels
75     l10=['band000000010']
76     # and we get all but band number 10
77     rest=list(filter(lambda x: x not in l10 ,labels))
78     wds=sla.WeightDefinitions()
79     wds.add('rest').addLabels(rest)
80     wds.add('b10').addLabels(l10)
81

```

(continues on next page)

(continued from previous page)

```

82     print( wds)
83
84     bw.addWeights(wds,s.dirname+'/+swwsum_sf')
85
86
87     # =====
88     #
89     # =====
90
91
92     if __name__ == '__main__':
93
94         work()
95

```

Now, we remove atom 20 in line 34. Have a look at `.in_final_PLlayer`, you will see that an atom at the top is missing.

In lines 72-84 we do something new. We define added band weights for the file `+sweights_sf`, which contains fatband data of layer-coordinate versus k-path. The band weights are referring to the bands not orbitals. This file tells us in which layer a particular bands has what weight. By defining added weights for band 10 and the rest we can see in [this Figure](#) (page 91) that the surface state in *10-cell slab, double in-plane cell, one atom removed*. (page 89) really lives in the first layer and the rest of the bands does not.

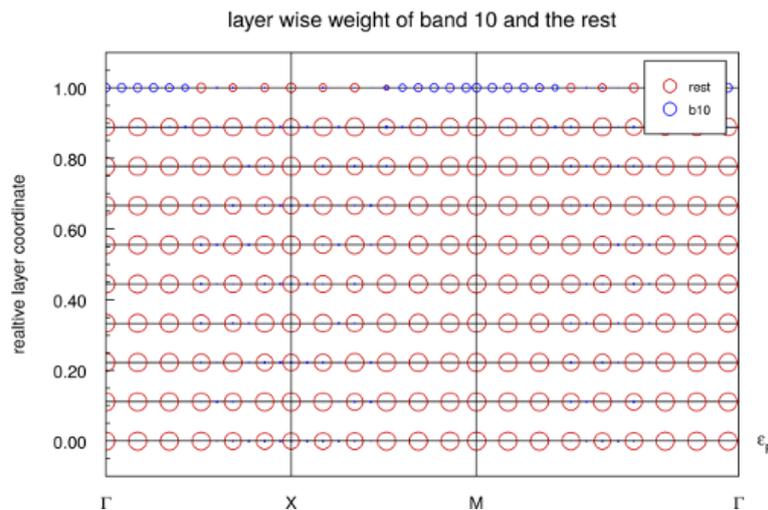


Fig. 3.8: Layer resolved weights of the bands.

### 3.1.8 Finite slab with 10 unit cells (doubled in-plane cell), 3 atoms removed

We construct a finite slab with 10 unit cells, doubled in the  $a, b$ -plane with one atom removed at the upper side and two at the lower side.

Please change into `slab10x2mod2/` and run the script (from `slab10x2mod2/README.rst`):

```

# run the scripts as in

./slabify.py | tee out
./cuts.py | tee out
xftp weights.cmd
xftp sweights.cmd
xftp wcuts.cmd

```

(continues on next page)

(continued from previous page)

```
# and have a look at the output and into slabifyres/

# alternatively run

python ./slabify.py | tee out
python ./cuts.py | tee out
xfbp weigths.cmd
xfbp sweigths.cmd
xfbp wcuts.cmd
```

You should see something like this. (page 92)

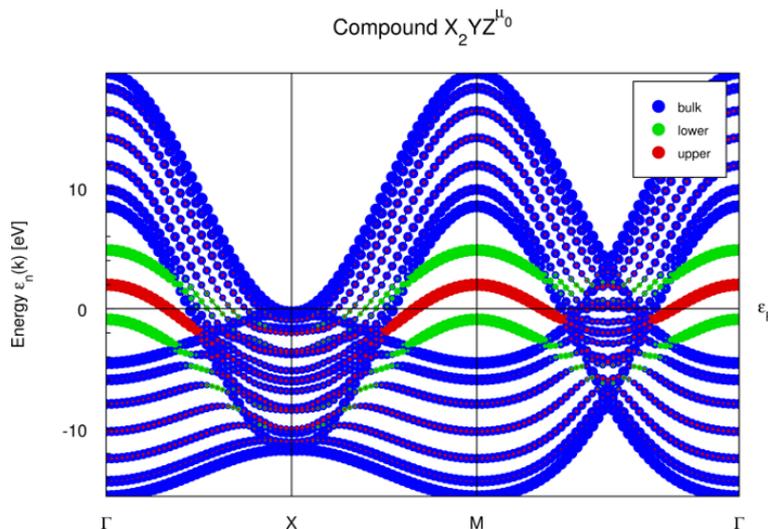


Fig. 3.9: 10-cell slab, doubled in-plane, 3 atoms removed.

Now, that is even more interesting. We see a surface state stemming from the removed atom at the upper surface and two surface states from the atom removal at the lower boundary.

Let's explain the script `slab10x2mod2/slabify.py` now.

```
1  #!/usr/bin/env python
2  from __future__ import print_function
3  import sys
4
5  # If your pyfplo is not found you could also
6  # explicitly specify the pyfplo version path:
7  #sys.path.insert(0, "/home/magru/FPLO/FPLO22.00-62/PYTHON/doc");
8
9  import numpy as np
10 import pyfplo.slabify as sla
11
12
13 print( '\npyfplo version=: {0}\nfrom: {1}\n'.format(sla.version,sla.__file__)
14 # protect against wrong version
15 #if fedit.version!='22.00': raise RuntimeError('pyfplo version is incorrect.')
16
17
18 # =====
19 #
20 # =====
21
22 def work():
```

(continues on next page)

(continued from previous page)

```

23
24
25 hamdata='../+hamdata'
26
27 s=sla.Slabify()
28
29
30 s.object='slab'
31 s.enlarge=[ [1,1,0], [-1,1,0], [0,0,1]]
32 s.anchor=-0.001
33 s.numberoflayers=10
34 s.cutatoms=[2,4,20]
35 s.printStructureSettings()
36
37 s.prepare(hamdata)
38
39 bp=sla.BandPlot()
40 bp.points=[
41     ['$~G', [0,0,0]],
42     ['X', [0.5,0,0]],
43     ['M', [0.5,0.5,0]],
44     ['$~G', [0,0,0]],
45     ]
46 bp.calculateBandPlotMesh(s.dirname)
47
48 s.calculateBandStructure(bp);
49
50
51
52 # all orbitals up to 10 length units at the upper surface
53 upper=s.orbitalNamesByDepth(-1,10)
54 # all orbitals up to 10 length units at the lower surface
55 lower=s.orbitalNamesByDepth(10,-1)
56 # a nice python trick to get the rest list
57 rest=list(set(s.orbitalNames())-set(upper)-set(lower))
58
59
60 wds=sla.WeightDefinitions()
61 wds.add('bulk').addLabels(rest)
62 wds.add('lower').addLabels(lower)
63 wds.add('upper').addLabels(upper)
64
65 bw=sla.BandWeights(s.dirname+'/+bweights_sf')
66 bw.addWeights(wds,s.dirname+'/+bwsum_sf')
67
68
69 # now we first create BandWeights
70 bw=sla.BandWeights(s.dirname+'/+sweights_sf')
71 # such that we can read its header and get the labels
72 labels=bw.header().labels
73
74
75
76 # and we get addweights for bands 8, 9, 10 and the rest
77
78 ibands=[8,9,10]
79
80 bands=['band{0:09d}'.format(x) for x in ibands]
81 rest=list(filter(lambda x: x not in bands,labels))
82 wds=sla.WeightDefinitions()
83 wds.add('rest').addLabels(rest)

```

(continues on next page)

(continued from previous page)

```

84     for ib,i in enumerate(ibands):
85         wds.add('b{0:02}'.format(i)).addLabels([bands[ib]])
86
87
88     bw.addWeights(wds,s.dirname+'/swwsum_sf',vlevel=sla.Vlevel.All)
89
90
91     # =====
92     #
93     # =====
94
95
96 if __name__ == '__main__':
97
98     work()
99

```

Line 34 now removes sites 2, 4 and 20. In lines 80-88 we use a different technique to extract the layer-wise fatbands for 3 bands and the rest resulting in *this Figure* (page 94).

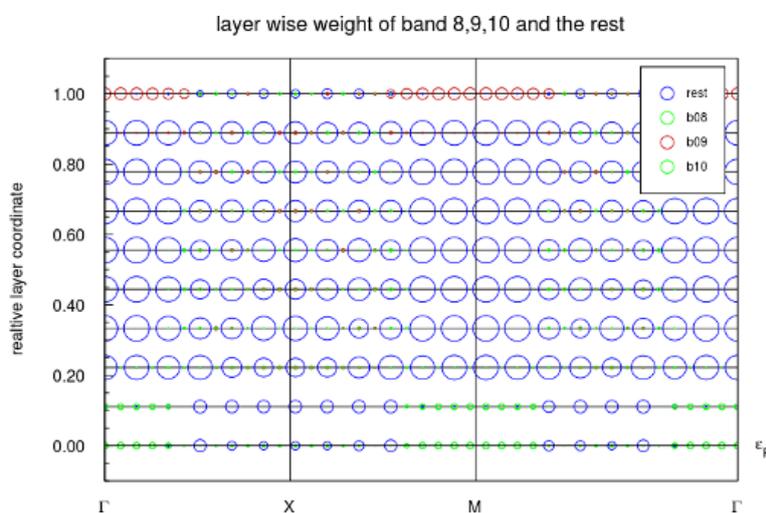


Fig. 3.10: Layer resolved weights of the bands.

The second script in this example is `slab10x2mod2/cuts.py`:

```

1  #!/usr/bin/env python
2  from __future__ import print_function
3  import sys
4
5  # If your pyfplo is not found you could also
6  # explicitly specify the pyfplo version path:
7  #sys.path.insert(0, "/home/magru/FPLO/FPLO22.00-62/PYTHON/doc");
8
9  import numpy as np
10 import pyfplo.slabify as sla
11
12
13 print( '\npyfplo version:= {0}\nfrom: {1}\n'.format(sla.version,sla.__file__)
14 # protect against wrong version
15 #if fedit.version!='22.00': raise RuntimeError('pyfplo version is incorrect.')
16
17
18 # =====

```

(continues on next page)

(continued from previous page)

```

19 #
20 # =====
21
22 def work():
23
24     hamdata='./+hamdata'
25
26     s=sla.Slabify()
27
28
29
30     s.object='slab'
31     s.enlarge=[ [1,1,0], [-1,1,0], [0,0,1]]
32     s.anchor=-0.001
33     s.numberoflayers=10
34     s.cutatoms=[2,4,20]
35     s.printStructureSettings()
36
37     s.prepare(hamdata)
38
39
40     # all orbitals up to 10 length units at the upper surface
41     upper=s.orbitalNamesByDepth(-1,10)
42     # all orbitals up to 10 length units at the lower surface
43     lower=s.orbitalNamesByDepth(10,-1)
44     # a nice python trick to get the rest list
45     rest=list(set(s.orbitalNames())-set(upper)-set(lower))
46
47
48     wds=sla.WeightDefinitions()
49     wds.add('bulk').addLabels(rest)
50     wds.add('lower').addLabels(lower)
51     wds.add('upper').addLabels(upper)
52
53
54     fso=sla.FermiSurfaceOptions()
55     fso.setMesh(200, [-0.5,0.5], 200, [-0.5,0.5])
56     fso.setPlane([1,0,0], [0,1,0], [0,0,1])
57     fso.fermienergy=0
58     s.calculateFermiSurfaceCuts(fso,wds=wds,suffix='E=0',
59                               forcerecalculation=True);
60
61     fso.fermienergy=-1.
62     s.calculateFermiSurfaceCuts(fso,wds=wds,suffix='E=-1.',
63                               forcerecalculation=False);
64
65
66
67
68 # =====
69 #
70 # =====
71
72
73 if __name__ == '__main__':
74
75     work()
76

```

Here we first define the addweights in lines 40-51. Then we give *wds* as argument to *Slabify.calculateFermiSurfaceCuts* (page 44) which triggers the creation of the files +cutswithweights. ... Note, that the flag *forcerecalculation=False* is used in the second call to *Slabify*.

`calculateFermiSurfaceCuts` (page 44). The resulting files are plotted via `xfbp wcuts.cmd` and produce *this Figure*. (page 96).

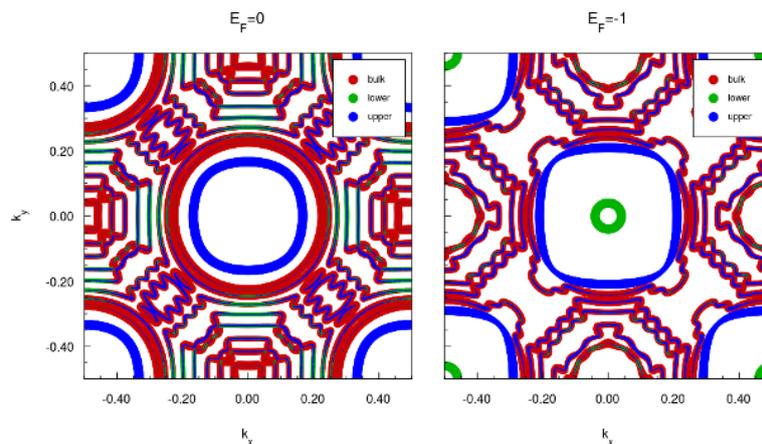


Fig. 3.11: Fermi surface cuts colored by bulk, lower and upper weights.

You can see that there are a lots of tiny wiggles in the bulk band region. This is so because we backfolded the planar unit cell. The only changes to the Hamiltonian are the cut-out atoms, which do not influence the bulk much. Together with the finite number of layers this is what you get. For infinite layers these bulk regions would fill up completely. What you also see however are the blue upper-surface surface-band for  $E = 0$  and additionally one green lower-surface surface-band for  $E = -1$ . Compare to *the band weights* (page 92).

### 3.1.9 Semi infinite slab

Finally we create spectral density plots for a semi infinite slab. In this case not much interesting is seen, since the surface states were a consequence of cutting out an atom, which currently is not yet implemented for 'semislabs' yet.

Please change into `semi/` and run the script (from `semi/README.rst`):

```
# Here we set up a semi infinite slab with a normal in-plane unit cell.
# run
./slabify.py | tee out
xfbp edc.cmd
xfbp fssemi.cmd

# have a look at slabifyres/=in_step... and slabifyres/=in_final_PLlayer.
# Try to understand how the structure settings work.
```

You should see something like *this* (page 97) and *something like this* (page 97). Compare this to *the bulk projected bands* (page 82).

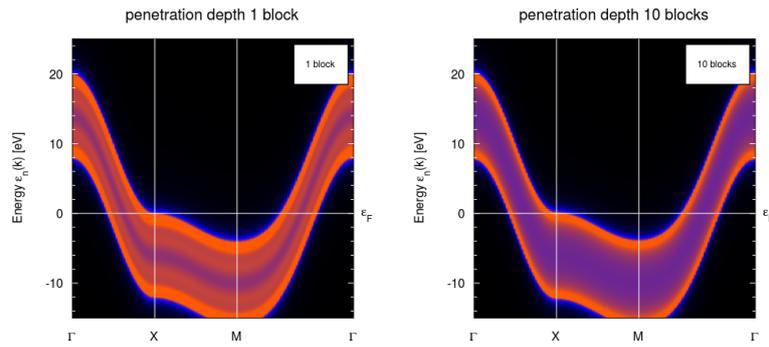


Fig. 3.12: Energy distribution curves for the semi-infinite slab.

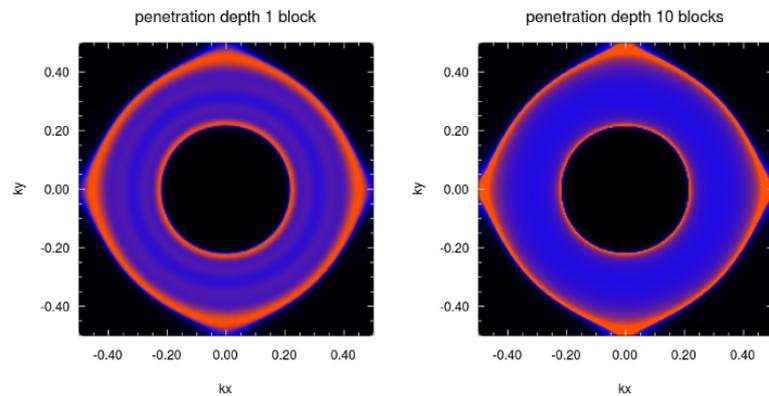


Fig. 3.13: Surface Fermi surface spectral function.

Let's explain the script `semi/slabify.py` now.

```

1  #!/usr/bin/env python
2
3  from __future__ import print_function
4  import sys
5
6  # If your pyfplo is not found you could also
7  # explicitly specify the pyfplo version path:
8  #sys.path.insert(0, "/home/magru/FPLO/FPLO22.00-62/PYTHON/doc");
9
10 import numpy as np
11 import pyfplo.slabify as sla
12
13
14 print( '\npyfplo version=: {0}\nfrom: {1}\n'.format(sla.version,sla.__file__)
15 # protect against wrong version
16 #if fedit.version!='22.00': raise RuntimeError('pyfplo version is incorrect.')
17
18
19
20 # =====
21 #
22 # =====
23
24 def work():
25
26
27     hamdata='../+hamdata'
28

```

(continues on next page)

```

29     s=sla.Slabify()
30
31
32     s.object='semislab'
33     s.anchor=-0.001
34     s.numberoflayers=4
35     s.printStructureSettings()
36
37     s.prepare(hamdata)
38
39     bp=sla.BandPlot()
40     bp.points=[
41         ['$~G', [0, 0, 0]],
42         ['X', [0.5, 0, 0]],
43         ['M', [0.5, 0.5, 0]],
44         ['$~G', [0, 0, 0]],
45     ]
46     bp.ndiv=60
47     bp.calculateBandPlotMesh(s.dirname)
48
49     ec=sla.EnergyContour(200, -15, 25)
50     print( ec)
51
52     bp.on()
53
54     # penetration one block (primary layer)
55     s.calculateEDC(bp, ec, penetrationdepth=-1, suffix='_1block')
56
57     # penetration 10 blocks (primary layer)
58     s.calculateEDC(bp, ec, penetrationdepth=-10, suffix='_10block')
59
60     fso=sla.FermiSurfaceOptions()
61     fso.setMesh(200, [-0.5, 0.5], 200, [-0.5, 0.5])
62     s.calculateFermiSurfaceSpectralDensity(fso, penetrationdepth=-1,
63                                         suffix='_1block')
64     s.calculateFermiSurfaceSpectralDensity(fso, penetrationdepth=-10,
65                                         suffix='_10block')
66
67
68
69
70
71
72 # =====
73 #
74 # =====
75
76
77 if __name__ == '__main__':
78
79     work()
80

```

In line 32 the structure type is set. We use `numberoflayers=4` in this case, although it is not really needed. See documentation of `numberoflayers` (page 55). In lines 39-47 a band structure path is setup for the energy distribution curve (EDC). Line 49 defines an energy contour (see `EnergyContour` (page 59)) with an automatic imaginary part. Line 55 and 58 actually execute the EDC calculation (`Slabify.calculateEDC` (page 45)) with two different `penetrationdepth`s. The resulting files are called `+akbl_sf.spin1` (or `spin2`), which we modified via the `suffix` argument. The name means:  $akbl=A_{Bl}(\mathbf{k})$  = Bloch-spectral-function and `sf` = slabify. `penetrationdepth` indicates to which depth the spectral density is collected. The larger the `penetrationdepth` the more bulk signal will be sampled.

Lines 60-65 setup and calculate the surface Fermi surface, which is basically the  $k_x, k_y$ -resolved spectral density for the 2d surface BZ (`Slabify.calculateFermiSurfaceSpectralDensity` (page 45)). The resulting files are called `+fs_sf.spin1` (or `spin2`), which we modified via the `suffix` argument. The name means `fs` = Fermi-Surface-Spectral-Function and `sf` = slabify.

### 3.1.10 Semi infinite slab, doubled planar cell

At very last, we double the planar unit cell for illustrative purpose.

Please change into `semix2/` and run the script (from `semix2/README.rst`):

```
# Here we set up a semi infinte slab with a normal in-plane unit cell.
# run
./slabify.py | tee out
xftp edc.cmd
xftp fssemi.cmd

# have a look at slabifyres/=in_step... and slabifyres/=in_final_PLlayer.
# Try to understand how the structure settings work.
```

You should see something like this (page 99) and something like this (page 99). Compare this to the left panel of the finite slab result (page 96) with removed atoms. It demonstrates the backfolding in the bulk projected bands region.

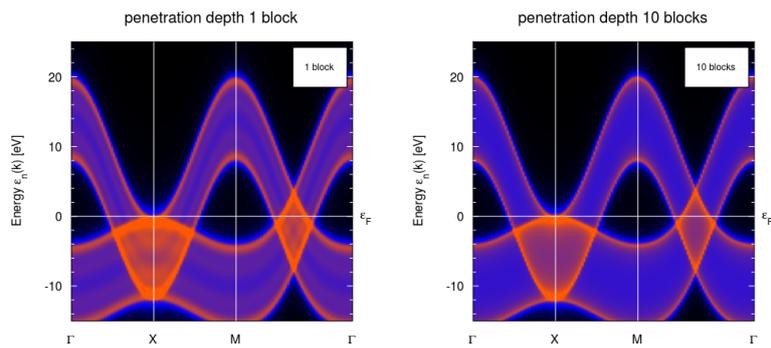


Fig. 3.14: The energy distribution curves.

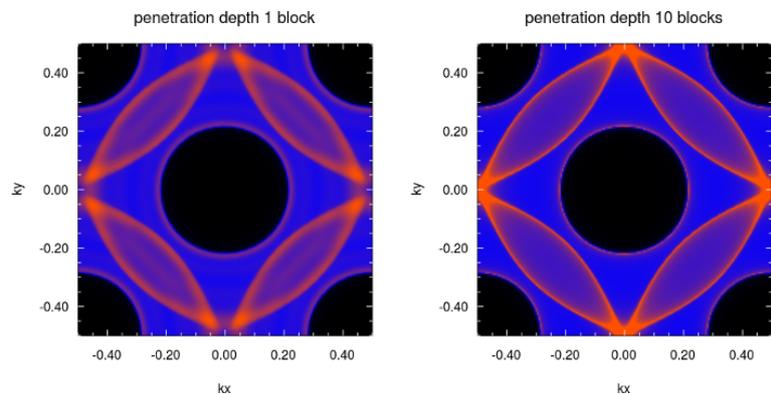


Fig. 3.15: Surface Fermi-surface spectral function.

Let's explain the script `semix2/slabify.py` now.

```

1  #!/usr/bin/env python
2
3  from __future__ import print_function
4  import sys
5
6  # If your pyfplo is not found you could also
7  # explicitly specify the pyfplo version path:
8  #sys.path.insert(0, "/home/magru/FPLO/FPLO22.00-62/PYTHON/doc");
9
10 import numpy as np
11 import pyfplo.slabify as sla
12
13
14 print( '\npyfplo version=: {0}\nfrom: {1}\n'.format(sla.version,sla.__file__) )
15 # protect against wrong version
16 #if fedit.version!='22.00': raise RuntimeError('pyfplo version is incorrect.')
17
18
19
20 # =====
21 #
22 # =====
23
24 def work():
25
26
27     hamdata='../+hamdata'
28
29     s=sla.Slabify()
30
31
32     s.object='semislab'
33     # make a larger 3d cell out of the simple cell
34     s.enlarge=[ [1,1,0], [-1,1,0], [0,0,1]]
35     s.anchor=-0.001
36     s.numberoflayers=4
37     s.printStructureSettings()
38
39     s.prepare(hamdata)
40
41     bp=sla.BandPlot()
42     bp.points=[
43         ['$~G', [0,0,0]],
44         ['X', [0.5,0,0]],
45         ['M', [0.5,0.5,0]],
46         ['$~G', [0,0,0]],
47     ]
48     bp.ndiv=60
49     bp.calculateBandPlotMesh(s.dirname)
50
51     ec=sla.EnergyContour(200,-15,25)
52     print( ec)
53
54     bp.on()
55
56     # penetration one block (primary layer)
57     s.calculateEDC(bp,ec,penetrationdepth=-1,suffix='_1block')
58
59     # penetration 10 blocks (primary layer)
60     s.calculateEDC(bp,ec,penetrationdepth=-10,suffix='_10block')
61
62     fso=sla.FermiSurfaceOptions()

```

(continues on next page)

(continued from previous page)

```

63     fso.setMesh(200, [-0.5, 0.5], 200, [-0.5, 0.5])
64     s.calculateFermiSurfaceSpectralDensity(fso, penetrationdepth=-1,
65                                           suffix='_1block')
66     s.calculateFermiSurfaceSpectralDensity(fso, penetrationdepth=-10,
67                                           suffix='_10block')
68
69
70
71
72
73
74     # =====
75     #
76     # =====
77
78
79     if __name__ == '__main__':
80
81         work()
82

```

The only thing we changed was line 34.

With this the introductory tutorial shall end.

## 3.2 2D topological insulator

- *The topological phase* (page 101)
- *The trivial insulator phase* (page 107)

---

**Note:** You need to use the newer **xfbp/xfplo** version, which comes with `pyfplo` in order for the `cmd` scripts to work properly.

---

This example tries to explain how to calculate the  $Z_2$  invariant of the 2d time reversal invariant topological insulator BHZ model (Bernevig, Hughes and Zhang). The tutorial files are in `FPLO.../DOC/pyfplo/Examples/slabify/BHZmodel` where `FPLO...` stands for your version's `FPLO` directory, e.g. `FPLO22.00-62`.

The method can be applied to any plane spanned by time reversal invariant points also in 3d lattices, e.g. in Weyl semi metals, where the  $Z_2$  invariant tells us how many pairs of surface states are to be expected to cross a given line – the projection of the plane onto the surface. (All assuming that the corresponding plane is gapped.)

We use a Hamiltonian file (`+hamdata`) containing the model data, which was created by `bhz.py`. Usually `+hamdata` is created by the Wannier function module of `fplo`. The model data are taken from [Yu2011] and modified a bit (`a=5`, `M_TI=-6`). The model defines a four orbital tight binding model. Depending on the parameter choice the model is either a trivial insulator or a topological insulator. The two cases are in subdirectories called `I` and `TI`.

### 3.2.1 The topological phase

In python script for the 2d band structure and the  $Z_2$  calculation is `TI/2d/Z2.py`

```

1  #! /usr/bin/env python
2

```

(continues on next page)

(continued from previous page)

```

3 from __future__ import print_function
4 import sys
5
6 # If your pyfplo is not found you could also
7 # explicitly specify the pyfplo version path:
8 #sys.path.insert(0, "/home/magru/FPLO/FPLO22.00-62/PYTHON/doc");
9
10 import numpy as np
11 import pyfplo.slabify as sla
12
13
14 print( '\npyfplo version=: {0}\nfrom: {1}\n'.format(sla.version,sla.__file__)
15 # protect against wrong version
16 #if fedit.version!='22.00': raise RuntimeError('pyfplo version is incorrect.')
17
18
19 # =====
20 #
21 # =====
22
23 def work():
24
25
26     hamdata='../+hamdata'
27
28
29     s=sla.Slabify()
30     s.object='3d'
31     s.printStructureSettings()
32     s.prepare(hamdata)
33
34
35     # prepare BandPlot
36     bp=sla.BandPlot()
37     bp.ndiv=100
38     bp.points=[
39         ['$~G', [0,0,0]],
40         ['X', [0.5,0,0]],
41         ['M', [0.5,0.5,0]],
42         ['$~G', [0,0,0]],
43         ['Y', [0,0.5,0]],
44     ]
45     bp.calculateBandPlotMesh(s.dirname)
46
47
48     s.calculateBandStructure(bp)
49
50
51     # wannier centers
52     s.calculateZ2Invariant([0,0,0],[0.5,0,0],[0,0.5,0],Nint=20,Nky=100,
53                           homos=[2,4])
54
55
56
57
58 # =====
59 #
60 # =====
61
62
63 if __name__ == '__main__':

```

(continues on next page)

(continued from previous page)

```
64
65     work()
66
```

Line 8 can be uncommented and edited to make python search for pyfplo in a particular location (also see [Setup](#) (page 1)). Line 14 shows which pyfplo version was loaded and from where. If you uncomment line 16, the script is showing an error message if the pyfplo version does not match a particular version.

Please run the script (from `TI/2d/README.rst`):

```
# run the script as in

./Z2.py | tee out
xftp bands.cmd

xftp Z2_homo2.cmd
xftp Z2_homo4.cmd

# and have a look at the output and into slabifyres/
```

In line 27 we set the proper `+hamdata` and in line 30 we set the object to `'3d'`, which in the case of a 2d lattice means 2d-bulk. In lines 35-48 the band structure is calculated (*2d bulk bands for the topological phase*, (page 103)). Note that there is a small gap close to the  $\Gamma$ -point where the orbital weight is inverted across the gap which already indicates the possibility of a topological phase.

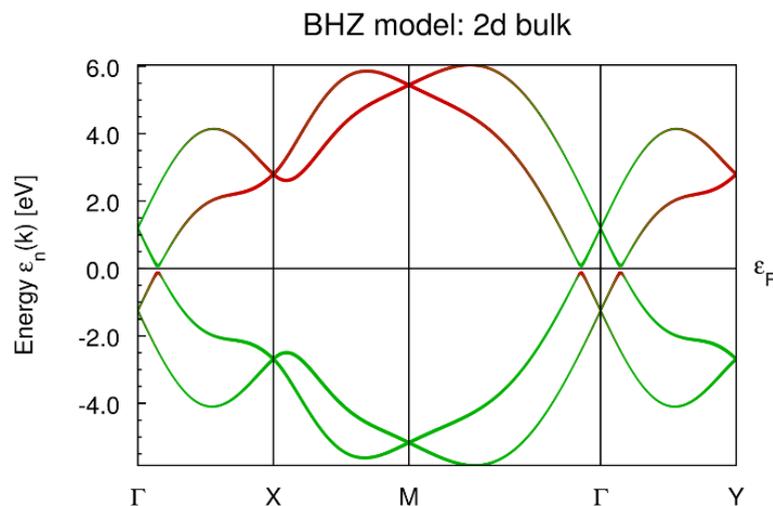


Fig. 3.16: 2d bulk bands for the topological phase.

In line 52 the Wannier centers are calculated. Consult [calculateZ2Invariant](#) (page 48) for more information. Note, that we have to choose three time reversal invariant points (TRIM) in the Brillouin zone in units of `kscale` (page 56). At the end of the output you will see the primitive reciprocal cell and the `kscale` factor (code lines 57-58). The TRIM points are always in the middle between two reciprocal cell vectors. This should explain our choice.

The output contains quite a lot of information. Please consider it all. For each homo which was specified a file containing the Wannier centers `slabifyres/+Z2_homo...` and a file containing a reference line `+zgap_homo...` are created. The reference line usually follows the largest gap between Wannier centers [Sol2011]. In our modification a number of such lines for some of the largest WC-gaps is created. Each yields a topological index. The majority result will win. The percentage of such lines with the majority result is called reliability in the output. The table of invariants for all homos in the output looks like

```
Invariants:
homo      Z2 (reliability)  smallest gap [eV]
  2        1 (100%)         0.20353
  4        0 (100%)         >10
```

If the reliability is 100% all WC-gap-following curves gave the same result. This does however not mean it is a save result. The exactness of this algorithm depends on the actual system. If the Berry curvature is strongly fluctuating a finer grid is needed. This often happens e.g. if the energy gaps are small or zero. In such cases the degeneracies at  $ky = 0$  and  $ky = \pi$  can be broken. Also discontinuities can occur and the results can depend on the evenness/oddness of  $Nint$ . All these are warning signs that the  $Z_2$  invariant might be nonsensical.

Additionally, the smallest energy gap for each homo on the grid defined by the parameters  $Nint$  and  $Nky$  is printed. This helps you decide, if the time reversal invariant plane is really gaped. Also this information is not fool proof, since the grid is discrete.

If  $efhomo$  is given to `calculateZ2Invariant` (page 48) a \* is printed after this homo number, usually ment to be the Fermi level. If the reliability is less then 99% a question mark is printed before the  $Z_2$  invariant, as in in the following example:

```
Invariants:
homo      Z2 (reliability)  smallest gap [eV]
  12      ? 1 (90%)         0.4
  14      *? 0 (67%)        0.13
  16      1 (100%)         0.56
```

Let's have a look at the corresponding Wannier center files (*Wannier centers for homo 2 in the topological phase.* (page 104) and *Wannier centers for homo 4 in the topological phase.* (page 104)) created by loading `Z2_homo . . . cmd` into **xfbp**.

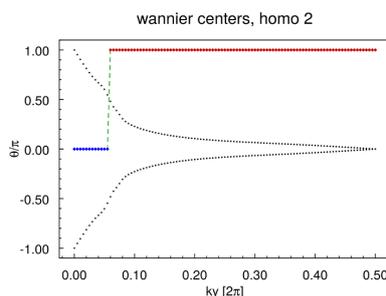


Fig. 3.17: Wannier centers for homo 2 in the topological phase.

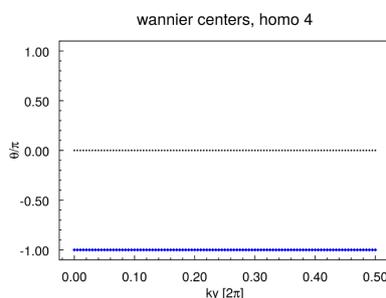


Fig. 3.18: Wannier centers for homo 4 in the topological phase.

The Soluyanov algorithm draws a curve through all wannier centers, such that it always stays in the largest gap between Wannier centers (which we modified to give more competing solutions). If the number of centers crossed so far at a certain  $ky$ -value is even a blue symbol is plotted if the number is odd a red symbol is plotted. If the last symbol ( $ky = \pi$ ) is red the  $Z_2$  invariant is odd. Of course the WC-curves are discrete and hence we cannot a priori decide about their connectedness, which leaves an uncertainty. Ultimately, the user has to look at these

pictures and decide for himself, how many WCs a chosen reference line crosses. In **xfbp** a right mouse click close to a curve tells you how many sets there are, which helps to verify the number of crossed Wannier centers. Any reference curve (also a straight horizontal line) is OK. The one shown in the plots is just the automatically determined one. Since we modified the algorithm, the automatic curve does not necessarily follow the largest WC-gaps. The two figures tell us that homo 2 has a non trivial  $Z_2$  invariant in the plane spanned by the TRIM points given to `calculateZ2Invariant` (page 48).

Now, we make a calculation of the 1d surface states. Go into `TI/semi` and run the script (from `TI/semi/README.rst`):

```
# run the script as in

./calcedc.py
xfbp edc.cmd

# and have a look at the output and into slabifyres/
```

The script looks like this:

```
1  #!/usr/bin/env python
2
3  from __future__ import print_function
4  import sys
5
6  # If your pyfplo is not found you could also
7  # explicitly specify the pyfplo version path:
8  #sys.path.insert(0, "/home/magru/FPLO/FPLO22.00-62/PYTHON/doc");
9
10 import numpy as np
11 import pyfplo.slabify as sla
12
13
14 print( '\npyfplo version: {0}\nfrom: {1}\n'.format(sla.version,sla.__file__))
15 # protect against wrong version
16 #if fedit.version!='22.00': raise RuntimeError('pyfplo version is incorrect.')
17
18
19 # =====
20 #
21 # =====
22
23 def work():
24
25
26
27     hamdata='../+hamdata'
28
29     s=sla.Slabify()
30     s.object='semislab'
31     s.zaxis=[0,1,0]
32     s.numberoflayers=2
33     s.printStructureSettings()
34     s.prepare(hamdata)
35
36
37     bp=sla.BandPlot()
38     bp.ndiv=100
39     bp.points=[
40         ['X', [0.5,0,0]],
41         ['$~G', [0,0,0]],
42         ['X', [0.5,0,0]],
```

(continues on next page)

(continued from previous page)

```

43 ]
44 bp.calculateBandPlotMesh(s.dirname)
45 ec=sla.EnergyContour(400,-6,6)
46 s.calculateEDC(bp,ec,penetrationdepth=-1,suffix='_1block')
47
48
49 # another zoom
50 s.dirname='zoom'
51 bp.points=[
52     ['$arrowleft', [0.15,0,0]],
53     ['$~G', [0,0,0]],
54     ['$arrowright X', [0.15,0,0]],
55 ]
56 bp.calculateBandPlotMesh(s.dirname)
57
58 ec=sla.EnergyContour(200,-0.5,0.5)
59 s.calculateEDC(bp,ec,penetrationdepth=-1,suffix='_1block')
60
61 # =====
62 #
63 # =====
64
65
66 if __name__ == '__main__':
67
68     work()
69

```

Lines 30-32 setup a `semislabs` with [010] surface and two layers in the primary layer. Since the bulk is 2d with a [001] surface this results in a semi infinite ribbon. (Actually, in the moment the 2d bulk is a 3d repeated monolayer slab.) After setup, the former [010] direction is the new c-direction (surface of the semi infinite ribbon) but still the cartesian y-axis (remember we use rotations internally to keep the original orientation). The global x-axis is now the (periodic) b-direction, while the global z-axis (as for the 2d bulk) is the direction perpendicular to the ribbon. If we chose the high symmetry points along the cartesian x-axis (lines 40-42, 52-54) we sample k-points in the periodic b-direction.

In lines 37-46 we set up a bandplot path and an energy contour and calculate the energy resolved Bloch spectral density. In line 50-59 a zoomed in version is calculated and placed into a different directory (line 50), because the `points` file is different for both calculations.

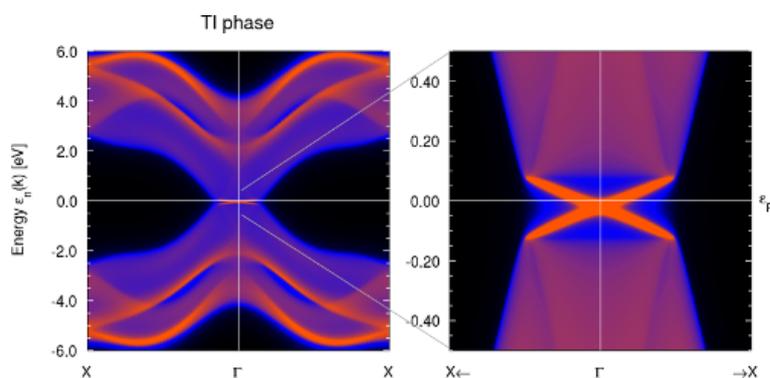


Fig. 3.19: Surface states in the topological phase.

The result looks like *this Figure* (page 106) and clearly shows a Dirac cone at the edge of the surface.

### 3.2.2 The trivial insulator phase

In python script for the 2d band structure and the  $Z_2$  calculation is `I/2d/Z2.py` and is identical to the one for the topological phase.

Please run the script (from `I/2d/README.rst`):

```
# run the script as in

./Z2.py | tee out
xftp bands.cmd

xftp Z2_homo2.cmd
xftp Z2_homo4.cmd

# and have a look at the output and into slabifyres/
```

Have a look at the *2d bulk bands for the trivial phase*. (page 107)

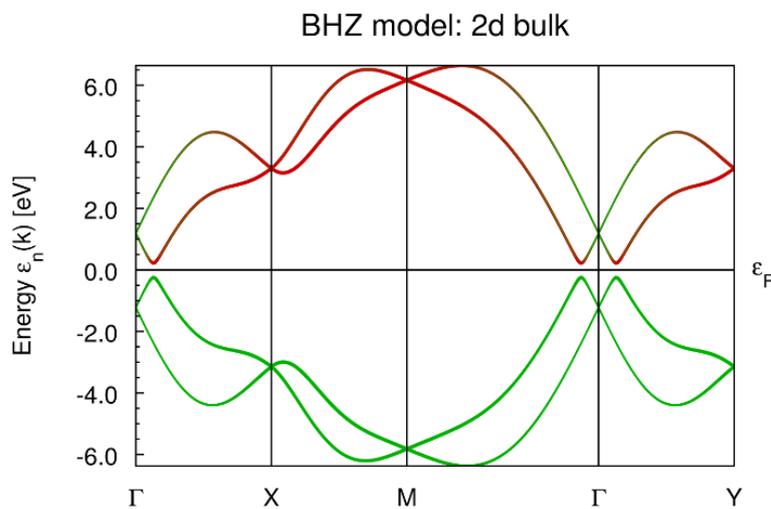


Fig. 3.20: 2d bulk bands for the trivial phase.

It looks pretty much like the *2d bulk bands for the topological phase*. (page 103) except that the band inversion across the gap is missing. Consequently, *Wannier centers for homo 2 in the trivial phase*. (page 107) (created by `xftp Z2_homo2.cmd`) clearly indicates a trivial phase as is also seen by the output:

```
Invariants:
homo    Z2 (reliability)  smallest gap [eV]
  2      0 (100%)        0.46419
  4      0 (100%)        >10
```

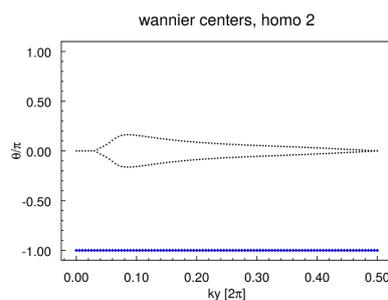


Fig. 3.21: Wannier centers for homo 2 in the trivial phase.

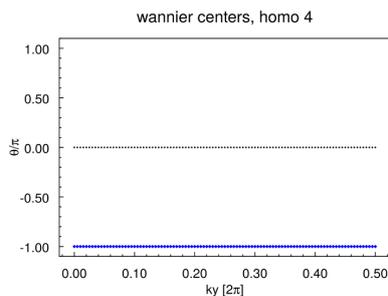


Fig. 3.22: Wannier centers for homo 4 in the trivial phase.

As a final confirmation let's have a look at the *Surface states in the trivial phase*. (page 108).

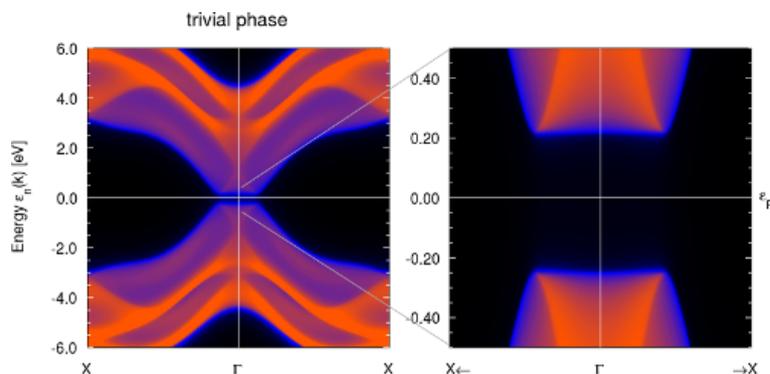


Fig. 3.23: Surface states in the trivial phase.

Run the script (from TI/semi/README.rst):

```
# run the script as in

./calcedc.py
xftp edc.cmd

# and have a look at the output and into slabifyres/
```

The python script is the same as for the topological phase.

### 3.3 3D topological insulator

---

**Note:** You need to use the newer **xftp/xfplo** version, which comes with **pyfplo** in order for the `cmd` scripts to work properly.

---

Here we discuss the calculation of the topological indices for 3d topological insulators. The algorithm is based on Wannier centers as in *2D topological insulator* (page 101). The only difference is that we need to calculate four  $Z_2$  invariants for 4 planes in the 3d BZ. In principle there are 6 different planes forming the faces of a parallelepiped whose corners are the eight distinct time reversal invariant momenta (TRIM) of any chosen smallest primitive reciprocal cell. However, if the electronic spectrum is gapped above a certain band there are only 4 independent  $Z_2$  invariants, which are usually grouped into four topological indices  $\nu_0; (\nu_1\nu_2\nu_3)$ .  $\nu_0 = 1$  indicates a strong topological insulator while any  $\nu_{1,2,3} = 1$  indicate conditions on the number of surface state pairs for certain surface orientations. If  $\nu_0 = 0$  but some  $\nu_{1,2,3} \neq 0$  it is a weak topological insulator. If all indices are zero it is a trivial insulator. The mapping of  $Z_2$  invariants to indices is as follows: if any two parallel planes of the

parallelepiped have differing invariants  $\nu_0 = 1$ . We take the two planes spanned by the first two reciprocal lattice vectors ( $g_{1,2}$ ) called  $z_0$  if the plane goes through the origin and  $z_1$  if it goes through  $g_3$ . The three weak indices are identical to the  $Z_2$  invariants of the three planes, which do not go through the origin, which we call  $x_1$  (spanned by  $g_{2,3}$  through  $g_1$ ),  $y_1$  (spanned by  $g_{3,1}$  through  $g_2$ ) and  $z_1$  (spanned by  $g_{1,2}$  through  $g_3$ ). The weak indices of course depend on the chosen TRIM points.

The algorithm selects the TRIM points and performs four  $Z_2$  calculations. It tries to determine the invariants via the automatic procedure explained in *2D topological insulator* (page 101). The results including reliability and electronic gap estimates are printed to the output and convenience files `Z2_3dTI_homo...cmd` for use in `xfbp` are created.

This algorithm works for centro symmetric and non-centro symmetric compounds. A version of this algorithm is linked directly into `fplo`. The difference is that when used from `fplo` a all-orbital Wannier basis is created instead of the reduces Wannier models, which usually are the basis of `pyfplo.slabify` (page 41). This leads to differently locking Wannier center curves, since there are more orbitals (semi core and such). If surface state calculations are planned it is anyway necessary to create a Wannier model first. The resulting  $Z_2$ -calculations are faster than in `fplo`.

For illustration we chose a system, which is centro symmetric, since in this case we know the invariants from the parity algorithm, which allows comparison. This example also shows some fo the complications. We will use both the `fplo` version and the `pyfplo` version of the Wannier center algorithm to understand the peculiarities.

The tutorial files are in `FPLO.../DOC/pyfplo/Examples/slabify/3dTI/Bi2Se3`.

To start with change into this directory and have a look into the topological insulator submenu in `fedit`. You will see that “Force wannier center” is switched on, to force the use of this algorithm despite the fact that this compound has an inversion center.

Now, have a look at `makewandef.py`. It illustrates how to create a `=.wandef` quickly. There is a particularity in the system, which is that the `Bi6p` and `Se4p` orbitals are separated from lower and higher bands through two gaps. This makes creating a Wannier model especially easy. We use the option `lbands` and `ubands` to define an energy window which excludes all other bands. Please execute

```
python makewandef.py
```

You should have a `=.wandef` file now. Next run `fplo`. E.g.:

```
fplo.... > out
```

to produce a bandstructure, `+wancoeff` and the output from the `fplo` Wannier center calculation. This will take some time, since the `fplo` version of the WC algorithm is not the fastest, due to the larger number of bands needed. Output related to the TI calculation can be found via:

```
grep TI: out
```

First, the invariants from parities are printed and then the Wannier center algorithm is executed. Various files are created. **Note, that the indices from parities can differ from the Wannier center results; first, because the WC algorithm is tricky or second, if the electronic gap for a certain homo is zero, in which case both algorithms makes no sense.**

You will see the following output from the Wannier centers:

```
TI: Invariants:
TI:  homo  invariants  E(k=0)                estim.gaps [eV] for
TI:                z0          z1          x1          y1
TI:    114  ? 0; (000)  -1.09567    0.05585    0.01917    0.01917    0.01917
TI:    116  ? 1; (111)  -0.79835    0.01306    0.03873    0.03873    0.03873
TI:    118  * 1; (000)  -0.22789    0.44699    0.43513    0.43513    0.43513
TI:    120  ? 1; (111)   0.28647    0.15850    0.14845    0.14845    0.14845
TI:    122  ? 0; (111)   1.42635    0.02117    0.17378    0.17378    0.17378
TI:    124  ? 0; (000)   1.44752    0.21087    0.29692    0.29692    0.29692
TI:
TI:  homo  invariants  E(k=0)                Z2 (reliability)
```

(continues on next page)

(continued from previous page)

TI:				z0	z1	x1	y1
TI:	114	? 0; (000)	-1.09567	0 ( 77%)	0 ( 77%)	0 ( 77%)	0 ( 77%)
TI:	116	? 1; (111)	-0.79835	0 ( 50%)	1 ( 73%)	1 ( 73%)	1 ( 73%)
TI:	118	* 1; (000)	-0.22789	1 (100%)	0 (100%)	0 (100%)	0 (100%)
TI:	120	? 1; (111)	0.28647	0 ( 92%)	1 ( 65%)	1 ( 65%)	1 ( 65%)
TI:	122	? 0; (111)	1.42635	1 ( 69%)	1 (100%)	1 (100%)	1 (100%)
TI:	124	? 0; (000)	1.44752	0 ( 80%)	0 (100%)	0 (100%)	0 (100%)

The first table shows homos, indices the energy at the Gamma point and estimates of the electronic gap (taken from the values on the finite grid used for the WC algorithm), while the second shows the  $Z_2$  invariants and their estimated reliability for the four planes of the parallelepiped mentioned above. If the reliability is less than 100% a question mark is printed after the homo. The homo, which is likely the highest occupied band is marked by a \* for orientation. This is of course only reliable if the system has a true gap.

In our case we see that there are rather small estimated gaps for lots of bands. Fortunately above the Fermi energy (homo 118) the gap is sizable. Not, surprisingly the reliabilities are all very high (actually 100%). We also find lots of question marks. Let's have a look at the *fplo and Wannier model bands (red/green)* (page 110).

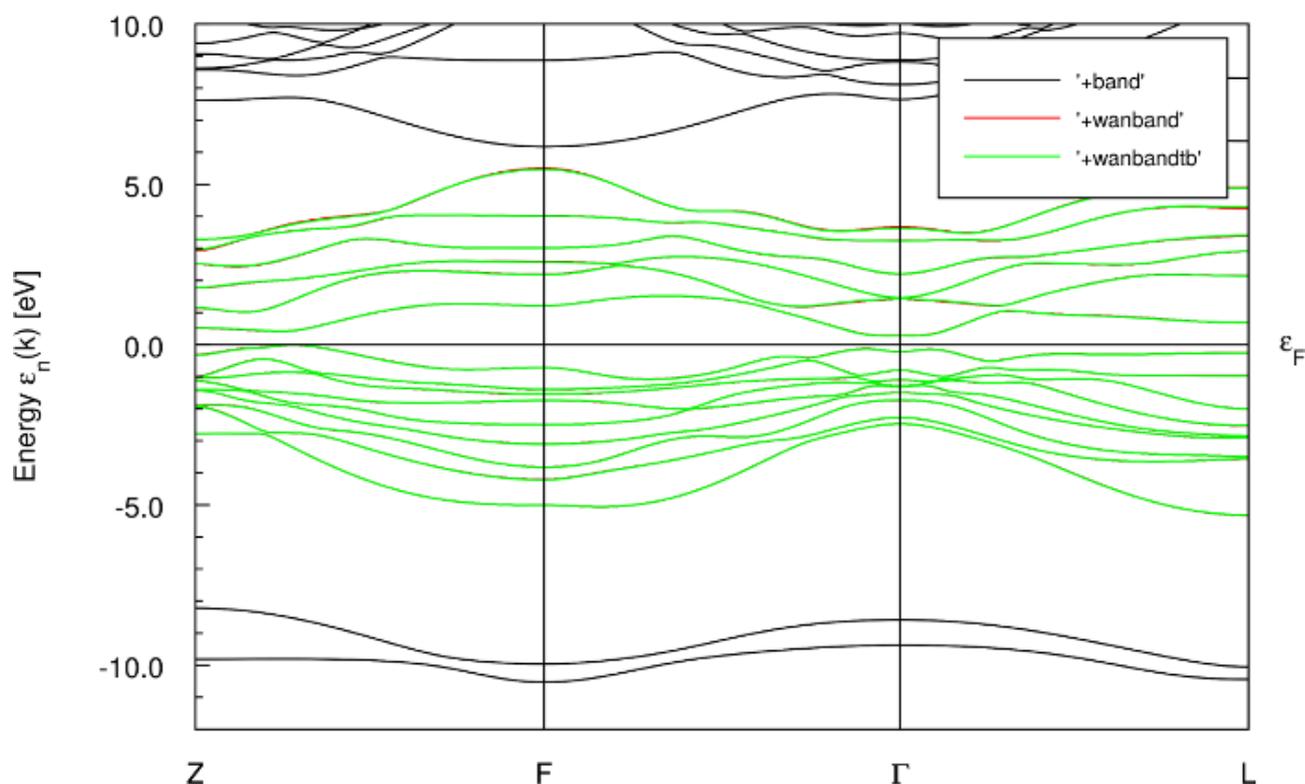


Fig. 3.24: fplo and Wannier model bands (red/green)

We see the two gaps, separating the p-bands as discussed above and that nearly all bands are separated by gaps (although small). This means that topological invariants have a meaning. Note, that it is not needed that the gap is visible in the DOS. All one needs is that the bands of homo and homo+1 never cross (warped gap).

Since this table is not ment for straight forward consumption we have to inspect the Wannier centers by hand. Run:

```
xfbp Z2_3dTI_homo118.cmd
```

and so forth. You will get several pictures. Let's start with *Wannier centers and reference line for homo 118* (page 111). As already pointed out in the table his band has a nice electronic gap above itself, happens to be the last occupied band below the Fermi level and has high reliability of the automatically determined  $Z_2$  invariants (reference line winding number).

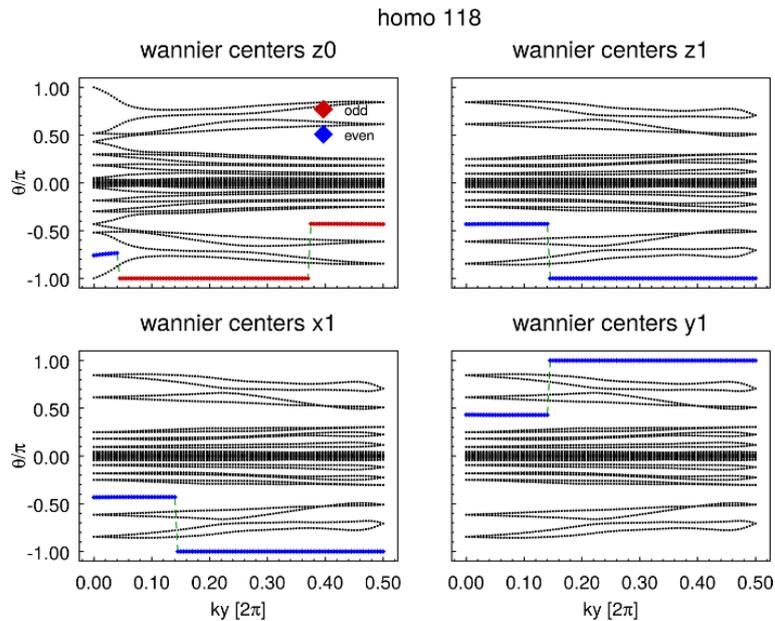


Fig. 3.25: Wannier centers and reference line for homo 118

It can be clearly seen that the  $x_1$ ,  $y_1$  and  $z_1$  planes have zero  $Z_2$  invariant, while plane  $z_0$  is non-trivial. To really verify this use the right mouse click in **xfbp** close to the highest Wannier center curve in the  $z_0$  panel close to  $\theta = 1$  and  $k_y = 0$ . It will show that only one center (called Set118 in **xfbp**) comes down from its degenerate value of  $+1$ . A clean straight reference line can be drawn for  $\theta = 0.9$  (not shown), which only crosses this center and hence crosses an odd number of Wannier centers, which results in  $Z_2 = 1$ . This and the fact that we can visually connect the Wannier center curves in a reasonable smooth way convinces us that the topological indices are  $1; (000)$ . (Remember  $z_0 \neq z_1 \rightarrow \nu_0 = 1$  and  $\nu_1 = Z_2(x_1, \dots)$ ) In fact due to symmetry the tree planes  $x_1$ ,  $y_1$  and  $z_1$  give the same result in this compound.

Next we have a look at all the other homos, just for learning purposes.

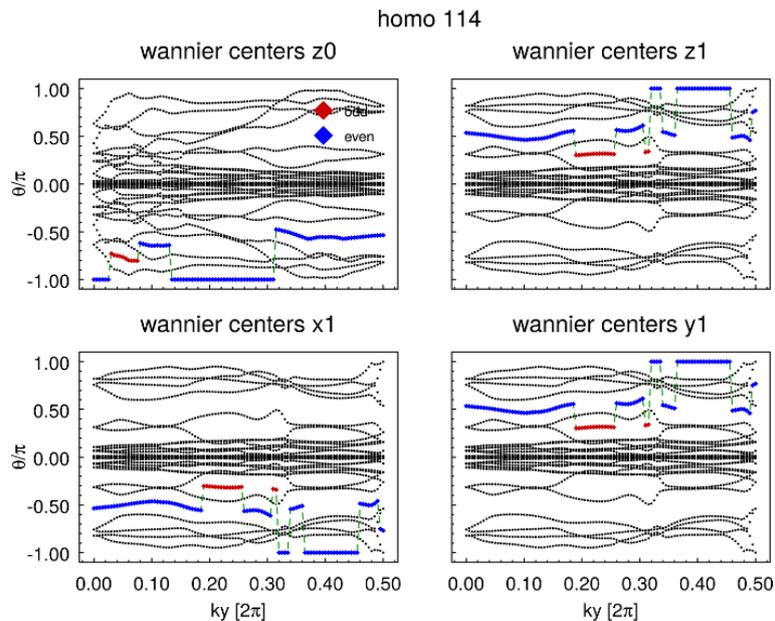


Fig. 3.26: Wannier centers and reference line for homo 114

The *Wannier centers and reference line for homo 114* (page 111) are quite messy. However, you should convince yourself that  $z_1$  has a region ( $\theta \approx 0.5$ ) without any centers crossing  $\rightarrow$  trivial, while for the  $z_0$ -plane an even

number of curves cross any reference line. Hence we get 0;(000).

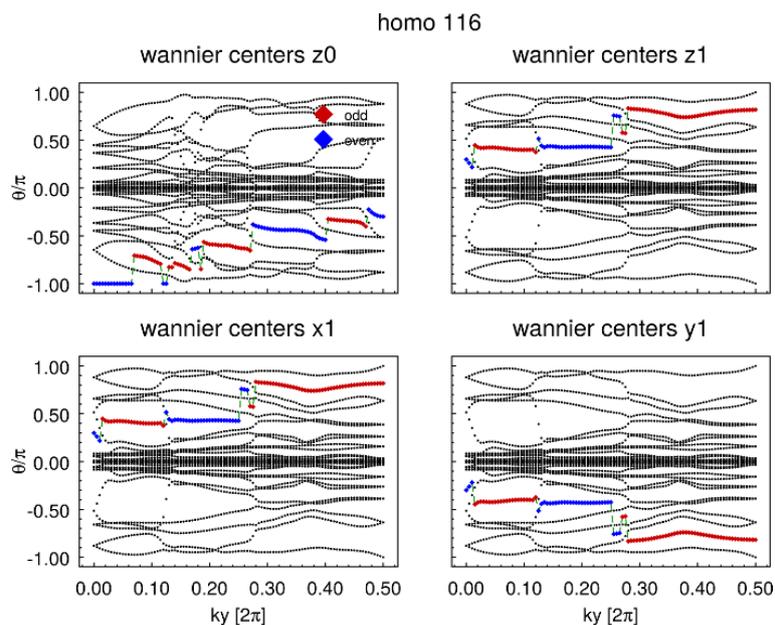


Fig. 3.27: Wannier centers and reference line for homo 116

The *Wannier centers and reference line for homo 116* (page 112) are also messy and reveal another possible complication. First, note that both 114 and 116 yield curves with fast varying (steep slopes) curves. These steep curves generate only very few points on their steep section. Depending on the density of the  $k_y$  mesh there might even be zero points on such sections, which would make it look as if certain curves end for some  $k_y$  value to reemerge for a later  $k_y$ -value. Any algorithm including our own judgment can fail in such cases. All we can do is to take a denser  $k_y$ -mesh. In this case the automatic reference line algorithm is correct as can be visually verified. There is an even/odd number of curves crossed by a suitably chosen reference curve for the  $z_0/z_1$  planes ( $x_1, y_1 = z_1$ ). Hence we get 1;(111). Note, however, that the electronic gaps above homos 114 and 116 are rather small, they could well be zero. If this homo were important we would try to use much finer meshes to make absolutely sure that the gap is finite.

Homo 122 also has a small electronic gap. If we assume that it is finite we can proceed analysing the invariants. The *Wannier centers and reference line for homo 120* (page 113) and the *Wannier centers and reference line for homo 122* (page 113) both show that the indices printed in the output table are wrong (as can be verified in this case due to the parity algorithm).

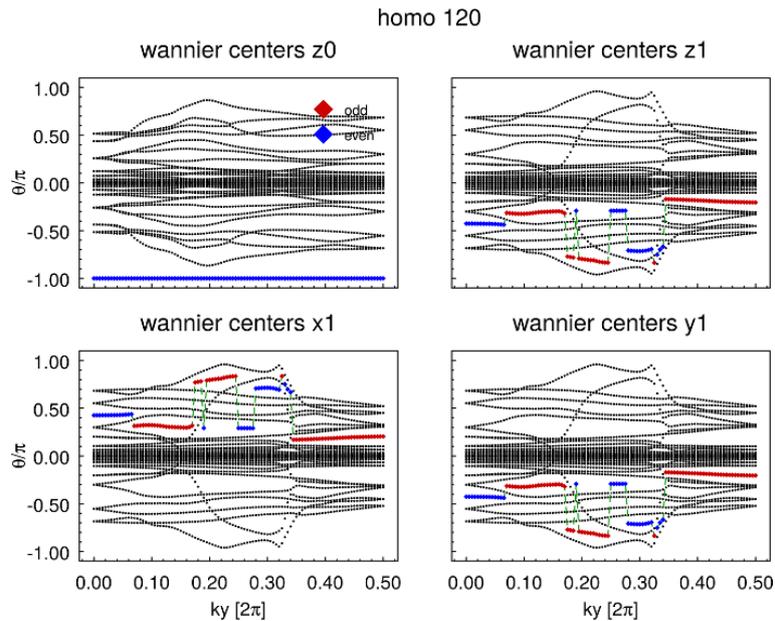


Fig. 3.28: Wannier centers and reference line for homo 120

For homo 120 the  $z_0$  plane is clearly trivial since we can draw a non crossing reference line at  $\theta = \pm 1$  (the automatic blue reference line is just that). For the other three planes the algorithm made a mistake around  $ky=0.33$ , where a single red dot appears. This mistake is due to a steep section at one curve. One can find hand drawn reference curves, which cross an even number of times and hence these planes are trival. The result should be 0;(000). Please also note that the reliability for homo 120 is high for the  $z_0$  plane but only 65% for the other three, which fits to the findings explained above. This is of course only an indicator.

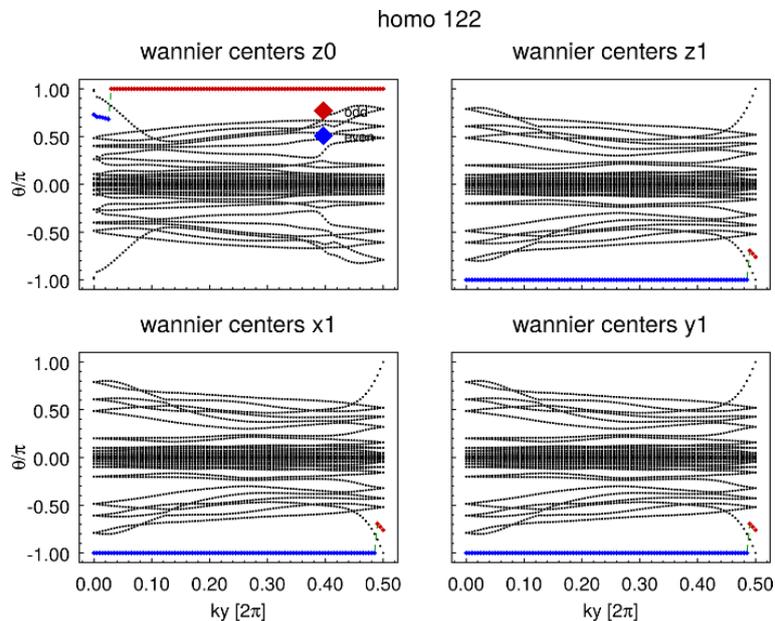


Fig. 3.29: Wannier centers and reference line for homo 122

For homo 122 the situation is even trickier. According to the reliabilities the  $z_0$  panel might have a problem. Indeed, the  $z_1$  plane (and  $x_1, y_1$ ) is clearly topological, but the  $z_0$  plane was determined as non trivial although it is not. Again due to small electronic gaps in the  $z_0$  plane we get very steep curves close to  $ky=0$ . In fact this curve is so steep that we see only one point in the steep section. In **xfbp** use the right mouse click at the dot in the upper left corner. It will show that there are two curves. Given the rest of the dots it is clear that one dot is connected with the declining curve and the other must fall very steeply to connect to a curve close to  $\theta = 0.28$ .

Hence the reference line actually crosses two curves and not one. Consequently,  $z_0 \neq z_1 \rightarrow 1;(111)$  in contrast with the result printed in the output table.

Homo 124 is again a simple case and will not be discussed here.

In summary we saw that the **fplo** based version of the procedure is rather slow, which is bad if a high grid density is needed to obtain reliable results. Let's continue with the `pyfplo` version then. We already set up the Wannier basis production in our first **fplo** run. Now, we can proceed by re-running:

```
fplo...
```

Note that we do not use an output file (we don't want to override the TI results). This will produce among others the file `+hamdata`, which we will use for the `pyfplo` based version of the TI procedure. If the `fplo` run is over we confirm the Wannier basis by checking:

```
xfbp wband.cmd
```

which should show green bands (finite cut-off tight-binding Wannier bands) on top of red bands (exact Wannier bands) on top of black bands (**fplo** bands). Please, change into `slabify/Z2` and execute:

```
python -u Z2.py | tee out
```

The flag `-u` un-buffers the output, such that the progress is updated in real time. The file's content is

```

1  #! /usr/bin/env python
2  from __future__ import print_function
3  import sys, copy
4  import numpy as np
5  import pyfplo.slabify as sla
6  import pyfplo.common as com
7  import pyfplo.fploio as fploio
8
9
10
11
12
13
14
15
16  # =====
17  #
18  # =====
19  def work():
20
21     print( '\nversion: ', sla.version, '\n')
22
23     np.set_printoptions(precision=5, suppress=True, linewidth=120)
24
25     hamdata='../..+hamdata'
26
27
28     s=sla.Slabify()
29     s.object='3d'
30     s.printStructureSettings()
31     s.prepare(hamdata)
32
33
34     p=fploio.INParser()
35     p.parseFile('../..=.in')
36     d=p( 'special_sympoints' )
37     l=[]
38     for i in range(d.size()):
39         l.append([ d[i]('label').S, d[i]('kpoint').listD])

```

(continues on next page)

(continued from previous page)

```

40
41 bp=com.BandPlot ()
42 bp.points=1
43 bp.ndiv=100
44 bp.calculateBandPlotMesh(s.dirname)
45
46 #help(sla)
47 s.calculateBandStructure(bp)
48
49 # 14 -> 0;(000)
50 # 16 -> 1;(111)
51 # 18 -> 1;(000)
52 # 20 -> 0;(000)
53 # 22 -> 1;(111)
54 # 24 -> 0;(000)
55
56 s.calculate3dTIInvariants(200,200,homos=range(14,25,2),efhomo=18)
57
58
59
60
61
62 return
63
64 # =====
65 #
66 # =====
67
68
69 if __name__ == '__main__':
70
71     work()
72
73

```

In lines 34-47 we calculate the band structure of the Wannier model along the same high symmetry points as were used in the fplo calculation. With the help of:

```
xfbp slabifyres/+band_sf
```

and a right mouse click we find out that the Fermi level is above homo 18 (just a coincidence that 118-18=100). This is used in line 56 to define a set of *homos* and *efhomo*. The grid is chosen finer for both the “integration” and the ky direction. See [calculate3dTIInvariants](#) (page 49).

The output table:

```
Invariants:
```

homo	invariants	E(k=0)	estim.gaps [eV] for			
			z0	z1	x1	y1
14	? 0;(000)	-1.09645	0.02338	0.01563	0.01563	0.01563
16	? 1;(111)	-0.79820	0.00531	0.02098	0.02098	0.02098 <-small
18	* 1;(000)	-0.22063	0.42925	0.43611	0.43611	0.43611
20	0;(000)	0.27854	0.14357	0.06937	0.06937	0.06937
22	1;(111)	1.42038	0.00911	0.17176	0.17176	0.17176 <-small
24	0;(000)	1.42948	0.20549	0.24304	0.24304	0.24304

homo	invariants	E(k=0)	Z2 (reliability)			
			z0	z1	x1	y1
14	? 0;(000)	-1.09645	0 (100%)	0 ( 93%)	0 ( 93%)	0 ( 93%)
16	? 1;(111)	-0.79820	0 (100%)	1 ( 86%)	1 ( 93%)	1 ( 93%)
18	* 1;(000)	-0.22063	1 (100%)	0 (100%)	0 (100%)	0 (100%)

(continues on next page)

(continued from previous page)

20	0; (000)	0.27854	0 (100%)	0 (100%)	0 (100%)	0 (100%)
22	1; (111)	1.42038	0 (100%)	1 (100%)	1 (100%)	1 (100%)
24	0; (000)	1.42948	0 (100%)	0 (100%)	0 (100%)	0 (100%)

looks a bit different from the previous one. First of all the gap values are different since there are always tiny shifts in a reduced Wannier model compared to a full band structure and second we used a finer grid. Furthermore, the reliabilities increased considerably, again mostly due to the finer grid. Finally, all topological indices are correct.

We can now load all the `Z2_3dTI_homo...cmd` files into `xfbp` and analyse the validity by hand. We will look at three Wannier center graphs. The *Wannier centers and reference line for homo 16* (page 116) show a quick change in the  $z_0$ -panel around  $ky=0.16$ , which is most likely due to the small electronic gap. Furthermore, the steep sections in the  $z_1$ -panel are now represented by more points, which makes the automatic reference line algorithm more reliable.

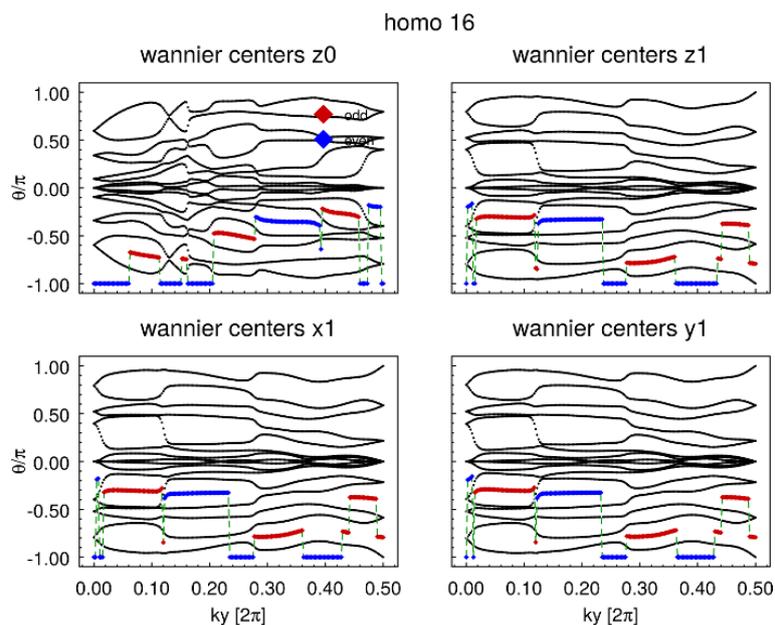


Fig. 3.30: Wannier centers and reference line for homo 16

The *Wannier centers and reference line for homo 20* (page 117) has the correct reference line due to the increased mesh size.

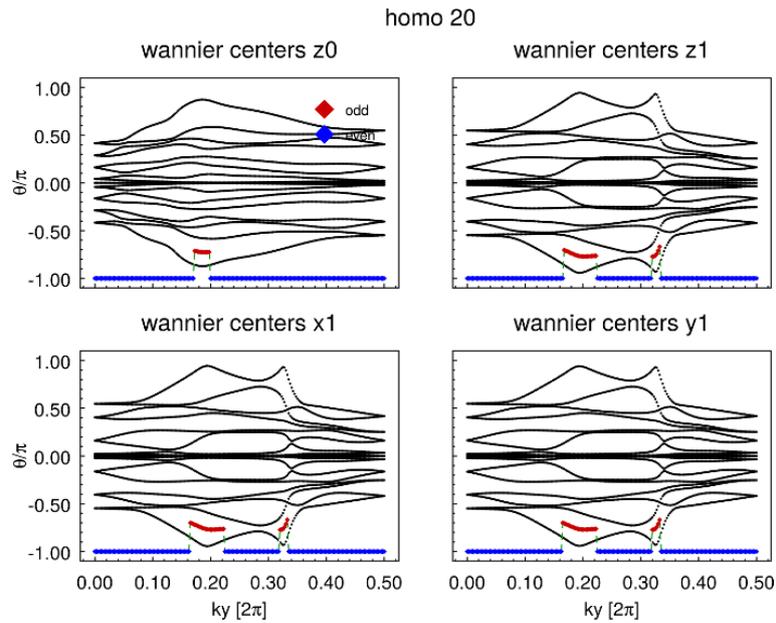


Fig. 3.31: Wannier centers and reference line for homo 20

The *Wannier centers and reference line for homo 22* (page 117) now clearly shows that two WC curves meet close to the upper left corner of the  $z_0$ -panel, which makes this plane trivial. Consequently, the invariants are correctly determined as 1;(111).

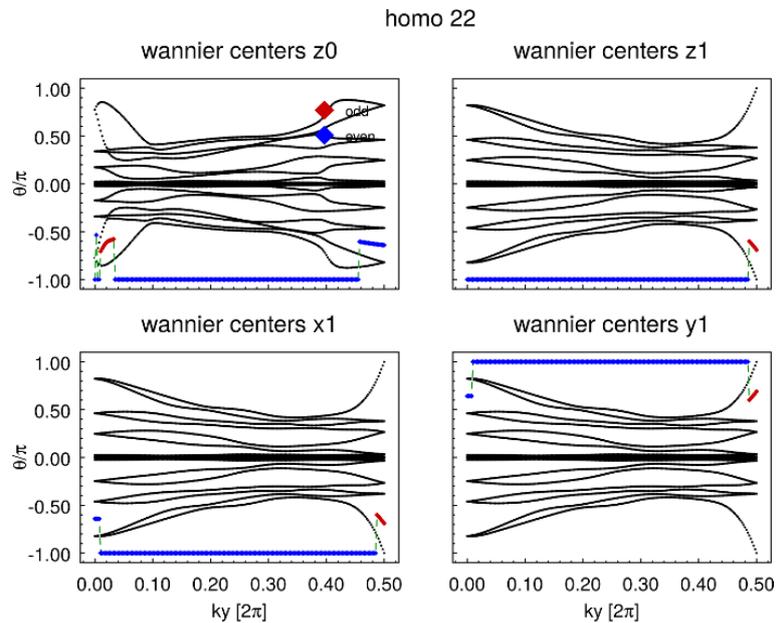


Fig. 3.32: Wannier centers and reference line for homo 22

In summary, the use of a reduce Wannier function model leads to a much faster TI procedure, which allows to use finer grids, which in turn increase the accuracy of the results. Additionally, one could calculate surface spectra using the same model.

### 3.4 Weyl semi metals

**Note:** You need to use the newer **xfbp/xfplo** version, which comes with `pyfplo` in order for the `xpy` scripts to work properly.

In this example we discuss methods for Weyl semi metals. We demonstrate how to find Weyl points, how to prove that they are indeed Weyl points and how to calculate surface spectra to analyse potential topological surface states (Fermi arcs).

We start with  $\text{MoTe}_2$ . This example is based on [Wang15]. The tutorial files are in `FPLO.../DOC/pyfplo/Examples/slabify/Weylpoints/MoTe2`. They contain a converged full relativistic calculation.

Figure *The bulk unit cell and band structure of  $\text{MoTe}_2$* . (page 118) shows the bulk unit cell and band structure. There is clearly an isolated band complex between -6 and +5 eV, which makes the creation of a Wannier function model especially easy.

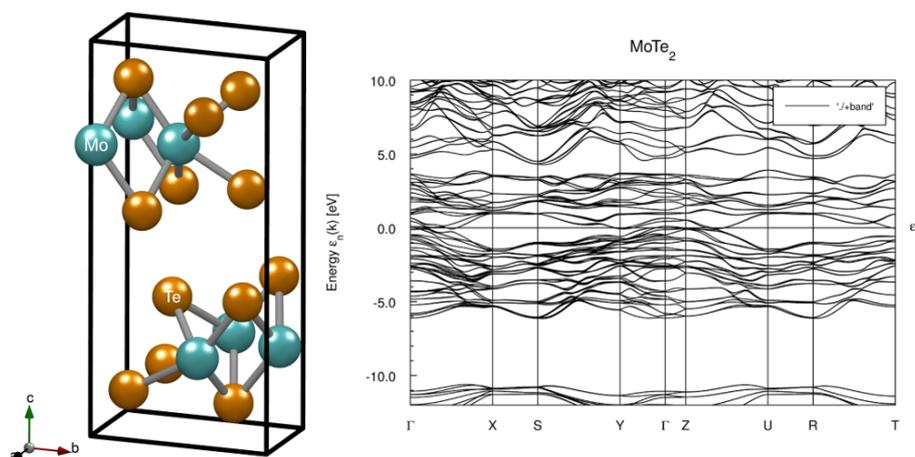


Fig. 3.33: The bulk unit cell and band structure of  $\text{MoTe}_2$ .

With a right mouse click close to the highest and lowest band in **xfbp** (when `+band` is loaded) we determined that this band complex has 88 bands. From the available valence orbitals we guess that these must be the 4d-bands of the 4 Mo atoms and the 5p-bands of the 8 Te atoms, which makes a total of  $4 \cdot 10 + 8 \cdot 6 = 88$  orbitals/bands. We could also have used band weights to determine this. Fortunately, none of these 88 bands has a section somewhere in the BZ where the sum of the weights of the considered orbitals is exactly zero. In other words there is no band section which needs other orbitals to get a non-zero WF projector. This allows to build a very simple WF model. The example directory contains the file `makewandef.py` which has the following content

```

1  #!/usr/bin/env python
2  from __future__ import print_function
3  import pyfplo.wanniertools as wt
4
5
6
7  # =====
8  def main():
9
10
11
12
13     wdc=wt.WanDefCreator(rcutoff=25,wftol=0.001,coeffformat='bin',
14                          wfgriddirections=[[1,0,0],[0,1,0],[0,0,1]],
15                          wfgridsubdiv=[1,1,1],savespininfo=False,
16                          keeprunning=True,opendxinterface=False)
17
18     emin=-6
19     emax= 4

```

(continues on next page)

(continued from previous page)

```

20 delower=1
21 deupper=0
22
23 wdc.add(wt.MultipleOrbitalWandef('Mo', range(1, 4+1), ['4db'],
24                               emin=emin, emax=emax,
25                               delower=delower, deupper=deupper))
26 wdc.add(wt.MultipleOrbitalWandef('Te', range(5, 12+1), ['5pb'],
27                               emin=emin, emax=emax,
28                               delower=delower, deupper=deupper))
29
30 wdc.writeFile('=.wandef')
31
32
33
34
35
36
37
38
39 return
40
41
42 # =====
43 # =====
44 # =====
45 # =====
46 if __name__ == '__main__':
47     main()

```

Line 23 defines the projectors for the 4d-orbitals of Mo sites 1–4. Note, that we use the python expression `range(1, 4+1)` to obtain the list `[1, 2, 3, 4]`. Similarly, line 26 defines the projectors for the 5p-orbitals of Te sites 5-12. The energy window encompasses the isolated band complex and we use a zero upper energy window tail to keep the higher bands out of the desired Hilbert space.

Please execute this script:

```
python makewandef.py
```

after which you should have the new file `=.wandef`. We run **fplo** now to obtain the necessary data for the WF calculation in `+wancoeff`:

```
fplo... > out
```

Convince yourself that the file `+wancoeff` has been created and re run **fplo** to calculate the Wannier functions:

```
fplo... > outwf
```

Now, the file `+hamdata` was created, which contains the WF model. In order to check the quality of the WF fit execute:

```
xfbp wband.xpy
```

You will see something like *The WF fit*. (page 120). The black curves are the DFT results, the red curves are the exact WF transformed bands and the green curves the bands resulting from the real space WF model.

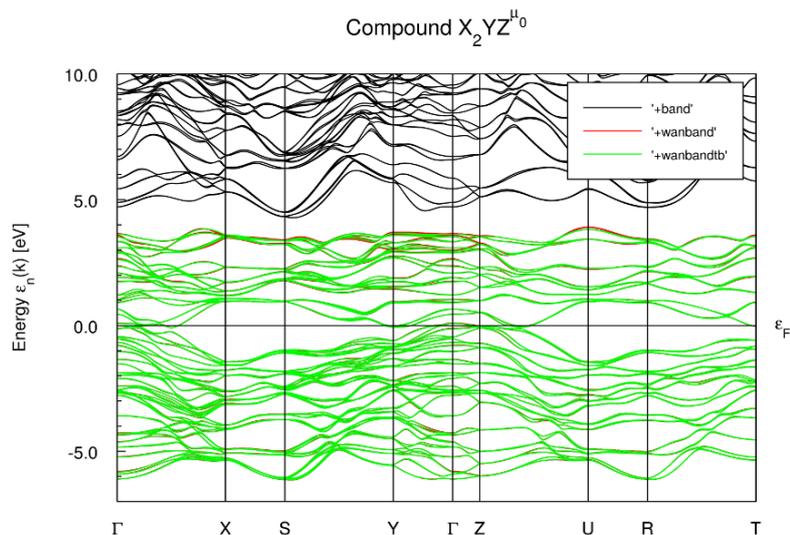


Fig. 3.34: The WF fit.

If you use a zoom function you will notice that the green curves deviate from the exact transform around the Fermi level by let's say 20meV. We could do better by using larger WF cutoffs and more SCF k-points, but for our purpose this is good enough.

To check the validity of the result we change into `slabify/3d`:

```
cd slabify/3d
```

and execute the two following commands:

```
python bands.py
xftp bands3d.xpy
```

with the result *The 3d bulk band structure from the WF model.* (page 120).

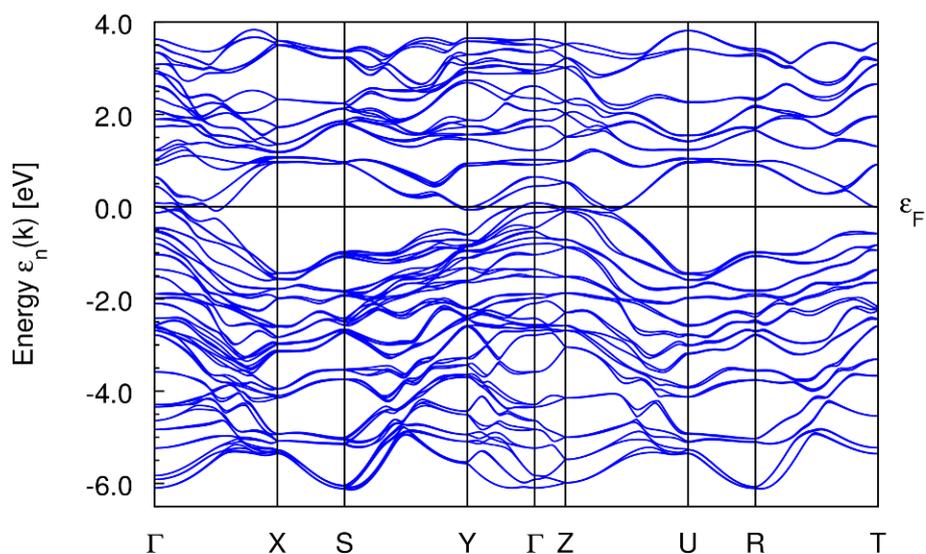


Fig. 3.35: The 3d bulk band structure from the WF model.

Have a look at the script `bands.py`

```

1  #!/usr/bin/env python
2
3  from __future__ import print_function
4  import sys
5
6  # If your pyfplo is not found you could also
7  # explicitly specify the pyfplo version path:
8  #sys.path.insert(0, "/home/magru/FPLO/FPLO22.00-62/PYTHON/doc");
9
10 import numpy as np
11 import pyfplo.slabify as sla
12 import pyfplo.fploio as fploio
13
14
15 print('\npyfplo version=: {0}\nfrom: {1}\n'.format(sla.version,sla.__file__))
16 # protect against wrong version
17 #if fedit.version!='22.00': raise RuntimeError('pyfplo version is incorrect.')
18
19
20 # =====
21 #
22 # =====
23
24 def work():
25
26
27     p=fploio.INParser()
28     p.parseFile('../../= .in')
29     d=p() ('special_sympoints')
30     l=[]
31     for i in range(d.size()):
32         l.append([ d[i] ('label') .S,d[i] ('kpoint') .listD])
33
34     hamdata='../../+hamdata'
35
36     s=sla.Slabify()
37     s.object='3d'
38     s.printStructureSettings()
39     s.prepare(hamdata)
40
41     bp=sla.BandPlot()
42     bp.points=l
43     bp.calculateBandPlotMesh(s.dirname)
44     s.calculateBandStructure(bp)
45
46
47 # =====
48 #
49 # =====
50
51
52 if __name__ == '__main__':
53
54     work()
55

```

In line 27-32 we read the high symmetry points from the **fplo** input file `=.in` which we then assign to the `BandPlot` (page 29) object in line 42. Line 43 prepares the path through the BZ and line 44 does the actual calculation. Do not forget to have a look at the **xfbp** script `bands3d.xpy` e.g. via the Edit->Script/Transformations editor of **xfbp**.

Next, in **xfbp** use a right mouse click at the highest band between X and S below the Fermi level to read off the band number of this band. You will get a popup menu with a number of bands (which might not be sorted). The highest number in this list will be 56. This will be the band for which we check the Weyl points.

Now, that we are confident that the model looks like the DFT result we will try to find Weyl points. For this we change into `../wpsearch`:

```
cd ../wpsearch
```

and have a look at the script `wpsearch.py`

```

1  #!/usr/bin/env python
2  # =====
3  # file:   auto.py
4  # author: k.koepernik@ifw-dresden.de
5  # date:   19 Jun 2017
6  from __future__ import print_function
7  import sys
8  import numpy as np
9  import numpy.linalg as LA
10
11 import pyfplo.slabify as sla
12
13 print( '\npyfplo version=: {0}\nfrom: {1}\n'.format(sla.version,sla.__file__) )
14
15
16
17
18 # =====
19 #
20 # =====
21 def work():
22
23
24     hamdata='../..+hamdata'
25     s=sla.Slabify()
26     s.object='3d'
27     s.prepare(hamdata)
28
29
30     homo=56
31
32     nk=10
33     tol=1e-4
34
35     # setup primitive reciprocal cell BoxMesh
36     G=s.hamdataRCell()
37     print( 'G=\n',G)
38     x=G[:,0]
39     y=G[:,1]
40     z=G[:,2]
41     dx=LA.norm(x)
42     dy=LA.norm(y)
43     dz=LA.norm(z)
44     # with most isotropic mesh subdivision
45     nx=max(2,nk)
46     ny=max(2,int(dy/dx*nx))
47     nz=max(2,int(dz/dx*nx))
48
49
50     # shift the origin by a small amount
51     k0=- (x/(nx-1)+y/(ny-1)+z/(nz-1))*0.5
52

```

(continues on next page)

(continued from previous page)

```

53     # define the BoxMesh
54     box=sla.BoxMesh()
55     box.setBox(xaxis=x,yaxis=y,zaxis=z,origin=k0)
56     box.setMesh(nx=nx,xinterval=[-dx/2,dx/2],
57               ny=ny,yinterval=[-dy/2,dy/2],
58               nz=nz,zinterval=[-dz/2,dz/2])
59
60     # and try to find Weyl points
61     s.findWeylPoints(box,[homo],tol)
62
63     try:
64         from weylpoints import wps
65     except:
66         raise RuntimeError('file weylpoints.py does not exist')
67
68     # the conventional cell
69     A=s.hamdataCCell()
70     # its lattice parameters
71     a=LA.norm(A[:,0])
72     b=LA.norm(A[:,1])
73     c=LA.norm(A[:,2])
74     boa=b/a
75     coa=c/a
76     # print in units of 2p/a_i
77     for i,w in enumerate(wps):
78         print( 'WP{0:4d}: {1:10.6f} {2:10.6f} {3:10.6f} {4:5.1f} {5:10.6f} {6:10d}
↪ {7}'\
79               .format(i,w.k[0],w.k[1]*boa,w.k[2]*coa,w.chirality,w.energy,
80                       w.homo,w.spin))
81
82     return
83
84
85     # =====
86     #
87     # =====
88
89 if __name__ == '__main__':
90     work()

```

In line 25-27 the default bulk 3d unit cell is setup. In line 30 we remember the number of the homo i.e. the band which forms the lower two legs of a potential Weyl point crossing. Line 32 contains the number of points of the initial search grid for the algorithm. Please consult *findWeylPoints* (page 53) for a description of the search algorithm. Line 33 defines the smallest bisection cell size.

In lines 35-47 we set up the primitive reciprocal unit cell vectors ( $x, y, z$ ) in units of *k*scale (page 56) (usually  $\frac{2\pi}{a_0}$ ), and their length ( $dx, dy$  and  $dz$ ) as well as a subdivision in each direction which results in the most isotropic mesh cells. These values are used to define a *BoxMesh* (page 56) in lines 54-58. Note line 51, in which we assign  $-1/2$  of the body diagonal of a mesh cell to  $k_0$ . This is used to shift the origin of the *BoxMesh* (page 56) such that we do not miss WPs which lay on a mesh cell boundary. Note, that *origin* (page 58) shifts the mesh and is not an absolute origin. Finally, line 61 executes the Weyl point search. First, for each cell of the *BoxMesh* (page 56) the Berry curvature is integrated over the cell surface, which is done with compact formulas as described for 2d in [Fukui05]. If the cell has a nonzero Chern number it is registered for future adaptive search. The resulting number of found Weyl points is shown in the output at the end of the progress message:

```
CPU TIME WeylScanner::scan box: 9.17 sec done 60% ETA=15.29 sec: 4 WP candidates
```

After the whole *BoxMesh* (page 56) was scanned a refined search takes place, in which each resulting cell is bisected, which leads to a small Mesh with 8 sub cells. For these sub cells a new search as described above is performed. Any cells with nonzero Chern number are registered for the next bisection step. The bisecting stops if the sub cell size is smaller than *tol*. The algorithm can search for several *homos* at once (just give a list). During

the bisections previously found WPs can vanish. This has to do with the fine structure of the Berry curvature and the limited accuracy of the 8-point integration of the Berry curvature. An example of this could look like this:

```
box size= 0.0155955, 4 WPs ...
... finer scan of the currently 4 WPs:

box size= 0.0077977, 3 WPs ...
... finer scan of the currently 3 WPs:
```

where two bisection steps are shown. After the first bisection one WP is lost. This also means that a large-*tol* result is indicative of the existence WPs (or false positives). If a WP gets lost the algorithm will redo the bisection step with a small shift of the cell. This seems to help in many cases. Yet, it does not guarantee that no loss occurs.

It is in general a good idea to try several mesh sizes (*nk*=), especially also in small steps. E.g. we could test for *nk*= 10, 11, 20, 21, 40, 41 and so on. Currently, there is no better way. It is **not safe** to assume that a large *nk* will detect all Weyl points! It is also very likely that the *origin* (page 58) matters. As you can see it can be quite tricky.

After the search is over the file `weylpoints.py` is created which contains a list of `WeylPoints` (page 64) At the end of the script (lines 63-66) this file is imported to make the WPs available and in lines 68-80 the WPs are printed in units of  $\frac{2\pi}{a_i}$ .

One last word about the *tol* parameter. If it is too large the accuracy of the resulting WP will be not so great. This is disadvantageous for future tests of these WPs.

We will now edit the script and put *nk*=10. Then we execute:

```
python -u wpsearch.py | tee out
```

The results are written to `stdout` and into the file `out`. You will notice that no WPs were found.

Next, set *nk* to 20 and rerun... again no WPs.

Next, set *nk* to 24 and rerun. You will have found 4 WPs after the initial scan and also after bisections. The output looks like:

```
WP[000]: k= -0.10121221 0.02978658 0.00001523 Chi=-1.00 E= 0.
↪055099\
 homo=56 spin=1
WP[001]: k= -0.10119098 -0.02980890 0.00001523 Chi= 1.00 E= 0.
↪055192\
 homo=56 spin=1
WP[002]: k= 0.10116975 0.02976426 0.00001523 Chi= 1.00 E= 0.
↪055209\
 homo=56 spin=1
WP[003]: k= 0.10119098 -0.02980890 0.00001523 Chi=-1.00 E= 0.
↪055192\
 homo=56 spin=1

-----
Slabify::findWeylPoints: END
-----

WP 0: -0.101212 0.054301 0.000061 -1.0 0.055099 56 1
WP 1: -0.101191 -0.054342 0.000061 1.0 0.055192 56 1
WP 2: 0.101170 0.054260 0.000061 1.0 0.055209 56 1
WP 3: 0.101191 -0.054342 0.000061 -1.0 0.055192 56 1
```

You will notice that the point coordinates, chirality and energy match the results of [Wang15]. You also notice that time reversal related WPs have the same chirality (WPs 1 and 2 or 0 and 3) while WPs connected by mirror symmetry have opposite chirality (WPs 0 and 1 or 2 and 3...). In principle we could have restricted the *BoxMesh* (page 56) to  $x, y, z \geq 0$  in lines 56-58. We did not do this to make the script most generic.

Let's now put  $nk=40$  and rerun... you got 4 WPs after the initial scan but magically found 5 more (of which #3 and #4 are identical) in the second bisection step:

```
WP[000]: k=   -0.10151555   -0.00971942    0.00001142  Chi=-1.00  E=   0.
↳016153\
  homo=56 spin=1
WP[001]: k=   -0.10151555    0.00975960    0.00001142  Chi= 1.00  E=   0.
↳016206\
  homo=56 spin=1
WP[002]: k=   -0.10121507   -0.02978769   -0.00001142  Chi= 1.00  E=   0.
↳055082\
  homo=56 spin=1
WP[003]: k=   -0.10119629    0.02978769    0.00001142  Chi=-1.00  E=   0.
↳055135\
  homo=56 spin=1
WP[004]: k=   -0.10117751    0.02976761    0.00001142  Chi=-1.00  E=   0.
↳055175\
  homo=56 spin=1
WP[005]: k=    0.10117751   -0.02976761    0.00001142  Chi=-1.00  E=   0.
↳055175\
  homo=56 spin=1
WP[006]: k=    0.10119629    0.02980778    0.00001142  Chi= 1.00  E=   0.
↳055162\
  homo=56 spin=1
WP[007]: k=    0.10155311   -0.00973951    0.00001142  Chi= 1.00  E=   0.
↳016018\
  homo=56 spin=1
WP[008]: k=    0.10155311    0.00973951   -0.00001142  Chi=-1.00  E=   0.
↳016018\
  homo=56 spin=1
```

```
-----
Slabify::findWeylPoints: END
-----
```

```
WP  0:  -0.101516  -0.017719   0.000046  -1.0   0.016153   56  1
WP  1:  -0.101516   0.017792   0.000046   1.0   0.016206   56  1
WP  2:  -0.101215  -0.054303  -0.000046   1.0   0.055082   56  1
WP  3:  -0.101196   0.054303   0.000046  -1.0   0.055135   56  1
WP  4:  -0.101178   0.054266   0.000046  -1.0   0.055175   56  1
WP  5:   0.101178  -0.054266   0.000046  -1.0   0.055175   56  1
WP  6:   0.101196   0.054340   0.000046   1.0   0.055162   56  1
WP  7:   0.101553  -0.017755   0.000046   1.0   0.016018   56  1
WP  8:   0.101553   0.017755  -0.000046  -1.0   0.016018   56  1
```

What must have happened is that the two WPs at low resolution in the initial step must have looked like a cell with chirality larger than one. This can easily happen because of the crude 8-point formula for the chirality of a mesh cell.

The 9 WPs are basically two different ones if we do not count symmetry equivalent WPs (look at the coordinates and energies). We cannot tell why this second WP was not been reported in [Wang15]. Either they missed it or it is a computational differences.

The occurrence of duplicates is due to the adaptive search and especially its internal cell shift to avoid loss of WPs.

After finding the WPs we need to check if they are false positives. Such things actually exist, e.g. if the berry curvature is locally cylindrical with outwards pointing vectors, in which case tiny roundoff errors will give a nonzero Chern number although it is not a 3d monopole. We do this with another script, which calculates the Chern number in a small sphere around the WP and which also produces a picture of the monopole (hedgehog). Have a look at the script `cherninsphere.py`.

```

1  #!/usr/bin/env python
2  # =====
3  # file:   auto.py
4  # author: k.koepernik@ifw-dresden.de
5  # date:   19 Jun 2017
6  from __future__ import print_function
7  import sys
8  import numpy as np
9  import numpy.linalg as LA
10
11 import pyfplo.slabify as sla
12
13 print('\npyfplo version=: {0}\nfrom: {1}\n'.format(sla.version,sla.__file__))
14
15
16
17
18 # =====
19 #
20 # =====
21 def work(iwp):
22
23
24     hamdata='../../+hamdata'
25     s=sla.Slabify()
26     s.object='3d'
27     s.prepare(hamdata)
28
29     try:
30         from weylpoints import wps
31     except:
32         raise RuntimeError('file weylpoints.py does not exist')
33
34     wp=wps[iwp]
35     s.calculateChernNumberInSphere(center=wp.k,
36                                 radius=wp.radius,homo=wp.homo,
37                                 nsubdiv=10,nradius=1)
38
39     return
40
41
42
43 # =====
44 #
45 # =====
46
47 if __name__ == '__main__':
48
49
50     if len(sys.argv)>1 and sys.argv[1]=='-h':
51         print("usage: {} [-h] -i wp-number".format(sys.argv[0]))
52         sys.exit(0)
53     try:
54         i=int(sys.argv.index("-i"))
55         iwp=int(sys.argv[i+1])
56     except:
57         raise RuntimeError('need option -i wp-number')
58
59
60     work(iwp)
61
62     sys.exit(0)

```

(continues on next page)

(continued from previous page)

63  
64  
65  
66

In lines 24-27 we have the usual 3d setup. In lines 30-33 we try to import `weylpoints.py` (result of `wpssearch.py`) and in lines 35-37 we pick a particular WP and calculate the Chern number in a sphere around the WP center. The center and homo are taken from `weylpoints.py` as well as the radius of the sphere. The radius saved in `weylpoints.py` is the size of the smallest bisection cell in which the WP was found. This is usually a sensible radius. In some cases it might be better to take a larger or smaller radius. If a smaller radius is chosen the WP might actually fall outside the sphere, which is not good. You most likely can judge this from the graphical representation (see below). The script takes a command line option to specify which WP to look at. Note, that WPs are numbered starting with 0.

Let's start with WP No. 0:

```
python -u cherninsphere.py -i 0
```

In the resulting output you find lines:

```
List of chern numbers with abs value larger than 0.01
chern up to band 56:                -0.387852580115874
```

In our case only one chern number is written (for homo 56, which is what we where looking at). However, the printed Chern number is far from integer. So, let's look at the picture first. The program writes files called `berrycurvsphere.net` and so forth. You need to have the program `opendx` installed for this to work. Please execute:

```
dx -execute_on_change -image berrycurvsphere.net
```

From the menu Options in the main window chose View Control. In the dialog click the Reset button at the bottom right. Then in the mode combobox select Rotate and use the left mouse click and drag to rotate the picture. You will see something like *The hedgehog (monopole) of WP 0*. (page 127). (If your `opendx` works as mine does you can use `Ctrl+F` for reset and `Ctrl+R` for rotate.)

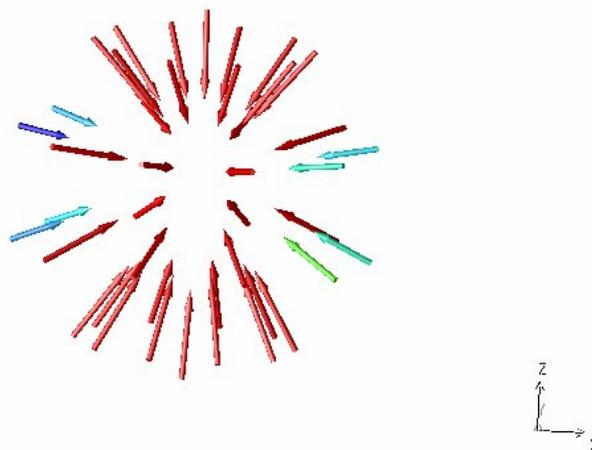


Fig. 3.36: The hedgehog (monopole) of WP 0.

Please note that all vectors point inside, which indicates negative chirality which fits the data stored in `weylpoints.py` and also the sign of the Chern number written to the output.

Now, in the control panel (menu windows -> Open All Control Panels) use the arrows on the AutoGlyph upper limit to decrease/increase the vector length limit. If you click the arrows for a longer

time the changes will speed up. If you go to the lowest value (left arrow) the arrow will be of the same length since each vector of larger length will be limited in length. If you however go to the highest limit (right arrow) essentially no limit is applied and you see the actual length of the Berry curvature field. You will notice that the vectors are rather long around the poles and very short around the equator. This is the reason for the non-integer Chern number. The curvature field is highly anisotropic. This is also an issue for the Weyl point finding algorithm (as you noticed it is not straight forward to find all 8 WPs). In such a situation we can increase the number of points on the sphere *nsubdiv* in line 37.

Please increase it to *nsubdiv=20* and rerun the script... the Chern number is -0.632.

Please increase it to *nsubdiv=40* and rerun the script... the Chern number is -0.911.

Please increase it to *nsubdiv=100* and rerun the script... the Chern number is -0.993.

So, we finally converged to the result, which we already knew. This is a somewhat extreme case. The more important point is the graphical representation. If it shows a monopole (especially for limited vector length) the Weyl point is valid. Btw. WP No. 3 is even more extreme.

Please revert the script to *nsubdiv=10* and check the other Weyl points: have a look at the graphical representation. We will have confirmed that all WPs are actually Weyl points by confirming the monopole. Indeed, it would have sufficed to check the two WPs with different energies since the others are obtained by symmetry.

With this the WP search is complete. We can proceed e.g. with surface spectra. Please change into *semi*:

```
cd ../semi
```

Have a look at the script *fs.py* which calculates the surface spectral function for  $E = 0$  in the surface Brillouin zone (the Fermi surface equivalent).

```

1  #! /usr/bin/env python
2
3  from __future__ import print_function
4  import sys
5
6  # If your pyfplo is not found you could also
7  # explicitly specify the pyfplo version path:
8  #sys.path.insert(0, "/home/magru/FPLO/FPLO22.00-62/PYTHON/doc");
9
10 import numpy as np
11 import numpy.linalg as LA
12 import pyfplo.slabify as sla
13
14
15 print( '\npyfplo version=: {0}\nfrom: {1}\n'.format(sla.version,sla.__file__)
16 # protect against wrong version
17 #if fedit.version!='22.00': raise RuntimeError('pyfplo version is incorrect.')
18
19
20 # =====
21 #
22 # =====
23
24 def work():
25
26
27     hamdata='../..+hamdata'
28
29     s=sla.Slabify()
30     s.object='semislab'
31     s.zaxis=[0,0,-1]
32     s.numberoflayers=1
33     s.anchor=0.001
34     s.printStructureSettings()
35     s.prepare(hamdata)

```

(continues on next page)

(continued from previous page)

```

36
37     # the conventional cell
38     A=s.hamdataCCell()
39     # its lattice parameters
40     a=LA.norm(A[:,0])
41     b=LA.norm(A[:,1])
42     aob=a/b
43
44     Nk=200
45     fso=sla.FermiSurfaceOptions()
46     fso.setMesh(Nk,[0,0.5],Nk,[0,0.5*aob])
47     fso.setPlane([1,0,0],[0,1,0],[0,0,0])
48     fso.fermienergy=-0.02
49     fso.fermienergyim=1./Nk*1.
50     print(fso)
51
52     s.calculateFermiSurfaceSpectralDensity(fso,penetrationdepth=-1.)
53
54     # =====
55     #
56     # =====
57
58
59     if __name__ == '__main__':
60
61         work()
62

```

In lines 29-35 we setup a semi infinite slab with a (00-1) surface. We anchor the surface such that a  $\text{MoTe}_2$  layer is the last layer. After executing the script (after the structure setup was executed, hence before the actual calculation starts) the directory `slabifyres` contains the file `.in_final_PLlayer`. If you load it into **xfplo** you will see the cell whose semi-infinite repetition will make up the semi-slab. Note, that the c-axis points downwards! Hence the surface is the lower face of the cell while the infinite side grows on the upper cell face (not shown). If you had chosen the (001) surface the c-axis would have pointed upwards. Note, that `numberoflayers` is set to 1. Our 3d unit cell is already long enough to fulfill the condition that no hopping reaches further then between two adjacent primary layers.

In lines 37-42 we setup the axis scales of the surface BZ and in lines 44-50 we set up the surface BZ and a mesh which represents the upper right quadrant of the surface BZ. We set the Fermi surface to -20meV as in [Wang15]. The imaginary part is chosen in a reasonable way such that the Lorentzian width is comparable to the mesh distance. If we would take a much smaller imaginary part, the picture would tend to look spotty. A larger value will smear it out.

In line 52 the actual calculation takes place. After the calculation finishes please execute:

```
xfbp fs.xpy
```

to produce *this Figure* (page 130). Compare this to Fig. 3 of [Wang15] and also read the discussion of the surface state therein. Please note, that we show the whole quadrant, while [Wang15] shows a smaller part of the whole surface BZ.

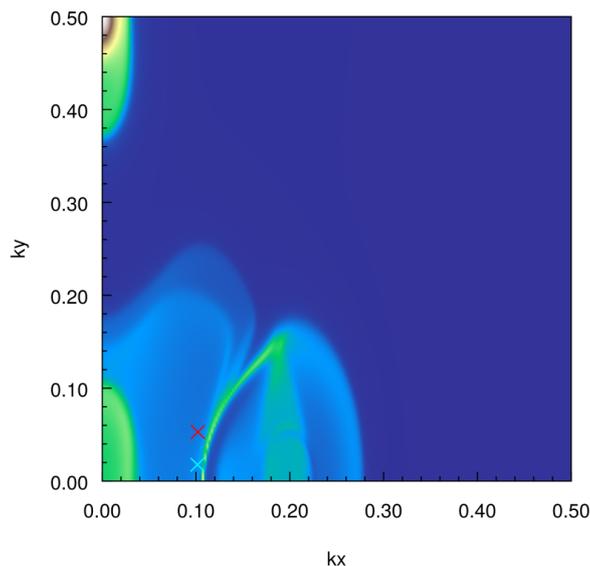


Fig. 3.37: The surface Fermi surface spectral function for  $E = -20$  meV. The two crosses denote the two WPs of this quadrant.

Finally, we could also calculate energy distribution curves. This was, however already discussed in other examples. With this the basic Weyl point example shall come to its end.

## 3.5 FEDIT scripting examples

- *Set bandplot points* (page 130)
- *Simple fedit example* (page 131)
- *BCC Iron* (page 133)
- *BCC and FCC Iron* (page 137)
- *Set Extended basis* (page 142)
- *BCC Iron, extended basis* (page 144)
- *mBJ XC-potential* (page 148)

### 3.5.1 Set bandplot points

The tutorial files are in `FPLO.../DOC/pyfplo/Examples/fploio/fedit_use/setbandplot` where `FPLO...` stands for your version's FPLO directory, e.g. `FPLO21.00-61`. Here are the files of this directory:

- *setbandplot.py* (page 130)

`setbandplot.py`

```

1  #! /usr/bin/env python
2  # =====
3  # file:   bandplot.py
4  # author: k.koepernik@ifw-dresden.de
5  # date:   19 Apr 2017

```

(continues on next page)

(continued from previous page)

```

6 from __future__ import print_function
7 import sys
8 import pyfplo.fedit as fedit
9 # =====
10 #
11 # =====
12 def work():
13
14
15     points=[
16         ['$~G', [0,0,0]],
17         ['X', [1,0,0]],
18         ['M', [1,1,0]],
19         ['$~G', [0,0,0]],
20         ['Z', [0,0,1]],
21     ]
22
23     # IMPORTANT:
24     # We only want to change bandplot settings in an existing/new =.in.
25     # Hence, we set recreate=False, which DOES NOT resets non-symmetry input.
26     fed=fedit.Fedit(recreate=False)
27     fed.bandplot(active=True,points=points,weights=True,
28                 interval=[2000,-1,1])
29     fed.pipeFedit()
30
31
32     return
33
34 # =====
35 #
36 # =====
37
38 if __name__ == '__main__':
39
40     work()
41
42     sys.exit(0)
43
44
45
46

```

### 3.5.2 Simple fedit example

This example runs **fplo** for pre-prepared input until convergence, then switches on the **bandplot** option and re-runs to calculate the band structure. The tutorial files are in `FPLO.../DOC/pyfplo/Examples/fploio/fedit_use/simple` where `FPLO...` stands for your version's `FPLO` directory, e.g. `FPLO21.00-61`. Here are the files of this directory:

- [README](#) (page 131)
- [wrapp.sh](#) (page 132)
- [simple.py](#) (page 132)

README

```

1 Read the script to understand what it is doing.
2 It just demonstrates how one could use python to run things.
3 The fedit part is actually very minimal.
4

```

(continues on next page)

(continued from previous page)

```

5 run simple.py as
6
7     simple.py
8
9 then have a look what was created.
10
11
12 If pyfplo path needs to be set use wrapp.sh. First edit pyfplopath
13 in wrapp.sh and then just put it in front as in any of the following.
14
15 wrapp.sh simple.py

```

A wrapper to setup paths wrapp.sh

```

1 #! /usr/bin/env sh
2 #
3 # Example wrapper script for path setting.
4 #
5 #####
6
7
8
9 # set your path here
10
11 pyfplopath=$HOME/FPLO/FPLO22.00-62/PYTHON/
12
13 export PYTHONPATH=$pyfplopath:$PYTHONPATH
14
15
16 $*

```

The python script simple.py

```

1 #! /usr/bin/env python
2 # =====
3 # file:    simple.py
4 # author:  k.koepernik@ifw-dresden.de
5 # date:    06 Jun 2017
6 from __future__ import print_function
7 import sys
8 import os
9 import pyfplo.fedit as fedit
10
11 print( '\npyfplo version=: {0}\nfrom: {1}\n'.format(fedit.version,fedit.__file__) )
12 # protect against wrong version
13 #if fedit.version!='22.00': raise RuntimeError('pyfplo version is incorrect.')
14
15 # =====
16 #
17 # =====
18 def work():
19
20     # rm old files
21     os.system('rm +dos* +band +bweigh*')
22
23     # get a Fedit instance
24     fed=fedit.Fedit(recreate=False)
25     # set some stuff for the SCF run
26     fed.bzintegration(nxyz=[6,6,6])
27
28     # switch off bandplot (if already set) so that we do not

```

(continues on next page)

(continued from previous page)

```

29     # calculate unnecessary stuff during self consistency
30     fed.bandplot(active=False)
31     # write input file
32     fed.pipeFedit()
33
34     # run fplo (SCF)
35     os.system(fedit.fploExecutable()+"|tee outscf")
36
37
38     # now we re-run for bandplot.
39     # Note that we use the Fedit() from before
40     fed.bandplot(active=True)
41     fed.pipeFedit()
42     os.system(fedit.fploExecutable()+"|tee outb")
43
44
45     return
46
47     # =====
48     #
49     # =====
50
51 if __name__ == '__main__':
52
53     work()
54
55     sys.exit(0)
56
57
58
59

```

### 3.5.3 BCC Iron

This example prepares **fplo** input and optionally runs the calculations. The tutorial files are in `FPLO.../DOC/pyfplo/Examples/fploio/fedit_use/bccFe` where `FPLO...` stands for your version's FPLO directory, e.g. `FPLO21.00-61`. Here are the files of this directory:

- [README](#) (page 133)
- [wrapp.sh](#) (page 134)
- [simple.py](#) (page 134)

README

```

1 Read the script to understand what it is doing.
2 It just demonstrates how one could use python to do things.
3 The fedit part is actually very small.
4
5 run simple.py as
6
7     simple.py
8
9 then have a look what was created.
10 Now run
11
12     simple.py -r -c
13
14 to do the calculations and collect the results.
15

```

(continues on next page)

(continued from previous page)

```

16 Or just
17
18     simple.py -c
19
20 to only collect results (if there are any yet).
21
22 If pyfplo path needs to be set use wrapp.sh. First edit pyfplopath
23 in wrapp.sh and then just put it in front as in any of the following.
24
25 wrapp.sh simple.py
26 wrapp.sh simple.py -r -c
27 wrapp.sh simple.py -c

```

A wrapper to setup paths wrapp.sh

```

1  #!/usr/bin/env sh
2  #
3  # Example wrapper script for path setting.
4  #
5  #####
6
7
8
9  # set your path here
10
11 pyfplopath=$HOME/FPLO/FPLO22.00-62/PYTHON/
12
13 export PYTHONPATH=$pyfplopath:$PYTHONPATH
14
15
16 $*

```

The python script simple.py

```

1  #!/usr/bin/env python
2  #
3  # Example script to create a series of calculations for varying lattice
4  # constant for bcc Fe.
5  #
6  #
7  #####
8  from __future__ import print_function
9  import sys
10
11 # If your pyfplo is not found you could als
12 # explicitly specify the pyfplo version path:
13 #sys.path.insert(0, "/home/magru/FPLO/FPLO22.00-62/PYTHON/doc");
14
15 import os
16 from optparse import OptionParser
17 import numpy as np
18 import pyfplo.fedit as fedit
19
20
21 print( '\npyfplo version=: {0}\nfrom: {1}\n'.format(fedit.version,fedit.__file__)
22 # protect against wrong version
23 #if fedit.version!='22.00': raise RuntimeError('pyfplo version is incorrect.')
24
25 # =====
26 #
27 # =====

```

(continues on next page)

(continued from previous page)

```

28
29 def INPUT(n):
30     if sys.version_info[0] == 3:
31         return input(n)
32     else:
33         return raw_input(n)
34
35 # =====
36 #
37 # =====
38
39 def work(fplo, runit=False):
40
41     # sanity check
42     if runit:
43         if INPUT("Shall I run the jobs: [y/n]") != 'y':
44             print( "\nOK no running then.\n")
45             sys.exit(0)
46     else:
47         if INPUT("Shall I (re)create the input: [y/n]") != 'y':
48             print( "\nOK, aborting.\n")
49             sys.exit(0)
50
51
52     # Give all directories a name prefixed with the name of the parameter,
53     # which is running, followed by the parameter itself
54     prefix='a0='
55
56     # Remember the current directory.
57     ROOT=os.getcwd()
58
59     # loop over the running parameter, in our case the lattice constant
60
61     for x in np.arange(4.4,6.01,0.4):
62         # make sure we are in the root directory of our data directory tree
63         os.chdir(ROOT)
64
65         # create the directory name as described above (example 'a0=6.00')
66         # Use explicit format for x to ensure 2 digits after the comma.
67
68         dir='{0}{1}'.format(prefix, '{0:12.2f}'.format(x).strip())
69
70
71         # input creation branch
72
73         # if the directory does not yet exist, create it
74         if not os.path.exists(dir):
75             os.mkdir(dir)
76             print( 'directory '+dir+' created')
77         else:
78             print( 'directory {0} exists already'.format(dir))
79
80         # change into the directory of paramter $xx
81         os.chdir(dir)
82
83
84         # do the fedit magic
85         fed=fedit.Fedit()
86         fed.resetPipeInput(recreate=True) # important
87         fed.symmetry(compound="Fe, a0-variation",
88                     spacegroup=229,

```

(continues on next page)

(continued from previous page)

```

89         type='cry',
90         units='bohr',
91         latcon=[x,x,x],
92         angles=['90.']*3,
93         atoms=[['fe',[0,0,0]]],
94     )
95     fed.bzintegration([16,16,16])
96     fed.vxc(version='5') # gga
97     fed.relativistic('scalar')
98     fed.spin(spin=2,initialspinsplit=1,initialspin=[[1,2.5]])
99     fed.pipeFedit()
100
101     # do we run the job?
102     if runit:
103
104         print( fplo+" running in "+dir+" ...")
105
106         # now execute, whatever is nessecary to launch job in the current
107         # directory (dir)
108
109         # START Example
110         # We just run the jobs sequentially on a single machine
111         # and redirect stdout to file 'out' and stderr to /dev/null.
112         # (In this way there will be no dangling output and the job could
113         # run savely in the background, which is not done in our example
114         # here.)
115
116         # Furthermore, we use the +yes-file mechanism to avoid a crash
117         # due to repeated inital polarization (spin split).
118         # The "y" below enforces fplo to continue in such situation
119         # without a repeated split and does nothing otherwise. See manual.
120         with open('+yes','w') as f:
121             f.write('y')
122
123         os.system('cat +yes | {0} 2>/dev/null > out'.format(fplo))
124         # END Example
125
126         # just in case
127         os.chdir(ROOT)
128     # and of x-loop
129
130     os.chdir(ROOT)
131
132     #
133     # After the run we should have a directory structure like
134     #
135     # ./a0=4.40/
136     # ./a0=4.80/
137     # ./a0=5.20/
138     # ./a0=5.60/
139     # ./a0=6.00/
140     # ./simple.py
141     #
142     # where every directory contains the same setup, except for the
143     # lattice constant.
144     # We may now perform converged calculations (option -r) in all directories.
145     # If we want to change, say, the number of k-points, we edit this number
146     # in the pipe-section above, re-run that script to change the input and
147     # re-converge the calculations (option -r).
148     #
149 
```

(continues on next page)

(continued from previous page)

```

150
151 # =====
152 #
153 # =====
154
155 def collect():
156
157     # collect the results
158     os.system("grepfplo -p 'a0=' -m EE | tee e")
159     os.system("grepfplo -p 'a0=' -m SS | tee s")
160     os.system("xfbp pic.cmd")
161
162
163 # =====
164 #
165 # =====
166 if __name__ == "__main__":
167
168     # Set an FPLO version, you need to set this according to your
169     # needs, including possibly a path. Or you use option -p.
170     # A guess for the default name:
171     FPLO=fedit.fploExecutable()
172
173     # scan command line options
174     usage = "usage: %prog [-c] [-r] [-h] [-p fploexecname]"
175     parser = OptionParser(usage)
176     parser.add_option('-r', '', action='store_true', dest='run', default=False,
177                       help='force fplo run')
178     parser.add_option('-c', '', action='store_true', dest='collect', default=False,
179                       help='collect results')
180     parser.add_option('-p', '', type='str', dest='fplo', default=FPLO,
181                       help='optional: the name of an FPLO executable\n'+
182                             'possibly with explicit path')
183     (options, args) = parser.parse_args()
184
185     # do the work
186     if not options.collect or options.run:
187         work(options.fplo, options.run)
188
189     if options.collect:
190         collect()

```

### 3.5.4 BCC and FCC Iron

A more complex example, preparing input and running **fplo** is shown here. Note, that we run sequentially. If you have a job queueing system you need to modify the script accordingly. The tutorial files are in `FPLO.../DOC/pyfplo/Examples/fploio/fedit_use/Fe` where `FPLO...` stands for your version's FPLO directory, e.g. `FPLO21.00-61`. Here are the files of this directory:

- [README](#) (page 137)
- [wrapp.sh](#) (page 138)
- [simple.py](#) (page 138)

README

```

1 Read the script to understand what it is doing.
2 It just demonstrates how one could use python to do things.
3 The fedit part is actually very small.
4

```

(continues on next page)

(continued from previous page)

```

5 run notsosimple.py as
6
7     notsosimple.py
8
9 then have a look what was created.
10 Now run
11
12     notsosimple.py -r -c
13
14 to do the calculations and collect the results.
15
16 Or just
17
18     notsosimple.py -c
19
20 to only collect results (if there are any yet).
21
22 If pyfplo path needs to be set use wrapp.sh. First edit pyfplopath
23 in wrapp.sh and then just put it in front as in any of the following.
24
25 wrapp.sh notsosimple.py
26 wrapp.sh notsosimple.py -r -c
27 wrapp.sh notsosimple.py -c

```

A wrapper to setup paths wrapp.sh

```

1 #!/usr/bin/env sh
2 #
3 # Example wrapper script for path setting.
4 #
5 #####
6
7
8
9 # set your path here
10
11
12 pyfplopath=$HOME/FPLO/FPLO22.00-62/PYTHON
13
14 export PYTHONPATH=$PYTHONPATH:$pyfplopath
15
16
17 $*

```

The python script notsosimple.py

```

1 #!/usr/bin/env python
2 #
3 # Example script to create a series of calculations for varying lattice
4 # constant for bcc and fcc Fe.
5 #
6 #
7 #####
8 from __future__ import print_function
9 import sys
10 # If your pyfplo is not found you could als
11 # explicitly specify the pyfplo version path:
12 #sys.path.insert(0, "/home/magru/FPLO/FPLO22.00-62/PYTHON/doc");
13
14 import os
15 from optparse import OptionParser

```

(continues on next page)

(continued from previous page)

```

16 import numpy as np
17 import pyfplo.fedit as fedit
18 import pyfplo.fploio as fploio
19
20 print( '\npyfplo version=: {0}\nfrom: {1}\n'.format(fedit.version,fedit.__file__)
21 # protect against wrong version
22 #if fedit.version!='22.00': raise RuntimeError('pyfplo version is incorrect.')
23
24
25 # =====
26 #
27 # =====
28
29 def makeInput(a0,structure,vxc,spin):
30
31     if structure=='bcc':
32         group=229
33     elif structure=='fcc':
34         group=225
35
36     if vxc=='lda':
37         xc='4'
38     elif vxc=='gga':
39         xc='5'
40
41     fed=fedit.Fedit()
42     fed.resetPipeInput(recreate=True) # important
43     fed.symmetry(compound="Fe, a0-variation",
44                 spacegroup=group,
45                 type='cry',
46                 units='bohr',
47                 latcon=[a0]*3,
48                 angles=['90.']*3,
49                 atoms=[['fe',[0,0,0]],
50
51
52
53
54
55
56
57
58
59
60
61 # =====
62 #
63 # =====
64
65 def work(fplo,structure,vxc,spin,runit=False):
66
67     # Give all directories a name prefixed with the name of the parameter,
68     # which is running, followed by the parameter itself
69     prefix='V='
70
71     # Remember the current directory.
72     ROOT=os.getcwd()
73
74     # loop over the running parameter, in our case the lattice constant (volume)
75
76     vvs=np.arange(60.,90.001,3)
77     if structure=='bcc':

```

(continues on next page)

(continued from previous page)

```

77     nsite=2
78     else:
79         nsite=4
80
81     spdir='NSP' if spin==1 else 'SP'
82     basedir=vxc+'/' + structure+'/' + spdir
83
84     for v in vvs:
85
86         x=(v*nsite)**(1./3)
87
88         # make sure we are in the root directory of our data directory tree
89         os.chdir(ROOT)
90
91         # create the directory name as described above (example 'V=60.00')
92         # Use explicit format for x to ensure 2 digits after the comma.
93         # This of course depends on the actual values.
94
95         dir='{2}/{0}{1}'.format(prefix, '{0:12.2f}'.format(v).strip(),
96                                basedir)
97
98
99         # input creation branch
100
101         # if the directory does not yet exist, create it
102         if not os.path.exists(dir):
103             os.makedirs(dir);
104             print( 'directory '+dir+' created')
105
106         # change into the directory of parameter v
107         os.chdir(dir)
108
109
110         # make input
111         makeInput(x, structure, vxc, spin)
112
113         # do we run the job?
114         if runit:
115
116             print( fplo+" running in "+dir+" ...")
117
118             # now execute, whatever is necessary to launch job in the current
119             # directory (dir)
120
121             # START Example
122             # We just run the jobs sequentially on a single machine
123             # and redirect stdout to file 'out' and stderr to /dev/null.
124             # (In this way there will be no dangling output and the job could
125             # run safely in the background, which is not done in our example
126             # here.)
127
128             # Furthermore, we use the +yes-file mechanism to avoid a crash
129             # due to repeated initial polarization (spin split).
130             # The "y" below enforces fplo to continue in such situation
131             # without a repeated split and does nothing otherwise. See manual.
132             with open('+yes', 'w') as f:
133                 f.write('y')
134
135             os.system('cat +yes | {0} 2>/dev/null > out'.format(fplo))
136             # END Example
137

```

(continues on next page)

(continued from previous page)

```

138
139
140     # just in case
141     os.chdir(ROOT)
142     # and of x-loop
143
144     os.chdir(ROOT)
145
146     if runit:
147         os.chdir(basedir)
148         os.system("grepfplo -p 'V=' -m EE | tee e".format(prefix))
149         os.system("grepfplo -p 'V=' -m SS | tee s".format(prefix))
150         os.chdir(ROOT)
151
152
153
154     # =====
155     #
156     # =====
157
158     def collect():
159
160         # collect the results
161         os.system("xftp pic.cmd")
162
163
164     # =====
165     #
166     # =====
167
168     def INPUT(n):
169         if sys.version_info[0] == 3:
170             return input(n)
171         else:
172             return raw_input(n)
173
174     # =====
175     #
176     # =====
177     if __name__ == "__main__":
178
179         # Set an FPLO version, you need to set this according to your
180         # needs, including possibly a path. Or you use option -p.
181         # A guess for the default name:
182         FPLO=fploio.fploExecutable()
183
184         # scan command line options
185         usage = "usage: %prog [-c] [-r] [-h] [-p fploexecname]"
186         parser = OptionParser(usage)
187         parser.add_option('-r', '', action='store_true', dest='run', default=False,
188                           help='force fplo run')
189         parser.add_option('-c', '', action='store_true', dest='collect', default=False,
190                           help='collect results')
191         parser.add_option('-p', '', type='str', dest='fplo', default=FPLO,
192                           help='optional: the name of an FPLO executable\n'+
193                                'possibly with explicit path')
194         (options, args) = parser.parse_args()
195
196         # sanity check
197         if (not options.collect) and options.run:
198             if INPUT("Shall I run the jobs: [y/n]")!='y':

```

(continues on next page)

(continued from previous page)

```

199     print( "\nOK no running then.\n")
200     sys.exit(0)
201     elif (not options.collect) and (not options.run):
202         if INPUT("Shall I (re)create the input: [y/n]")!='y':
203             print( "\nOK, aborting.\n")
204             sys.exit(0)
205
206
207     # do the work
208     if not options.collect or options.run:
209         for structure in ['bcc','fcc']:
210             for vxc in ['lda','gga']:
211                 for spin in [1,2]:
212                     if structure=='fcc' and spin==2: continue
213                     work(options.fplo,structure,vxc,spin,options.run)
214
215     if (not options.collect) and options.run:
216         print( '\nTo show results rerun with option -c.')
217         print( 'To rerun (shorter output files) rerun -r or -r -c.\n')
218
219     if (not options.collect) and (not options.run):
220         print( '\nTo calculate run with option -r or -r -c\n')
221
222     if options.collect:
223         collect()

```

### 3.5.5 Set Extended basis

To simply switch on an extended basis (as was used in some **fplo** publications) use the following example as orientation (the actual code is three lines) The tutorial files are in `FPLO.../DOC/pyfplo/Examples/fploio/set_extended_basis` where `FPLO...` stands for your version's `FPLO` directory, e.g. `FPLO21.00-61`. Here are the files of this directory:

- *README* (page 142)
- *wrapp.sh* (page 142)
- *simple.py* (page 143)

README

```

1 Read the script to understand what it is doing.
2 It just demonstrates how one can set an extended basis
3 in an existing =.in.
4
5 run extended_basis.py as
6
7     extended_basis.py
8
9 then have a look at =.in via fedit.
10
11
12 If pyfplo path needs to be set use wrapp.sh. First edit pyfplopath
13 in wrapp.sh and then just put it in front as in any of the following.
14
15 wrapp.sh extended_basis.py

```

A wrapper to setup paths `wrapp.sh`

```

1 #! /usr/bin/env sh
2 #

```

(continues on next page)

(continued from previous page)

```

3 # Example wrapper script for path setting.
4 #
5 #####
6
7
8
9 # set your path here
10
11 pyfplopath=$HOME/FPLO/FPLO22.00-62/PYTHON/
12
13 export PYTHONPATH=$pyfplopath:$PYTHONPATH
14
15
16 $*

```

The python script `extended_basis.py`

```

1  #!/usr/bin/env python
2  # =====
3  # file:   extended_basis.py
4  # author: k.koepernik@ifw-dresden.de
5  # date:   06 Jun 2017
6  from __future__ import print_function
7  import sys
8  import os
9  import pyfplo.fedit as fedit
10
11 print( '\npyfplo version=: {0}\nfrom: {1}\n'.format(fedit.version,fedit.__file__)
12 # protect against wrong version
13 #if fedit.version!='22.00': raise RuntimeError('pyfplo version is incorrect.')
14
15 # =====
16 #
17 # =====
18 def work() :
19     # Get an Fedit instance:
20     # We want to change the basis in an existing =.in so recreate=False !!
21     fed=fedit.Fedit(recreate=False)
22     # Set extensionlevel=2 and add H-3d and f-orbital if needed and
23     # leave the rest as it is:
24     fed.basis(extensionlevel=2,add3d=True,addf=True)
25     # If the rest of the basis menu should be reset to default use:
26     fed.basis(extensionlevel=2,add3d=True,addf=True,
27             multicore=[],multisemicore=[],
28             core4f=[],core4fNoValenceF=[])
29
30     fed.pipeFedit(prot=True)
31     return
32
33 # =====
34 #
35 # =====
36
37 if __name__ == '__main__' :
38
39     work()
40
41     sys.exit(0)
42
43
44

```

(continues on next page)

### 3.5.6 BCC Iron, extended basis

This example prepares **fplo** input and optionally runs the calculations to show the influence of an extended basis. The tutorial files are in `FPLO.../DOC/pyfplo/Examples/fploio/fedit_use/bccFe_extended_basis` where `FPLO...` stands for your version's `FPLO` directory, e.g. `FPLO21.00-61`. Here are the files of this directory:

- [README](#) (page 155)
- [wrapp.sh](#) (page 144)
- [extbasis.py](#) (page 145)

README

```

1 Read the script to understand what it is doing.
2 It just demonstrates how to use pyfplo.fedit to
3 choose the basis.
4
5 run extbasis.py as
6
7     extbasis.py
8
9 then have a look what was created.
10 Now run
11
12     extbasis.py -r -c
13
14 to do the calculations and collect the results.
15
16 Or just
17
18     extbasis.py -c
19
20 to only collect results (if there are any yet).
21
22 If pyfplo path needs to be set use wrapp.sh. First edit pyfplopath
23 in wrapp.sh and then just put it in front as in any of the following.
24
25 wrapp.sh extbasis.py
26 wrapp.sh extbasis.py -r -c
27 wrapp.sh extbasis.py -c

```

A wrapper to setup paths `wrapp.sh`

```

1 #! /usr/bin/env sh
2 #
3 # Example wrapper script for path setting.
4 #
5 #####
6
7
8
9 # set your path here
10
11 pyfplopath=$HOME/FPLO/FPLO22.00-62/PYTHON/
12
13 export PYTHONPATH=$pyfplopath:$PYTHONPATH
14

```

(continues on next page)

(continued from previous page)

15  
16 \$\*The python script `extbasis.py`

```

1  #!/usr/bin/env python
2  #
3  # Example script to create a series of calculations for varying lattice
4  # constant for bcc Fe.
5  #
6  #
7  #####
8  from __future__ import print_function
9  import sys
10
11 # If your pyfplo is not found you could als
12 # explicitly specify the pyfplo version path:
13 #sys.path.insert(0, "/home/magru/FPLO/FPLO22.00-62/PYTHON/doc");
14
15 import os
16 from optparse import OptionParser
17 import numpy as np
18 import pyfplo.fedit as fedit
19
20
21 print( '\npyfplo version=: {0}\nfrom: {1}\n'.format(fedit.version,fedit.__file__))
22 # protect against wrong version
23 #if fedit.version!='22.00': raise RuntimeError('pyfplo version is incorrect.')
24
25 # =====
26 #
27 # =====
28
29 def INPUT(n):
30     if sys.version_info[0] == 3:
31         return input(n)
32     else:
33         return raw_input(n)
34
35 # =====
36 #
37 # =====
38
39 def work(fplo,bases,runit=False):
40
41     # sanity check
42     if runit:
43         if INPUT("Shall I run the jobs: [y/n]")!='y':
44             print( "\nOK no running then.\n")
45             sys.exit(0)
46     else:
47         if INPUT("Shall I (re)create the input: [y/n]")!='y':
48             print( "\nOK, aborting.\n")
49             sys.exit(0)
50
51
52     # Give all directories a name prefixed with the name of the parameter,
53 # which is running, followed by the parameter itself
54     prefix='a0='
55
56     # Remember the current directory.

```

(continues on next page)

(continued from previous page)

```

57 ROOT=os.getcwd()
58
59 # loop over the running parameter, in our case the lattice constant
60
61 for bas in bases:
62     os.chdir(ROOT)
63
64     if not os.path.exists(bas): os.mkdir(bas)
65     os.chdir(bas)
66
67     for x in np.arange(4.4,6.01,0.4):
68         # make sure we are in the root directory of our data directory tree
69
70         # create the directory name as described above (example 'a0=6.00')
71         # Use explicit format for x to ensure 2 digits after the comma.
72
73         dir='{0}{1}'.format(prefix,'{0:12.2f}'.format(x).strip())
74
75
76         # input creation branch
77
78         # if the directory does not yet exist, create it
79         if not os.path.exists(dir):
80             os.mkdir(dir)
81             print( 'directory '+dir+' created')
82         else:
83             print( 'directory {0} exists already'.format(dir))
84
85         # change into the directory of paramter $xx
86         os.chdir(dir)
87
88
89         # do the fedit magic
90         fed=fedit.Fedit()
91         fed.resetPipeInput(recreate=True) # important
92         fed.symmetry(compound="Fe, a0-variation",
93                     spacegroup=229,
94                     type='cry',
95                     units='bohr',
96                     latcon=[x,x,x],
97                     angles=['90.']*3,
98                     atoms=[['fe',[0,0,0]]],
99                 )
100         fed.bzintegration([16,16,16])
101         fed.vxc(version='5') # gga
102         fed.relativistic('scalar')
103         if bas=='DT+f':
104             fed.basis(extensionlevel=2,addf=True,add3d=True)
105         fed.spin(spin=2,initialspinsplit=1,initialspin=[[1,2.5]])
106         fed.pipeFedit()
107
108         # do we run the job?
109         if runit:
110
111             print( fplo+" running in "+dir+" ...")
112
113             # now execute, whatever is nessecary to launch job in the current
114             # directory (dir)
115
116             # START Example
117             # We just run the jobs sequentially on a single machine

```

(continues on next page)

(continued from previous page)

```

118         # and redirect stdout to file 'out' and stderr to /dev/null.
119         # (In this way there will be no dangling output and the job could
120         # run savely in the background, which is not done in our example
121         # here.)
122
123         # Furthermore, we use the +yes-file mechanism to avoid a crash
124         # due to repeated inital polarization (spin split).
125         # The "y" below enforces fplo to continue in such situation
126         # without a repeated split and does nothing otherwise. See manual.
127         with open('+yes', 'w') as f:
128             f.write('y')
129
130         os.system('cat +yes | {0} 2>/dev/null > out'.format(fplo))
131         # END Example
132
133         # just in case
134         os.chdir(ROOT)
135         os.chdir(bas)
136         # and of x-loop
137         # and of bas-loop
138
139     os.chdir(ROOT)
140
141     #
142     # After the run we should have a directory structure like
143     #
144     # ./a0=4.40/
145     # ./a0=4.80/
146     # ./a0=5.20/
147     # ./a0=5.60/
148     # ./a0=6.00/
149     # ./simple.py
150     #
151     # where every directory contains the same setup, except for the
152     # lattice constant.
153     # We may now perform converged calculations (option -r) in all directories.
154     # If we want to change, say, the number of k-points, we edit this number
155     # in the pipe-section above, re-run that script to change the input and
156     # re-converge the calculations (option -r).
157     #
158
159
160     # =====
161     #
162     # =====
163
164     def collect(bases):
165
166         # collect the results
167         ROOT=os.getcwd()
168         for bas in bases:
169             os.chdir(ROOT)
170             os.chdir(bas)
171             os.system("grepfplo -p 'a0=' -m EE | tee e")
172             os.system("grepfplo -p 'a0=' -m SS | tee s")
173
174         os.chdir(ROOT)
175         os.system("xftp pic.xpy")
176
177
178     # =====

```

(continues on next page)

(continued from previous page)

```

179 #
180 # =====
181 if __name__ == "__main__":
182
183     # Set an FPLO version, you need to set this according to your
184     # needs, including possibly a path. Or you use option -p.
185     # A guess for the default name:
186     FPLO=fedit.fploExecutable()
187
188     # scan command line options
189     usage = "usage: %prog [-c] [-r] [-h] [-p fploexecname]"
190     parser = OptionParser(usage)
191     parser.add_option('-r', '', action='store_true', dest='run', default=False,
192                      help='force fplo run')
193     parser.add_option('-c', '', action='store_true', dest='collect', default=False,
194                      help='collect results')
195     parser.add_option('-p', '', type='str', dest='fplo', default=FPLO,
196                      help='optional: the name of an FPLO executable\n'+
197                          'possibly with explicit path')
198     (options, args) = parser.parse_args()
199
200
201     bases=['def', 'DT+f']
202     # do the work
203     if not options.collect or options.run:
204         work(options.fplo,bases,options.run)
205
206     if options.collect:
207         collect(bases)

```

### 3.5.7 mBJ XC-potential

This example prepares **fplo** input and optionally runs the calculations to show the influence of an extended basis on the gap-results for the mBJ XC-potential. The tutorial files are in `FPLO.../DOC/pyfplo/Examples/fploio/fedit_use/mBJ` where `FPLO...` stands for your version's FPLO directory, e.g. `FPLO21.00-61`. Here are the files of this directory:

- [README](#) (page 148)
- [wrapp.sh](#) (page 149)
- [mbj.py](#) (page 149)

README

```

1 This example illustrates the use of fedit, mBJ-xc-potential,
2 an extended basis and pyfplo.fploio.OutGrep. It wi
3
4 to only create the input run as
5
6     mbj.py
7
8 run the calculations serially ( you need to modify to run on batch systems)
9
10     mbj.py -r
11
12 if done collect results like so
13
14     mbj.py -c
15
16

```

(continues on next page)

(continued from previous page)

```

17
18 If pyfplo path needs to be set use wrapp.sh. First edit pyfplopath
19 in wrapp.sh and then just put it in front as in any of the following.
20
21 wrapp.sh mbj.py -r
22
23 wrapp.sh mbj.py -c
24
25

```

#### A wrapper to setup paths wrapp.sh

```

1  #! /usr/bin/env sh
2  #
3  # Example wrapper script for path setting.
4  #
5  #####
6
7
8
9  # set your path here
10
11 pyfplopath=$HOME/FPLO/FPLO22.00-62/PYTHON/
12
13 export PYTHONPATH=$pyfplopath:$PYTHONPATH
14
15
16 $*

```

#### The python script mbj.py

```

1  #! /usr/bin/env python3
2  # =====
3  # file:   mbj.py
4  # author: k.koepernik@ifw-dresden.de
5  # date:   30 Jun 2022
6
7  from __future__ import print_function
8  import sys,os
9  import numpy as np
10 import argparse
11 import pyfplo.fedit as fedit
12 import pyfplo.fploio as fploio
13
14 ROOT=os.getcwd()
15 FPLO=fedit.fploExecutable()
16 # =====
17 #
18 # =====
19 def mkdir(dir):
20     if not os.path.exists(dir): os.mkdir(dir)
21     return dir
22
23 # =====
24 #
25 # =====
26 def work(basis,xcfunctionals,cases,options):
27
28     os.chdir(ROOT)
29
30

```

(continues on next page)

(continued from previous page)

```

31     for case in cases:
32         comp=case['compound']
33         a0=case['a0']
34         b0=case['b0'] if 'b0' in case else a0
35         c0=case['c0'] if 'c0' in case else a0
36         setting=case['setting'] if 'setting' in case else None
37         rel=case['rel'] if 'rel' in case else 'S'
38         spin=case['spin'] if 'spin' in case else 1
39         initialspin=(case['initialspin'] if 'initialspin' in case
40                     else None)
41
42
43     os.chdir(ROOT)
44     dir=mkdir(dir='{comp}'.format(comp=comp))
45     for bas in basis:
46         dir=mkdir('{comp}/{bas}'.format(comp=comp,bas=bas))
47         for xc in xcfunctionals:
48             dir=mkdir('{comp}/{bas}/{xc}'
49                     .format(comp=comp,bas=bas,xc=xc[1]))
50             os.chdir(dir)
51
52         if(not options.collect):
53             fed=fedit.Fedit(recreate=True)
54             fed.symmetry(spacegroup=case['spgr'],
55                         setting=setting,
56                         latcon=[a0,b0,c0],
57                         units='ang',
58                         atoms=case['atoms'])
59             fed.spin(spin=spin,initialspinsplit=True,
60                    initialspin=initialspin,
61                    fsm=[True if spin==2 else False,0])
62             fed.vxc(version=xc[0])
63             fed.relativistic(mode=rel)
64             if bas=='DT+f':
65                 fed.basis(extensionlevel=2,
66                          addf=True,add3d=True)
67             fed.pipeFedit()
68
69         if(options.run):
70             print('running {} in {}'.format(FPLO,dir))
71             with open('+yes','w') as fh:
72                 fh.write('y\ny\ny\n')
73             os.system('cat +yes | {} > out '.format(FPLO))
74
75         if(options.collect):
76             og=fploio.OutGrep('out')
77             try:
78                 gap=float(og.grep('gap')[-1])
79             except:
80                 gap=-1e6
81             print('{comp:<12s} {bas:<10s} {xc:<12s} '+
82                  '{gap:10.3f} eV   last dev={it}')
83                 .format(comp=comp,bas=bas,xc=xc[1],gap=gap,
84                        it=og.grep('it')[-1])
85             os.chdir(ROOT)
86
87     return
88
89 # =====
90 #
91 # =====
92 def options():
93     parser = argparse.ArgumentParser(description='',

```

(continues on next page)

(continued from previous page)

```

92             conflict_handler='resolve',
93             epilog="")
94 parser.add_argument('-c', '--collect', dest='collect',
95                   action='store_true',
96                   help='', default=False)
97 parser.add_argument('-r', '--run', dest='run', action='store_true',
98                   help='', default=False)
99 args = parser.parse_args()
100 return args
101 # =====
102 #
103 # =====
104
105 if __name__ == '__main__':
106     args=options()
107
108     work(basis=['def', 'DT+f'],
109         xcffunctionals=[['4', 'PW92'], ['9', 'mBJLDAc']],
110         cases=[{'compound': 'C', 'spgr': '227', 'a0': 3.567,
111               'atoms': [['C', ['1/8', '1/8', '1/8']]], 'rel': 'S'},
112               {'compound': 'CaF2', 'spgr': '225', 'a0': 5.462,
113               'atoms': [['Ca', ['0', '0', '0']],
114                       ['F', ['1/4', '1/4', '1/4']]], 'rel': 'S'},
115               {'compound': 'AlAs', 'spgr': '216', 'a0': 5.6620,
116               'atoms': [['Al', ['0', '0', '0']],
117                       ['As', ['1/4', '1/4', '1/4']]], 'rel': 'F'},
118               {'compound': 'HfS2', 'spgr': '164',
119               'a0': 3.631, 'c0': 5.841,
120               'atoms': [['Hf', ['0', '0', '0']],
121                       ['S', ['1/3', '2/3', '1/4']]], 'rel': 'F'},
122               {'compound': 'NiO', 'spgr': '166', 'setting': 'E',
123               'a0': np.sqrt(0.5)*4.176, 'c0': np.sqrt(12.)*4.176,
124               'atoms': [['Ni', ['0', '0', '0']],
125                       ['Ni', ['0', '0', '1/2']],
126                       ['O', ['0', '0', '1/4']]],
127               'spin': 2, 'initialspin': [[1, 4], [2, -4], [3, 0]]},
128         ],
129         ,options=args
130     )
131     sys.exit(0)
132

```

## 3.6 FPLOIO examples

- *Reading =.in files* (page 152)
- *=.files to json* (page 153)
- *Reading cif-files* (page 155)
- *Write =.in with low level routines* (page 158)
- *Write =.in with mid level routines* (page 159)
- *Extract default basis into =.basdef* (page 161)
- *User defined basis (in =.basdef)* (page 163)
- *Extract =.basdef from output file* (page 168)

- [Grep results](#) (page 170)

### 3.6.1 Reading `=.in` files

This tutorial shows how to use *INParser* (page 16) and *PObj* (page 17) to read `=.in`-files.

The tutorial files are in `FPLO.../DOC/pyfplo/Examples/fploio/fploio/readinfile` where `FPLO...` stands for your version's `FPLO` directory, e.g. `FPLO21.00-61`. Here are the files of this directory:

- [readinfile.py](#) (page 152)
- `=.in`

`readinfile.py`

```

1  #!/usr/bin/env python
2  # =====
3  # file:   bandplot.py
4  # author: k.koepernik@ifw-dresden.de
5  # date:   19 Apr 2017
6  from __future__ import print_function
7  import sys
8  import pyfplo.fploio as fploio
9  # =====
10 #
11 # =====
12 def work():
13
14
15     print( '\nThis example shows how to parse =.in files.\n')
16
17     p=fploio.INParser()
18     p.parseFile('=.in')
19     d=p()
20
21     print( 'sorts:',d('nsort').L)
22     print( 'lattice constants:',d('lattice_constants').listD)
23     print( 'or\nlattice constants:',d('lattice_constants').listS)
24     print( '\nWyckoff positions')
25     dw=d('wyckoff_positions')
26     for i in range(dw.size()):
27         dd=dw[i]
28         print( dd('element').S,dd('tau').listS)
29
30
31     print( '\nAlternative way of doing it')
32     for i in range(dw.size()):
33         dd=dw[i]
34         print( dd('element').S,dd('tau')[0].S,dd('tau')[1].S,dd('tau')[2].S)
35
36
37     print( '\nYet another way')
38     for i in range(dw.size()):
39         dd=dw[i]
40         print( dd('element').S,dd('tau[0]').S,dd('tau[1]').S,dd('tau[2]').S)
41
42     print( '\nsingle shot')
43     print( d('wyckoff_positions[0].element').S,
44               d('wyckoff_positions[0].tau').listS)
45
46     print( '\noptions')

```

(continues on next page)

(continued from previous page)

```

47 do=d('options')
48 for i in range(do.size()):
49     print( do[i].S,)
50     if ((i+1)%4)==0: print()
51 print()
52 print( '\nsetting CALC_PLASMON_FREQ and CALC_DOS')
53 do['CALC_PLASMON_FREQ'].L=True
54 do['CALC_DOS'].L=True
55 print( '\nwhich now are',bool(do['CALC_PLASMON_FREQ'].L)
56     , 'and',bool(do['CALC_DOS'].L))
57 print( '\nhave another look at options')
58 do=d('options')
59 for i in range(do.size()):
60     print( do[i].S,)
61     if (i%4)==0: print()
62 print()
63
64
65
66 return
67
68 # =====
69 #
70 # =====
71
72 if __name__ == '__main__':
73
74     work()
75
76     sys.exit(0)
77
78
79
80

```

### 3.6.2 =.files to json

This tutorial shows how to convert files with the =.in syntax into json-files. It demonstrates the usage of *PObj* (page 17).

The tutorial files are in `FPLO.../DOC/pyfplo/Examples/fploio/equaldot2json` where `FPLO...` stands for your version's `FPLO` directory, e.g. `FPLO21.00-61`. Here are the files of this directory:

- *equaldot2json.py* (page 153)
- =.in

`equaldot2json.py`

```

1  #! /usr/bin/env python
2  # =====
3  # file:   testscan.py
4  # author: k.koepernik@ifw-dresden.de
5  # date:   04 Okt 2018
6
7  from __future__ import print_function
8  import pyfplo.fploio as fploio
9  import argparse
10 import json
11
12 version = 20181002

```

(continues on next page)

(continued from previous page)

```

13
14 parser = argparse.ArgumentParser(description='Translates `=*.*` files \
15                                     (e.g., `=.in`, `=.xef`, `=.dens`) into \
16                                     a JSON file without calling pyfplo. \
17                                     Works with Python 2, but the order \
18                                     of entries is retained in Python 3, \
19                                     only.', conflict_handler='resolve',
20                                     epilog="Please report any bugs to \
21                                     Oleg Janson <olegjanson@gmail.com>.")
22 parser.add_argument('-i', '--input', default='=.in',
23                     type=str, dest='input',
24                     help='FPLO input file (default:%(default)s)')
25 parser.add_argument('-o', '--output', type=argparse.FileType('w'),
26                     default='-', help='JSON output file')
27 parser.add_argument('-n', '--nice', dest='nice', action='store_true',
28                     help='add indents (otherwise just a single line)')
29 parser.set_defaults(nice=False)
30 parser.add_argument('-v', '--version', action='version', \
31                     help='print the version', \
32                     version='% (prog) s version {:d}'.format(version))
33 args = parser.parse_args()
34 jsonprintoptions = ( {}, {'indent':4} ) [ args.nice ]
35
36 # =====
37 #
38 # =====
39 def scanStruct(d, dictdata, ind=''):
40     n=d.first()
41     while True:
42         if n.isScalar():
43             if n.isInt():
44                 dictdata[n.name()]=n.L
45             elif n.isReal():
46                 dictdata[n.name()]=n.D
47             elif n.isLogical():
48                 dictdata[n.name()]=bool(n.L)
49             elif n.isFlag():
50                 dictdata[n.name()]=bool(n.L)
51             else:
52                 dictdata[n.name()]=n.S
53         elif n.isArray():
54             dictdata[n.name()]=[]
55             scanArray(n, dictdata[n.name()], n.sizes(), len(n.sizes()))
56         elif n.isStruct():
57             dictdata[n.name()]={}
58             scanStruct(n, dictdata[n.name()], ind+'    ')
59         elif n.isStructArray():
60             dictdata[n.name()]=[]
61             for i in range(n.size()):
62                 dictdata[n.name()].append({})
63                 scanStruct(n[i], dictdata[n.name()][-1], ind+'    ')
64
65         if not n.hasNext(): break
66         n=n.next()
67
68     return
69 # =====
70 #
71 # =====
72
73 def scanArray(d, dictdata, sizes, dim, idx=[], ind=''):

```

(continues on next page)

(continued from previous page)

```

74
75     if dim==len(sizes):
76         idx=[0]*dim
77     for i in range(sizes[dim-1]):
78         idx[dim-1]=i
79         if dim==1:
80             n=d[tuple(idx)]
81             if n.isInt():
82                 dictdata.append(n.L)
83             elif n.isReal():
84                 dictdata.append(n.D)
85             elif n.isLogical():
86                 dictdata.append(bool(n.L))
87             elif n.isFlag():
88                 dictdata.append({n.S[0:-3]:bool(n.L)})
89             else:
90                 dictdata.append(n.S)
91         else:
92             dictdata.append([])
93             scanArray(d,dictdata[-1],sizes,dim-1,idx,ind='')
94
95     return
96
97 # =====
98 #
99 # =====
100
101 if __name__ == '__main__':
102
103     p=fploio.INParser()
104     p.parseFile(args.input)
105
106     dictdata={}
107     scanStruct(p(),dictdata)
108
109
110     with args.output as jsonfile:
111         json.dump(dictdata, jsonfile, **jsonprintoptions)
112
113
114
115
116
117

```

### 3.6.3 Reading cif-files

This tutorial demonstrates how to read cif-files. See [structureFromCIFFile](#) (page 21). The tutorial files are in `FPLO.../DOC/pyfplo/Examples/fploio/cif` where `FPLO...` stands for your version's FPLO directory, e.g. `FPLO21.00-61`. Here are the files of this directory:

- [README](#) (page 155)
- [fromcif.py](#) (page 156)
- `RuW.cif`

README

```

1 This demonstrates the reading of cif files and some of the possible options.
2

```

(continues on next page)

(continued from previous page)

```

3 Execute:
4
5 python fromcif.py raw
6
7 which loads the cif as it is (which happens to have spacegroup 1).
8 Have a look at the output, especially the Wyckoff positions.
9
10
11
12 Next, execute:
13
14 python fromcif.py smoothed
15
16 and have a look at the output, especially the Wyckoff positions,
17 which are now fractionals.
18
19
20
21 Next, execute:
22
23 python fromcif.py raw detsym
24
25 and have a look at the output: we got space group 63 and
26 2 Ru positions, due to the approximate fractional 0.1666 and such.
27
28
29 Finally, execute:
30
31 python fromcif.py smoothed detsym
32
33 Now we have one Ru and W position with fractionals and spacegroup 194

```

fromcif.py

```

1  #!/usr/bin/env python
2  # =====
3  # file:    fromcif.py
4  # author:  k.koepernik@ifw-dresden.de
5  # date:    22 Mar 2018
6  from __future__ import print_function
7  import sys
8  import pyfplo.fploio as fploio
9  import pyfplo.fedit as fedit
10
11 # =====
12 #
13 # =====
14 def work(mode='raw', detsym=False):
15
16
17     print( '\nThis example shows how to import cif files.')
18
19     whtol=1e-4 if mode=='smoothed' else 1e-6
20
21     # Create FPLOInput object and read =.in or create new parser
22     # content if =.in does not exist.
23     fio=fploio.FPLOInput('=.in')
24     # read cif file into parser
25     fio.structureFromCIFFile('RuW.cif', wyckofftolerance=whtol,
26                               determinesymmetry=detsym)
27     # write =.in

```

(continues on next page)

(continued from previous page)

```

28     fio.writeFile("=.in")
29
30     # here we have =.in with the structure from the cif file
31
32
33     # use fedit to set further input
34     fed=fedit.Fedit()
35     fed.iteration(n=100)
36     fed.bzintegration(nxyz=[12,12,6])
37     fed.pipeFedit()
38
39
40     print ('\n\nNow we have read the raw cif content as is,'+
41           ' which results in \n'+
42           'the following symmetry settings:\n\n')
43
44     printsettings()
45
46
47     return
48
49     # =====
50     #
51     # =====
52
53     def printsettings():
54         fio=fploio.FPLOInput('=.in')
55         par=fio.parser()
56         d=par()
57         print( 'spacegroup number : ',d('spacegroup.number').S)
58         print( 'spacegroup setting: ',d('spacegroup.setting').S)
59         print( 'lattice constants : ',d('lattice_constants').listS)
60         print( 'axis angle          : ',d('axis_angles').listS)
61         dw=d('wyckoff_positions')
62         print( 'Wyckoff positions: ',dw.size())
63         for i in range(dw.size()):
64             taus=dw[i]('tau').listS
65             print( '{0:>2s} {1:>20s} {2:>20s} {3:>20s}'
66                   .format(dw[i]('element').S,taus[0],taus[1],taus[2]))
67
68     # =====
69     #
70     # =====
71
72     def usage():
73         print( 'usage: ',sys.argv[0],' ( raw | smoothed) [detsym]')
74     # =====
75     #
76     # =====
77
78     if __name__ == '__main__':
79
80
81
82
83
84         if len(sys.argv)<2:
85             usage()
86             sys.exit(0)
87         mode=sys.argv[1]
88         mode=mode.replace('-','')

```

(continues on next page)

(continued from previous page)

```

89
90     if not (mode=='raw' or mode=='smoothed'):
91         usage()
92         sys.exit(0)
93
94     detsym=False
95     if len(sys.argv)>=3:
96         detsym=True
97
98
99     work(mode,detsym)
100
101     sys.exit(0)
102
103
104
105

```

### 3.6.4 Write =.in with low level routines

This tutorial shows how to use *INParser* (page 16) and *PObj* (page 17) to modify =.in-files. This should not be done for symmetry input!

The tutorial files are in `FPLO.../DOC/pyfplo/Examples/fploio/fploio/writinfilelowlevel` where `FPLO...` stands for your version's FPLO directory, e.g. `FPLO21.00-61`. Here are the files of this directory:

- *writinfilelowlevel.py* (page 158)
- =.in

writinfilelowlevel.py

```

1  #!/usr/bin/env python
2  # =====
3  # file:   bandplot.py
4  # author: k.koepernik@ifw-dresden.de
5  # date:   19 Apr 2017
6  from __future__ import print_function
7  import sys
8  import pyfplo.fploio as fploio
9  # =====
10 #
11 # =====
12 def work():
13
14
15     print( '\nThis example shows simple low level =.in writing.')
16     print( 'DO NOT DO THIS UNLESS YOU ARE CONFIDENT YOUR DOING THE '+
17           'RIGHT THING.\n')
18     p=fploio.INParser()
19     p.parseFile('=.in')
20     d=p()
21     d('bzone_integration.nkxyz').listL=[16,16,16]
22     d('spin.mspin').L=2
23     d('relativistic.type').L=3
24     # This illustrates one of the problems with the low level.
25     # In order to keep =.in consistent for human readers
26     # we should set the descriptions too.
27     # It is better to use pyfplo.fedit instead.
28     d('relativistic.description').S='full relativistic'

```

(continues on next page)

(continued from previous page)

```

29     p.writeFile('=.in')
30
31     print( 'done')
32
33
34     return
35
36     # =====
37     #
38     # =====
39
40 if __name__ == '__main__':
41
42     work()
43
44     sys.exit(0)
45
46
47
48

```

### 3.6.5 Write =.in with mid level routines

This tutorial shows how to use *FPLOInput* (page 20) to modify =.in-files. This should not be done for symmetry input!

The tutorial files are in `FPLO.../DOC/pyfplo/Examples/fploio/fploio/writinfilemidlevel` where `FPLO...` stands for your version's `FPLO` directory, e.g. `FPLO21.00-61`. Here are the files of this directory:

- *writinfilemidlevel.py* (page 159)
- =.in

writinfilemidlevel.py

```

1  #!/usr/bin/env python
2  # =====
3  # file:   bandplot.py
4  # author: k.koepernik@ifw-dresden.de
5  # date:   19 Apr 2017
6  from __future__ import print_function
7  import sys,os
8  import pyfplo.fploio as fploio
9  # =====
10 #
11 # =====
12 def work():
13
14
15     print( '\nThis fictitious example shows simple mid level =.in writing.')
16     print( 'DO NOT DO THIS UNLESS YOU ARE CONFIDENT YOUR DOING THE '+
17           'RIGHT THING.\n')
18
19     fio=fploio.FPLOInput()
20     if os.path.exists('=.in'):
21         fio.parseInFile(True)
22     else:
23         fio.createNewFileContent()
24
25     p=fio.parser()

```

(continues on next page)

```

26     d=p() # the root of the data tree
27
28
29     # set length units
30     d("lengthunit.type").L=2
31     # not really needed but good for human readers of =.in
32     d("lengthunit.description").S='angstroem'
33
34
35     # set wyckoff positions
36     d("nsort").L=3
37     # yes, we need to do this too.
38     dw=d("wyckoff_positions")
39     dw.resize(3)
40
41     i=0 ;
42     di=dw[i]
43     di('element').S='Fe'
44     di('tau').listD=[0,0,0]
45
46     i+=1 ; di=dw[i]
47     di('element').S='Al'
48     di('tau').listS=['1/2','1/2','1/2']
49
50     i+=1 ; di=dw[i]
51     di('element').S='Mn'
52     di('tau').listS=['1/4','1/4','1/4']
53
54     d('lattice_constants').listD=[5.4,5.4,5.4]
55     #or
56     d('lattice_constants').listS=['5.4','5.4','5.4']
57
58     # symmetry update required
59     msg=fio.symmetryUpdate();
60     print( msg)
61
62     # reset all other input, to get a default file
63     fio.resetNonSymmetrySections()
64
65     # here we can set other things
66     d('spin.mspin').L=2
67
68     fio.writeFile("=.in")
69
70     print( 'done')
71
72
73     return
74
75     # =====
76     #
77     # =====
78
79     if __name__ == '__main__':
80
81         work()
82
83         sys.exit(0)
84
85
86

```

### 3.6.6 Extract default basis into `=.basdef`

This example extracts the default basis. The tutorial files are in `FPLO.../DOC/pyfplo/Examples/fploio/basis/extract_default_basdef` where `FPLO...` stands for your version's `FPLO` directory, e.g. `FPLO21.00-61`. Here are the files of this directory:

- `README` (page 161)
- `wrapp.sh` (page 161)
- `extractdefaultbasis.py` (page 161)

README

```

1 Read the script to understand what it is doing.
2 It demonstrates how to extract the default basis from an existing =.in
3 into =.basdef
4 for manual basis manipulation, which could be done by a script (demonstrated_
  ↳elsewhere).
5
6
7 run extractdefaultbasis.py as
8
9   extractdefaultbasis.py
10
11 then have a look at =.basdef.
12
13
14 If pyfplo path needs to be set use wrapp.sh. First edit pyfplopath
15 in wrapp.sh and then just put it in front as in any of the following.
16
17 wrapp.sh extractdefaultbasis.py

```

A wrapper to setup paths `wrapp.sh`

```

1 #!/usr/bin/env sh
2 #
3 # Example wrapper script for path setting.
4 #
5 #####
6
7
8
9 # set your path here
10
11 pyfplopath=$HOME/FPLO/FPLO22.00-62/PYTHON/
12
13 export PYTHONPATH=$pyfplopath:$PYTHONPATH
14
15
16 $*

```

The python script `extractdefaultbasis.py`

```

1 #!/usr/bin/env python
2 # =====
3 # file:   basis.py
4 # author: k.koepernik@ifw-dresden.de
5 # date:   06 Jun 2017

```

(continues on next page)

(continued from previous page)

```

6 from __future__ import print_function
7 import sys
8 import os
9 import pyfplo.fedit as fedit
10 import pyfplo.fploio as fploio
11
12 print( '\npyfplo version=: {0}\nfrom: {1}\n'.format(fedit.version,fedit.__file__)
13 # protect against wrong version
14 #if fedit.version!='22.00': raise RuntimeError('pyfplo version is incorrect.')
15
16 # =====
17 #
18 # =====
19 def work_short_version(prot=True):
20
21
22     if not os.path.exists('=.in'):
23         print('ERROR: this script assumes the existence of an =.in-file.')
24         sys.exit(1)
25
26
27     p=fploio.INParser()
28     p.parseFile('=.in')
29     d=p() ('wyckoff_positions')
30     elements=[d[i] ('element') .S for i in range(d.size())]
31     bastype=p() ('basis.version.type') .L
32     b=fploio.Basis(bastype,elements)
33     b.writeFile()
34
35     os.system('ls -ltr ; echo ; cat =.basdef')
36
37     return
38
39 # =====
40 #
41 # =====
42 def work_long_version(prot=True):
43
44     if not os.path.exists('=.in'):
45         print('ERROR: this script assumes the existence of an =.in-file.')
46         sys.exit(1)
47
48
49     # 1: Get info for hand-made basis creation
50
51     # Read =.in to get basisversion (optional) and
52     # element (optionally: atomic number) list
53     p=fploio.INParser()
54     p.parseFile('=.in')
55     d=p() ('wyckoff_positions')
56     elements=[d[i] ('element') .S for i in range(d.size())]
57     atomicnumbers=list(map(lambda x: fploio.c_elements.index(x),elements))
58     if prot:
59         print('\n=.in contains Wyckoff positions with\n\telements      '
60               +'\n\tatomic numbers  }\n'
61               .format(elements,atomicnumbers))
62     basversion=(p() ('basis.version.type') .L,p() ('basis.version.description') .S)
63
64
65     print('basis version in =.in: key={0[0]} name="{0[1]}"\n'.format(basversion))
66     print('available versions:\n',fploio.Basis.versions,'\n')

```

(continues on next page)

(continued from previous page)

```

67
68     # 2: Get default basis, compatible with =.in-content.
69     #     We could use basversion[0], or basversion[1] as argument
70     #     to fploio.Basis or (in later fplo versions) another basis ID.
71     b=fploio.Basis('default FPLO9 basis',elements)
72     # equivalent:
73     # b=fploio.Basis(1,elements)
74     # or like this:
75     # b=fploio.Basis('default FPLO9 basis',atomicnumbers)
76
77     # 3: write basdef file (=basdef):
78     b.writeFile()
79     #or
80     #b.writeFile('=.basdef')
81
82     # Now we have the default basis in '=.basdef' which would be used by fplo
83     # on running. We could modify it by hand too.
84
85     os.system('ls -ltr ; echo ; cat =.basdef')
86
87     return
88
89     # =====
90     #
91     # =====
92
93     if __name__ == '__main__':
94
95         print('-'*72,'\nlong version\n','-'*72)
96         work_long_version()
97
98         print('-'*72,'\nshort version\n','-'*72)
99         work_short_version()
100
101         sys.exit(0)
102
103
104
105

```

### 3.6.7 User defined basis (in =.basdef)

This example extracts the default basis, modifies it and creates =.basdef. The tutorial files are in FPLO.../DOC/pyfplo/Examples/fploio/basis/modify\_basdef where FPLO... stands for your version's FPLO directory, e.g. FPLO21.00-61. Here are the files of this directory:

- [README](#) (page 163)
- [wrapp.sh](#) (page 164)
- [basis.py](#) (page 164)

README

```

1 Read the script to understand what it is doing.
2 It demonstrates how to extract the default basis into =.basdef and how to
3 modify it to achieve non-standard basis settings.
4 In fact it mostly replicates the fedit basis modification options using
5 low-level manipulation. This serves as a starting point for user-made
6 basis modifications.
7

```

(continues on next page)

(continued from previous page)

```

8
9 run basis.py as
10
11     basis.py
12
13
14
15
16 If pyfplo path needs to be set use wrapp.sh. First edit pyfplopath
17 in wrapp.sh and then just put it in front as in any of the following.
18
19 wrapp.sh basis.py

```

A wrapper to setup paths wrapp.sh

```

1  #! /usr/bin/env sh
2  #
3  # Example wrapper script for path setting.
4  #
5  #####
6
7
8
9  # set your path here
10
11 pyfplopath=$HOME/FPLO/FPLO22.00-62/PYTHON/
12
13 export PYTHONPATH=$pyfplopath:$PYTHONPATH
14
15
16 $*

```

The python script basis.py

```

1  #! /usr/bin/env python
2  # =====
3  # file:    basis.py
4  # author:  k.koepernik@ifw-dresden.de
5  # date:    06 Jun 2017
6  from __future__ import print_function
7  import sys
8  import os
9  import pyfplo.fedit as fedit
10 import pyfplo.fploio as fploio
11
12 print( '\npyfplo version=: {0}\nfrom: {1}\n'.format(fedit.version,fedit.__file__) )
13 # protect against wrong version
14 #if fedit.version!='22.00': raise RuntimeError('pyfplo version is incorrect.')
15
16 FPLO=fploio.fploExecutable()
17
18 # =====
19 #
20 # =====
21 def makeINFile(extensionlevel=1,core4f=None,core4fNoValenceF=None,
22               addf=False,add3d=False,multicore=None,):
23     ''' make a (hypothetical compound) =.in-file '''
24     fed=fedit.Fedit(recreate=True)
25     fed.symmetry(spacegroup=123,latcon=[8,8,8],atoms=
26                 [
27                     ['fe',[0,0,0]],

```

(continues on next page)

(continued from previous page)

```

28         ['eu', ['1/2', '1/2', '1/2']],
29         ['H', ['1/2', '0', '0']],
30     ])
31     fed.basis(extensionlevel=extensionlevel, core4f=core4f,
32               core4fNoValenceF=core4fNoValenceF, addf=addf, add3d=add3d,
33               multicore=multicore)
34     # write input file
35     fed.pipeFedit()
36
37     return
38 # =====
39 #
40 # =====
41
42 def getBasisIngredient(prot=False):
43     '''
44     Read .in to get basisversion (optional) and
45     element (optinal: atomic number) list
46     '''
47     p=fploio.INParser()
48     p.parseFile('=.in')
49     d=p()['wyckoff_positions']
50     elements=[d[i]['element']].S for i in range(d.size())
51     atomicnumbers=list(map(lambda x: fploio.c_elements.index(x), elements))
52     if prot:
53         print('\n=.in contains Wyckoff positions with\n\telements      '
54               +'\n\tatomic numbers  {}\n'
55               .format(elements, atomicnumbers))
56     basversion=(p()['basis.version.type']).L, p()['basis.version.description'].S
57     return (basversion, elements, atomicnumbers)
58
59 # =====
60 #
61 # =====
62
63 def writeBasdef(basdeffile='=.basdef', extensionlevel=1, core4f=[],
64                core4fNoValenceF=[], addf=False, add3d=False,
65                multicore=None,
66                makesingle=False, doubleSemiCoreS=False):
67
68     (basversion, elements, atomicnumbers)=getBasisIngredient()
69     b=fploio.Basis('default FPL09 basis', elements)
70
71     # modify
72
73     for bd in b:
74         for l in range(1, extensionlevel):
75             for o in bd.valence:
76                 mu=o.multiplicity
77                 o.append(Q=o.Q(mu-1)+2, P=max(min(o.P(mu-1), 1.), 0.85))
78
79     if add3d:
80         for bd in b:
81             haved=any([o.name[1]=='d' for o in bd.core])
82             haved=haved or any([o.name[1]=='d' for o in bd.semicore])
83             haved=haved or any([o.name[1]=='d' for o in bd.valence])
84             if not haved:
85                 bd.valence.append('3d', Q=[5], P=[1])
86
87     # add S*f (D*f if uncommented)
88

```

(continues on next page)

(continued from previous page)

```

89     if addf:
90         for bd in b:
91             nmain=3
92             for o in bd.core:
93                 if o.name[1]=='f':
94                     nmain=max(nmain,int(o.name[0]))
95             for o in bd.semicore:
96                 if o.name[1]=='f':
97                     nmain=max(nmain,int(o.name[0]))
98             for o in bd.valence:
99                 if o.name[1]=='f':
100                     nmain=max(nmain,int(o.name[0])+o.multiplicity-1)
101
102             if nmain<4:
103                 bd.valence.append('{nm}f'.format(nm=nmain+1),Q=[5],P=[1])
104                 #bd.valence.append('{nm}f'.format(nm=nmain+1),Q=[5,7],P=[1,1])
105
106             # move 4f to core, leave remaining f-valence
107             for c in core4f:
108                 if isinstance(c,int):
109                     isort=c
110                 else:
111                     isort=elements.index(c.capitalize()+1)
112             bd=b[isort-1] # isort is one-based as in FPLO
113             for o in bd.valence:
114                 if o.name=='4f':
115                     o.removeFirst()
116                     bd.core.append('4f')
117
118             # move 4f to core, no f-valence
119             for c in core4fNoValenceF:
120                 if isinstance(c,int):
121                     isort=c
122                 else:
123                     isort=elements.index(c.capitalize()+1)
124             bd=b[isort-1] # isort is one-based as in FPLO
125             for i,o in enumerate(bd.valence):
126                 if o.name=='4f':
127                     bd.valence.remove(i)
128                     bd.core.append('4f')
129                 break
130
131             # make all valence MultiOrbitals single orbitals
132             # This option does not exist in fedit.
133             if makesingle:
134                 for bd in b:
135                     for o in bd.valence:
136                         while o.multiplicity>1: o.removeLast()
137
138             # make only semicore s-orbitals double
139             # This option does not exist in fedit.
140             if doubleSemiCoreS:
141                 for bd in b:
142                     for o in bd.semicore:
143                         mu=o.multiplicity
144                         if o.name[1]=='s':
145                             o.append(Q=0,S=5,P=max(min(o.P(mu-1),1.),0.85))
146                             #o.set(0,Q=0,S=-5)
147
148             # make double core
149             if (multicore is not None) and len(multicore)>0:

```

(continues on next page)

(continued from previous page)

```

150     for bd in b:
151         for o in bd.core:
152             o.set(Q=multicore[0][0],S=multicore[0][1])
153             for m in range(1, len(multicore)):
154                 o.append(Q=multicore[m][0],S=multicore[m][1])
155
156
157
158     b.writeFile(basdeffile)
159
160     return
161 # =====
162 #
163 # =====
164 def extractBasDefFromOut (outfile='out',basdeffile='=.basdef'):
165     with open(outfile,'r') as fh:
166         lines=fh.readlines()
167
168     with open(basdeffile,'w') as fh:
169         start=False
170         for line in lines:
171             if line.startswith('Start: content of =.basdef'):
172                 start=True
173                 continue
174             if line.startswith('End : content of =.basdef'):
175                 break
176             if start:
177                 if line.startswith('---'): continue
178                 fh.write(line)
179
180     return
181 # =====
182 #
183 # =====
184 def work():
185
186
187
188     # -----
189     # With basis level2 (from fedit):
190     makeINFile(extensionlevel=2,addf=True,add3d=True,
191               core4fNoValenceF=[2], # sort of Eu
192               multicore=[[0,0],[0,10]])
193     # Here we are all set ... including level2 basis defined in =.in.
194     # -----
195
196     # -----
197     # For later diff-checking, run fplo to extract the basis as defined
198     # in =.in from the outfile section :
199     #     Start: content of =.basdef
200     #     ...
201     #     End : content of =.basdef
202     #
203
204
205     fed=fedit.Fedit(recreate=False)
206     fed.iteration(n=1) # single step, since we run for =.basdef extraction
207     fed.pipeFedit()
208
209     # we want =.in to determine the basis, so delete =.basdef
210     os.system('rm -f =.basdef')

```

(continues on next page)

(continued from previous page)

```

211 print('running fplo to extract which basis was used, wait a few secs...')
212 os.system('{} > out'.format(FPLO))
213
214 extractBasDefFromOut('out', basdef_file='.basdef_as_defined_by_in_file')
215 # -----
216
217
218
219
220 # -----
221 # Extract default =.basdef and modify it
222 writeBasDef(basdef_file='.basdef', extension_level=2,
223             addf=True, add3d=True, core4fNoValenceF=[2],
224             multicore=[[0,0],[0,10]],
225             makesingle=False, doubleSemiCoreS=False)
226 # which now should have the same basis modifications as defined in =.in.
227 # Compare the two
228 print('Executing: diff =.basdef =.basdef_as_defined_by_in_file ":") # same
229 os.system('diff =.basdef =.basdef_as_defined_by_in_file') # same
230 print('Nothing should be shown from the diff!')
231 # -----
232
233
234
235 return
236
237 # =====
238 #
239 # =====
240
241 if __name__ == '__main__':
242
243     work()
244
245     sys.exit(0)
246
247
248
249

```

### 3.6.8 Extract =.basdef from output file

This example extracts =.basdef from an *fplo* output file. The tutorial files are in `FPLO.../DOC/pyfplo/Examples/fploio/basis/extract_basdef_from_outfile` where `FPLO...` stands for your version's `FPLO` directory, e.g. `FPLO21.00-61`. Here are the files of this directory:

- [README](#) (page 168)
- [wrapp.sh](#) (page 169)
- [extractbasdeffromout.py](#) (page 169)

README

```

1 Read the script to understand what it is doing.
2 It extracts =.basdef from an FPLO output file. This =.basdef
3 will contain the basis actually used during the calculation
4 (as opposed to the default =.basdef). Once extracted, it
5 will be used henceforth during calculations (and can be modified).
6
7

```

(continues on next page)

(continued from previous page)

```

8 run extractbasdeffromout.py as
9
10     extractbasdeffromout.py
11
12 then have a look at =.basdef.
13
14
15 If pyfplo path needs to be set use wrapp.sh. First edit pyfplopath
16 in wrapp.sh and then just put it in front as in any of the following.
17
18 wrapp.sh extractbasdeffromout.py

```

A wrapper to setup paths wrapp.sh

```

1 #! /usr/bin/env sh
2 #
3 # Example wrapper script for path setting.
4 #
5 #####
6
7
8
9 # set your path here
10
11 pyfplopath=$HOME/FPLO/FPLO22.00-62/PYTHON/
12
13 export PYTHONPATH=$pyfplopath:$PYTHONPATH
14
15
16 $*

```

The python script extractbasdeffromout.py

```

1 #! /usr/bin/env python3
2 # =====
3 # file:     extractbasdeffromout.py
4 # author:  k.koepernik@ifw-dresden.de
5 # date:    24 Jun 2022
6
7 from __future__ import print_function
8 import sys
9 import numpy as np
10
11 # =====
12 #
13 # =====
14 def extractBasDefFromOut(outfile='out',basdeffile='=.basdef'):
15
16     with open(outfile,'r') as fh:
17         lines=fh.readlines()
18
19     with open(basdeffile,'w') as fh:
20         start=False
21         for line in lines:
22             if line.startswith('Start: content of =.basdef'):
23                 start=True
24                 continue
25             if line.startswith('End : content of =.basdef'):
26                 break
27             if start:
28                 if line.startswith('---'): continue

```

(continues on next page)

```

29         fh.write(line)
30     return
31
32     # =====
33     #
34     # =====
35
36 if __name__ == '__main__':
37
38     extractBasDefFromOut (outfile='out',basdeffile='=.basdef')
39
40     sys.exit(0)
41
42
43
44

```

### 3.6.9 Grep results

These examples extract all (or the latest) occurrences via *OutGrep* (page 22) of some results in the outfile.

The tutorial files are in `FPLO.../DOC/pyfplo/Examples/fploio/grep/` where `FPLO...` stands for your version's `FPLO` directory, e.g. `FPLO21.00-61`. Here are the files of this directory:

- *README* (page 170)
- *wrapp.sh* (page 171)
- *grepintodict.py* (page 171)
- *grepEtot.py* (page 172)
- *grepiterationprogress.py* (page 172)

#### README

```

1 Demonstration of how to grep from FPLO output from within python scripts.
2 Read the scripts to understand what they are doing.
3
4
5 run scripts as
6
7     grepintodict.py
8
9 or
10
11     grepEtot.py
12
13 or
14
15     grepiterationprogress.py
16
17
18
19 If pyfplo path needs to be set use wrapp.sh. First edit pyfplopath
20 in wrapp.sh and then just put it in front as in any of the following.
21
22 wrapp.sh grepintodict.py
23
24 wrapp.sh grepEtot.py
25
26 wrapp.sh grepiterationprogress.py

```

A wrapper to setup paths `wrapp.sh`

```

1  #! /usr/bin/env sh
2  #
3  # Example wrapper script for path setting.
4  #
5  #####
6
7
8
9  # set your path here
10
11 pyfplopath=$HOME/FPLO/FPLO22.00-62/PYTHON/
12
13 export PYTHONPATH=$pyfplopath:$PYTHONPATH
14
15
16 $*
```

The python script `grepintodict.py`

```

1  #! /usr/bin/env python3
2  # =====
3  # file:   g.py
4  # author: k.koepernik@ifw-dresden.de
5  # date:   22 Jun 2022
6
7  from __future__ import print_function
8  import sys
9  import numpy as np
10 import pyfplo.fploio as fploio
11
12 print( '\npyfplo version=: {0}\nfrom: {1}\n'.format(fploio.version,fploio.__file__
13 ↪) )
14 # protect against wrong version
15 #if fploio.version!='22.00': raise RuntimeError('pyfplo version is incorrect.')
16
17 # =====
18 #
19 # =====
20 def work(printmodes=True):
21
22     if printmodes: # print grep modes
23         for k in fploio.OutGrep.modes.keys():
24             print('mode: {0:15s} : {1}'.format(k,fploio.OutGrep.modes[k]))
25             print('-'*72,'\n')
26
27     # this will read the file
28     og=fploio.OutGrep('out')
29     si=og.sites()
30
31     # grep some results from last appearance in file (last iteration)
32     results={}
33     results['etot'] = float(og.grep('EE')[-1])
34     results['total spin']= float(og.grep('SS')[-1])
35
36     for i,s in enumerate(si):
37         site=i+1
38         results['spin {0}{1:<3d}'.format(s.element,site)]\
39             =float(og.grep('SSat',site)[-1])
40
41
```

(continues on next page)

(continued from previous page)

```

42     for k in results.keys():
43         print('{0:<20s} {1:>20.10f}'.format(k, results[k]))
44
45
46     if printmodes: print('\n', '-'*72)
47
48     return
49
50     # =====
51     #
52     # =====
53
54 if __name__ == '__main__':
55
56     work()
57
58     sys.exit(0)
59

```

The python script `grepEtot.py`

```

1  #!/usr/bin/env python3
2  # =====
3  # file:    g.py
4  # author:  k.koepernik@ifw-dresden.de
5  # date:    22 Jun 2022
6
7  from __future__ import print_function
8  import pyfplo.fploio as fploio
9
10 print( '\npyfplo version=: {0}\nfrom: {1}\n'.format(fploio.version, fploio.__file__
11 ↪))
12 # protect against wrong version
13 #if fploio.version!='22.00': raise RuntimeError('pyfplo version is incorrect.')
14
15 og=fploio.OutGrep('out')
16 print('etot=', og.grep('EE')[-1], ', gap=', og.grep('gap')[-1])

```

The python script `grepiterationprogress.py`

```

1  #!/usr/bin/env python3
2  # =====
3  # file:    g.py
4  # author:  k.koepernik@ifw-dresden.de
5  # date:    22 Jun 2022
6
7  from __future__ import print_function
8  import os
9  import pyfplo.fploio as fploio
10
11 print( '\npyfplo version=: {0}\nfrom: {1}\n'.format(fploio.version, fploio.__file__
12 ↪))
13 # protect against wrong version
14 #if fploio.version!='22.00': raise RuntimeError('pyfplo version is incorrect.')
15
16 og=fploio.OutGrep('out')
17 si=og.sites()
18
19 with open('res', 'w') as fh:
20     res=og.grep('it')

```

(continues on next page)

(continued from previous page)

```
20 fh.write('# step last_dev\n')
21 for i,r in enumerate(res):
22     fh.write('{} {}\n'.format(i,r))
23 fh.write('\n')
24
25 res=og.grep('EE')
26 fh.write('# step Etot\n')
27 for i,r in enumerate(res):
28     fh.write('{} {}\n'.format(i,r))
29 fh.write('\n')
30
31 res=og.grep('gap')
32 E0=float(res[0])
33 fh.write('# step gap\n')
34 for i,r in enumerate(res):
35     fh.write('{} {}\n'.format(i,r))
36 fh.write('\n')
37
38 with open('resspins','w') as fh:
39
40     for i,s in enumerate(si):
41         site=i+1
42         res=og.grep('SSat',site)
43         fh.write('# step \'atom spin {}{}\n'.format(s.element,site))
44         for it,r in enumerate(res):
45             fh.write('{} {}\n'.format(it,r))
46         fh.write('\n')
47 os.system('xftp grepiterationprogress.xpy')
```



## BIBLIOGRAPHY

- [Yu2011] Rui Yu et.al. Phys. Rev. B **84**, 075119 (2011). <https://doi.org/10.1103/PhysRevB.84.075119>
- [Sol2011] A. A. Soluyanov, D. Vanderbilt Phys. Rev. B **83**, 235401 (2011)  
<https://doi.org/10.1103/PhysRevB.83.235401>
- [Wang15] Zhijun Wang et.a. arxiv:1511.07440 (2015)
- [Fukui05] [Takahiro Fukui et.al. J. Phys. Soc. Jpn. 74, 1674 (2005)]



## PYTHON MODULE INDEX

### p

`pyfplo.common`, 27

`pyfplo.fedit`, 3

`pyfplo.fploio`, 15

`pyfplo.slabify`, 41

`pyfplo.wanniertools`, 68



## Symbols

\_\_call\_\_() (*INParser method*), 16  
 \_\_call\_\_() (*PObj method*), 17  
 \_\_eq\_\_() (*Version method*), 39  
 \_\_getitem\_\_() (*BasDefSection method*), 26  
 \_\_getitem\_\_() (*Basis method*), 25  
 \_\_getitem\_\_() (*BerryCurvatureData method*), 67  
 \_\_getitem\_\_() (*OptionSet method*), 36  
 \_\_getitem\_\_() (*PObj method*), 18  
 \_\_len\_\_() (*BasDefSection method*), 25  
 \_\_ne\_\_() (*Version method*), 39  
 \_\_setitem\_\_() (*OptionSet method*), 36  
 \_\_str\_\_() (*BandHeader method*), 32  
 \_\_str\_\_() (*BasDef method*), 25  
 \_\_str\_\_() (*BasDefSection method*), 26  
 \_\_str\_\_() (*BfieldConfig method*), 65  
 \_\_str\_\_() (*BoxMesh method*), 58  
 \_\_str\_\_() (*EnergyContour method*), 59  
 \_\_str\_\_() (*FermiSurfaceOptions method*), 62  
 \_\_str\_\_() (*GreenOptions method*), 63  
 \_\_str\_\_() (*MultiOrbital method*), 27  
 \_\_str\_\_() (*OptionSet method*), 36  
 \_\_str\_\_() (*PObj method*), 19  
 \_\_str\_\_() (*Site method*), 37  
 \_\_str\_\_() (*Version method*), 39  
 \_\_str\_\_() (*WFSymOp method*), 66  
 \_\_str\_\_() (*WeightDefinition method*), 36  
 \_\_str\_\_() (*WeightDefinitions method*), 35  
 \_\_str\_\_() (*WeylPoint method*), 64

## A

absToRel() (*BoxMesh method*), 57  
 active (*BandPlot attribute*), 31  
 active (*FermiSurfaceOptions attribute*), 62  
 add() (*WanDefCreator method*), 70  
 add() (*WeightDefinitions method*), 34  
 addAtoms() (*WeightDefinition method*), 35  
 addContrib() (*WanDef method*), 73  
 addLabels() (*WeightDefinition method*), 35  
 addToPipeInput() (*Fedit method*), 5  
 addWeights() (*BandWeights method*), 33  
 All (*Vlevel attribute*), 40  
 alpha (*WFSymOp attribute*), 66  
 anchor (*Slabify attribute*), 55  
 append() (*BasDefSection method*), 25  
 append() (*MultiOrbital method*), 26

axis1 (*WeylPoint attribute*), 64  
 axis2 (*WeylPoint attribute*), 64  
 axis3 (*WeylPoint attribute*), 64

## B

BandFileContext (*class in pyfplo.common*), 27  
 BandFileContext (*in module pyfplo.slabify*), 68  
 BandHeader (*class in pyfplo.common*), 32  
 BandHeader (*in module pyfplo.slabify*), 68  
 BandPlot (*class in pyfplo.common*), 29  
 BandPlot (*in module pyfplo.slabify*), 67  
 bandplot() (*Fedit method*), 8  
 BandWeights (*class in pyfplo.common*), 33  
 BandWeights (*in module pyfplo.slabify*), 67  
 BasDef (*class in pyfplo.fploio*), 25  
 BasDefSection (*class in pyfplo.fploio*), 25  
 Basis (*class in pyfplo.fploio*), 24  
 basis() (*Fedit method*), 13  
 berryCurvature() (*Slabify method*), 52  
 BerryCurvatureData (*class in pyfplo.slabify*), 67  
 bfield (*Slabify attribute*), 56  
 BfieldConfig (*class in pyfplo.slabify*), 65  
 BoxMesh (*class in pyfplo.slabify*), 56  
 bzintegration() (*Fedit method*), 6

## C

c\_abtoang (*in module pyfplo.common*), 40  
 c\_angstroem\_m (*in module pyfplo.common*), 40  
 c\_echarge\_C (*in module pyfplo.common*), 40  
 c\_elements (*in module pyfplo.common*), 40  
 c\_elements (*in module pyfplo.fploio*), 27  
 c\_elements (*in module pyfplo.slabify*), 67  
 c\_hatoev (*in module pyfplo.common*), 40  
 c\_hbar Js (*in module pyfplo.common*), 40  
 c\_me\_kg (*in module pyfplo.common*), 40  
 c\_speed\_of\_light\_mpers (*in module pyfplo.common*), 40  
 calculate3dTIIInvariants() (*Slabify method*), 49  
 calculateBandPlotMesh() (*BandPlot method*), 29  
 calculateBandStructure() (*Slabify method*), 43  
 calculateBerryCurvatureOnBox() (*Slabify method*), 47

- calculateBulkProjectedEDC() (*Slabify method*), 43
- calculateBulkProjectedFS() (*Slabify method*), 43
- calculateChernNumberInSphere() (*Slabify method*), 47
- calculateEDC() (*Slabify method*), 45
- calculateFermiSurfaceCuts() (*Slabify method*), 44
- calculateFermiSurfaceSpectralDensity() (*Slabify method*), 45
- calculateMirrorChernNumbers() (*Slabify method*), 54
- calculateZ2Invariant() (*Slabify method*), 48
- charges() (*Fedit method*), 7
- chirality (*WeylPoint attribute*), 64
- close() (*BandFileContext method*), 28
- close() (*DensPlotContext method*), 63
- coDiagonalize() (*Slabify method*), 52
- Contrib (*class in pyfplo.wanniertools*), 73
- core (*BasDef attribute*), 25
- coreoccupation() (*Fedit method*), 11
- createNewFileContent() (*FPLOInput method*), 22
- cutatoms (*Slabify attribute*), 55
- cutlayersat (*Slabify attribute*), 55
- ## D
- D (*PObj attribute*), 20
- DensPlotContext (*class in pyfplo.slabify*), 63
- dhva() (*Fedit method*), 12
- dhvaison() (*Fedit method*), 11
- diagonalize() (*Slabify method*), 51
- diagonalizeUnitary() (*Slabify method*), 52
- dirname (*Slabify attribute*), 54
- Dk (*WFSymOp attribute*), 66
- ## E
- e0 (*EnergyContour attribute*), 59
- e1 (*EnergyContour attribute*), 59
- element (*Site attribute*), 37
- energy (*WeylPoint attribute*), 64
- EnergyContour (*class in pyfplo.slabify*), 59
- enlarge (*Slabify attribute*), 55
- equivalentSites (*WFSymOp attribute*), 67
- ewindow (*BandPlot attribute*), 32
- ## F
- Fedit (*class in pyfplo.fedit*), 3
- fermienergy (*FermiSurfaceOptions attribute*), 62
- fermienergyim (*FermiSurfaceOptions attribute*), 62
- FermiSurfaceOptions (*class in pyfplo.slabify*), 60
- findWeylPoints() (*Slabify method*), 53
- finuc() (*Fedit method*), 8
- first() (*PObj method*), 18
- forces() (*Fedit method*), 14
- fploExecutable() (*in module pyfplo.fedit*), 3
- fploExecutable() (*in module pyfplo.fploio*), 15
- FPLOInput (*class in pyfplo.fploio*), 20
- fullName() (*PObj method*), 17
- ## G
- go() (*Watch method*), 38
- GreenOptions (*class in pyfplo.slabify*), 63
- grep() (*OutGrep method*), 23
- gridoutput() (*Fedit method*), 10
- ## H
- hamAtKPoint() (*Slabify method*), 50
- hamdataCCell() (*Slabify method*), 42
- hamdataCell() (*Slabify method*), 42
- hamdataRCell() (*Slabify method*), 42
- hasbasisconnection (*Slabify attribute*), 56
- hasNext() (*PObj method*), 18
- hassigma (*Slabify attribute*), 56
- hasxcfield (*Slabify attribute*), 56
- header() (*BandWeights method*), 33
- homo (*WeylPoint attribute*), 64
- ## I
- ilower (*BandHeader attribute*), 33
- ime (*EnergyContour attribute*), 59
- index (*WFSymOp attribute*), 66
- Info (*Vlevel attribute*), 40
- INParser (*class in pyfplo.fploio*), 16
- isArray() (*PObj method*), 18
- isChar() (*PObj method*), 19
- isFlag() (*PObj method*), 19
- isimproper (*WFSymOp attribute*), 66
- isinlittlegroup (*WFSymOp attribute*), 66
- isInt() (*PObj method*), 19
- isLogical() (*PObj method*), 19
- isReal() (*PObj method*), 19
- isScalar() (*PObj method*), 18
- isString() (*PObj method*), 19
- isStruct() (*PObj method*), 18
- isStructArray() (*PObj method*), 18
- iteration() (*Fedit method*), 14
- iupper (*BandHeader attribute*), 33
- ## K
- k (*WeylPoint attribute*), 64
- kdists (*BandPlot attribute*), 32
- kpnts (*BandPlot attribute*), 32
- kyscale (*Slabify attribute*), 56
- ## L
- L (*PObj attribute*), 19
- labels (*BandHeader attribute*), 33
- layerCell() (*Slabify method*), 42
- layerSites() (*Slabify method*), 42
- listD (*PObj attribute*), 19
- listL (*PObj attribute*), 19

listS (*PObj* attribute), 19  
 lowerdepthdatalimit (*BandPlot* attribute), 32  
 lsda() (*Fedit* method), 9

## M

mainVersion() (*Version* method), 39  
 Many (*Vlevel* attribute), 40  
 mesh() (*BoxMesh* method), 57  
 mesh() (*EnergyContour* method), 59  
 mesh() (*FermiSurfaceOptions* method), 60  
 modes (*OutGrep* attribute), 23  
 More (*Vlevel* attribute), 40  
 MultiOrbital (*class in pyfplo.fploio*), 26  
 MultipleOrbitalWandef (*class in pyfplo.wanniertools*), 71  
 multiplicity (*MultiOrbital* attribute), 27

## N

name (*MultiOrbital* attribute), 27  
 name() (*PObj* method), 18  
 names (*OptionSet* attribute), 36  
 nband (*BandHeader* attribute), 32  
 ndiv (*BandPlot* attribute), 31  
 ne (*EnergyContour* attribute), 59  
 next() (*PObj* method), 18  
 nkp (*BandHeader* attribute), 32  
 norb (*BandHeader* attribute), 33  
 nsigiter (*GreenOptions* attribute), 64  
 nspin (*BandHeader* attribute), 32  
 nspin (*Slabify* attribute), 55  
 numberoflayers (*Slabify* attribute), 55  
 numerics() (*Fedit* method), 14  
 nvdim (*Slabify* attribute), 54  
 nx (*BoxMesh* attribute), 58  
 nx (*FermiSurfaceOptions* attribute), 62  
 ny (*BoxMesh* attribute), 58  
 ny (*FermiSurfaceOptions* attribute), 62  
 nz (*BoxMesh* attribute), 58

## O

object (*Slabify* attribute), 55  
 off() (*BandPlot* method), 31  
 off() (*FermiSurfaceOptions* method), 60  
 on() (*BandPlot* method), 31  
 on() (*FermiSurfaceOptions* method), 60  
 opc() (*Fedit* method), 8  
 openBandFile() (*BandPlot* method), 30  
 openDensPlotFile() (*FermiSurfaceOptions* method), 61  
 optics() (*Fedit* method), 7  
 options (*Slabify* attribute), 55  
 options() (*Fedit* method), 6  
 OptionSet (*class in pyfplo.common*), 36  
 OptionSet (*in module pyfplo.slabify*), 68  
 orbitalIndicesByDepth() (*Slabify* method), 45  
 orbitalIndicesBySite() (*Slabify* method), 46  
 orbitalNames() (*Slabify* method), 45  
 orbitalNamesByDepth() (*Slabify* method), 46

origin (*BoxMesh* attribute), 58  
 origin (*FermiSurfaceOptions* attribute), 62  
 OutGrep (*class in pyfplo.fploio*), 22  
 outputpartoccubands (*BandPlot* attribute), 32

## P

P() (*MultiOrbital* method), 26  
 parseFile() (*INParser* method), 16  
 parseInFile() (*FPLOInput* method), 20  
 parser() (*FPLOInput* method), 22  
 partoccuoffset (*BandPlot* attribute), 32  
 pipeFedit() (*Fedit* method), 5  
 PObj (*class in pyfplo.fploio*), 17  
 points (*BandPlot* attribute), 32  
 prepare() (*Slabify* method), 42  
 printProgress() (*Watch* method), 38  
 printStructureSettings() (*Slabify* method), 43  
 pyfplo.common (*module*), 27  
 pyfplo.fedit (*module*), 3  
 pyfplo.fploio (*module*), 15  
 pyfplo.slabify (*module*), 41  
 pyfplo.wanniertools (*module*), 68

## Q

Q() (*MultiOrbital* method), 26  
 qns() (*MultiOrbital* method), 26

## R

radius (*WeylPoint* attribute), 64  
 readBandPlotMesh() (*BandPlot* method), 30  
 readBands() (*BandPlot* method), 31  
 readBandWeights() (*BandWeights* method), 34  
 relativistic() (*Fedit* method), 6  
 release() (*Version* method), 39  
 relToAbs() (*BoxMesh* method), 57  
 remove() (*BasDefSection* method), 25  
 removeFirst() (*MultiOrbital* method), 26  
 removeLast() (*MultiOrbital* method), 26  
 reset() (*FPLOInput* method), 22  
 reset() (*Watch* method), 38  
 resetNonSymmetrySections() (*FPLOInput* method), 21  
 resetPipeInput() (*Fedit* method), 4  
 resize() (*PObj* method), 17

## S

S (*PObj* attribute), 20  
 S() (*MultiOrbital* method), 26  
 semicore (*BasDef* attribute), 25  
 set() (*MultiOrbital* method), 26  
 setBox() (*BoxMesh* method), 56  
 setGlobalField() (*BfieldConfig* method), 65  
 setLocalFields() (*BfieldConfig* method), 65  
 setMesh() (*BoxMesh* method), 56  
 setMesh() (*EnergyContour* method), 59  
 setMesh() (*FermiSurfaceOptions* method), 60

setOutputRestrictions() (*BandPlot method*), 31  
 setPlane() (*FermiSurfaceOptions method*), 60  
 setProgress() (*Watch method*), 38  
 sigitermethod (*GreenOptions attribute*), 64  
 sigitertol (*GreenOptions attribute*), 64  
 Silent (*Vlevel attribute*), 40  
 SingleOrbitalWandef (class in *pyfplo.wanniertools*), 70  
 Site (class in *pyfplo.common*), 37  
 Site (in module *pyfplo.slabify*), 67  
 sites() (*OutGrep method*), 23  
 size() (*PObj method*), 17  
 sizes() (*PObj method*), 17  
 Slabify (class in *pyfplo.slabify*), 41  
 sort (*Site attribute*), 37  
 spin (*WeylPoint attribute*), 65  
 spin() (*Fedit method*), 6  
 status() (*Watch method*), 38  
 stop() (*Watch method*), 37  
 structureFromCIFFile() (*FPLOInput method*), 21  
 symbol (*WFSymOp attribute*), 66  
 symmetry() (*Fedit method*), 5  
 symmetryUpdate() (*FPLOInput method*), 21

## T

tau (*Site attribute*), 37  
 tau (*WFSymOp attribute*), 66  
 ti() (*Fedit method*), 8  
 timerev (*WFSymOp attribute*), 66  
 type (*Site attribute*), 37

## U

upperdepthdatalimit (*BandPlot attribute*), 32

## V

valence (*BasDef attribute*), 25  
 varExists() (*INParser method*), 16  
 varExists() (*PObj method*), 17  
 verbosity() (*Fedit method*), 8  
 Version (class in *pyfplo.common*), 39  
 version (in module *pyfplo.common*), 40  
 Version (in module *pyfplo.fploio*), 27  
 version (in module *pyfplo.fploio*), 27  
 Version (in module *pyfplo.slabify*), 67  
 version (in module *pyfplo.slabify*), 67  
 versions (*Basis attribute*), 25  
 Vlevel (class in *pyfplo.common*), 39  
 Vlevel (in module *pyfplo.slabify*), 68  
 vxc() (*Fedit method*), 7

## W

Wandef (class in *pyfplo.wanniertools*), 71  
 WanDefCreator (class in *pyfplo.wanniertools*), 69  
 wannierCenterMatrix() (*Slabify method*), 46  
 Watch (class in *pyfplo.common*), 37  
 WeightDefinition (class in *pyfplo.common*), 35

WeightDefinition (in module *pyfplo.slabify*), 67  
 WeightDefinitions (class in *pyfplo.common*), 34  
 WeightDefinitions (in module *pyfplo.slabify*), 67  
 WeylPoint (class in *pyfplo.slabify*), 64  
 WFSymOp (class in *pyfplo.slabify*), 66  
 write() (*BandFileContext method*), 28  
 write() (*DensPlotContext method*), 63  
 writeFile() (*Basis method*), 25  
 writeFile() (*FPLOInput method*), 21  
 writeFile() (*INParser method*), 16  
 writeFile() (*WanDefCreator method*), 70

## X

xaxis (*BoxMesh attribute*), 58  
 xaxis (*FermiSurfaceOptions attribute*), 62  
 xcoptions() (*Fedit method*), 6  
 xinterval (*BoxMesh attribute*), 58  
 xinterval (*FermiSurfaceOptions attribute*), 62  
 xmesh() (*BoxMesh method*), 56  
 xmesh() (*FermiSurfaceOptions method*), 60  
 xyzFromIndex() (*BoxMesh method*), 57

## Y

yaxis (*BoxMesh attribute*), 58  
 yaxis (*FermiSurfaceOptions attribute*), 63  
 yinterval (*BoxMesh attribute*), 58  
 yinterval (*FermiSurfaceOptions attribute*), 62  
 ymesh() (*BoxMesh method*), 56  
 ymesh() (*FermiSurfaceOptions method*), 60

## Z

zaxis (*BoxMesh attribute*), 58  
 zaxis (*Slabify attribute*), 55  
 zinterval (*BoxMesh attribute*), 58  
 zmesh() (*BoxMesh method*), 56