Appendix D

The PythTB Package

Numerous examples and exercises in this book make use of the PythTB software package. This package, which is written in the Python programming language, is designed to allow the user to construct and solve tight-binding (TB) models of the electronic structure of finite clusters and of systems that display periodicity in one or more dimensions, such as polymer chains, ribbons, slabs, and 3D crystals. It is also designed to provide convenient features for computing geometric or topological properties such as Berry phases, Berry curvatures, and Chern numbers. The code was developed beginning in 2010 and is maintained principally by Sinisa Coh and David Vanderbilt, although significant contributions have been provided by others (see `http://www.physics.rutgers.edu/pythtb/about.html#History`).

The examples and exercises used in this book assume the use of version 1.7.2 of PythTB, which is the current version at the time of this writing. PythTB version 1.7.2 is designed for compatibility with both Python2.7 and Python3.x (Python2.6 and below are not recommended). If you already have one of these Python versions and pip installed on a Linux system, you should be able to install the latest version of PythTB just by issuing the command

```
pip install pythtb --upgrade
```

at a command prompt if you have root permission, or

```
pip install pythtb --upgrade --user
```

to install it in your home folder if you do not. If you wish to install the package without pip or otherwise need further help with installation, see the package web pages at `http://www.physics.rutgers.edu/pythtb`. You will also need access to the standard numpy (numerical) and matplotlib (plotting) Python packages; install these too if they are not already present.

https://doi.org/10.1017/9781316662205.012 Published online by Cambridge University Press

The example programs, together with their printed and plotted outputs, are available at `http://www.cambridge.org/9781107157651#resources`. This is referred to as "the website" in the remainder of this appendix. All efforts will be made to keep future upgrades of PythTB backwardcompatible with the example programs provided here, but in case of doubt you can obtain version 1.7.2 specifically by issuing the command

```
pip install pythtb==1.7.2
```

or download and install PythTB 1.7.2 from the "Installation" link on the website.

Each program starts with the header lines

```
#!/usr/bin/env python
from __future__ import print_function # python3 style print
```

The first identifies the script as a PYTHON program. The second is ignored when running under PYTHON3, but is needed for compatibility with PYTHON2 so that `print` is interpreted as a function, as is standard in PYTHON3. These two lines are typically followed by some comment lines (starting with '#') that identify the program.

The next few lines import the PythTB module and any other modules needed for execution of the script. Typical entries include

```
from pythtb import *
import numpy as np
import matplotlib.pyplot as plt
```

The first line imports the PythTB module. All of the example programs also make use of the NUMPY mathematical subroutine library, which is imported on the second line in such a way that NUMPY '`func`' will be called as `np.func`. This line is actually unnecessary because the import is done automatically when PythTB is imported, but it may be included to make the code more readable; this has been done in the first example in Appendix D.1, but not in subsequent ones. Finally, the import of `matplotlib.pyplot` on the third line provides a MATLIB-like plotting environment for those example programs that produce plotted output.[1]

PythTB defines two *classes*: `tb_model` and `wf_array`.[2] A command such as `my_model=tb_model(`*arguments*`)` defines an *instance* of the `tb_model` class, an object that carries all the relevant information about the TB model. The (*arguments*) given initially define some general characteristics of the model (dimensionality, whether it is a spinor model, and so on), but then parameters such as on-site energies and hopping amplitudes are set subsequently by making

[1] The precise appearance of a plot may depend on the installed version of MATPLOTLIB (version 1.5.3 was used here) and on any `matplotlibrc` configuration files that may be present.

[2] It also defines a third, `w90`, which provides an interface to the WANNIER90 code package (Mostofi et al., 2008, 2014). We shall not make use of this class here.

https://doi.org/10.1017/9781316662205.012 Published online by Cambridge University Press

use of the `set_onsite` and `add_hop` methods. For periodic models, the next step is usually to set up a mesh of **k**-points on which the model is to be solved, with the solution typically done by calling the `solve_all` method. Alternatively, an array object can be defined as an instance of the `wf_array` class and solved by the `solve_on_grid` method of this class; this is especially useful when computing Berry phases and curvatures, since the `wf_array` class provides methods specifically for such purposes. Finally, the results are printed or plotted for inspection.

All this is best illustrated by browsing the example programs and their outputs in the sections that follow.

D.1 Water Molecule

Program `h2o.py` computes the eigenstates of a TB model for the water molecule as described in Section 2.2.2. The first few lines define the molecular geometry (bond length and angle) and the TB parameters (*s* and *p* site energies and *s*-*s* and *p*-*p* hoppings). The `lat=` line specifies three basis vectors (here Cartesian), in terms of which the atom coordinates will be given, and the `orb` line specifies those coordinates. The next few lines define the TB model `my_model`, while `my_model.display` prints a summary of it and `my_model.solve_all` solves for the eigenvalues and eigenvectors. Since the model is described by a real Hamiltonian, the imaginary parts of the eigenvectors are zero to numerical accuracy and are discarded. The last few lines print the result; `np.set_printoptions` is a NUMPY function that sets the format for subsequent print statement, and the eigenvalues and eigenvectors are printed at the end.

h2o.py

```
#!/usr/bin/env python
from __future__ import print_function # python3 style print

# ------------------------------------------------------------
# Tight-binding model for H2O molecule
# ------------------------------------------------------------

# import the pythtb module
from pythtb import *
import numpy as np

# geometry: bond length and half bond-angle
b=1.0; angle=54.0*np.pi/180

# site energies [O(s), O(p), H(s)]
eos=-1.5; eop=-1.2; eh=-1.0

# hoppings [O(s)-H(s), O(p)-H(s)]
ts=-0.4; tp=-0.3
```

https://doi.org/10.1017/9781316662205.012 Published online by Cambridge University Press

```
# define frame for defining vectors: 3D Cartesian
lat=[[1.0,0.0,0.0],[0.0,1.0,0.0],[0.0,0.0,1.0]]

# define coordinates of orbitals: O(s,px,py,pz) ; H(s) ; H(s)
orb=[ [0.,0.,0.], [0.,0.,0.], [0.,0.,0.], [0.,0.,0.],
      [b*np.cos(angle), b*np.sin(angle),0.],
      [b*np.cos(angle),-b*np.sin(angle),0.] ]

# define model
my_model=tbmodel(0,3,lat,orb)
my_model.set_onsite([eos,eop,eop,eop,eh,eh])
my_model.set_hop(ts,0,4)
my_model.set_hop(ts,0,5)
my_model.set_hop(tp*np.cos(angle),1,4)
my_model.set_hop(tp*np.cos(angle),1,5)
my_model.set_hop(tp*np.sin(angle),2,4)
my_model.set_hop(-tp*np.sin(angle),2,5)

# print model
my_model.display()

# solve model
(eval,evec)=my_model.solve_all(eig_vectors=True)

# the model is real, so OK to discard imaginary parts of eigenvectors
evec=evec.real

# optional: choose overall sign of evec according to some specified rule
# (here, we make the average oxygen p component positive)
for i in range(len(eval)):
  if sum(evec[i,1:4]) < 0:
    evec[i,:]=-evec[i,:]

# print results, setting numpy to format floats as xx.xxx
np.set_printoptions(formatter={'float': '{: 6.3f}'.format})
# print eigenvalues and real parts of eigenvectors, one to a line
print("  n   eigval   eigvec")
for n in range(6):
    print(" %2i  %7.3f  " % (n,eval[n]), evec[n,:])
```

D.2 Benzene Molecule

Program `benzene.py`, discussed in Section 2.2.2, computes the eigenstates for a TB model of the p_π manifold (i.e., p orbitals oriented normal to the plane of the molecule) for benzene (CH_6). The structure is much like that of `h2o.py` in the preceding section. See also Ex. 2.5.

benzene.py

```
#!/usr/bin/env python
from __future__ import print_function # python3 style print

# ------------------------------------------------------------
# Tight-binding model for p_z states of benzene molecule
# ------------------------------------------------------------

from pythtb import *

# set up molecular geometry
lat=[[1.0,0.0],[0.0,1.0]]        # define coordinate frame: 2D Cartesian
```

https://doi.org/10.1017/9781316662205.012 Published online by Cambridge University Press

```
r=1.2                              # distance of atoms from center
orb=np.zeros((6,2),dtype=float)    # initialize array for orbital positions
for i in range(6):                 # define coordinates of orbitals
  angle=i*np.pi/3.0
  orb[i,:]= [r*np.cos(angle), r*np.sin(angle)]

# set site energy and hopping amplitude, respectively
ep=-0.4
t=-0.25

# define model
my_model=tbmodel(0,2,lat,orb)
my_model.set_onsite([ep,ep,ep,ep,ep,ep])
my_model.set_hop(t,0,1)
my_model.set_hop(t,1,2)
my_model.set_hop(t,2,3)
my_model.set_hop(t,3,4)
my_model.set_hop(t,4,5)
my_model.set_hop(t,5,0)

# print model
my_model.display()

# solve model and print results
(eval,evec)=my_model.solve_all(eig_vectors=True)

# print results, setting numpy to format floats as xx.xxx
np.set_printoptions(formatter={'float': '{: 6.3f}'.format})
# print eigenvalues and real parts of eigenvectors, one to a line
print("  n   eigval   eigvec")
for n in range(6):
    print(" %2i  %7.3f  " % (n,eval[n]), evec[n,:].real)
```

D.3 bcc Li Crystal

Program `li.py` computes the band structure of bcc Li based on a TB model with one *s* orbital per atomic site, as introduced in Section 2.2.4. Now the vectors `lat` have to be a set of primitive real-space lattice vectors that define the periodicity of the crystal, and subsequent coordinates must be given in terms of these. In particular, the neighboring orbitals specified by the calls to `set_hop` are given in these lattice coordinates, with the translation to Cartesian coordinates given in comments at the end of each line. Orbital locations also need to be defined in lattice coordinates, but there is only one atom in the unit cell and we place it at the origin. (The value of the lattice constant has no effect on the results to be computed here, so it is set to unity for convenience.)

The next few lines after `my_model.display()` specify a list of three special **k** points, or "nodes," that define the path along which the band structure will be computed and plotted. The `path` and `label` variables are lists of the node coordinates and their labels, respectively, and the `k_path` method takes `path` as input and constructs a list of **k** points tracing a path along straight-line segments between these nodes. The returned variables `k_vec` and `k_dist` are arrays containing the coordinates of the **k** points and corresponding accumulated distance

https://doi.org/10.1017/9781316662205.012 Published online by Cambridge University Press

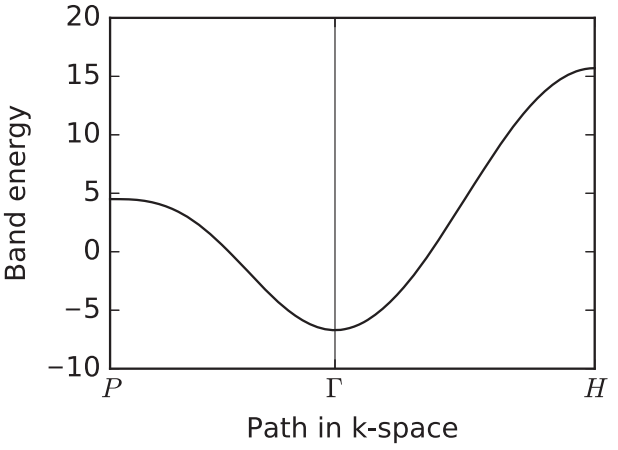

Figure D.1 Band structure of bcc Li as generated by the `li.py` program.

along the path for each point, while `k_node` gives the accumulated distance just for the nodes. These are used in the calls to the PYPLOT plotting routines to generate the band-structure plot which is output as `li_band.pdf`, shown in Fig. D.1. This reproduces the results shown in Fig. 2.4 in Section 2.2.3.

Note that the node positions in `path` must be specified in *reciprocal lattice coordinates*, as linear combinations of the primitive reciprocal lattice vectors that are dual to the real-space ones. (The same convention is used for the output array `k_vec`.) It is the responsibility of the user to specify `path` in this way, but the "k_path report" generated by the `k_path` function is useful for checking that this has been done correctly. For this program, for example, see the lines between `k_path report begin` and `k_path report end` in the printed output on the website.

li.py

```
#!/usr/bin/env python
from __future__ import print_function # python3 style print

# 3D model of Li on bcc lattice, with s orbitals only

from pythtb import * # import TB model class
import matplotlib.pyplot as plt

# define lattice vectors
lat=[[-0.5, 0.5, 0.5],[ 0.5,-0.5, 0.5],[ 0.5, 0.5,-0.5]]
# define coordinates of orbitals
orb=[[0.0,0.0,0.0]]

# make 3D model
my_model=tb_model(3,3,lat,orb)

# set model parameters
# lattice parameter implicitly set to a=1
Es= 4.5    # site energy
t =-1.4    # hopping parameter
```

https://doi.org/10.1017/9781316662205.012 Published online by Cambridge University Press

```
# set on-site energy
my_model.set_onsite([Es])
# set hoppings along four unique bonds
# note that neighboring cell must be specified in lattice coordinates
# (the corresponding Cartesian coords are given for reference)
my_model.set_hop(t, 0, 0, [1,0,0])    # [-0.5, 0.5, 0.5] cartesian
my_model.set_hop(t, 0, 0, [0,1,0])    # [ 0.5,-0.5, 0.5] cartesian
my_model.set_hop(t, 0, 0, [0,0,1])    # [ 0.5, 0.5,-0.5] cartesian
my_model.set_hop(t, 0, 0, [1,1,1])    # [ 0.5, 0.5, 0.5] cartesian

# print tight-binding model
my_model.display()

# generate k-point path and labels
# again, specified in reciprocal lattice coordinates
k_P     = [0.25,0.25,0.25]            # [ 0.5, 0.5, 0.5] cartesian
k_Gamma = [ 0.0, 0.0, 0.0]            # [ 0.0, 0.0, 0.0] cartesian
k_H     = [-0.5, 0.5, 0.5]            # [ 1.0, 0.0, 0.0] cartesian
path=[k_P,k_Gamma,k_H]
label=(r'$P$',r'$\Gamma $',r'$H$')
(k_vec,k_dist,k_node)=my_model.k_path(path,101)

print('---------------------------------------')
print('starting calculation')
print('---------------------------------------')
print('Calculating bands...')

# solve for eigenenergies of Hamiltonian on
# the set of k-points from above
evals=my_model.solve_all(k_vec)

# plotting of band structure
print('Plotting band structure...')

# first make a figure object
fig, ax = plt.subplots(figsize=(4.,3.))

# specify horizontal axis details
ax.set_xlim([0,k_node[-1]])
ax.set_xticks(k_node)
ax.set_xticklabels(label)
for n in range(len(k_node)):
  ax.axvline(x=k_node[n], linewidth=0.5, color='k')

# plot bands
ax.plot(k_dist,evals[0],color='k')
# put title
ax.set_xlabel("Path in k-space")
ax.set_ylabel("Band energy")
# make a PDF figure of a plot
fig.tight_layout()
fig.savefig("li_bsr.pdf")

print('Done.\n')
```

D.4 Alternating Site Model

Program `chain_alt.py` computes the band structure of the 1D alternating site chain model of Fig. 2.6 in Section 2.2.4. The lattice constant is set to unity for convenience. The orbitals at $x = 0$ and $x = 1/2$ are assigned site energies Δ and

https://doi.org/10.1017/9781316662205.012 Published online by Cambridge University Press

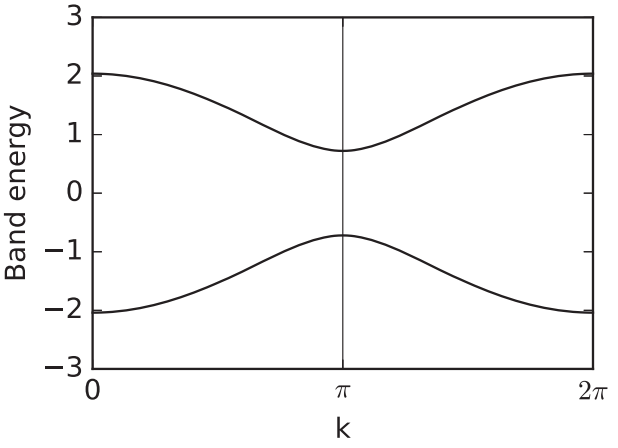

Figure D.2 Band structure resulting from the 1D alternating chain model of Fig. 2.6 as computed by `chain_alt.py`.

$-\Delta$, respectively, and the alternating hopping strengths are $t + \delta t$ and $t - \delta t$. This time a subroutine `set_model` is defined to specify the model; this will be convenient later when looping over parameters of the model. The resulting plot appears in Fig. D.2.

chain_alt.py

```
#!/usr/bin/env python
from __future__ import print_function # python3 style print

# Chain with alternating site energies and hoppings

from pythtb import *
import matplotlib.pyplot as plt

# define function to set up model for a given paramter set
def set_model(t,del_t,Delta):
  # 1D model with two orbitals per cell
  lat=[[1.0]]
  orb=[[0.0],[0.5]]
  my_model=tbmodel(1,1,lat,orb)
  # alternating site energies (let average be zero)
  my_model.set_onsite([Delta,-Delta])
  # alternating hopping strengths
  my_model.add_hop(t+del_t, 0, 1, [0])
  my_model.add_hop(t-del_t, 1, 0, [1])
  return my_model

# set reference hopping strength to unity to set energy scale
t=-1.0
# set alternation strengths
del_t=-0.3   # bond strength alternation
Delta= 0.4   # site energy alternation

# set up the model
my_model=set_model(t,del_t,Delta)
```

https://doi.org/10.1017/9781316662205.012 Published online by Cambridge University Press

```
# construct the k-path
(k_vec,k_dist,k_node)=my_model.k_path('full',121)
k_lab=(r'0',r'$\pi$',r'$2\pi$')

# solve for eigenvalues at each point on the path
evals=my_model.solve_all(k_vec)

# set up the figure and specify details
fig, ax = plt.subplots(figsize=(4.,3.))
ax.set_xlim([0,k_node[-1]])
ax.set_xticks(k_node)
ax.set_xticklabels(k_lab)
ax.axvline(x=k_node[1],linewidth=0.5, color='k')
ax.set_xlabel("k")
ax.set_ylabel("Band energy")

# plot first and second bands
ax.plot(k_dist,evals[0],color='k')
ax.plot(k_dist,evals[1],color='k')

# save figure as a PDF
fig.tight_layout()
fig.savefig("chain_alt.pdf")
```

D.5 Graphene

Program `graphene.py` computes the band structure for the model of the π orbitals of graphene discussed in Section 2.2.4 and illustrated in Fig. 2.7(a). The conventional Wigner–Seitz Brillouin zone (BZ) is shown in Fig. 2.7(b) with high-symmetry points labeled; the `my_model.k_path` line of the program specifies a path from Γ to K to M and back to Γ (see the "k_path report" in the printed output available on the website). The resulting band structure plot appears as Fig. 2.8 in Section 2.2.4.

graphene.py

```
#!/usr/bin/env python
from __future__ import print_function # python3 style print

# Simple model of pi manifold of graphene

from pythtb import * # import TB model class
import matplotlib.pyplot as plt

# define lattice vectors
lat=[[1.0,0.0],[0.5,np.sqrt(3.0)/2.0]]
# define coordinates of orbitals
orb=[[1./3.,1./3.],[2./3.,2./3.]]

# make 2D tight-binding graphene model
my_model=tb_model(2,2,lat,orb)

# set model parameters
delta=0.0
t=-1.0
```

https://doi.org/10.1017/9781316662205.012 Published online by Cambridge University Press

```
my_model.set_onsite([-delta,delta])
my_model.set_hop(t, 0, 1, [ 0, 0])
my_model.set_hop(t, 1, 0, [ 1, 0])
my_model.set_hop(t, 1, 0, [ 0, 1])

# print out model details
my_model.display()

# list of k-point nodes and their labels defining the path for the
#   band structure plot
path=[[0.,0.],[2./3.,1./3.],[.5,.5],[0.,0.]]
label=(r'$\Gamma $',r'$K$', r'$M$', r'$\Gamma $')

# construct the k-path
nk=121
(k_vec,k_dist,k_node)=my_model.k_path(path,nk)

# solve for eigenvalues at each point on the path
evals=my_model.solve_all(k_vec)

# generate band structure plot

fig, ax = plt.subplots(figsize=(4.,3.))
# specify horizontal axis details
ax.set_xlim([0,k_node[-1]])
ax.set_ylim([-3.4,3.4])
ax.set_xticks(k_node)
ax.set_xticklabels(label)
# add vertical lines at node positions
for n in range(len(k_node)):
  ax.axvline(x=k_node[n],linewidth=0.5, color='k')
# put titles
ax.set_xlabel("Path in k-space")
ax.set_ylabel("Band energy")

# plot first and second bands
ax.plot(k_dist,evals[0],color='k')
ax.plot(k_dist,evals[1],color='k')

# save figure as a PDF
fig.tight_layout()
fig.savefig("graphene.pdf")
```

D.6 Trimer Molecule

Program `trimer.py` computes the eigenfunctions of the trimer molecule discussed in Section 3.2.4. In addition, it produces plots of Berry phases and Berry curvatures of the ground state resulting from variation of the parameters.

A `set_model` subroutine has been defined so as to return an instance of the model for a given set of the parameters defined in Eq. (3.43), and another subroutine `get_evecs` calls `set_model` and returns the computed eigenvectors. Predefining these functions simplifies the process of looping over the values of φ and α as needed later. This time one of the hoppings is generally complex, so TR symmetry is broken and the eigenvectors are generally complex as well.

https://doi.org/10.1017/9781316662205.012 Published online by Cambridge University Press

The plotted output of this program appears as Fig. 3.11 in the main text. For pedagogical purposes the Berry phase is computed in several ways. First, for a single given value of α, the program computes the discrete Berry phase explicitly according to Eq. (3.1) as φ runs from 0 to 2π on a dense mesh (increments of $\pi/30$) at a fixed value of α. The product of the inner products appearing in Eq. (3.1) is accumulated in the variable `prod` and the Berry phase is computed explicitly by taking the imaginary part of the log using the NUMPY function `angle`.

Second, on a coarse mesh of α values (in increments of $\pi/6$), the eigenvectors on a dense mesh of φ values are stored into a 1D PYTHTB wave-function array object `evec_array` that is initialized as an instance of the PYTHTB `wf_array` class. The Berry phase is computed by the `berry_phase` method associated with this class, and the results are plotted as the black dots in Fig. 3.11(a). The same procedure is repeated to compute the Berry phases in the α direction on a coarse mesh of φ values and plot them as the dots in Fig. 3.11(b).

Next, a 2D `wf_array` object (also named `evec_array`) is filled with eigenvectors on a fine mesh in both the φ and α directions. The `berry_phase` method is then used to compute the Berry phases first in the φ direction and then in the α direction, and these (together with values shifted by integer multiples of 2π) are plotted as the continuous lines in Fig. 3.11(a–b). By default, the `berry_phase` procedure enforces a smooth evolution of the Berry phase, so we do not see the same kind of discontinuities that appeared when using the previous method.

Finally, the `berry_flux` method is applied to the same 2D `evec_array` array to compute the Berry flux through each of the 60×60 plaquettes, and the results are divided by parametric area to obtain the Berry curvature and used to generate the contour plot in Fig. 3.11(c). The PYPLOT `subplots` method is used to combine the three panels into a single PDF output file.

trimer.py

```
#!/usr/bin/env python
from __future__ import print_function # python3 style print

# Tight-binding model for trimer with magnetic flux

from pythtb import *
import matplotlib.pyplot as plt

# ------------------------------------------------------------
# define function to set up model for given (t0,s,phi,alpha)
# ------------------------------------------------------------
def set_model(t0,s,phi,alpha):

  # coordinate space is 2D
  lat=[[1.0,0.0],[0.0,1.0]]
  # finite model with three orbitals forming a triangle at unit
  # distance from the origin
  sqr32=np.sqrt(3.)/2.
```

https://doi.org/10.1017/9781316662205.012 Published online by Cambridge University Press

```
  orb=np.zeros((3,2),dtype=float)
  orb[0,:]=[0.,1.]          # orbital at top vertex
  orb[1,:]=[-sqr32,-0.5]    # orbital at lower left
  orb[2,:]=[ sqr32,-0.5]    # orbital at lower right

  # compute hoppings [t_01, t_12, t_20]
  # s is distortion amplitude; phi is "pseudorotation angle"
  tpio3=2.0*np.pi/3.0
  t=[ t0+s*np.cos(phi), t0+s*np.cos(phi-tpio3), t0+s*np.cos(phi-2.0*tpio3) ]

  # alpha is fraction of flux quantum passing through the triangle
  # magnetic flux correction, attached to third bond
  t[2]=t[2]*np.exp((1.j)*alpha)

  # set up model (leave site energies at zero)
  my_model=tbmodel(0,2,lat,orb)
  my_model.set_hop(t[0],0,1)
  my_model.set_hop(t[1],1,2)
  my_model.set_hop(t[2],2,0)
  return(my_model)

# ------------------------------------------------------------
# define function to return eigenvectors for given (t0,s,phi,alpha)
# ------------------------------------------------------------
def get_evecs(t0,s,phi,alpha):
  my_model=set_model(t0,s,phi,alpha)
  (eval,evec)=my_model.solve_all(eig_vectors=True)
  return(evec)        # evec[bands,orbitals]

# ------------------------------------------------------------
# begin regular execution
# ------------------------------------------------------------
# for the purposes of this problem we keep t0 and s fixed
t0 =-1.0
s  =-0.4
ref_model=set_model(t0,s,0.,1.)  # reference with phi=alpha=0
ref_model.display()

# define two pi
twopi=2.*np.pi

# ------------------------------------------------------------
# compute Berry phase for phi loop explicitly at alpha=pi/3
# ------------------------------------------------------------
alpha=np.pi/3.
n_phi=60
psi=np.zeros((n_phi,3),dtype=complex)    # initialize wavefunction array
for i in range(n_phi):
  phi=float(i)*twopi/float(n_phi)        # 60 equal intervals
  psi[i]=get_evecs(t0,s,phi,alpha)[0]    # psi[i] is short for psi[i,:]
prod=1.+0.j                              # final [0] picks out band 0
for i in range(1,n_phi):
  prod=prod*np.vdot(psi[i-1],psi[i]) # <psi_0|psi_1>...<psi_58|psi_59>
prod=prod*np.vdot(psi[-1],psi[0])    # include <psi_59|psi_0>
berry=-np.angle(prod)                # compute Berry phase
print("Explicitly computed phi Berry phase at alpha=pi/3 is %6.3f"% berry)

# ------------------------------------------------------------
# compute Berry phases for phi loops for several alpha values
# using pythtb wf_array() method
# ------------------------------------------------------------
```

https://doi.org/10.1017/9781316662205.012 Published online by Cambridge University Press

```
alphas=np.linspace(0.,twopi,13)   # 0 to 2pi in 12 increments
berry_phi=np.zeros_like(alphas)      # same shape and type array (empty)
print("\nBerry phases for phi loops versus alpha")
for j,alpha in enumerate(alphas):

  # let phi range from 0 to 2pi in equally spaced steps
  n_phi=61
  phit=np.linspace(0.,twopi,n_phi)

  # set up empty wavefunction array object using pythtb wf_array()
  # creates 1D array of length [n_phi], with hidden [nbands,norbs]
  evec_array=wf_array(ref_model,[n_phi])

  # run over values of phi and fill the array
  for k,phi in enumerate(phit[0:-1]):    # skip last point of loop
    evec_array[k]=get_evecs(t0,s,phi,alpha)
  evec_array[-1]=evec_array[0]   # copy first point to last point of loop

  # now compute and store the Berry phase
  berry_phi[j]=evec_array.berry_phase([0])   # [0] specifices lowest band
  print("%3d %7.3f %7.3f"% (j, alpha, berry_phi[j]))

# ------------------------------------------------------------
# compute Berry phases for alpha loops for several phi values
# using pythtb wf_array() method
# ------------------------------------------------------------
phis=np.linspace(0.,twopi,13)   # 0 to 2pi in 12 increments
berry_alpha=np.zeros_like(phis)
print("\nBerry phases for alpha loops versus phi")
for j,phi in enumerate(phis):
  n_alpha=61
  alphat=np.linspace(0.,twopi,n_alpha)
  evec_array=wf_array(ref_model,[n_alpha])
  for k,alpha in enumerate(alphat[0:-1]):
    evec_array[k]=get_evecs(t0,s,phi,alpha)
  evec_array[-1]=evec_array[0]
  berry_alpha[j]=evec_array.berry_phase([0])
  print("%3d %7.3f %7.3f"% (j, phi, berry_alpha[j]))

# ------------------------------------------------------------
# now illustrate use of wf_array() to set up 2D array
# recompute Berry phases and compute Berry curvature
# ------------------------------------------------------------
n_phi=61
n_alp=61
n_cells=(n_phi-1)*(n_alp-1)
phi=np.linspace(0.,twopi,n_phi)
alp=np.linspace(0.,twopi,n_alp)
evec_array=wf_array(ref_model,[n_phi,n_alp])  # empty 2d wavefunction array
for i in range(n_phi):
  for j in range(n_alp):
    evec_array[i,j]=get_evecs(t0,s,phi[i],alp[j])
evec_array.impose_loop(0)   # copy first to last points in each dimension
evec_array.impose_loop(1)

bp_of_alp=evec_array.berry_phase([0],0)  # compute phi Berry phases vs. alpha
bp_of_phi=evec_array.berry_phase([0],1)  # compute alpha Berry phases vs. phi

# compute 2D array of Berry fluxes for band 0
flux=evec_array.berry_flux([0])
print("\nFlux = %7.3f = 2pi * %7.3f"% (flux, flux/twopi))
```

https://doi.org/10.1017/9781316662205.012 Published online by Cambridge University Press

```
curvature=evec_array.berry_flux([0],individual_phases=True)*float(n_cells)

# ------------------------------------------------------------
# plots
# ------------------------------------------------------------
fig,ax=plt.subplots(1,3,figsize=(10,4),gridspec_kw={'width_ratios':[1,1,2]})
(ax0,ax1,ax2)=ax

ax0.set_xlim(0.,1.)
ax0.set_ylim(-6.5,6.5)
ax0.set_xlabel(r"$\alpha/2\pi$")
ax0.set_ylabel(r"Berry phase $\phi(\alpha)$ for $\varphi$ loops")
ax0.set_title("Berry phase")
for shift in (-twopi,0.,twopi):
  ax0.plot(alp/twopi,bp_of_alp+shift,color='k')
ax0.scatter(alphas/twopi,berry_phi,color='k')

ax1.set_xlim(0.,1.)
ax1.set_ylim(-6.5,6.5)
ax1.set_xlabel(r"$\varphi/2\pi$")
ax1.set_ylabel(r"Berry phase $\phi(\varphi)$ for $\alpha$ loops")
ax1.set_title("Berry phase")
for shift in (-twopi,0.,twopi):
  ax1.plot(phi/twopi,bp_of_phi+shift,color='k')
ax1.scatter(phis/twopi,berry_alpha,color='k')

X=alp[0:-1]/twopi + 0.5/float(n_alp-1)
Y=phi[0:-1]/twopi + 0.5/float(n_phi-1)
cs=ax2.contour(X,Y,curvature,colors='k')
ax2.clabel(cs, inline=1, fontsize=10)
ax2.set_title("Berry curvature")
ax2.set_xlabel(r"$\alpha/2\pi$")
ax2.set_xlim(0.,1.)
ax2.set_ylim(0.,1.)
ax2.set_ylabel(r"$\varphi/2\pi$")

fig.tight_layout()
fig.savefig("trimer.pdf")
```

D.7 Berry Phase of Alternating Site Chain

Program `chain_alt_bp.py` returns to the same alternating chain model presented in Appendix D.4 and computes the Berry phase of the lower band as the Bloch wavevector k is cycled around the BZ, as presented in Section 3.4. The Berry phase is again calculated first explicitly and then again using the `wf_array` method. In the explicit calculation, one has to be sure to account for the extra phase factor in Eq. (3.74) as discussed on p. 108. The `wf_array` method, by contrast, takes care of this automatically.

chain_alt_bp.py

```
#!/usr/bin/env python
from __future__ import print_function # python3 style print

# Chain with alternating site energies and hoppings

from pythtb import *
import matplotlib.pyplot as plt
```

https://doi.org/10.1017/9781316662205.012 Published online by Cambridge University Press

```
# define function to set up model for a given parameter set
def set_model(t,del_t,Delta):
  lat=[[1.0]]
  orb=[[0.0],[0.5]]
  my_model=tbmodel(1,1,lat,orb)
  my_model.set_onsite([Delta,-Delta])
  my_model.add_hop(t+del_t, 0, 1, [0])
  my_model.add_hop(t-del_t, 1, 0, [1])
  return my_model

# set parameters of model
t=-1.0       # average hopping
del_t=-0.3   # bond strength alternation
Delta= 0.4   # site energy alternation
my_model=set_model(t,del_t,Delta)
my_model.display()

# ----------------------------------
# explicit calculation of Berry phase
# ----------------------------------

# set up and solve the model on a discretized k mesh
nk=61          # 60 equal intervals around the unit circle
(k_vec,k_dist,k_node)=my_model.k_path('full',nk,report=False)
(eval,evec)=my_model.solve_all(k_vec,eig_vectors=True)
evec=evec[0]   # pick band=0 from evec[band,kpoint,orbital]
               # now just evec[kpoint,orbital]

# k-points 0 and 60 refer to the same point on the unit circle
# so we will work only with evec[0],...,evec[59]

# compute Berry phase of lowest band
prod=1.+0.j
for i in range(1,nk-1):            # <evec_0|evec_1>...<evec_58|evec_59>
  prod*=np.vdot(evec[i-1],evec[i]) # a*=b means a=a*b

# now compute the phase factors needed for last inner product
orb=np.array([0.0,0.5])            # relative coordinates of orbitals
phase=np.exp((-2.j)*np.pi*orb)     # construct phase factors
evec_last=phase*evec[0]            # evec[60] constructed from evec[0]
prod*=np.vdot(evec[-2],evec_last)  # include <evec_59|evec_last>

print("Berry phase is %7.3f"% (-np.angle(prod)))

# ----------------------------------
# Berry phase via the wf_array method
# ----------------------------------

evec_array=wf_array(my_model,[61])       # set array dimension
evec_array.solve_on_grid([0.])           # fill with eigensolutions
berry_phase=evec_array.berry_phase([0])  # Berry phase of bottom band

print("Berry phase is %7.3f"% berry_phase)
```

D.8 Infinite and Finite Three-Site Chain

Program `chain_3_site.py` solves a 1D TB model with three sites per unit cell and computes the Wannier function centers in the multiband context as discussed in Section 3.6.3. The nearest-neighbor hoppings are all equal, but the site energies are

https://doi.org/10.1017/9781316662205.012 Published online by Cambridge University Press

modulated according to Eq. (3.132) in such a way that the evolution of parameter λ from 0 to 2π represents a sliding charge-density wave, as illustrated in Fig. 1.4 in Chapter 1.

The Wannier centers are first computed for the periodic bulk via calculations of multiband Berry phases, first for the lowest-energy band, then for the next lowest, then by computing the centers of the maximally localized Wannier functions (MLWFs) from the multiband Berry phases for the two lowest bands taken as a group (this requires specifying `berry_evals=True` in the call to `berry_phase`).

Next, the finite-system MLWF centers are computed for a finite chain of 10 unit cells cut from the infinite bulk model using the `cut_piece` method of the PYTHTB `tb_model` class. Again this is done for the lowest band alone (more precisely, the 10 lowest eigenstates), the second band (next 10 eigenstates), and the joint group including both (20 lowest states). The MLWF centers are obtained by diagonalizing the 10×10 or 20×20 X position matrix of Eq. (3.129) using the `position_hwf` method of the PYTHTB package.

The resulting printout is reproduced on p. 134. The format is designed to facilitate comparison between the two sets of results. In each case it is evident that while the Wannier centers deviate from the bulk positions near the ends of the chain, they converge rapidly to those bulk values as one goes deeper into bulk-like region of the chain.

chain_3_site.py

```
#!/usr/bin/env python
from __future__ import print_function # python3 style print

# Chain with three sites per cell

from pythtb import *
import matplotlib.pyplot as plt

# define function to construct model
def set_model(t,delta,lmbd):
    lat=[[1.0]]
    orb=[[0.0],[1.0/3.0],[2.0/3.0]]
    model=tb_model(1,1,lat,orb)
    model.set_hop(t, 0, 1, [0])
    model.set_hop(t, 1, 2, [0])
    model.set_hop(t, 2, 0, [1])
    onsite_0=delta*(-1.0)*np.cos(2.0*np.pi*(lmbd-0.0/3.0))
    onsite_1=delta*(-1.0)*np.cos(2.0*np.pi*(lmbd-1.0/3.0))
    onsite_2=delta*(-1.0)*np.cos(2.0*np.pi*(lmbd-2.0/3.0))
    model.set_onsite([onsite_0,onsite_1,onsite_2])
    return(model)

# construct the model
t=-1.3
delta=2.0
lmbd=0.3
my_model=set_model(t,delta,lmbd)
```

https://doi.org/10.1017/9781316662205.012 Published online by Cambridge University Press

```
# compute the results on a uniform k-point grid
evec_array=wf_array(my_model,[21])       # set array dimension
evec_array.solve_on_grid([0.])           # fill with eigensolutions

# obtain Berry phases and convert to Wannier center positions
#   constrained to the interval [0.,1.]
wfc0=evec_array.berry_phase([0])/(2.*np.pi)%1.
wfc1=evec_array.berry_phase([1])/(2.*np.pi)%1.
x=evec_array.berry_phase([0,1],berry_evals=True)/(2.*np.pi)%1.
gwfc0=x[0]
gwfc1=x[1]

print ("Wannier centers of bands 0 and 1:")
print(("  Individual"+" Wannier centers: "+2*"%7.4f") % (wfc0,wfc1))
print(("  Multiband "+" Wannier centers: "+2*"%7.4f") % (gwfc1,gwfc0))
print()

# construct and solve finite model by cutting 10 cells from infinite chain
finite_model=my_model.cut_piece(10,0)
(feval,fevec)=finite_model.solve_all(eig_vectors=True)

print ("Finite-chain eigenenergies associated with")
print(("Band 0:"+10*"%6.2f")% tuple(feval[0:10]))
print(("Band 1:"+10*"%6.2f")% tuple(feval[10:20]))

# find maxloc Wannier centers in each band subspace
xbar0=finite_model.position_hwf(fevec[0:10,],0)
xbar1=finite_model.position_hwf(fevec[10:20,],0)
xbarb=finite_model.position_hwf(fevec[0:20,],0)

print ("\nFinite-chain Wannier centers associated with band 0:")
print((10*"%7.4f")% tuple(xbar0))
x=10*(wfc0,)
print(("Compare with bulk:\n"+10*"%7.4f")% x)
print ("\nFinite-chain Wannier centers associated with band 1:")
print((10*"%7.4f")% tuple(xbar1))
x=10*(wfc1,)
print(("Compare with bulk:\n"+10*"%7.4f")% x)
print ("\nFirst 10 finite-chain Wannier centers associated with bands"+
  "0 and 1:")
print((10*"%7.4f")% tuple(xbarb[0:10]))
x=5*(gwfc0,gwfc1)
print(("Compare with bulk:\n"+10*"%7.4f")% x)
```

D.9 Adiabatic Cycle for Infinite Three-Site Chain

Program `chain_3_cycle.py` treats the same model as in Appendix D.8, but this time treating λ as a cyclic parameter that runs from 0 to 2π and focusing on tracking the bulk Wannier center position of the lowest band through this cycle. The approach is almost the same as is used in the "one-dimensional cycle of 1D tight-binding model" example program in the standard PythTB distribution, except for the usage of the `wf_array` method in two-dimensional (k, λ) space there, which has the advantage of providing automatic enforcement of continuity with respect to λ. By contrast, continuity is enforced explicitly in the version here in the lines following the comment line `#enforce smooth evolution of xbar`. The plotted output of this program appears as Fig. 4.5 in Section 4.2.3.

https://doi.org/10.1017/9781316662205.012 Published online by Cambridge University Press

chain_3_cycle.py

```
#!/usr/bin/env python
from __future__ import print_function # python3 style print

# Chain with three sites per cell - cyclic variation

from pythtb import *
import matplotlib.pyplot as plt

# define function to construct model
def set_model(t,delta,lmbd):
    lat=[[1.0]]
    orb=[[0.0],[1.0/3.0],[2.0/3.0]]
    model=tb_model(1,1,lat,orb)
    model.set_hop(t, 0, 1, [0])
    model.set_hop(t, 1, 2, [0])
    model.set_hop(t, 2, 0, [1])
    onsite_0=delta*(-1.0)*np.cos(lmbd)
    onsite_1=delta*(-1.0)*np.cos(lmbd-2.0*np.pi/3.0)
    onsite_2=delta*(-1.0)*np.cos(lmbd-4.0*np.pi/3.0)
    model.set_onsite([onsite_0,onsite_1,onsite_2])
    return(model)

def get_xbar(band,model):
    evec_array=wf_array(model,[21])        # set array dimension
    evec_array.solve_on_grid([0.])         # fill with eigensolutions
    wfc=evec_array.berry_phase([band])/(2.*np.pi)  # Wannier centers
    return(wfc)

# set fixed parameters
t=-1.3
delta=2.0

# obtain results for an array of lambda values
lmbd=np.linspace(0.,2.*np.pi,61)
xbar=np.zeros_like(lmbd)
for j,lam in enumerate(lmbd):
    my_model=set_model(t,delta,lam)
    xbar[j]=get_xbar(0,my_model)    # Wannier center of bottom band

# enforce smooth evolution of xbar
for j in range(1,61):
    delt=xbar[j]-xbar[j-1]
    delt=-0.5+(delt+0.5)%1.  # add integer to enforce |delt| < 0.5
    xbar[j]=xbar[j-1]+delt

# set up the figure
fig, ax = plt.subplots(figsize=(5.,3.))
ax.set_xlim([0.,2.*np.pi])
ax.set_ylim([-0.6,1.1])
ax.set_xlabel(r"Parameter $\lambda$")
ax.set_ylabel(r"Wannier center position")
xlab=[r"0",r"$\pi/3$",r"$2\pi/3$",r"$\pi$",r"$4\pi/3$",r"$5\pi/3$",r"$2\pi$"]
ax.set_xticks(np.linspace(0.,2.*np.pi,num=7))
ax.set_xticklabels(xlab)
ax.plot(lmbd,xbar,'k')    # plot Wannier center and some periodic images
ax.plot(lmbd,xbar-1.,'k')
ax.plot(lmbd,xbar+1.,'k')
ax.axhline(y=1.,color='k',linestyle='dashed') # horizontal reference lines
ax.axhline(y=0.,color='k',linestyle='dashed')
fig.tight_layout()
fig.savefig("chain_3_cycle.pdf")
```

https://doi.org/10.1017/9781316662205.012 Published online by Cambridge University Press

D.10 Surface Properties of Alternating Site Model

Program `chain_alt_surf.py` returns to the alternating site model of Appendices D.4 and D.7, but now focuses on computing the surface properties of a finite chain. Two types of variation are considered: modification of the site energy of the last orbital on the chain, and modification of the bulk Hamiltonian along an adiabatic cycle corresponding to a charge pump. The calculations are discussed in Section 4.5.3, where the resulting plot is presented as Fig. 4.16. For each kind of parameter variation, three plots are presented, showing the variation of the energy eigenvalues, the Wannier center positions (computed along the same lines as in Appendix D.8), and the surface charge computed using the ramp-function method of Eqs. (4.78) and (4.79).

chain_alt_surf.py

```
#!/usr/bin/env python
from __future__ import print_function # python3 style print

# Chain with alternating site energies and hoppings
# Study surface properties of finite chain

from pythtb import *
import matplotlib as mpl
import matplotlib.pyplot as plt

# to set up model for given surface energy shift and lambda
def set_model(n_cell,en_shift,lmbd):

  # set parameters of model
  t=-1.0       # average hopping
  Delta=-0.4*np.cos(lmbd)   # site energy alternation
  del_t=-0.3*np.sin(lmbd)   # bond strength alternation

  # construct bulk model
  lat=[[1.0]]
  orb=[[0.0],[0.5]]
  bulk_model=tbmodel(1,1,lat,orb)
  bulk_model.set_onsite([Delta,-Delta])
  bulk_model.add_hop(t+del_t, 0, 1, [0])
  bulk_model.add_hop(t-del_t, 1, 0, [1])

  # cut chain of length n_cell and shift energy on last site
  finite_model=bulk_model.cut_piece(n_cell,0)
  finite_model.set_onsite(en_shift,ind_i=2*n_cell-1,mode='add')

  return finite_model

# set Fermi energy and number of cells
Ef=0.18
n_cell=20
n_orb=2*n_cell

# set number of parameter values to run over
n_param=101

# initialize arrays
params=np.linspace(0.,1.,n_param)
eig_sav=np.zeros((n_orb,n_param),dtype=float)
```

https://doi.org/10.1017/9781316662205.012 Published online by Cambridge University Press

```
xbar_sav=np.zeros((n_orb,n_param),dtype=float)
nocc_sav=np.zeros((n_param),dtype=int)
surf_sav=np.zeros((n_param),dtype=float)
count=np.zeros((n_orb),dtype=float)

# initialize plots
mpl.rc('font',size=10)  # set global font size
fig,ax=plt.subplots(3,2,figsize=(7.,6.),
    gridspec_kw={'height_ratios':[2,1,1]},sharex="col")

# loop over two cases: vary surface site energy, or vary lambda
for mycase in ['surface energy','lambda']:

  if mycase == 'surface energy':
    (ax0,ax1,ax2)=ax[:,0]   # axes for plots in left panels
    ax0.text(-0.30,0.90,'(a)',size=22.,transform=ax0.transAxes)
    lmbd=0.15*np.pi*np.ones((n_param),dtype=float)
    en_shift=-3.0+6.0*params
    abscissa=en_shift
  elif mycase == 'lambda':
    (ax0,ax1,ax2)=ax[:,1]   # axes for plots in right panels
    ax0.text(-0.30,0.90,'(b)',size=22.,transform=ax0.transAxes)
    lmbd=params*2.*np.pi
    en_shift=0.2*np.ones((n_param),dtype=float)
    abscissa=params

  # loop over parameter values
  for j in range(n_param):

    # set up and solve model; store eigenvalues
    my_model=set_model(n_cell,en_shift[j],lmbd[j])
    (eval,evec)=my_model.solve_all(eig_vectors=True)

    # find occupied states
    nocc=(eval<Ef).sum()
    ovec=evec[0:nocc,:]

    # get Wannier centers
    xbar_sav[0:nocc,j]=my_model.position_hwf(ovec,0)

    # get electron count on each site
    # convert to charge (2 for spin; unit nuclear charge per site)
    # compute surface charge down to depth of 1/3 of chain
    for i in range(n_orb):
      count[i]=np.real(np.vdot(evec[:nocc,i],evec[:nocc,i]))
    charge=-2.*count+1.
    n_cut=int(0.67*n_orb)
    surf_sav[j]=0.5*charge[n_cut-1]+charge[n_cut:].sum()

    # save information for plots
    nocc_sav[j]=nocc
    eig_sav[:,j]=eval

  ax0.set_xlim(0.,1.)
  ax0.set_ylim(-2.8,2.8)
  ax0.set_ylabel(r"Band energy")
  ax0.axhline(y=Ef,color='k',linewidth=0.5)
  for n in range(n_orb):
    ax0.plot(abscissa,eig_sav[n,:],color='k')

  ax1.set_xlim(0.,1.)
```

https://doi.org/10.1017/9781316662205.012 Published online by Cambridge University Press

```
  ax1.set_ylim(n_cell-4.6,n_cell+0.4)
  ax1.set_yticks(np.linspace(n_cell-4,n_cell,5))
  #ax1.set_ylabel(r"$\bar{x}$")
  ax1.set_ylabel(r"Wannier centers")
  for j in range(n_param):
    nocc=nocc_sav[j]
    ax1.scatter([abscissa[j]]*nocc,xbar_sav[:nocc,j],color='k',
      s=3.,marker='o',edgecolors='none')

  ax2.set_ylim(-2.2,2.2)
  ax2.set_yticks([-2.,-1.,0.,1.,2.])
  ax2.set_ylabel(r"Surface charge")
  if mycase == 'surface energy':
    ax2.set_xlabel(r"Surface site energy")
  elif mycase == 'lambda':
    ax2.set_xlabel(r"$\lambda/2\pi$")
  ax2.set_xlim(abscissa[0],abscissa[-1])
  ax2.scatter(abscissa,surf_sav,color='k',s=3.,marker='o',edgecolors='none')

  # vertical lines denote surface state at right end crossing the
  # Fermi energy
  for j in range(1,n_param):
    if nocc_sav[j] != nocc_sav[j-1]:
      n=min(nocc_sav[j],nocc_sav[j-1])
      frac=(Ef-eig_sav[n,j-1])/(eig_sav[n,j]-eig_sav[n,j-1])
      a_jump=(1-frac)*abscissa[j-1]+frac*abscissa[j]
      if mycase == 'surface energy' or nocc_sav[j] < nocc_sav[j-1]:
        ax0.axvline(x=a_jump,color='k',linewidth=0.5)
        ax1.axvline(x=a_jump,color='k',linewidth=0.5)
        ax2.axvline(x=a_jump,color='k',linewidth=0.5)

fig.tight_layout()
plt.subplots_adjust(left=0.12,wspace=0.4)
fig.savefig("chain_alt_surf.pdf")
```

D.11 Band Structure of the Haldane Model

Program `haldane_bsr.py` computes the band structure of the Haldane model of Eq. (5.1) for four different parameter sets, with the plotted output appearing as Fig. 5.2 in Section 5.1.1. In addition to plotting the band structure, the program places filled or open circles on the bands at the high-symmetry K and K' points according to which site dominates the character of the lower-energy band, thereby highlighting the band inversion.

haldane_bsr.py

```
#!/usr/bin/env python
from __future__ import print_function # python3 style print

# Band structure of Haldane model

from pythtb import * # import TB model class
import matplotlib.pyplot as plt

# set model parameters
```

https://doi.org/10.1017/9781316662205.012 Published online by Cambridge University Press

```
delta=0.7   # site energy shift
t=-1.0      # real first-neighbor hopping
t2=0.15     # imaginary second-neighbor hopping

def set_model(delta,t,t2):
  lat=[[1.0,0.0],[0.5,np.sqrt(3.0)/2.0]]
  orb=[[1./3.,1./3.],[2./3.,2./3.]]
  model=tb_model(2,2,lat,orb)
  model.set_onsite([-delta,delta])
  for lvec in ([ 0, 0], [-1, 0], [ 0,-1]):
    model.set_hop(t, 0, 1, lvec)
  for lvec in ([ 1, 0], [-1, 1], [ 0,-1]):
    model.set_hop(t2*1.j, 0, 0, lvec)
  for lvec in ([-1, 0], [ 1,-1], [ 0, 1]):
    model.set_hop(t2*1.j, 1, 1, lvec)
  return model

# construct path in k-space and solve model
path=[[0.,0.],[2./3.,1./3.],[.5,.5],[1./3.,2./3.], [0.,0.]]
label=(r'$\Gamma $',r'$K$', r'$M$', r'$K^\prime$', r'$\Gamma $')
(k_vec,k_dist,k_node)=set_model(delta,t,t2).k_path(path,101)

# set up band structure plots
fig, ax = plt.subplots(2,2,figsize=(8.,6.),sharex=True,sharey=True)
ax=ax.flatten()

t2_values=[0.,-0.06,-0.1347,-0.24]
labs=['(a)','(b)','(c)','(d)']
for j in range(4):

  my_model=set_model(delta,t,t2_values[j])
  evals=my_model.solve_all(k_vec)

  ax[j].set_xlim([0,k_node[-1]])
  ax[j].set_xticks(k_node)
  ax[j].set_xticklabels(label)
  for n in range(len(k_node)):
    ax[j].axvline(x=k_node[n],linewidth=0.5, color='k')
  ax[j].set_ylabel("Energy")
  ax[j].set_ylim(-3.8,3.8)
  for n in range(2):
    ax[j].plot(k_dist,evals[n],color='k')

  # filled or open dots at K and K' following band inversion
  for m in [1,3]:
    kk=k_node[m]
    (en,ev)=my_model.solve_one(path[m],eig_vectors=True)
    if np.abs(ev[0,0]) > np.abs(ev[0,1]):   #ev[band,orb]
      en=[en[1],en[0]]
    ax[j].scatter(kk,en[0],s=40.,marker='o',edgecolors='k',
      facecolors='w',zorder=4)
    ax[j].scatter(kk,en[1],s=40.,marker='o',color='k',zorder=6)

  ax[j].text(0.20,3.1,labs[j],size=18.)

# save figure as a PDF
fig.tight_layout()
fig.savefig("haldane_bsr.pdf")
```

https://doi.org/10.1017/9781316662205.012 Published online by Cambridge University Press

D.12 Berry Curvature of the Haldane Model

Program `haldane_bcurv.py` generates contour plots of the Berry curvature of the Haldane model for three parameter sets. The model is solved on a 2D mesh of **k**-points using the `solve_on_grid` method of `wf_array`. Then the `berry_flux` method is used in two different ways: once with `individual_phases=True` to obtain the flux through each plaquette of the mesh, and once with the default `False` value to obtain the total flux through the entire 2D BZ. When normalized, the first gives the Berry curvature, which is plotted using the `contour` method of PYPLOT, and the second gives the Chern number, which is printed out. The plot appears as Fig. 5.3 in Section 5.1.1.

haldane_bcurv.py

```
#!/usr/bin/env python
from __future__ import print_function # python3 style print

# Berry curvature of Haldane model

from pythtb import * # import TB model class
import matplotlib.pyplot as plt

# define setup of Haldane model
def set_model(delta,t,t2):
  lat=[[1.0,0.0],[0.5,np.sqrt(3.0)/2.0]]
  orb=[[1./3.,1./3.],[2./3.,2./3.]]
  model=tb_model(2,2,lat,orb)
  model.set_onsite([-delta,delta])
  for lvec in ([ 0, 0], [-1, 0], [ 0,-1]):
    model.set_hop(t, 0, 1, lvec)
  for lvec in ([ 1, 0], [-1, 1], [ 0,-1]):
    model.set_hop(t2*1.j, 0, 0, lvec)
  for lvec in ([-1, 0], [ 1,-1], [ 0, 1]):
    model.set_hop(t2*1.j, 1, 1, lvec)
  return model

# miscellaneous setup
delta=0.7   # site energy shift
t=-1.0      # real first-neighbor hopping

nk=61
dk=2.*np.pi/(nk-1)
k0=(np.arange(nk-1)+0.5)/(nk-1)
kx=np.zeros((nk-1,nk-1),dtype=float)
ky=np.zeros((nk-1,nk-1),dtype=float)
sq3o2=np.sqrt(3.)/2.
for i in range(nk-1):
  for j in range(nk-1):
    kx[i,j]=sq3o2*k0[i]
    ky[i,j]= -0.5*k0[i]+k0[j]

fig,ax=plt.subplots(1,3,figsize=(11,4))
labs=['(a)','(b)','(c)']
```

https://doi.org/10.1017/9781316662205.012 Published online by Cambridge University Press

```
# compute Berry curvature and Chern number for three values of t2
for j,t2 in enumerate([0.,-0.06,-0.24]):
  my_model=set_model(delta,t,t2)
  my_array=wf_array(my_model,[nk,nk])
  my_array.solve_on_grid([0.,0.])
  bcurv=my_array.berry_flux([0],individual_phases=True)/(dk*dk)
  chern=my_array.berry_flux([0])/(2.*np.pi)
  print('Chern number =',"%8.5f"%chern)

  # make contour plot of Berry curvature
  pos_lvls= 0.02*np.power(2.,np.linspace(0,8,9))
  neg_lvls=-0.02*np.power(2.,np.linspace(8,0,9))
  ax[j].contour(kx,ky,bcurv,levels=pos_lvls,colors='k')
  ax[j].contour(kx,ky,bcurv,levels=neg_lvls,colors='k',linewidths=1.4)

  # remove rectangular box and draw parallelogram, etc.
  ax[j].xaxis.set_visible(False)
  ax[j].yaxis.set_visible(False)
  for loc in ["top","bottom","left","right"]:
    ax[j].spines[loc].set_visible(False)
  ax[j].set(aspect=1.)
  ax[j].plot([0,sq3o2,sq3o2,0,0],[0,-0.5,0.5,1,0],color='k',linewidth=1.4)
  ax[j].set_xlim(-0.05,sq3o2+0.05)
  ax[j].text(-.35,0.88,labs[j],size=24.)

fig.savefig("haldane_bcurv.pdf")
```

D.13 Hybrid Wannier Centers and Edge States of the Haldane Model

Program `haldane_topo.py` produces plots of the hybrid Wannier center flow in the bulk, and of the edge band structure of a finite-width ribbon, for the Haldane model with two different parameter sets covering both the trivial and topological cases. The plotted output appears as Fig. 5.4 in Section 5.1.1. The methods used to obtain the bulk Wannier centers should be familiar by now. The `cut_piece` method of the `tb_model class` is applied to the 2D bulk model `my_model` to obtain the new 1D `ribbon_model`, extending indefinitely in the horizontal direction but of finite width vertically, here chosen to be 20 cells high. The band structure of the ribbon is plotted using the PYPLOT `scatter` method, instead of as a continuous line, so that the identity of the states localized at the top and bottom edges of the ribbon can be distinguished. This is done by assigning each plotted point a weight that is normally 1, but that is reduced for states at the bottom edge. These are identified by using the `position_expectation` method of PYTHTB to compute the component of the position expectation value in the vertical direction for each Bloch eigenvector.

haldane_topo.py

```
#!/usr/bin/env python
from __future__ import print_function # python3 style print

# Band structure of Haldane model
```

https://doi.org/10.1017/9781316662205.012 Published online by Cambridge University Press

```
from pythtb import * # import TB model class
import matplotlib.pyplot as plt

# define setup of Haldane model
def set_model(delta,t,t2):
  lat=[[1.0,0.0],[0.5,np.sqrt(3.0)/2.0]]
  orb=[[1./3.,1./3.],[2./3.,2./3.]]
  model=tb_model(2,2,lat,orb)
  model.set_onsite([-delta,delta])
  for lvec in ([ 0, 0], [-1, 0], [ 0,-1]):
    model.set_hop(t, 0, 1, lvec)
  for lvec in ([ 1, 0], [-1, 1], [ 0,-1]):
    model.set_hop(t2*1.j, 0, 0, lvec)
  for lvec in ([-1, 0], [ 1,-1], [ 0, 1]):
    model.set_hop(t2*1.j, 1, 1, lvec)
  return model

# set model parameters and construct bulk model
delta=0.7   # site energy shift
t=-1.0      # real first-neighbor hopping
nk=51

# For the purposes of plot labels:
#   Real space is (r1,r2) in reduced coordinates
#   Reciprocal space is (k1,k2) in reduced coordinates
# Below, following Python, these are (r0,r1) and (k0,k1)

# set up figures
fig,ax=plt.subplots(2,2,figsize=(7,6))

# run over two choices of t2
for j2,t2 in enumerate([-0.06,-0.24]):

  # solve bulk model on grid and get hybrid Wannier centers along r1
  # as a function of k0
  my_model=set_model(delta,t,t2)
  my_array=wf_array(my_model,[nk,nk])
  my_array.solve_on_grid([0.,0.])
  rbar_1 = my_array.berry_phase([0],1,contin=True)/(2.*np.pi)

  # set up and solve ribbon model that is finite along direction 1
  width=20
  nkr=81
  ribbon_model=my_model.cut_piece(width,fin_dir=1,glue_edgs=False)
  (k_vec,k_dist,k_node)=ribbon_model.k_path('full',nkr,report=False)
  (rib_eval,rib_evec)=ribbon_model.solve_all(k_vec,eig_vectors=True)

  nbands=rib_eval.shape[0]
  (ax0,ax1)=ax[j2,:]

  # hybrid Wannier center flow
  k0=np.linspace(0.,1.,nk)
  ax0.set_xlim(0.,1.)
  ax0.set_ylim(-1.3,1.3)
  ax0.set_xlabel(r"$\kappa_1/2\pi$")
  ax0.set_ylabel(r"HWF centers")
  for shift in (-2.,-1.,0.,1.):
    ax0.plot(k0,rbar_1+shift,color='k')

  # edge band structure
  k0=np.linspace(0.,1.,nkr)
```

https://doi.org/10.1017/9781316662205.012 Published online by Cambridge University Press

```
    ax1.set_xlim(0.,1.)
    ax1.set_ylim(-2.5,2.5)
    ax1.set_xlabel(r"$\kappa_1/2\pi$")
    ax1.set_ylabel(r"Edge band structure")
    for (i,kv) in enumerate(k0):

      # find expectation value <r1> at i'th k-point along direction k0
      pos_exp=ribbon_model.position_expectation(rib_evec[:,i],dir=1)

      # assign weight in [0,1] to be 1 except for edge states near bottom
      weight=3.0*pos_exp/width
      for j in range(nbands):
        weight[j]=min(weight[j],1.)

      # scatterplot with symbol size proportional to assigned weight
      s=ax1.scatter([k_vec[i]]*nbands, rib_eval[:,i],
           s=0.6+2.5*weight, c='k', marker='o', edgecolors='none')

# save figure as a PDF
aa=ax.flatten()
for i,lab in enumerate(['(a)','(b)','(c)','(d)']):
  aa[i].text(-0.45,0.92,lab,size=18.,transform=aa[i].transAxes)
fig.tight_layout()
plt.subplots_adjust(left=0.15,wspace=0.6)
fig.savefig("haldane_topo.pdf")
```

D.14 Entanglement Spectrum of the Haldane Model

Program `haldane_entang.py` produces plots of the entanglement spectrum for the Haldane model for two parameter sets, one in the trivial phase and one in the topological phase. The entanglement spectrum, mentioned briefly at the end of Section 5.1.1, is defined as follows. For an infinite 1D chain, one arbitrarily chooses a dividing surface that partitions the chain into a lower region A and an upper region B. The entanglement spectrum is then given by computing the eigenvalues of the reduced density matrix $\rho_A = \mathcal{P}_A \rho \mathcal{P}_A$, where $\rho_{ij} = \sum_n^{\text{occ}} \psi_{nk}^*(i)\psi_{nk}(j)$ is the one-particle density matrix in the TB basis and $\mathcal{P}_A$ is the projector onto subregion A. For a 2D system, one plots these eigenvalues for the effective 1D system extending along lattice direction $\mathbf{a}_2$ as a function of κ_1, as shown in Fig. D.3. In practice, `haldane_entang.py` obtains the entanglement spectrum from the same finite-width ribbon constructed in `haldane_topo.py`, partitioned in the middle; edge states on the physical boundary of the ribbon have little influence since they are far from the dividing surface.

A comparison of Fig. D.3(b) and Fig. 5.4(c) shows that an upward flow of the hybrid Wannier function (HWF) centers with increasing κ_1 corresponds to a downward flow of the ρ_A eigenvalues. This is only natural, since an upward migration of Wannier functions corresponds to a depletion of occupation in the lower region A.

https://doi.org/10.1017/9781316662205.012 Published online by Cambridge University Press

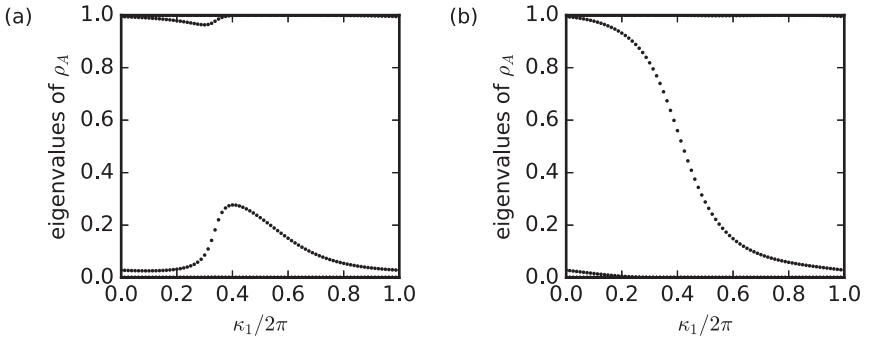

Figure D.3 Entanglement spectrum of the Haldane model with $\Delta = 0.7$ and $t_1 = -1$. A ribbon geometry of width 20 unit cells along the κ_2 direction was subdivided into bottom and top halves A and B, respectively; eigenvalues of the reduced density matrix ρ_A (see text) are plotted vertically. (a) Trivial phase ($t_2 = 0.10$) with $C = 0$. (b) Topological phase ($t_2 = -0.24$) with $C = 1$.

The origin of the name "entanglement spectrum" is associated with the definition of an *entanglement entropy* given by $\mathrm{Tr}\,[-\rho_A \ln \rho_A - (1-\rho_A)\ln(1-\rho_A)]$. This object is dominated by eigenvalues of ρ_A lying in the middle of the interval $[0, 1]$, so roughly speaking it counts the number of states that cannot be localized entirely in A or in B, but instead are "entangled" across the dividing surface.

The flow of the single-particle entanglement spectrum, as shown in Fig. D.3, can be used as a tool to extract the Chern index of a 2D magnetic insulator, much like the edge band structures and HWF flow patterns in Fig. 5.4. The three methods are, in a sense, close cousins. However, the HWF flow approach has the advantage of requiring only a simple bulk calculation, with no introduction of artificial bounding surfaces or edge terminations. When generalized to the many-body context, however, the entanglement entropy becomes a useful tool for the study of strongly correlated topological states with long-range entanglement, such as those mentioned in Section 5.5.3.

haldane_entang.py

```
#!/usr/bin/env python
from __future__ import print_function # python3 style print

# Entanglement spectrum of Haldane model

from pythtb import * # import TB model class
import matplotlib.pyplot as plt

# define setup of Haldane model
def set_model(delta,t,t2):
  lat=[[1.0,0.0],[0.5,np.sqrt(3.0)/2.0]]
  orb=[[1./3.,1./3.],[2./3.,2./3.]]
  model=tb_model(2,2,lat,orb)
  model.set_onsite([-delta,delta])
  for lvec in ([ 0, 0], [-1, 0], [ 0,-1]):
```

https://doi.org/10.1017/9781316662205.012 Published online by Cambridge University Press

```
      model.set_hop(t, 0, 1, lvec)
    for lvec in ([ 1, 0], [-1, 1], [ 0,-1]):
      model.set_hop(t2*1.j, 0, 0, lvec)
    for lvec in ([-1, 0], [ 1,-1], [ 0, 1]):
      model.set_hop(t2*1.j, 1, 1, lvec)
    return model
# set model parameters and construct bulk model
delta=0.7   # site energy shift
t=-1.0      # real first-neighbor hopping

# set up figures
fig,ax=plt.subplots(1,2,figsize=(7,3))

# run over two choices of t2
for j2,t2 in enumerate([-0.10,-0.24]):

  my_model=set_model(delta,t,t2)

  # set up and solve ribbon model that is finite along direction 1
  width=20
  nkr=81
  ribbon_model=my_model.cut_piece(width,fin_dir=1,glue_edgs=False)
  (k_vec,k_dist,k_node)=ribbon_model.k_path('full',nkr,report=False)
  (rib_eval,rib_evec)=ribbon_model.solve_all(k_vec,eig_vectors=True)

  nbands=rib_eval.shape[0]
  ax1=ax[j2]

  # entanglement spectrum
  k0=np.linspace(0.,1.,nkr)
  ax1.set_xlim(0.,1.)
  ax1.set_ylim(0.,1)
  ax1.set_xlabel(r"$\kappa_1/2\pi$")
  ax1.set_ylabel(r"eigenvalues of $\rho_A$")

  (nband,nk,norb)=rib_evec.shape
  ncut=norb/2
  nocc=nband/2

  for (i,kv) in enumerate(k0):

    # construct reduced density matrix for half of the chain
    dens_mat=np.zeros((ncut,ncut),dtype=complex)
    for nb in range(nocc):
      for j1 in range(ncut):
        for j2 in range(ncut):
          dens_mat[j1,j2] += np.conj(rib_evec[nb,i,j1])*rib_evec[nb,i,j2]

    # diagonalize
    spect=np.real(np.linalg.eigvals(dens_mat))
    # scatterplot
    s=ax1.scatter([k_vec[i]]*nocc, spect,
         s=4, c='k', marker='o', edgecolors='none')

# save figure as a PDF
aa=ax.flatten()
for i,lab in enumerate(['(a)','(b)']):
  aa[i].text(-0.45,0.92,lab,size=18.,transform=aa[i].transAxes)
fig.tight_layout()
plt.subplots_adjust(left=0.15,wspace=0.6)
fig.savefig("haldane_entang.pdf")
```

https://doi.org/10.1017/9781316662205.012 Published online by Cambridge University Press

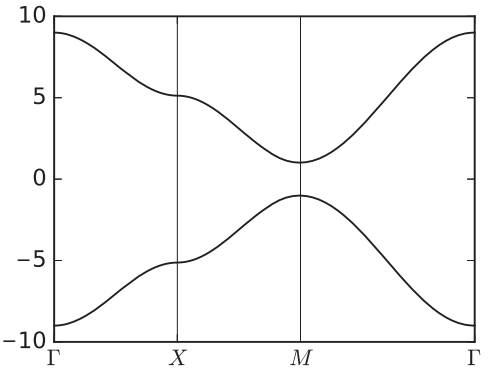

Figure D.4 Band structure of the 2D checkerboard model.

D.15 Band Structure of the Checkerboard Model

Program `checkerboard.py` computes the band structure for the 2D checkerboard model introduced in Ex. 5.3. The output appears here as Fig. D.4.

checkerboard.py

```
#!/usr/bin/env python
from __future__ import print_function # python3 style print

from pythtb import * # import TB model class
import matplotlib.pyplot as plt

# set geometry
lat=[[1.0,0.0],[0.0,1.0]]
orb=[[0.0,0.0],[0.5,0.5]]
my_model=tbmodel(2,2,lat,orb)

# set model
Delta  = 5.0
t_0    = 1.0
tprime = 0.4
my_model.set_sites([-Delta,Delta])
my_model.add_hop(-t_0, 0, 0, [ 1, 0])
my_model.add_hop(-t_0, 0, 0, [ 0, 1])
my_model.add_hop( t_0, 1, 1, [ 1, 0])
my_model.add_hop( t_0, 1, 1, [ 0, 1])
my_model.add_hop( tprime   , 1, 0, [ 1, 1])
my_model.add_hop( tprime*1j, 1, 0, [ 0, 1])
my_model.add_hop(-tprime   , 1, 0, [ 0, 0])
my_model.add_hop(-tprime*1j, 1, 0, [ 1, 0])
my_model.display()

# generate k-point path and labels and solve Hamiltonian
path=[[0.0,0.0],[0.0,0.5],[0.5,0.5],[0.0,0.0]]
k_lab=(r'$\Gamma $',r'$X$', r'$M$', r'$\Gamma $')
(k_vec,k_dist,k_node)=my_model.k_path(path,121)
evals=my_model.solve_all(k_vec)

# plot band structure
fig, ax = plt.subplots(figsize=(4.,3.))
```

https://doi.org/10.1017/9781316662205.012 Published online by Cambridge University Press

```
ax.set_xlim([0,k_node[-1]])
ax.set_xticks(k_node)
ax.set_xticklabels(k_lab)
for n in range(len(k_node)):
  ax.axvline(x=k_node[n], linewidth=0.5, color='k')
ax.plot(k_dist,evals[0],color='k')
ax.plot(k_dist,evals[1],color='k')
fig.savefig("checkerboard_bsr.pdf")
```

D.16 Band Structure of the Kane–Mele Model

Program `kanemele_bsr.py` computes the band structure of the Kane–Mele model of Eq. (5.17) in Section 5.2.2 for two parameter sets exemplifying the normal and topological regimes. This is a spinor model, so we set `nspin=2` when creating the model. To simplify the specification of the spin-dependent hoppings in the model, we have predefined Pauli matrices σ_x, σ_y, and σ_z in Cartesian directions, and then σ_a, σ_b, and σ_c in three relevant in-plane directions related by 120° rotations. Note that `mode="add"` is needed when we specify a second component of the first-neighbor hoppings.

The results are plotted as Fig. 5.12. Filled and open circles are added to denote the orbital character for the highest occupied and lowest unoccupied states at K and K' to highlight the band inversion. Here we have cheated, with the filled and open circles simply exchanged in the second plot; a proper implementation should follow the example of `haldane_bsr.py` in Appendix D.11.

kanemele_bsr.py

```
#!/usr/bin/env python
from __future__ import print_function # python3 style print

# Tight-binding 2D Kane-Mele model
# C.L. Kane and E.J. Mele, PRL 95, 146802 (2005)

from pythtb import * # import TB model class
import matplotlib.pyplot as plt

# set model parameters
delta=0.7     # site energy
t=-1.0        # spin-independent first-neighbor hop
rashba=0.05   # spin-flip first-neighbor hop
soc_list=[-0.06,-0.24] # spin-dependent second-neighbor hop

def set_model(t,soc,rashba,delta):

  # set up Kane-Mele model
  lat=[[1.0,0.0],[0.5,np.sqrt(3.0)/2.0]]
  orb=[[1./3.,1./3.],[2./3.,2./3.]]
  model=tb_model(2,2,lat,orb,nspin=2)
  model.set_onsite([delta,-delta])

  # definitions of Pauli matrices
  sigma_x=np.array([0.,1.,0.,0])
  sigma_y=np.array([0.,0.,1.,0])
```

https://doi.org/10.1017/9781316662205.012 Published online by Cambridge University Press

```
  sigma_z=np.array([0.,0.,0.,1])
  r3h =np.sqrt(3.0)/2.0
  sigma_a= 0.5*sigma_x-r3h*sigma_y
  sigma_b= 0.5*sigma_x+r3h*sigma_y
  sigma_c=-1.0*sigma_x

  # spin-independent first-neighbor hops
  for lvec in ([ 0, 0], [-1, 0], [ 0,-1]):
    model.set_hop(t, 0, 1, lvec)
  # spin-dependent second-neighbor hops
  for lvec in ([ 1, 0], [-1, 1], [ 0,-1]):
    model.set_hop(soc*1.j*sigma_z, 0, 0, lvec)
  for lvec in ([-1, 0], [ 1,-1], [ 0, 1]):
    model.set_hop(soc*1.j*sigma_z, 1, 1, lvec)
  # spin-flip first-neighbor hops
  model.set_hop(1.j*rashba*sigma_a, 0, 1, [ 0, 0], mode="add")
  model.set_hop(1.j*rashba*sigma_b, 0, 1, [-1, 0], mode="add")
  model.set_hop(1.j*rashba*sigma_c, 0, 1, [ 0,-1], mode="add")

  return model

# construct path in k-space and solve model
path=[[0.,0.],[2./3.,1./3.],[.5,.5],[1./3.,2./3.], [0.,0.]]
label=(r'$\Gamma $',r'$K$', r'$M$', r'$K^\prime$', r'$\Gamma $')
(k_vec,k_dist,k_node)=set_model(t,0.,rashba,delta).k_path(path,101)

# set up band structure plots
fig, ax = plt.subplots(1,2,figsize=(8.,3.))

labs=['(a)','(b)']
for j in range(2):

  my_model=set_model(t,soc_list[j],rashba,delta)
  evals=my_model.solve_all(k_vec)

  ax[j].set_xlim([0,k_node[-1]])
  ax[j].set_xticks(k_node)
  ax[j].set_xticklabels(label)
  for n in range(len(k_node)):
    ax[j].axvline(x=k_node[n],linewidth=0.5, color='k')
  ax[j].set_ylabel("Energy")
  ax[j].set_ylim(-3.8,3.8)
  for n in range(4):
    ax[j].plot(k_dist,evals[n],color='k')

  for m in [1,3]:
    kk=k_node[m]
    en=my_model.solve_one(path[m])
    en=en[1:3]  # pick out second and third bands
    if j==1:    # exchange them in second plot
      en=[en[1],en[0]]
    ax[j].scatter(kk,en[0],s=40.,marker='o',color='k',zorder=6)
    ax[j].scatter(kk,en[1],s=40.,marker='o',edgecolors='k',
      facecolors='w',zorder=4)

  ax[j].text(-0.45,3.5,labs[j],size=18.)

# save figure as a PDF
fig.tight_layout()
plt.subplots_adjust(wspace=0.35)
fig.savefig("kanemele_bsr.pdf")
```

https://doi.org/10.1017/9781316662205.012 Published online by Cambridge University Press

D.17 Hybrid Wannier Centers and Edge States of the Kane–Mele Model

Program `kanemele_topo.py` produces plots of the hybrid Wannier center flow in the bulk, and of the edge band structure of a finite-width ribbon, for the Kane–Mele model with two different parameter sets covering both the trivial and topological cases, following the example of `haldane_topo.py` in Appendix D.13. The plotted output appears as Fig. 5.13 in Section 5.2.2.

kanemele_topo.py

```
#!/usr/bin/env python
from __future__ import print_function # python3 style print

# Tight-binding 2D Kane-Mele model
# C.L. Kane and E.J. Mele, PRL 95, 146802 (2005)

from pythtb import * # import TB model class
import matplotlib.pyplot as plt

# set model parameters
delta=0.7     # site energy
t=-1.0        # spin-independent first-neighbor hop
soc=0.06      # spin-dependent second-neighbor hop
rashba=0.05   # spin-flip first-neighbor hop
soc_list=[-0.06,-0.24] # spin-dependent second-neighbor hop

def set_model(t,soc,rashba,delta):

  # set up Kane-Mele model
  lat=[[1.0,0.0],[0.5,np.sqrt(3.0)/2.0]]
  orb=[[1./3.,1./3.],[2./3.,2./3.]]
  model=tb_model(2,2,lat,orb,nspin=2)
  model.set_onsite([delta,-delta])

  # definitions of Pauli matrices
  sigma_x=np.array([0.,1.,0.,0])
  sigma_y=np.array([0.,0.,1.,0])
  sigma_z=np.array([0.,0.,0.,1])
  r3h =np.sqrt(3.0)/2.0
  sigma_a= 0.5*sigma_x-r3h*sigma_y
  sigma_b= 0.5*sigma_x+r3h*sigma_y
  sigma_c=-1.0*sigma_x

  # spin-independent first-neighbor hops
  for lvec in ([ 0, 0], [-1, 0], [ 0,-1]):
    model.set_hop(t, 0, 1, lvec)
  # spin-dependent second-neighbor hops
  for lvec in ([ 1, 0], [-1, 1], [ 0,-1]):
    model.set_hop(soc*1.j*sigma_z, 0, 0, lvec)
  for lvec in ([-1, 0], [ 1,-1], [ 0, 1]):
    model.set_hop(soc*1.j*sigma_z, 1, 1, lvec)
  # spin-flip first-neighbor hops
  model.set_hop(1.j*rashba*sigma_a, 0, 1, [ 0, 0], mode="add")
  model.set_hop(1.j*rashba*sigma_b, 0, 1, [-1, 0], mode="add")
  model.set_hop(1.j*rashba*sigma_c, 0, 1, [ 0,-1], mode="add")

  return model

# For the purposes of plot labels:
#   Real space is (r1,r2) in reduced coordinates
```

https://doi.org/10.1017/9781316662205.012 Published online by Cambridge University Press

```
#   Reciprocal space is (k1,k2) in reduced coordinates
# Below, following Python, these are (r0,r1) and (k0,k1)

# set up figures
fig,ax=plt.subplots(2,2,figsize=(7,6))

nk=51
# run over two choices of t2
for je,soc in enumerate(soc_list):

  # solve bulk model on grid and get hybrid Wannier centers along r1
  # as a function of k0
  my_model=set_model(t,soc,rashba,delta)
  my_array=wf_array(my_model,[nk,nk])
  my_array.solve_on_grid([0.,0.])
  rbar = my_array.berry_phase([0,1],1,berry_evals=True,contin=True)/
    (2.*np.pi)

  # set up and solve ribbon model that is finite along direction 1
  width=20
  nkr=81
  ribbon_model=my_model.cut_piece(width,fin_dir=1,glue_edgs=False)
  (k_vec,k_dist,k_node)=ribbon_model.k_path('full',nkr,report=False)
  (rib_eval,rib_evec)=ribbon_model.solve_all(k_vec,eig_vectors=True)

  nbands=rib_eval.shape[0]
  (ax0,ax1)=ax[je,:]

  # hybrid Wannier center flow
  k0=np.linspace(0.,1.,nk)
  ax0.set_xlim(0.,1.)
  ax0.set_ylim(-1.3,1.3)
  ax0.set_xlabel(r"$\kappa_1/2\pi$")
  ax0.set_ylabel(r"HWF centers")
  ax0.axvline(x=0.5,linewidth=0.5, color='k')
  for shift in (-1.,0.,1.,2.):
    ax0.plot(k0,rbar[:,0]+shift,color='k')
    ax0.plot(k0,rbar[:,1]+shift,color='k')

  # edge band structure
  k0=np.linspace(0.,1.,nkr)
  ax1.set_xlim(0.,1.)
  ax1.set_ylim(-2.5,2.5)
  ax1.set_xlabel(r"$\kappa_1/2\pi$")
  ax1.set_ylabel(r"Edge band structure")
  for (i,kv) in enumerate(k0):

    # find expectation value <r1> at i'th k-point along direction k0
    pos_exp=ribbon_model.position_expectation(rib_evec[:,i],dir=1)

    # assign weight in [0,1] to be 1 except for edge states near bottom
    weight=3.0*pos_exp/width
    for j in range(nbands):
      weight[j]=min(weight[j],1.)

    # scatterplot with symbol size proportional to assigned weight
    s=ax1.scatter([k_vec[i]]*nbands, rib_eval[:,i],
         s=0.6+2.5*weight, c='k', marker='o', edgecolors='none')

    #   ax0.text(-0.45,0.92,'(a)',size=18.,transform=ax0.transAxes)
    #   ax1.text(-0.45,0.92,'(b)',size=18.,transform=ax1.transAxes)
```

https://doi.org/10.1017/9781316662205.012 Published online by Cambridge University Press

```
# save figure as a PDF
aa=ax.flatten()
for i,lab in enumerate(['(a)','(b)','(c)','(d)']):
  aa[i].text(-0.45,0.92,lab,size=18.,transform=aa[i].transAxes)
fig.tight_layout()
plt.subplots_adjust(left=0.15,wspace=0.6)
fig.savefig("kanemele_topo.pdf")
```

D.18 Fu–Kane–Mele Model

Program `fkm.py` solves the Fu–Kane–Mele model of Eq. (5.28) in Section 5.3.4. This is again a spinor model, and the spin-dependent hoppings are now specified using the convention that `spin` is a NUMPY array containing the coefficients of $\mathbb{1}$, σ_x, σ_y, and σ_z respectively, with the list of directions `dir_list` specifying the last three of those coefficients (the first is set to zero).

The program generates two plot files, `fkm_bsr.pdf` and `fkm_topo.pdf`. The first is the band-structure plot that appears as Fig. D.5. The second is a plot of the hybrid Wannier center flow in two different κ_1–κ_2 planes, at $\kappa_3 = 0$ and at $\kappa_3 = \pi$, which appears as Fig. 5.19 in Section 5.3.4.

fkm.py

```
#!/usr/bin/env python
from __future__ import print_function # python3 style print

# Three-dimensional Fu-Kane-Mele model
# Fu, Kane and Mele, PRL 98, 106803 (2007)

from pythtb import * # import TB model class
import matplotlib.pyplot as plt

# set model parameters
t=1.0      # spin-independent first-neighbor hop
```

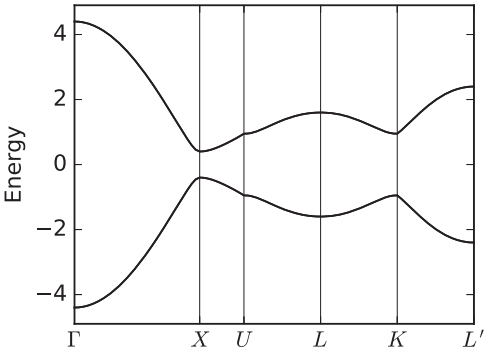

Figure D.5 Band structure of the Fu–Kane-Mele model of Eq. (5.28). Parameter values are $t_0 = 1$, $\Delta t = 0.4$, and $\lambda_{SO} = 0.125$.

https://doi.org/10.1017/9781316662205.012 Published online by Cambridge University Press

```
dt=0.4     # modification to t for (111) bond
soc=0.125  # spin-dependent second-neighbor hop

def set_model(t,dt,soc):

  # set up Fu-Kane-Mele model
  lat=[[.0,.5,.5],[.5,.0,.5],[.5,.5,.0]]
  orb=[[0.,0.,0.],[.25,.25,.25]]
  model=tb_model(3,3,lat,orb,nspin=2)

  # spin-independent first-neighbor hops
  for lvec in ([0,0,0],[-1,0,0],[0,-1,0],[0,0,-1]):
    model.set_hop(t,0,1,lvec)
  model.set_hop(dt,0,1,[0,0,0],mode="add")

  # spin-dependent second-neighbor hops
  lvec_list=([1,0,0],[0,1,0],[0,0,1],[-1,1,0],[0,-1,1],[1,0,-1])
  dir_list=([0,1,-1],[-1,0,1],[1,-1,0],[1,1,0],[0,1,1],[1,0,1])
  for j in range(6):
    spin=np.array([0.]+dir_list[j])
    model.set_hop( 1.j*soc*spin,0,0,lvec_list[j])
    model.set_hop(-1.j*soc*spin,1,1,lvec_list[j])

  return model

my_model=set_model(t,dt,soc)
my_model.display()

# first plot: compute band structure
# ----------------------------------

# construct path in k-space and solve model
path=[[0.,0.,0.],[0.,.5,.5],[0.25,.625,.625],
         [.5,.5,.5],[.75,.375,.375],[.5,0.,0.]]
label=(r'$\Gamma$',r'$X$',r'$U$',r'$L$',r'$K$',r'$L^\prime$')
(k_vec,k_dist,k_node)=my_model.k_path(path,101)

evals=my_model.solve_all(k_vec)

# band structure plot
fig, ax = plt.subplots(1,1,figsize=(4.,3.))
ax.set_xlim([0,k_node[-1]])
ax.set_xticks(k_node)
ax.set_xticklabels(label)
for n in range(len(k_node)):
  ax.axvline(x=k_node[n],linewidth=0.5, color='k')
ax.set_ylabel("Energy")
ax.set_ylim(-4.9,4.9)
for n in range(4):
  ax.plot(k_dist,evals[n],color='k')
fig.tight_layout()
fig.savefig("fkm_bsr.pdf")

# second plot: compute Wannier flow
# ---------------------------------

# initialize plot
fig, ax = plt.subplots(1,2,figsize=(5.4,2.6),sharey=True)

# Obtain eigenvectors on 2D grid on slices at fixed kappa_3
# Note physical (kappa_1,kappa_2,kappa_3) have python indices (0,1,2)
```

https://doi.org/10.1017/9781316662205.012 Published online by Cambridge University Press

```
kappa2_values=[0.,0.5]
labs=[r'$\kappa_3$=0',r'$\kappa_3$=$\pi$']
nk=41
dk=1./float(nk-1)
wf=wf_array(my_model,[nk,nk])

#loop over slices
for j in range(2):
  for k0 in range(nk):
    for k1 in range(nk):
      kvec=[k0*dk,k1*dk,kappa2_values[j]]
      (eval,evec)=my_model.solve_one(kvec,eig_vectors=True)
      wf[k0,k1]=evec
  wf.impose_pbc(mesh_dir=0,k_dir=0)
  wf.impose_pbc(mesh_dir=1,k_dir=1)
  hwfc=wf.berry_phase([0,1],dir=1,contin=True,berry_evals=True)/
    (2.*np.pi)

  ax[j].set_xlim([0.,1.])
  ax[j].set_xticks([0.,0.5,1.])
  ax[j].set_xlabel(r"$\kappa_1/2\pi$")
  ax[j].set_ylim(-0.5,1.5)
  for n in range(2):
    for shift in [-1.,0.,1.]:
      ax[j].plot(np.linspace(0.,1.,nk),hwfc[:,n]+shift,color='k')
  ax[j].text(0.08,1.20,labs[j],size=12.,bbox=dict(facecolor='w',
    edgecolor='k'))

ax[0].set_ylabel(r"HWF center $\bar{s}_2$")
fig.tight_layout()
plt.subplots_adjust(left=0.15,wspace=0.2)
fig.savefig("fkm_topo.pdf")
```

https://doi.org/10.1017/9781316662205.012 Published online by Cambridge University Press