Tutorial

The following tutorial will guide you through the basic steps of using the TBmodels package: Creating, evaluating and saving a `Model` instance.

Contents

Creating a Model instance

The `Model` class describes tight-binding models within TBmodels. Instances of this class can be created in various ways, some of which I will describe here.

Using Wannier90 output

First off, a model can be created from the output of a Wannier90 calculation. Note that `use_ws_distance = .true.` should be used (available since Wannier90 2.1) to get the correct tight-binding model. Wannier90 creates several files that can be read by TBmodels:

- The `*_hr.dat` file contains the hopping terms. Use `write_hr = .true.` to generate this file.
- The `*_wsvec.dat` file contains correction terms for the lattice vectors of the hopping terms.
- The `*_centres.xyz` file contains the Wannier centers. They can be used to determine the positions of the orbitals in TBmodels.
- The Wannier90 input file `*.win` can be used to determine the unit cell dimensions.

To generate a TBmodels `Model` instance, the `*_hr.dat` file is required, and the other files can be added optionally. However, it is recommended to at least also use the `*_wsvec.dat` file.

```
import tbmodels

# use only *_hr.dat
model = tbmodels.Model.from_wannier_files(hr_file='path_to_directory/wannier90_hr.dat')

# use all files
model = tbmodels.Model.from_wannier_files(
    hr_file='path_to_directory/wannier90_hr.dat',
    wsvec_file='path_to_directory/wannier90_wsvec.dat',
    xyz_file='path_to_directory/wannier90_centres.xyz',
    win_file='path_to_directory/wannier90.win'
)
```

Constructing models manually

Alternatively, a `Model` instance can be created directly using the constructor. The following example shows how to create a model with two orbitals. The orbitals have on-site energies `1` and `-1` (the unit can be arbitrary, but must be consistent), and there is one occupied state. The system is three-dimensional, and the orbitals are located at the origin and `[0.5, 0.5, 0.]` (in reduced coordinates), respectively.

In a second step, hopping terms between the two orbitals (nearest-neighbour interaction) and between the same orbital in different unit cells (next-nearest-neighbour) are added using the `add_hop()` method.

```
# (c) 2015-2018, ETH Zurich, Institut fuer Theoretische Physik
# Author: Dominik Gresch <greschd@gmx.ch>

import tbmodels
import itertools

model = tbmodels.Model(
    on_site=[1, -1], dim=3, occ=1, pos=[[0.0, 0.0, 0.0], [0.5, 0.5, 0.0]]
)

t1, t2 = (0.1, 0.2)
for phase, R in zip([1, -1j, 1j, -1], itertools.product([0, -1], [0, -1], [0])):
    model.add_hop(t1 * phase, 0, 1, R)

for R in ((r[0], r[1], 0) for r in itertools.permutations([0, 1])):
    model.add_hop(t2, 0, 0, R)
    model.add_hop(-t2, 1, 1, R)
```

Evaluating the model

Once created, a `Model` instance can be evaluated at different k-points in the Brillouin zone using the `hamilton()` and `eigenval()` methods. These methods take a single k-point, given in reduced coordinates, as argument.

```
print(model.hamilton(k=[0., 0., 0.]))
print(model.eigenval(k=[0., 0., 0.]))
```

Saving the model to a file

There are different ways of saving the model to a file. To save the model for later use, I recommend using the `to_hdf5_file()` method. This will preserve the model exactly as it is.

```
model.to_hdf5_file('model.hdf5')
model2 = tbmodels.Model.from_hdf5_file('model.hdf5') # model2 is an exact copy of model
```

If compatibility with other codes operating on Wannier90's `*hr.dat` format is needed, the `to_hr_file()` method can be used. However, this preserves only the hopping terms, not the positions of the atoms or shape of the unit cell. Also, the precision of the hopping terms is truncated.

```
model.to_hr_file('model_hr.dat')
model3 = tbmodels.Model.from_hr_file('model_hr.dat') # model3 might differ from model
```

Finally, the `Model` class is also compatible with Python's built-in `pickle` module. However, data saved with `pickle` may not be readable with different versions of TBmodels since pickle serialization depends on the specific names of classes and their attributes. The use of `pickle` compatibility is to enable `multiprocessing` with TBmodels.

Command-line interface

TBmodels has a built-in command line interface, which is designed to perform common operations, such as converting Wannier90 output into a TBmodels file. It can be accessed via the `tbmodels` command, and its documentation can be accessed with the `--help` flag or in the reference section. For example, `tbmodels --help` gives a list of possible commands, and `tbmodels parse --help` gives specific information about the `parse` command.

Note

The command line commands may change in future releases, because this feature is still in early development.