Symmetrization

In this short tutorial, we introduce the use of the symmetrization feature in TBmodels. The theoretical background for this feature is described in this paper (arxiv version). The example we describe here is to symmetrize a model of diamond silicon. The initial model, and code for this example, can also be found in the TBmodels source on GitHub.

We start by loading the initial model:

```
In [1]: import tbmodels
```

The model is located in the `examples` directory of the TBmodels source. You will have to change the following part to point to your examples directory:

```
In [2]: import os
   ...: import pathlib
   ...: EXAMPLES_DIR = pathlib.Path('../examples')
   ...:
```

```
In [3]: model_nosym =  tbmodels.io.load(
   ...:     EXAMPLES_DIR / 'symmetrization' / 'nonsymmorphic_Si' / 'data' /
'model_nosym.hdf5'
   ...: )
   ...:
```

Setting up orbitals

The most difficult step in using the symmetrization feature is setting up the symmetry operations. These need to be in the format of the `symmetry_representation` module, which is also part of the Z2Pack ecosystem.

Starting with the 0.2 release, `symmetry-representation` contains some handy helper functions which allow us to construct the symmetry operations automatically by specifying the orbitals which constitute the tight-binding model. Thus, the first step to obtain the symmetry operations is to define the orbitals.

We can see that our initial model has two positions, which we store in the `coords` variable:

```
In [4]: model_nosym.pos
Out[4]:
array([[0.5 , 0.5 , 0.5 ],
       [0.5 , 0.5 , 0.5 ],
       [0.5 , 0.5 , 0.5 ],
       [0.5 , 0.5 , 0.5 ],
       [0.75, 0.75, 0.75],
       [0.75, 0.75, 0.75],
       [0.75, 0.75, 0.75],
       [0.75, 0.75, 0.75],
       [0.5 , 0.5 , 0.5 ],
       [0.5 , 0.5 , 0.5 ],
       [0.5 , 0.5 , 0.5 ],
       [0.5 , 0.5 , 0.5 ],
       [0.75, 0.75, 0.75],
       [0.75, 0.75, 0.75],
       [0.75, 0.75, 0.75],
       [0.75, 0.75, 0.75]])

In [5]: coords = ((0.5, 0.5, 0.5), (0.75, 0.75, 0.75))
```

Since the initial model was calculated with Wannier90 and `sp3` projections, we know that the orbitals are ordered as follows:

- spin up
  - first coordinate $(0.5, 0.5, 0.5)$
    - sp^3 orbitals
  - second coordinate $(0.75, 0.75, 0.75)$
    - sp^3 orbitals
- spin down
  - first coordinate $(0.5, 0.5, 0.5)$
    - sp^3 orbitals
  - second coordinate $(0.75, 0.75, 0.75)$
    - sp^3 orbitals

Consequently, we construct a list of `symmetry_representation.Orbital` orbitals in this order:

```
In [6]: import symmetry_representation as sr

In [7]: orbitals = []

In [8]: for spin in (sr.SPIN_UP, sr.SPIN_DOWN):
   ...:     for pos in coords:
   ...:         for fct in sr.WANNIER_ORBITALS['sp3']:
   ...:             orbitals.append(sr.Orbital(
   ...:                 position=pos,
   ...:                 function_string=fct,
   ...:                 spin=spin
   ...:             ))
   ...:
```

Here we used constants defined by `symmetry_representation` to specify the spin up / down components, and the sp^3 orbitals in the order produced by Wannier90.

```
In [9]: sr.SPIN_UP, sr.SPIN_DOWN
Out[9]:
(Spin(total=Fraction(1, 2), z_component=Fraction(1, 2)),
 Spin(total=Fraction(1, 2), z_component=Fraction(-1, 2)))

In [10]: sr.WANNIER_ORBITALS['sp3']
Out[10]: ['1 + x + y + z', '1 + x - y  - z', '1 - x + y - z', '1 - x - y + z']
```

The `function_string` argument is a string which describes the orbital in terms of the cartesian coordinates `x`, `y` and `z`. The `symmetry-representation` code will use `sympy` to apply the symmetry operations to these functions and figure out which orbitals these are mapped to.

Creating symmetry operations

Having created the orbitals which describe our system, we can immediately generate the symmetry operation for time-reversal symmetry:

```
In [11]: time_reversal = sr.get_time_reversal(orbitals=orbitals, numeric=True)
```

Note that we use the `numeric=True` flag here. This keyword is used to switch between output using `numpy` arrays with numeric content, and `sympy` matrices with analytic content. Mixing these two formats is a bad idea, since basic operations between them don't work as one might expect. For the use in TBmodels, we can **always** choose the `numeric=True` option.

Next, we use `pymatgen` to determine the space group symmetries of our crystal:

```
In [12]: from pymatgen.core import Structure
    ....: from pymatgen.symmetry.analyzer import SpacegroupAnalyzer
    ....:

In [13]: structure = Structure(
    ....:     lattice=model_nosym.uc, species=['Si', 'Si'], coords=np.array(coords)
    ....: )
    ....:

In [14]: analyzer = SpacegroupAnalyzer(structure)

In [15]: sym_ops = analyzer.get_symmetry_operations(cartesian=False)

In [16]: sym_ops_cart = analyzer.get_symmetry_operations(cartesian=True)
```

Again, we can use a helper function from the `symmetry-representation` code to construct the symmetry operations automatically. Note that we need both the cartesian *and* the reduced symmetry operations:

```
In [17]: symmetries = []

In [18]: for sym, sym_cart in zip(sym_ops, sym_ops_cart):
   ....:     symmetries.append(sr.SymmetryOperation.from_orbitals(
   ....:         orbitals=orbitals,
   ....:         real_space_operator=sr.RealSpaceOperator.from_pymatgen(sym),
   ....:         rotation_matrix_cartesian=sym_cart.rotation_matrix,
   ....:         numeric=True
   ....:     ))
   ....:
```

Applying the symmetries

Finally, the simple task of applying the symmetries to the initial tight-binding model remains. We first apply the time-reversal symmetry.

```
In [19]: model_tr = model_nosym.symmetrize([time_reversal])
```

Note that, unlike the space group symmetries, the time-reversal symmetry does not constitute a full group. As a result, TBmodels will apply not only time-reversal $\mathcal{T}$, but also $\mathcal{T}^2 = -\mathbb{1}$, $\mathcal{T}^3 = -\mathcal{T}$, and the identity. For the space group, this extra effort is not needed since we already have the full group. This can be specified with the `full_group=True` flag:

```
In [20]: model_sym = model_tr.symmetrize(symmetries, full_group=True)
```

By comparing eigenvalues, we can see for example that the symmetrized model is two-fold degenerate at the Γ point, while the initial model is not:

```
In [21]: model_nosym.eigenval((0, 0, 0))
Out[21]:
array([-5.74263139, -5.74262186,  6.21206569,  6.21207289,  6.25909606,
        6.25910078,  6.25910527,  6.25910906,  8.80451686,  8.80452023,
        8.83906923,  8.83907362,  8.83907975,  8.83908456,  9.56537577,
        9.56538949])

In [22]: model_sym.eigenval((0, 0, 0))
Out[22]:
array([-5.74262663, -5.74262663,  6.2129382 ,  6.2129382 ,  6.25823388,
        6.25823388,  6.25910279,  6.25910279,  8.80515697,  8.80515697,
        8.83843836,  8.83843836,  8.83907679,  8.83907679,  9.56538263,
        9.56538263])
```