

pyqula user guide

Contents

1	Introduction	2
2	Building a Hamiltonian	3
3	Operators	8
4	Non-Hermitian Hamiltonians	16
5	Band structures and density of states	21
6	Local density of states	25
7	Spectral functions and Fermi surfaces	28
8	Quasiparticle interference	32
9	Band unfolding	38
10	Altermagnets	42
11	Superconductivity	47
12	Mean-field Hubbard models	52
13	Exchange interactions	58
14	Pseudofermion mean field	63
15	Topological invariants	67
16	Quantum geometry and Berry curvature	77
17	Topology in real space	82
18	Entanglement	87
19	Optical conductivity	89
20	Charge and generic response functions	91
21	Spin susceptibility, RPA and magnons	95
22	Excitons	104
23	Quantum transport	111
24	Single defects in infinite systems	121

25 Kernel polynomial method	123
26 Wannierization	128
27 Twisted bilayer graphene structural relaxation	132
28 Classical spin models	134
29 Lattice gas models	136
30 Ising models	138
31 Parallelism and reproducibility	139
32 Running on a GPU	140
33 Files written to the working directory	140
34 Errors and unsupported inputs	141
35 Geometry functions and methods	142
36 Hamiltonian functions and methods	143
37 Other classes	167

1 Introduction

pyqula is a Python library for tight-binding models of electrons on lattices. It computes band structures, densities of states and spectral functions, self-consistent mean-field Hamiltonians for interacting electrons, superconductivity, topological invariants, response functions and quantum transport, and also classical spin and lattice-gas models.

This guide explains how to compute physical quantities. For each quantity it gives the Hamiltonian or the formula, its physical meaning, the code, and the keywords that matter. The code is on [GitHub](#), and the whole guide is also available as one [PDF](#).

1.1 Installation

```
pip install pyqula
```

pyqula is written in Python, with numba for the heavy loops, so it needs no compiler. To use the code of a clone of the repository, run `pip install -e .` from its root.

Note that `import pyqula` alone gives access to nothing. Always import the module you need, for example `from pyqula import geometry`.

1.2 A first calculation

Almost every calculation has four steps: we build a lattice, we generate its tight-binding Hamiltonian, we add terms to it, and we compute an observable.

```
from pyqula import geometry
g = geometry.honeycomb_lattice() # 1. a lattice
h = g.get_hamiltonian()          # 2. its tight-binding Hamiltonian
h.add_zeeman([0.,0.,0.3])        # 3. add terms (in place)
(k,e) = h.get_bands()            # 4. compute an observable
```

Here the lattice is the honeycomb lattice of graphene, the Hamiltonian has first-neighbor hopping $t = 1$ and is spinful, and the Zeeman field $B_z = 0.3$ splits the two Dirac cones of the two spins by 0.6. `h.get_bands()` returns

the k-points along a path in the Brillouin zone and the energies at each of them. Energies are in units of the hopping t , and lengths in units of the first-neighbor distance.

Each code block in this guide has its own imports and runs alone, except when it continues the block just before it on the same page. The blocks stop at the computed arrays, and the plotting is left to you.

1.3 Where to find more

- `examples/` in the repository contains several hundred runnable scripts, ordered by dimension (0d/ 1d/ 2d/ 3d/) and by topic (`transport/`, `embedding/`, `wannier/`, `classicalspin/`, `latticegas/` and others). Most of them end with a figure.
- `jupyter-notebooks/functionalities/` contains one executed notebook for each functionality listed in the README, with the physics and the plots. Look there to see what a result should look like.
- The Reference tab lists the methods of **Geometry** and **Hamiltonian** and their arguments.

2 Building a Hamiltonian

Every calculation in pyqula follows the same steps. We build a geometry, we generate a tight-binding Hamiltonian from it, we add terms to this Hamiltonian, and we compute a quantity from it. In this page we see the first two steps, and the simplest terms: longer-range hopping, an onsite energy, a Zeeman field, an orbital magnetic field and a filling.

2.1 The first-neighbor chain

The simplest tight-binding model is a one-dimensional chain with hopping between first neighbors,

$$H = t \sum_n \left(c_n^\dagger c_{n+1} + c_{n+1}^\dagger c_n \right),$$

where $c_n^\dagger$ creates an electron at site n . The term $c_n^\dagger c_{n+1}$ moves an electron from site $n + 1$ to site n , and the second term moves it back. In pyqula the hopping is the unit of energy, $t = 1$, and the distance between first neighbors is the unit of length. The Hamiltonian is diagonal in momentum,

$$H = \sum_k \epsilon_k c_k^\dagger c_k, \quad \epsilon_k = 2t \cos k,$$

so the chain has a single band of width $4t$. With pyqula:

```
from pyqula import geometry
g = geometry.chain() # geometry of the 1D chain
h = g.get_hamiltonian() # generate the Hamiltonian
(k,e) = h.get_bands() # compute band structure
```

The geometry `g` contains the positions of the sites and the lattice vectors. `g.get_hamiltonian()` generates the Hamiltonian with first-neighbor hopping $t = 1$. `h.get_bands()` returns the k-points along a path and the energies at each k-point, here the cosine band $\epsilon_k = 2 \cos k$.

The Hamiltonian is spinful by default, so every band appears twice, once for each spin. A spinless Hamiltonian is generated with `g.get_hamiltonian(has_spin=False)`. All the methods that add a term are called `h.add_...()`, and they change `h` in place, so a calculation is a sequence of calls on the same object. If you want to keep the original Hamiltonian, make a copy first with `h.copy()`.

The computed energies (dots) lie on the cosine $2t \cos k$ (dashed line). The band goes from $-2t$ at $k = \pi$ to $2t$ at $k = 0$, so its width is $4t$.

The code of this figure is in `documentation/figures/hamiltonian_chain.py`.

See `examples/1d/linear_chain/main.py` for a runnable script, and `jupyter-notebooks/functionalities/single_particle` for spinless, spinful and Nambu Hamiltonians in zero to three dimensions.

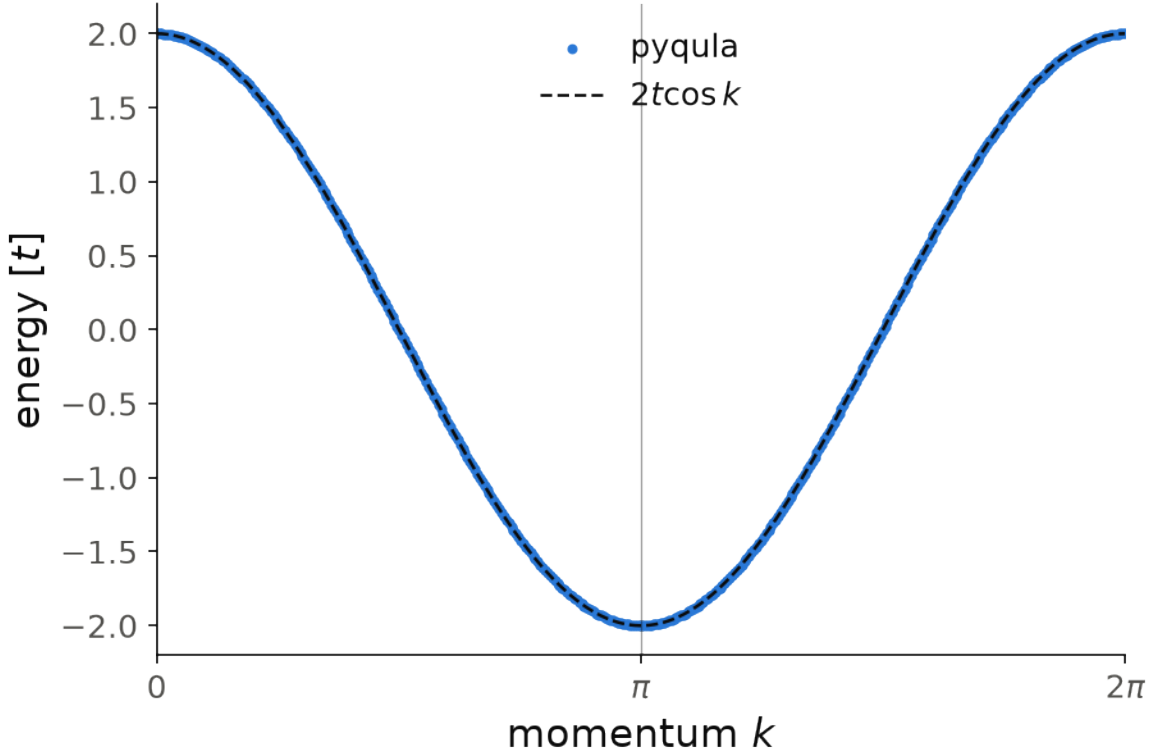


Figure 1: Band structure of the first-neighbor chain, with the cosine dispersion on top.

2.2 Second and third neighbor hopping

We now add hopping to sites further away. The Hamiltonian is

$$H = \sum_n c_n^\dagger c_{n+1} + t_2 \sum_n c_n^\dagger c_{n+2} + t_3 \sum_n c_n^\dagger c_{n+3} + h.c.,$$

where t_2 and t_3 are the second- and third-neighbor hoppings, in units of the first-neighbor one. The list of hoppings is given to `g.get_hamiltonian()`, first neighbor first. For $t_2 = 0.2$ and $t_3 = 0.3$:

```
from pyqula import geometry
g = geometry.chain() # geometry of the 1D chain
h = g.get_hamiltonian(tij=[1.0,0.2,0.3]) # Hamiltonian with t1,t2,t3
(k,e) = h.get_bands() # compute band structure
```

The dispersion is now

$$\epsilon_k = 2 \cos k + 2t_2 \cos 2k + 2t_3 \cos 3k.$$

In the band structure you see that the band is not symmetric anymore between positive and negative energies. The reason is the second-neighbor hopping: it connects two sites of the same sublattice, so it breaks the chiral (sublattice) symmetry $\epsilon_k \rightarrow -\epsilon_{k+\pi}$, while the first- and third-neighbor hoppings keep it. The same list works for any lattice. Instead of a list, `tij` can also be a function of the two positions; see `g.get_hamiltonian()` in the Geometry reference.

With $t_2 = 0.2$ and $t_3 = 0.3$ the band goes up to $3t$ at $k = 0$ and down to $-2.2t$ at $k = \pi$, so it is not symmetric around zero energy anymore. The first-neighbor band (gray) is shown as a reference.

The code of this figure is in `documentation/figures/hamiltonian_longrange.py`.

See `examples/1d/NNN_chain/main.py` for a runnable script.

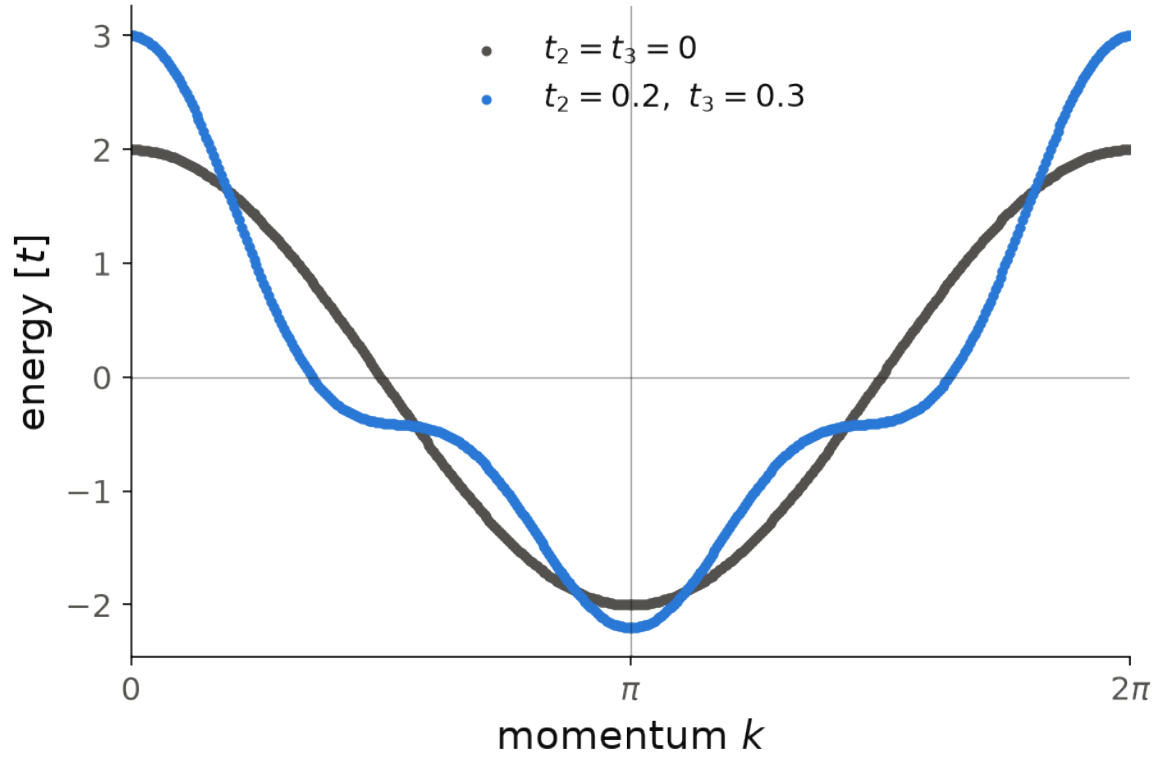


Figure 2: Band structure of the chain with first-, second- and third-neighbor hopping.

2.3 Onsite energy

An onsite energy changes the energy of an electron on a site, without moving it. With the same value μ on every site, it is a chemical potential,

$$H_\mu = \mu \sum_n c_n^\dagger c_n,$$

and it shifts the whole band structure rigidly by μ . All the observables in pyqula take zero energy as the Fermi energy, so adding a uniform onsite energy is the way to move the Fermi energy. It is added with `h.add_onsite()`:

```
from pyqula import geometry
g = geometry.chain() # geometry of the 1D chain
h = g.get_hamiltonian() # generate the Hamiltonian
mu = 0.3 # value of the onsite
h.add_onsite(mu) # add onsite energy
```

The argument can be

- a float: the same onsite energy on every site.
- a list or array: one onsite energy per site of the geometry.
- a function of the position $\mathbf{r}$: the onsite energy of each site is the function evaluated at its position.

A site-dependent onsite energy is how we build a potential: a sublattice imbalance in the honeycomb lattice, which opens a gap at the Dirac points; a single impurity, with a large onsite energy on one site (see Quasiparticle interference); or a smooth electrostatic potential from a gate.

2.4 Zeeman field

We now add an external magnetic field that acts on the spin of the electrons, with `h.add_zeeman()`. The Hamiltonian is $H = H_0 + H_Z$, with

$$H_0 = \sum_{n,s} c_{n,s}^\dagger c_{n+1,s} + h.c., \quad H_Z = \sum_{n,s,s'} \vec{B} \cdot \vec{\sigma}_{s,s'} c_{n,s}^\dagger c_{n,s'},$$

where n runs over the sites, s, s' over the spin, $\vec{B} = (B_x, B_y, B_z)$ is the field in units of energy, and $\vec{\sigma}$ are the Pauli matrices. The field splits each band into two spin bands separated by $2|\vec{B}|$, with the spin parallel and antiparallel to $\vec{B}$. For $\vec{B} = (0.1, 0.2, 0.3)$:

```
from pyqula import geometry
g = geometry.chain() # geometry of the 1D chain
h = g.get_hamiltonian() # generate the Hamiltonian
h.add_zeeman([0.1, 0.2, 0.3]) # add the Zeeman field (modifies h in place)
(k, e) = h.get_bands() # compute band structure
```

In the band structure, the two copies of the cosine band are now shifted by $\pm|\vec{B}| = \pm 0.37$. If the Hamiltonian is spinless, the call first makes it spinful. The field can also change from site to site: give one vector per site, or a function of the position.

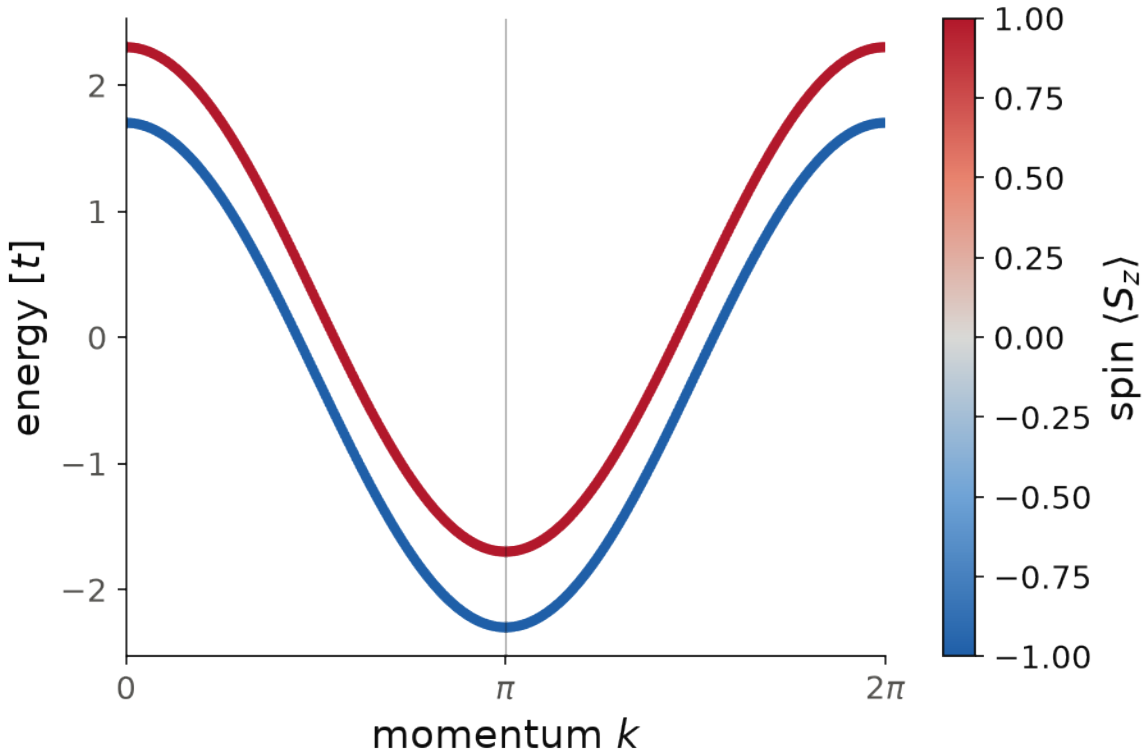


Figure 3: Band structure of the chain with a Zeeman field along z , colored by the spin of each state.

Here the field is along z , $\vec{B} = (0, 0, 0.3)$, so that the color can show $\langle S_z \rangle$ of each state. The two spin bands are rigid copies of the cosine band, shifted by ± 0.3 . The spin parallel to the field is the higher band, because the term is $+\vec{B} \cdot \vec{\sigma}$.

The code of this figure is in `documentation/figures/hamiltonian_zeeman.py`.

`h.add_exchange()` adds exactly the same term. We use the name Zeeman when the field is external, and exchange when it is the magnetization of the material itself, for example the one we obtain in Mean-field Hubbard models.

2.5 Orbital magnetic field

A magnetic field also acts on the orbital motion of the electrons. In a tight-binding model this enters as a phase in the hopping, the Peierls substitution,

$$t_{\alpha\beta} \rightarrow t_{\alpha\beta} e^{i \int_{\mathbf{r}_\alpha}^{\mathbf{r}_\beta} \vec{A} \cdot d\vec{l}},$$

where $\vec{A}$ is the vector potential, $\vec{B} = \nabla \times \vec{A}$, and the integral goes along the bond. An electron that goes around a closed loop picks up a phase $2\pi\Phi/\Phi_0$, with Φ the flux through the loop and Φ_0 the flux quantum. We apply it to a ribbon, a system finite in one direction and periodic in the other, with `h.add_orbital_magnetic_field()`:

```
from pyqula import geometry
N = 20 # number of unit cells as the width
g = geometry.square_ribbon(N) # ribbon
h = g.get_hamiltonian() # generate the Hamiltonian
B = 0.02 # magnetic field in quantum flux unit
h.add_orbital_magnetic_field(B) # add an out-of plane magnetic field
(k,e) = h.get_bands() # compute the Landau-level band structure
```

B is the flux through a plaquette of unit area, in units of Φ_0 , so $B = 0.02$ is one flux quantum every 50 plaquettes. The band structure shows the quantum Hall effect. There are flat bands, the Landau levels, and the lowest one is close to $-4 + 2\pi B$ for the square lattice, since near the band bottom the electrons behave as free electrons with cyclotron energy $4\pi B$. Between the Landau levels there are dispersive bands that cross the gap: these are the chiral edge states, one at each edge of the ribbon, moving in opposite directions. Each of them carries one quantum of conductance e^2/h , which is what a Hall measurement sees. For a honeycomb ribbon the same call gives the Landau levels of a Dirac equation, with one level at zero energy.

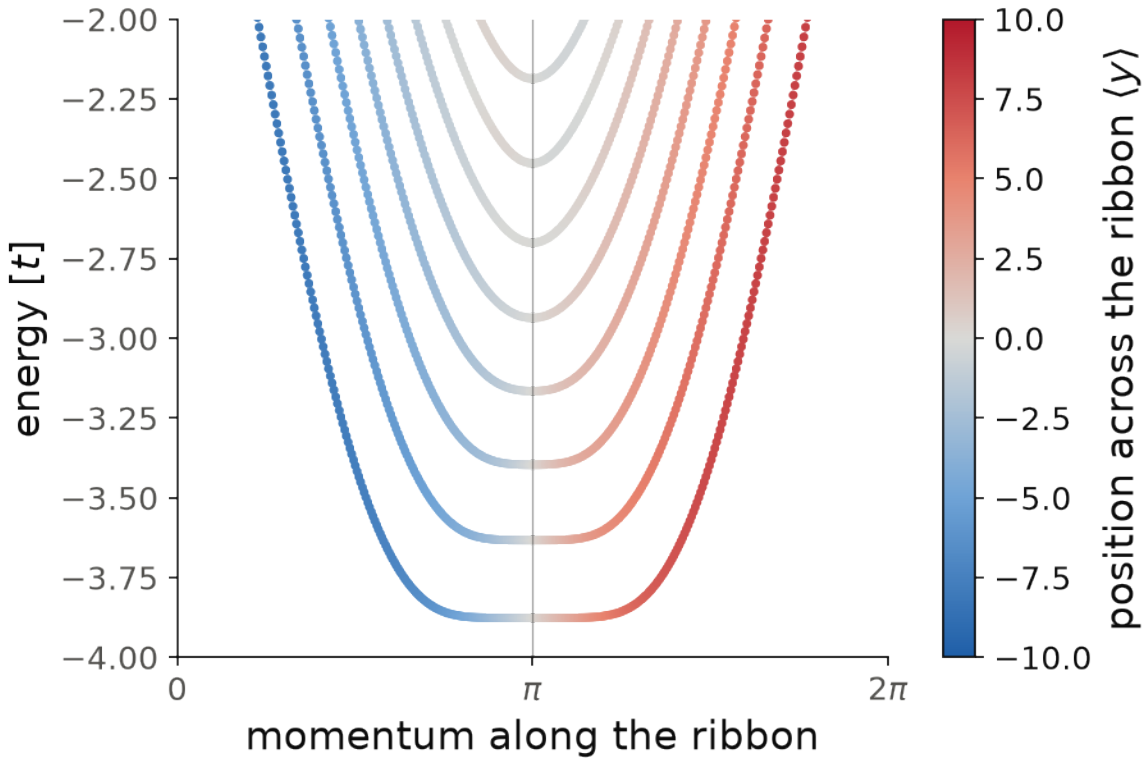


Figure 4: Bands of a square ribbon in a magnetic field, colored by the position of each state across the ribbon.

The color is the position of each state across the ribbon, computed with `h.get_bands(operator="yposition")`. In the middle of the k -path the bands are flat and white: these are the Landau levels, in the bulk of the ribbon.

The lowest one is at -3.87 , close to $-4 + 2\pi B = -3.87$. Away from the middle the levels disperse and turn red or blue: they become edge states, localized at one edge or the other, and moving in opposite directions.

The code of this figure is in `documentation/figures/hamiltonian_landau.py`.

See `examples/1d/landau_levels_zigzag_ribbon/main.py` for a runnable script on a honeycomb ribbon.

2.6 Filling

The filling ν is the fraction of the states that are occupied,

$$\nu = \frac{1}{N} \sum_n \langle c_n^\dagger c_n \rangle,$$

where the sum goes over all the N states of the unit cell (sites, and spins in a spinful system). Half filling is $\nu = 0.5$, and for the first-neighbor chain it is the case of the Fermi energy at zero. To fix another filling we use `h.set_filling()`:

```
from pyqula import geometry
g = geometry.chain() # chain
h = g.get_hamiltonian()
h.set_filling(0.7) # enforce a filling
```

`h.set_filling()` finds the Fermi energy E_F that gives this filling, using a k-mesh of `nk` points per direction, and then adds the onsite energy $-E_F$. After this, the Fermi energy is at zero, and every observable is computed at the filling you asked for. For a metal the k-mesh matters, because the filling changes continuously with E_F . For an insulator any E_F inside the gap gives the same filling. The same keyword `filling` appears again in `h.get_mean_field_hamiltonian()`, where the filling is kept fixed during the self-consistent loop (see Mean-field Hubbard models).

Sometimes we want a different filling on each site, for example a charge imbalance between two sublattices, created by a gate or a substrate. One Fermi energy cannot do this. Instead, we give one filling per site, and `h.set_filling()` finds one onsite energy per site so that each site has its own filling:

```
import numpy as np
from pyqula import geometry
g = geometry.honeycomb_lattice() # honeycomb lattice, two sites per unit cell
h = g.get_hamiltonian() # first-neighbor Hamiltonian
h.set_filling(np.array([0.4, 0.6])) # filling 0.4 on sublattice A, 0.6 on sublattice B
occ = h.get_vev() # occupation of each site
```

Each number has the same meaning as before, the fraction of the states of that site that are occupied. In this spinful example a site has two states, so `occ` is roughly 0.8 and 1.2 electrons, and the average filling is 0.5. A single number with `average=False` gives the same filling on every site, instead of only on average.

On a finite island the levels are discrete and the number of electrons is an integer. A set of fillings that gives a fractional number of electrons is only reached because of the broadening $\delta = 0.01$ that the solver uses, so in this case the occupations should be read with `h.get_vev(delta=1e-2)`.

3 Operators

An operator lets us ask a question about each eigenstate: which spin it has, where it is located, in which valley it is. In this page we see the operators that pyqula provides, their definition and their physical meaning.

All of them are obtained with `h.get_operator(name)`, and they are used through the `operator=` argument of `h.get_bands()`, `h.get_dos()`, `h.get_ldos()`, `h.get_kdos_bands()` and `h.get_vev()`. For an eigenstate $|\Psi_n(\mathbf{k})\rangle$ the quantity computed is the expectation value

$$O_n(\mathbf{k}) = \langle \Psi_n(\mathbf{k}) | O | \Psi_n(\mathbf{k}) \rangle.$$

With `h.get_bands()` this gives one extra array, the value $O_n(\mathbf{k})$ of each state, so you can color the band structure with it. With `h.get_dos()` each state enters the density of states with weight $O_n(\mathbf{k})$, so you get, for example, the spin-up minus spin-down density of states. Operators can also be added and multiplied, and some of them depend on $\mathbf{k}$ or are not even linear, as we see below.

See the executed notebooks `jupyter-notebooks/functionalities/single_particle_hamiltonians/04_state_resolved_b` and `06_operator_resolved_dos.ipynb` for operator-resolved band structures and densities of states.

3.1 Spin operators

The simplest operators are the spin operators,

$$S_\alpha = \sum_n \sum_{\mu\nu} \sigma_\alpha^{\mu\nu} c_{n,\mu}^\dagger c_{n,\nu}, \quad \alpha = x, y, z,$$

where n runs over the sites, μ, ν over the two spin states, and σ_α are the Pauli matrices. Note that there is no factor $1/2$, so the expectation value goes from -1 to $+1$. They need a spinful Hamiltonian.

```
from pyqula import geometry
g = geometry.triangular_lattice() # get the geometry
h = g.get_hamiltonian() # get the Hamiltonian
sx = h.get_operator("sx") # Spin x component
sy = h.get_operator("sy") # Spin y component
sz = h.get_operator("sz") # Spin z component
```

With `h.get_bands(operator=sz)` each band carries its spin projection. In a Zeeman field along z the two spin bands get $+1$ and -1 ; with spin-orbit coupling the spin changes along the band. The names also work as strings, `operator="sz"`. In the same way you can resolve by spin the density of states, the Fermi surface and the momentum-resolved spectral function, which is what spin-resolved ARPES measures.

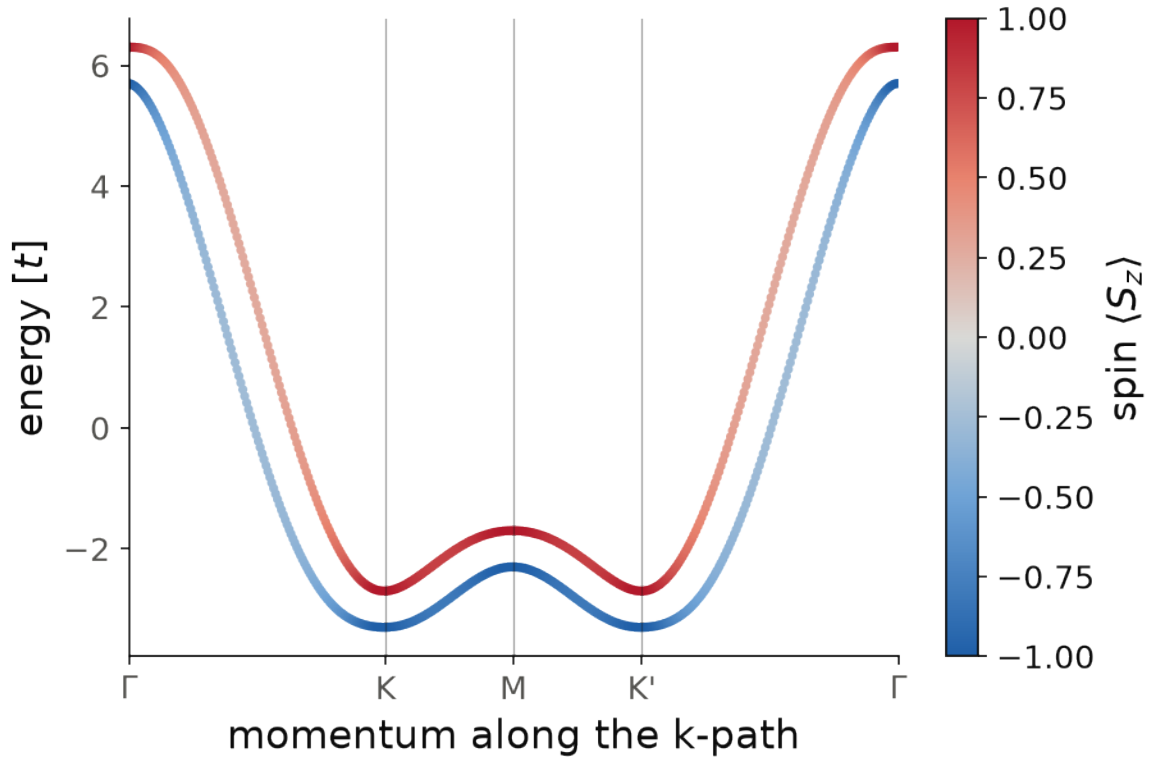


Figure 5: Bands of the triangular lattice with Rashba coupling and a Zeeman field, colored by the spin of each state.

To see the spin change along a band, we add to the snippet a Rashba coupling 0.3 and a Zeeman field 0.3 along z , and we color the bands with the expectation value of $\mathbf{sz}$. At Γ the two bands are split by $0.6 = 2 \times 0.3$, and their spin is ± 1 , as in a pure Zeeman field. Between Γ and K the Rashba coupling turns the spin into the plane, and $|\langle S_z \rangle|$ goes down to 0.27. At K , M and K' the two bands are again polarized along z , with $\langle S_z \rangle = \pm 1.00$.

The code of this figure is in `documentation/figures/operators_spin.py`.

See `examples/2d/operator_fermi_surface/main.py` for a spin-resolved Fermi surface, and `jupyter-notebooks/functionality` for a spin-resolved spectral function.

3.2 Listing the accepted names

The `operator` argument accepts an operator object, a matrix, a Hamiltonian, a function of the position, or a name. Other arguments are also chosen by a name: the initial guess of a mean-field calculation (`mf=`), the pairing symmetry (`mode=` of `h.add_pairing()`), the labels of the high-symmetry k-points, and the quantities of `h.extract()`. If you do not remember a name, you can print the list:

```
from pyqula import operatorlist, meanfield, extract
from pyqula.sctk import pairing
from pyqula.kpointstk import labels

print(operatorlist.get_operator_names()) # every name h.get_operator accepts
print(meanfield.get_guess_names())      # every mf= mean-field initialization
print(pairing.get_pairing_modes())      # every h.add_pairing(mode=...) symmetry
print(labels.get_label_names())         # every high-symmetry kpoint label
print(extract.get_extractable_names()) # every h.extract(...) quantity
```

A wrong name raises a `ValueError` that lists the accepted names.

3.3 Location operator

The position operators tell where a state is located in real space,

$$R_\alpha = \sum_{n,s} r_n^\alpha c_{n,s}^\dagger c_{n,s}, \quad \alpha = x, y, z,$$

where r_n^α is the α component of the position of site n , and s runs over the spin. The expectation value is the mean position of the state, $\langle R_\alpha \rangle = \sum_n r_n^\alpha |\Psi(n)|^2$.

```
from pyqula import geometry
g = geometry.triangular_lattice() # get the geometry
h = g.get_hamiltonian()           # get the Hamiltonian
x = h.get_operator("xposition")   # x component
y = h.get_operator("yposition")   # y component
z = h.get_operator("zposition")   # z component
```

In a ribbon periodic along x , the value of y says where across the width a state is. With `h.get_bands(operator=y)` the states at the two edges get opposite values, and the bulk states are close to the center. This is the continuous version of the bulk-edge operator below.

The triangular lattice of the snippet is periodic in both directions, so for the figure we use a zigzag honeycomb ribbon, with a Haldane coupling 0.1 to have states at the edges, and we color the bands with the expectation value of `yposition`. The ribbon goes from $y = -14.5$ to $y = 14.5$. The bulk bands are white, with $\langle y \rangle$ close to zero, and they have a gap between -0.48 and 0.48 . Inside the gap there are two bands that cross at $k = \pi$: the red one has $\langle y \rangle \approx 14$ and the blue one $\langle y \rangle \approx -14$, so they live at the two opposite edges. They also have opposite velocities: these are the chiral edge states of the Haldane model.

The code of this figure is in `documentation/figures/operators_position.py`.

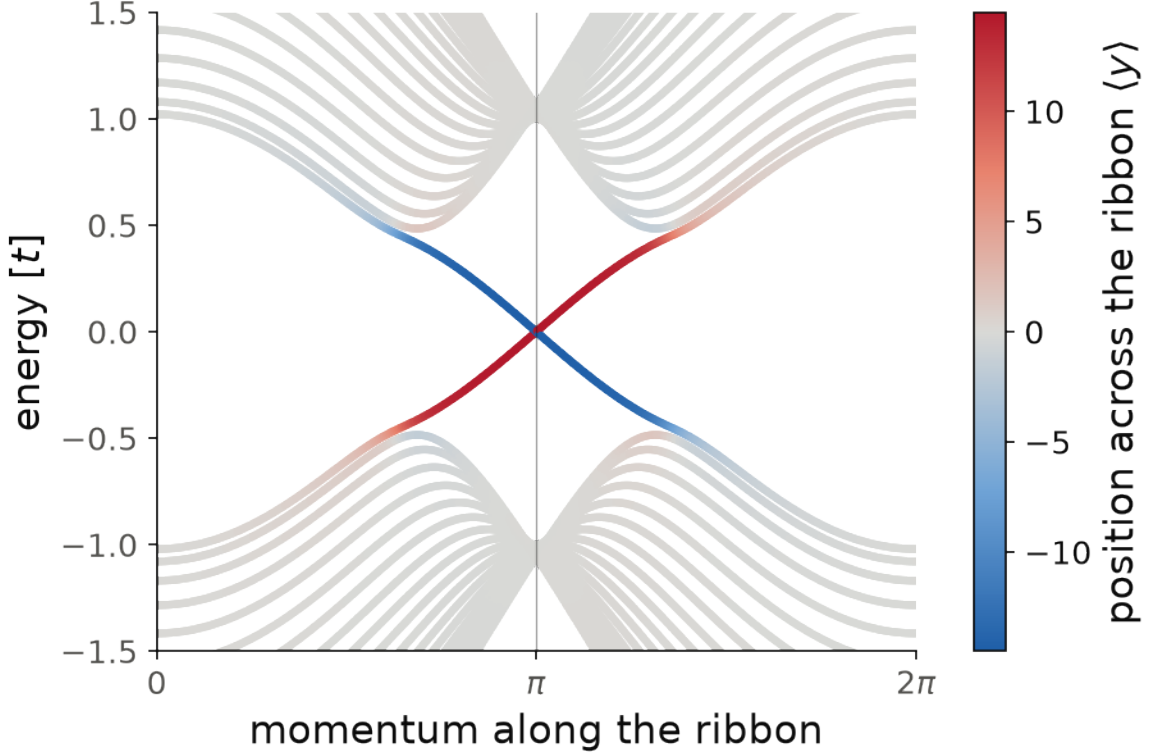


Figure 6: Bands of a zigzag honeycomb ribbon with a Haldane coupling, colored by the position of each state across the ribbon.

3.4 Bulk-edge operator

To know if a state sits at the edge or in the bulk, we use the edge and the bulk operators,

$$\hat{E} = \sum_{n \in \text{edge}, s} c_{n,s}^\dagger c_{n,s}, \quad \hat{B} = \sum_{n \in \text{bulk}, s} c_{n,s}^\dagger c_{n,s}, \quad \hat{E} + \hat{B} = 1.$$

The expectation value of $\hat{E}$ is the weight of the state on the edge sites, between 0 and 1.

```
from pyqula import geometry
g = geometry.honeycomb_zigzag_ribbon() # get the geometry
h = g.get_hamiltonian() # get the Hamiltonian
b = h.get_operator("bulk") # bulk operator
e = h.get_operator("edge") # edge operator
```

The edge sites are defined by their position. In a ribbon they are the sites in the outer 10% of the width on each side. In a finite island they are the sites in the outer part of the radius, and in a two-dimensional multilayer the outer layers along z . In the zigzag ribbon above, `h.get_bands(operator=e)` gives a weight close to 1 for the flat band at zero energy, the zigzag edge states, and a small weight for the dispersive bulk bands. So this is a quick way to find in-gap edge states in a band structure.

The flat band at zero energy is bright: its weight on the edge grows from about 0.5, where it joins the bulk bands, to 1.00 at $k = \pi$. The bulk bands are dark, with weights below 0.30. The edge sites are 20% of the ribbon, so a state spread over the whole ribbon has a weight close to 0.2, and this is the typical value of the bulk bands.

The code of this figure is in `documentation/figures/operators_edge.py`.

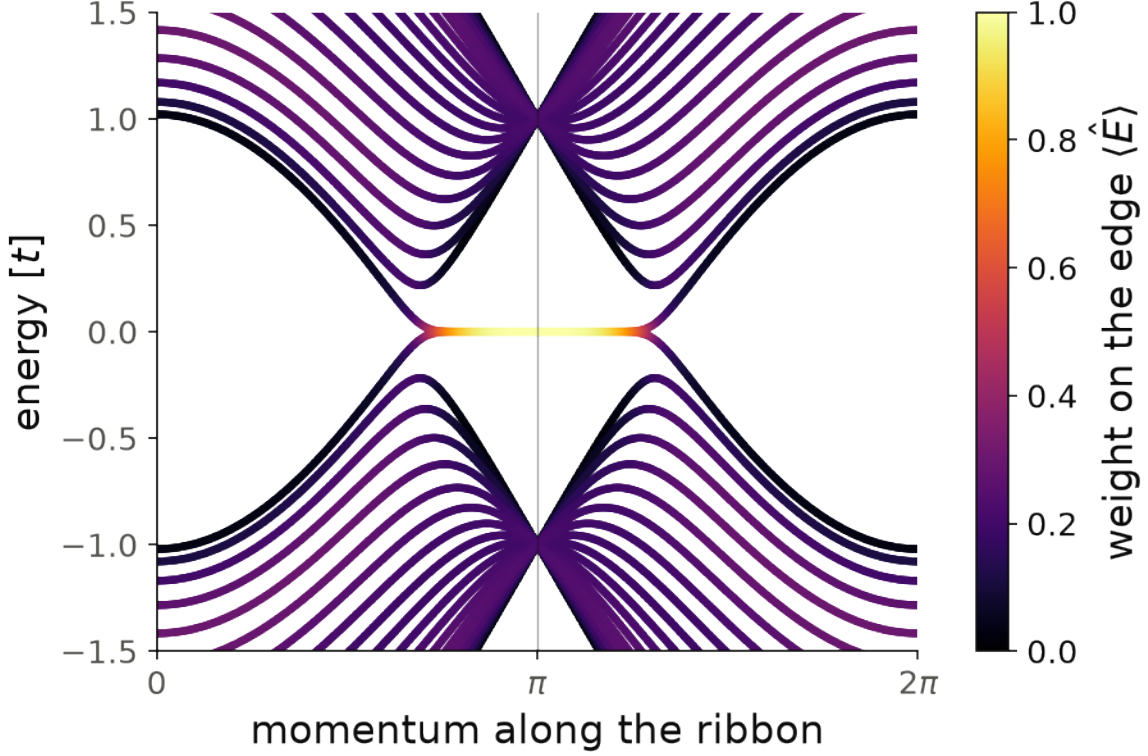


Figure 7: Bands of a zigzag honeycomb ribbon, colored by the weight of each state on the edge sites.

3.5 Valley operator

In honeycomb lattices, and also in aligned and twisted multilayers, the low-energy states sit in two valleys, around K and K' . The valley operator τ_z tells in which valley a state is. It is obtained with `h.get_operator("valley")`, and it is a second-neighbor hopping of the Haldane type,

$$\tau_z \propto i \sum_{\langle\langle ij \rangle\rangle, s} \nu_{ij} \eta_i c_{i,s}^\dagger c_{j,s},$$

where the sum goes over second neighbors, $\nu_{ij} = \pm 1$ for a clockwise or anticlockwise path from j to i , and $\eta_i = \pm 1$ for sublattice A or B. The factor η_i is what makes it different from the Haldane term. The Haldane term is a mass, with opposite sign on the two sublattices. This “modified Haldane” coupling has the same sign on both sublattices and opposite sign in the two valleys. The operator is normalized so that its expectation value is close to +1 in one valley and -1 in the other.

```
from pyqula import geometry
g = geometry.honeycomb_zigzag_ribbon() # get the geometry
h = g.get_hamiltonian() # get the Hamiltonian
vall = h.get_operator("valley") # valley operator
```

With `h.get_bands(operator=vall)` the states around K and around K' get opposite signs. This is most useful in a supercell or a moiré system, where the bands of the two valleys are folded on top of each other, and the valley operator is the way to separate them.

In the zigzag ribbon the two valleys are projected on two different momenta along the ribbon, one on each side of $k = \pi$. The bands on the left are blue and the ones on the right are red, with $\langle \tau_z \rangle$ up to ± 0.98 . The flat edge band goes from blue to red through white at $k = \pi$, where the edge state is an equal mix of the two valleys.

The code of this figure is in `documentation/figures/operators_valley.py`.

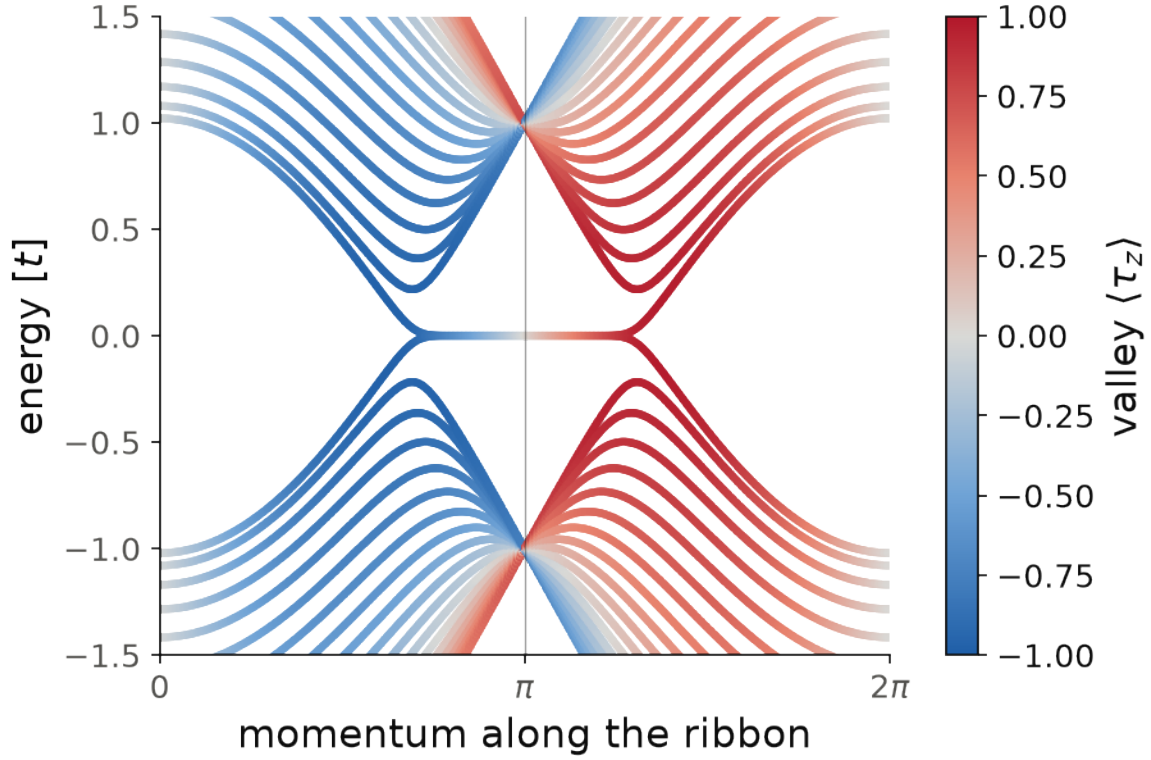


Figure 8: Bands of a zigzag honeycomb ribbon, colored by the valley of each state.

See `examples/1d/valley_operator/main.py` for the valley-resolved bands of a zigzag ribbon, `examples/2d/valley_bands/main.py` for a honeycomb supercell, and `jupyter-notebooks/functionalities/single_particle_hamiltonians/04_state_resolved` for more details.

3.6 In-plane valley operators

The valley operator above is the z component of a valley pseudospin. We now see the two other components, τ_x and τ_y , with `h.get_operator("valley_x")` and `h.get_operator("valley_y")`. Together they give the full valley vector (τ_x, τ_y, τ_z) , in the same way as (S_x, S_y, S_z) for the spin. While τ_z measures in which valley a state is, τ_x and τ_y measure the coherent mixing of the two valleys, the intervalley coherence, which is what a Kekulé bond order or an intervalley-coherent state has. They are built from a Kekulé coupling, which connects K and K' , averaged over the three Kekulé patterns so that the operator has the C_3 symmetry of the lattice around every site.

```
from pyqula import geometry
g = geometry.honeycomb_lattice().supercell(3) # Kekule-commensurate cell
h = g.get_hamiltonian(has_spin=False) # get the Hamiltonian
taux = h.get_operator("valley_x") # tau_x
tauy = h.get_operator("valley_y") # tau_y
```

Both need a honeycomb-like lattice with a sublattice index. In a periodic system the Kekulé pattern must fit in the unit cell, so we need a 3×3 supercell (or a multiple of 3) of the primitive cell. This is why we use `supercell(3)` above. A finite flake has no such condition.

A single vacancy in a honeycomb flake scatters electrons between the two valleys, and creates a vortex of the in-plane valley pseudospin around it. We compute (τ_x, τ_y) on every site:

```
from pyqula import islands
from pyqula import spectrum
g = islands.get_geometry(name="honeycomb", n=8, nedges=6)
gv = g.remove(g.get_central()[0]) # flake with a single vacancy
```

```

hv = gv.get_hamiltonian(has_spin=False)
dvx = spectrum.real_space_vev(hv,operator=hv.get_operator("valley_x"))
dvy = spectrum.real_space_vev(hv,operator=hv.get_operator("valley_y"))

```

`dvx` and `dvy` contain the expectation value of each component on every site, summed over the occupied states. If you draw (τ_x, τ_y) as an arrow on each site, the largest arrows, up to 0.29, are at the edges of the flake, because the zigzag edges have their own valley texture. Around the vacancy the arrows are up to 0.20, and they wind once: when we go counterclockwise around the vacancy, on a ring of radius 1 to 3, the arrow turns once clockwise, a winding number of -1 . To keep only the part created by the vacancy, we compute the same quantity on the pristine flake and subtract it, as `examples/0d/valley_vortex_vacancy/main.py` does. After this the arrows at the edges are below 0.002, and the winding -1 around the vacancy stays.

The valley vector can also enter the Hamiltonian. `h.add_valley_exchange(v)`, with $\vec{v} = (v_x, v_y, v_z)$, adds the term

$$H_v = \vec{v} \cdot \vec{\tau} = v_x \tau_x + v_y \tau_y + v_z \tau_z,$$

a valley Zeeman field, in the same way as `h.add_exchange()` for the spin:

```

from pyqula import geometry
g = geometry.honeycomb_lattice().supercell(3) # Kekule-commensurate cell
h = g.get_hamiltonian(has_spin=False)
h.add_valley_exchange([0.1,0.05,0.2]) # (vx,vy,vz)

```

See `examples/0d/valley_vortex_vacancy/main.py` and `examples/2d/valley_vortex/main.py` for runnable scripts of the vacancy vortex.

3.7 Nambu operators

In a superconductor the Hamiltonian is written in the Nambu basis, with an electron and a hole component for each site and spin (see Superconductivity). The operators "electron" and "hole" project on these two components,

$$P_e = \sum_{n,s} |n, s, e\rangle \langle n, s, e|, \quad P_h = \sum_{n,s} |n, s, h\rangle \langle n, s, h|, \quad P_e + P_h = 1.$$

The Hamiltonian must already be in the Nambu basis, so some pairing must be added first, or `h.setup_nambu_spinor()` must be called. Otherwise the hole projector raises an error.

```

from pyqula import geometry
g = geometry.triangular_lattice() # get the geometry
h = g.get_hamiltonian() # get the Hamiltonian
h.add_swave(0.2) # add pairing, doubling the basis into Nambu space
electron = h.get_operator("electron") # electron component
hole = h.get_operator("hole") # hole component
(k,e,c) = h.get_bands(operator=electron) # electron weight of each state

```

For a Bogoliubov quasiparticle $\gamma = u c + v c^\dagger$ the electron weight is $|u|^2$. Far from the Fermi energy a state is a pure electron ($|u|^2 = 1$) or a pure hole ($|u|^2 = 0$). Only close to the gap do the two mix, and exactly at the Fermi momentum $|u|^2 = |v|^2 = 1/2$. So the electron weight separates the physical band structure from its hole copy, and it is the weight used to obtain the electron spectral function from the full Bogoliubov-de Gennes spectrum.

The red band is the band of the triangular lattice, a pure electron with $|u|^2 = 1$, with its minimum -3 at K and K' . The blue band is its hole copy, the same band with the opposite energy, with $|u|^2 = 0$. They meet at the Fermi momentum, where the pairing opens a gap between -0.2 and 0.2 , so $2\Delta = 0.4$, and only there the color goes through white, $|u|^2 = 1/2$. We use `nk=1500` k-points instead of the default, because the band is steep where it crosses the Fermi energy.

The code of this figure is in `documentation/figures/operators_nambu.py`.

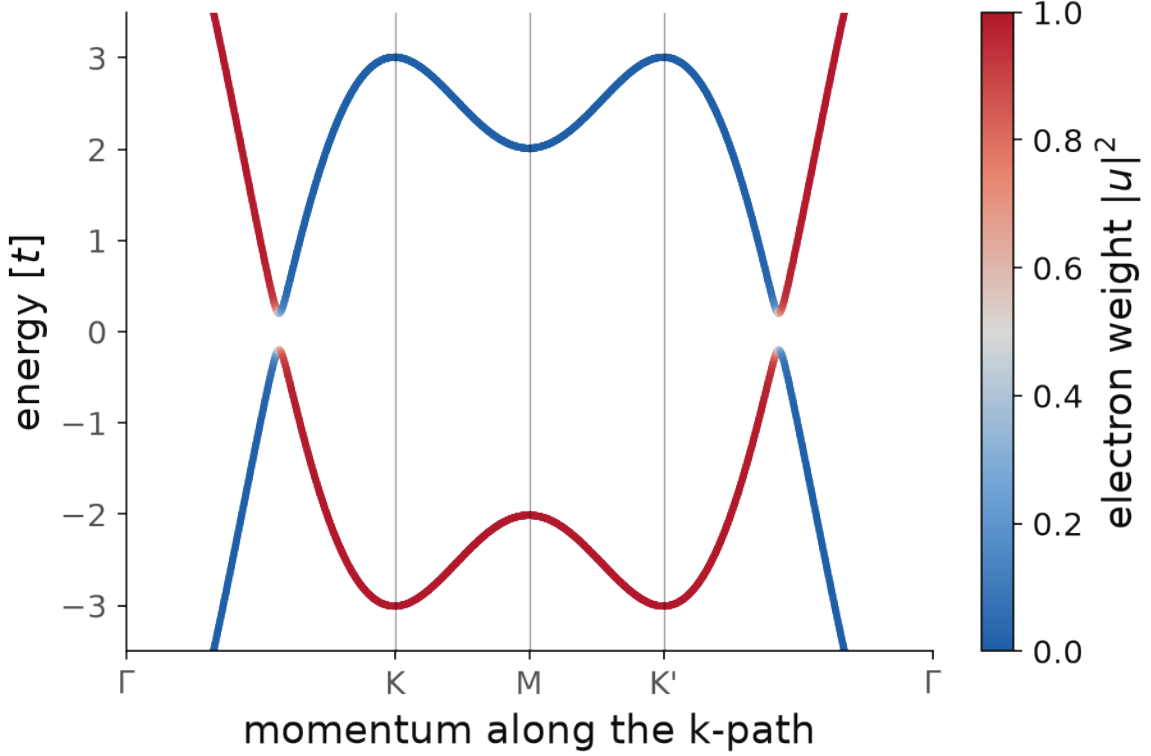


Figure 9: Bands of an s-wave superconductor on the triangular lattice, colored by the electron weight of each state.

3.8 Berry curvature operator

The Berry curvature operator is an example of an operator that depends on $\mathbf{k}$. It acts on an eigenstate as

$$O|\Psi_n(\mathbf{k})\rangle = \Omega(\mathbf{k}, \epsilon_n(\mathbf{k}))|\Psi_n(\mathbf{k})\rangle,$$

where $\Omega(\mathbf{k}, \omega)$ is the Berry curvature at momentum $\mathbf{k}$, computed from the Green's function at the energy $\omega = \epsilon_n(\mathbf{k})$ of the state, with a broadening `delta` (0.1 by default). It is obtained with `h.get_operator("berry")`, and it is defined only for two-dimensional Hamiltonians. With `h.get_bands(operator="berry")` you see which states of a Chern insulator carry the Berry curvature, usually the states close to the gap edge. The integral of the Berry curvature of the occupied states over the Brillouin zone is the Chern number (see Topological invariants).

Here we take the spinless honeycomb lattice with a Haldane coupling 0.1, `h.add_haldane(0.1)`, a Chern insulator with Chern number 1, and color its bands with `operator="berry"`. The Berry curvature is large only close to the gap edges at K and K' , at energies ± 0.45 , where it reaches ± 13.6 , and it is zero at Γ . It has the same sign at K and at K' , which is why the Chern number is not zero, and the opposite sign in the two bands.

The code of this figure is in `documentation/figures/operators_berry.py`.

3.9 Inverse participation ratio operator

All the operators above are linear, meaning that

$$A(|\Psi_1\rangle + |\Psi_2\rangle) = A|\Psi_1\rangle + A|\Psi_2\rangle.$$

The inverse participation ratio (IPR) is an operator that is not linear. It is defined as

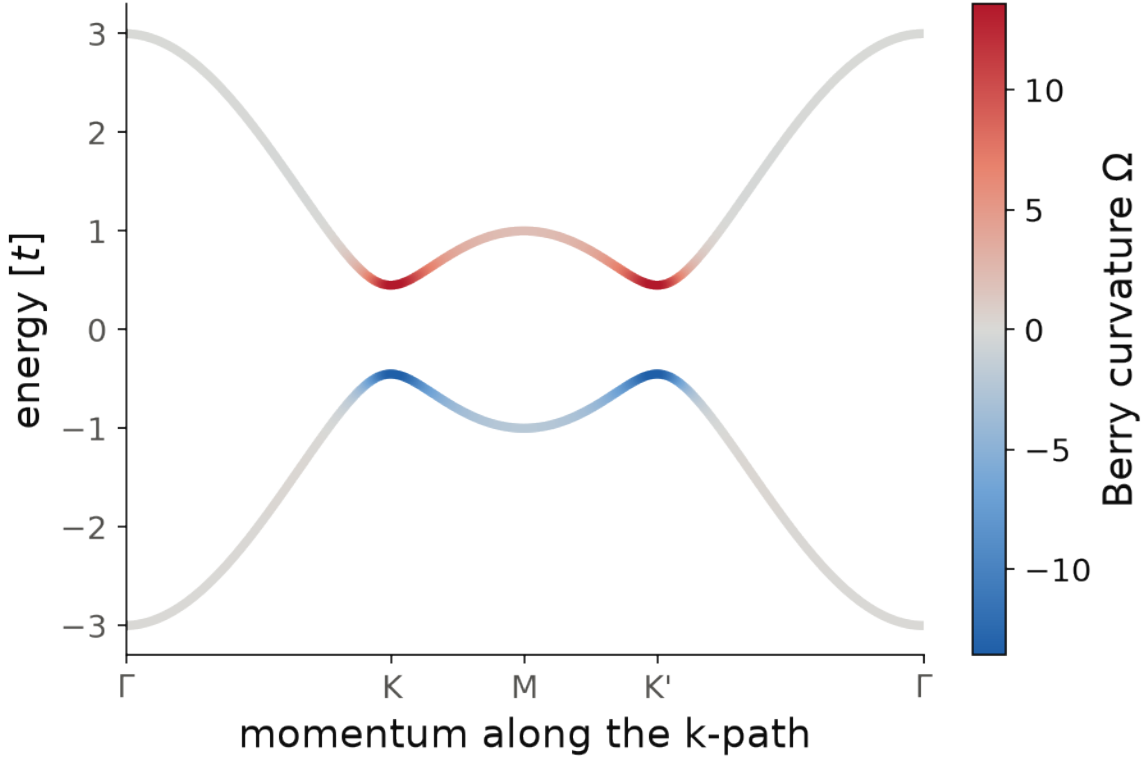


Figure 10: Bands of the Haldane model on the honeycomb lattice, colored by the Berry curvature of each state.

$$O|\Psi\rangle = \left(\sum_i |\langle i|\Psi\rangle|^4 \right) |\Psi\rangle,$$

where i runs over all the components of the state (sites and spin). Its expectation value $\sum_i |\langle i|\Psi\rangle|^4$ is 1 for a state on a single site, and $1/N$ for a state spread equally over N sites. So a large value means a localized state, for example an impurity state or an edge state.

```
from pyqula import geometry
g = geometry.honeycomb_zigzag_ribbon() # get the geometry
h = g.get_hamiltonian() # get the Hamiltonian
h.add_onsite(0.3) # add a uniform onsite energy
ipr = h.get_operator("IPR") # IPR operator
```

The onsite energy 0.3 only shifts the bands rigidly. In this zigzag ribbon, `h.get_bands(operator=ipr)` gives the flat edge states an IPR more than ten times larger than the bulk bands: close to the zone boundary the edge states live on a single site of the edge.

The color scale is logarithmic. The bulk bands have an IPR between 0.02 and 0.05, close to $1/40 = 0.025$, the value for a state spread over the 40 sites of the ribbon. The flat edge band at $E = 0.3$ goes up to 1.00 close to $k = \pi$. Some points of this band have 0.5 instead of 1: the two edge states at the same energy are degenerate, and the diagonalization can mix them into a state that lives on both edges.

The code of this figure is in `documentation/figures/operators_ipr.py`.

4 Non-Hermitian Hamiltonians

An open system exchanges particles or energy with an environment. Such a system is often described by an effective Hamiltonian that is not Hermitian. Gain and loss appear as imaginary onsite energies, $\epsilon_n \rightarrow \epsilon_n + i\gamma_n$, with $\gamma_n > 0$ for gain and $\gamma_n < 0$ for loss. A non-reciprocal hopping, $t_{ij} \neq t_{ji}^*$, appears as a hopping matrix that is

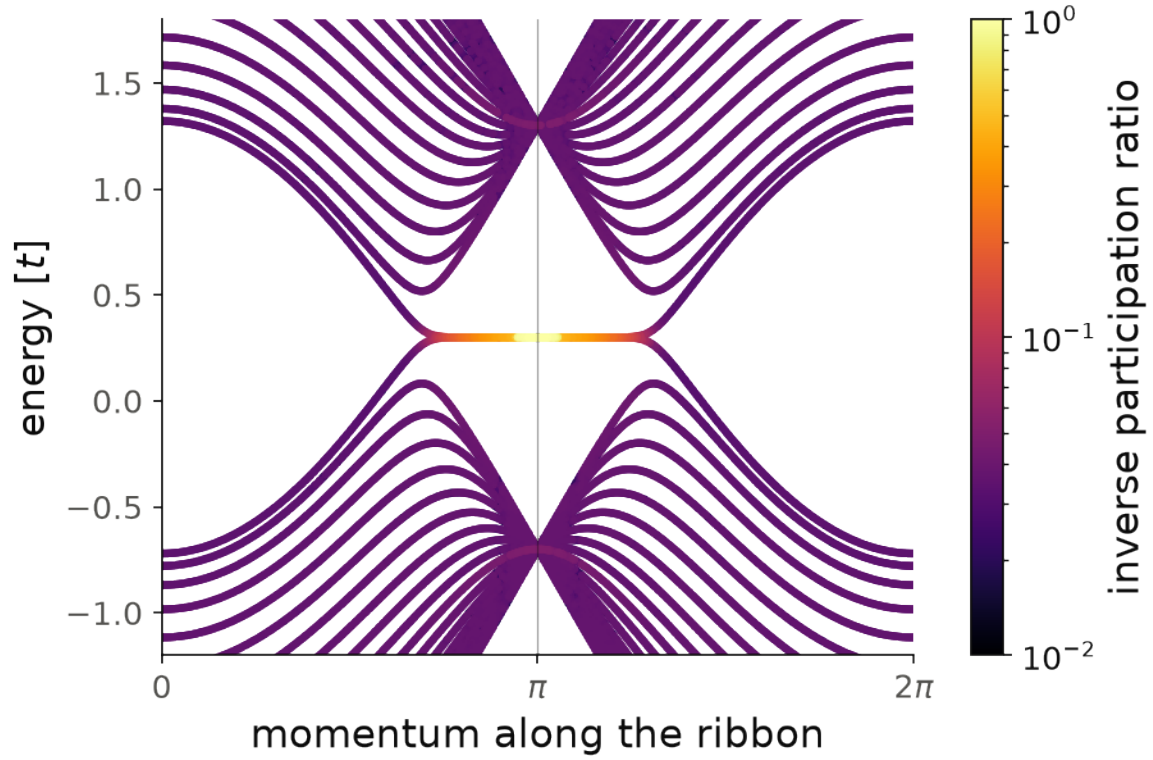


Figure 11: Bands of a zigzag honeycomb ribbon, colored by the inverse participation ratio of each state.

not Hermitian. The eigenvalues E_n are then complex. The real part is an energy, and the imaginary part is a rate: a state evolves as $e^{-iE_n t}$, so it grows if $\text{Im } E_n > 0$ and decays if $\text{Im } E_n < 0$. The eigenvectors are not orthogonal anymore. This is the physics of photonic and acoustic lattices with gain and loss, of the non-Hermitian skin effect, and of $\mathcal{PT}$ -symmetric models, and also of quasiparticles with a finite lifetime.

A non-Hermitian Hamiltonian is generated with `non_hermitian=True` in `g.get_hamiltonian()`. The model is then built as usual; for example, gain and loss are added with `h.add_onsite()` and a complex function. Let us take a chain of twenty sites with a purely imaginary modulation of the onsite energy,

$$H = - \sum_n (c_n^\dagger c_{n+1} + h.c.) + i 0.6 \sum_n \cos\left(\frac{2\pi x_n}{20}\right) c_n^\dagger c_n,$$

with gain where the cosine is positive and loss where it is negative:

```
import numpy as np
from pyqula import geometry

n = 20
g = geometry.chain().get_supercell(n)
h = g.get_hamiltonian(has_spin=False, non_hermitian=True, tij=[-1.])
# a purely imaginary Aubry-Andre modulation, with no Hermitian part at
# all: gain where the cosine is positive, loss where it is negative
h.add_onsite(lambda r: 0.6j*np.cos(2.*np.pi*r[0]/n))

(ks, es) = h.get_bands(kpath=[[0., 0., 0.]], write=False)
print("eigenvalues are complex:", es.dtype)
print("largest gain rate Im(E):", np.round(np.max(es.imag), 4))
```

The eigenvalues are complex: twenty of them, one per site. The real parts are inside the band of the chain,

between -2 and 2 . The imaginary parts are between -0.48 and $+0.48$, smaller than the 0.6 of the modulation, because the hopping mixes sites with gain and sites with loss. To plot them you have to choose: `es.real` for the energy, `es.imag` for the gain or decay rate, or both in the complex plane.

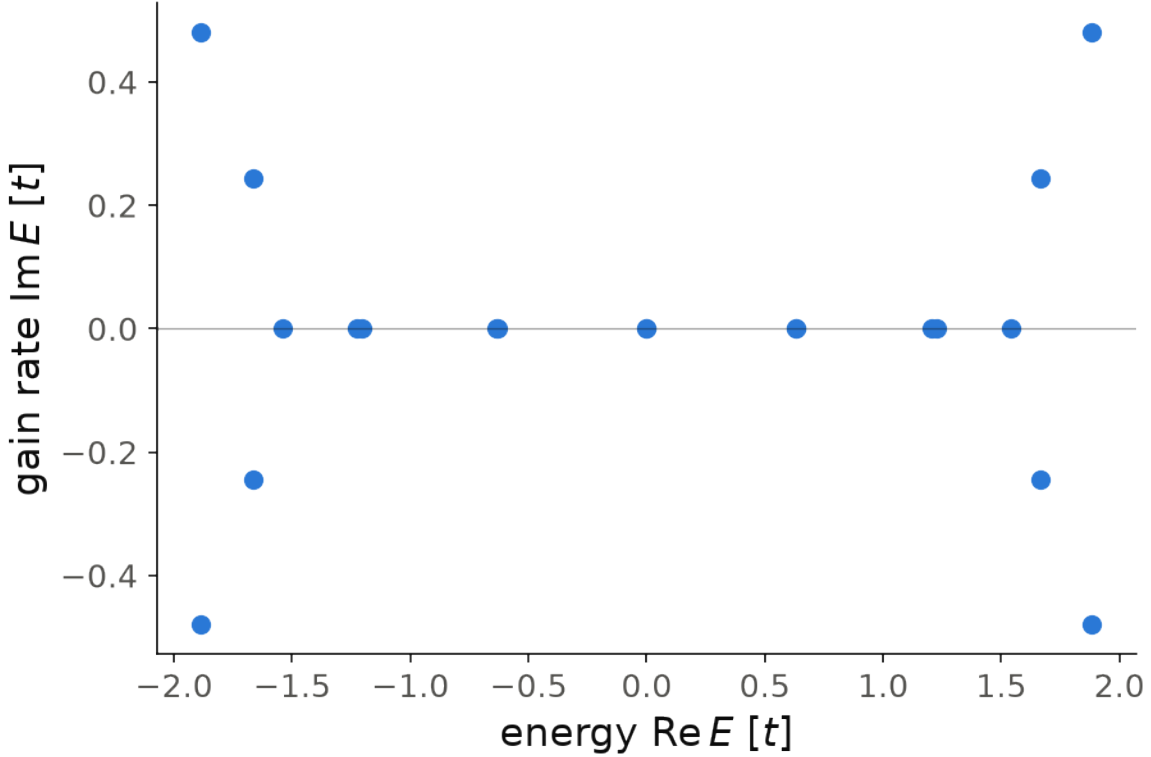


Figure 12: The twenty complex eigenvalues of the chain with gain and loss, in the complex plane.

Each dot is one eigenvalue, with the energy on the horizontal axis and the gain rate on the vertical one. Twelve of the twenty eigenvalues are real, between -1.54 and 1.54 : these states are spread equally over the gain and the loss, so they neither grow nor decay. The other eight are close to the band edges, at $|\text{Re } E|$ between 1.67 and 1.88 , and come in pairs E and E^* , with rates ± 0.25 and ± 0.48 . The spectrum is also symmetric under $E \rightarrow -E$.

The code of this figure is in `documentation/figures/nonhermitian_spectrum.py`.

4.1 Choosing which part is the energy: eigmode

The argument `eigmode` says which part of the complex eigenvalue is used as the energy. It can be

- "complex" (default): the whole complex eigenvalue.
- "real": the real part, the energy.
- "imag": the imaginary part, the gain or decay rate.

In `h.get_bands()` it decides what the returned energies contain. With "complex" the file `BANDS.OUT` has one more column, with the columns `k`, `Re E`, `Im E`, and then one column per operator.

In `h.get_ldos()` it decides on which axis the energy `e` is. With `eigmode="imag"` the states are selected by their gain rate and not by their energy. In this way we can ask where the most amplified state of the chain above is located:

```
import numpy as np
from pyqula import geometry
n = 20
g = geometry.chain().get_supercell(n)
h = g.get_hamiltonian(has_spin=False, non_hermitian=True, tij=[-1.])
```

```
h.add_onsite(lambda r: 0.6j*np.cos(2.*np.pi*r[0]/n))
(ks,es) = h.get_bands(kpath=[[0.,0.,0.]],write=False)
```

```
imax = np.argmax(es.imag) # the most amplified state
(x,y,d) = h.get_ldos(e=es[imax].imag,delta=1e-2,eigmode="imag",nrep=1)
print("it lives at x =",x[np.argmax(d)])
```

The answer is $x = -0.5$, the site where the gain is largest. As expected, the most amplified state lives where the imaginary onsite energy is largest.

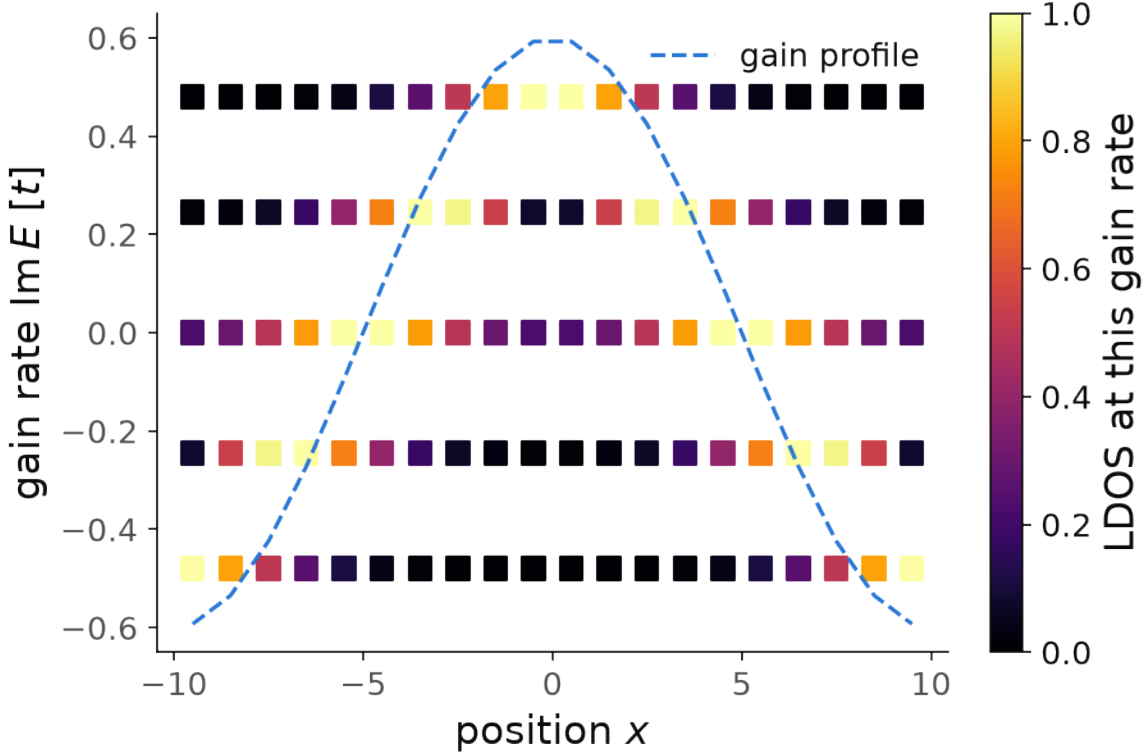


Figure 13: Local density of states of the chain with gain and loss, for each gain rate, as a function of the position.

Each row is the map d at one of the five gain rates of the spectrum, computed in the same way as in the snippet, and normalized to its largest value. The dashed line is the gain $0.6 \cos(2\pi x/20)$ of each site. The states with the largest gain, $\text{Im } E = 0.48$, live around $x = 0$, where the gain is largest, and the states with the largest loss, $\text{Im } E = -0.48$, live around $x = \pm 10$, where the loss is largest. The real states, $\text{Im } E = 0$, have their weight around $x = \pm 5$, where the onsite energy is zero.

The code of this figure is in `documentation/figures/nonhermitian_eigmode.py`.

`h.get_dos()` always broadens the real part when `eigmode` is "complex" or "real", so both give the same density of states. With "imag" it gives the distribution of the gain and decay rates.

4.2 The two spectral functions of a non-Hermitian Hamiltonian

We now see that a non-Hermitian Hamiltonian has two different momentum-resolved spectral functions, and how to compute each one with `h.get_kdos_bands()`. A non-Hermitian Hamiltonian has right eigenvectors $|R_n\rangle$ and left eigenvectors $\langle L_n|$,

$$H|R_n\rangle = E_n|R_n\rangle, \quad \langle L_n|H = E_n\langle L_n|,$$

and they are different. The weight of an operator O in a state can be defined with the right eigenvector alone, or with both:

$$O_n^R = \frac{\langle R_n | O | R_n \rangle}{\langle R_n | R_n \rangle}, \quad O_n^{LR} = \frac{\langle L_n | O | R_n \rangle}{\langle L_n | R_n \rangle}.$$

The first one is the default of `h.get_kdos_bands()`. Each state gives a Lorentzian of width `delta` at $\text{Re } E_n$, with a positive weight for a positive operator. This is the usual picture of a band structure.

The second one is what the Green's function gives,

$$A(\mathbf{k}, \omega) = -\frac{1}{\pi} \text{Im Tr} [O (\omega + i\delta - H_{\mathbf{k}})^{-1}] = -\frac{1}{\pi} \text{Im} \sum_n \frac{O_n^{LR}}{\omega + i\delta - E_n}.$$

Here a state with $\text{Im } E_n < 0$ is broadened by its own decay rate, its finite lifetime. This is how an effective non-Hermitian Hamiltonian describes quasiparticles that decay, and it is the spectral function that ARPES measures for such quasiparticles. It is computed with `biorthogonal=True`:

```
import numpy as np
from pyqula import geometry
g = geometry.chain().get_supercell(4, store_primal=True) # supercell of a chain
h = g.get_hamiltonian(has_spin=False, non_hermitian=True) # non-Hermitian Hamiltonian
h.add_onsite(lambda r: -0.3j*(1.+np.cos(np.pi*r[0]/2.))) # modulated loss
kpath = g.get_unfolded_kpath(nk=400) # primitive k-path, in supercell coordinates
(k,e,d) = h.get_kdos_bands(operator="unfold", kpath=kpath,
    biorthogonal=True) # unfolded Green's-function spectral function
```

Compared with the default, each unfolded band is broader and lower, because of the loss of its states. The two spectral functions are the same when the spectrum is real. Otherwise they differ in two ways. The Green's function one includes the lifetime of each state. And its weight can be negative for a state with gain, $\text{Im } E_n > 0$, whose pole is on the wrong side of the real axis. Close to an exceptional point, where two eigenvectors become the same, the biorthogonal weights of the two states become large and cancel each other, and only their sum has a meaning.

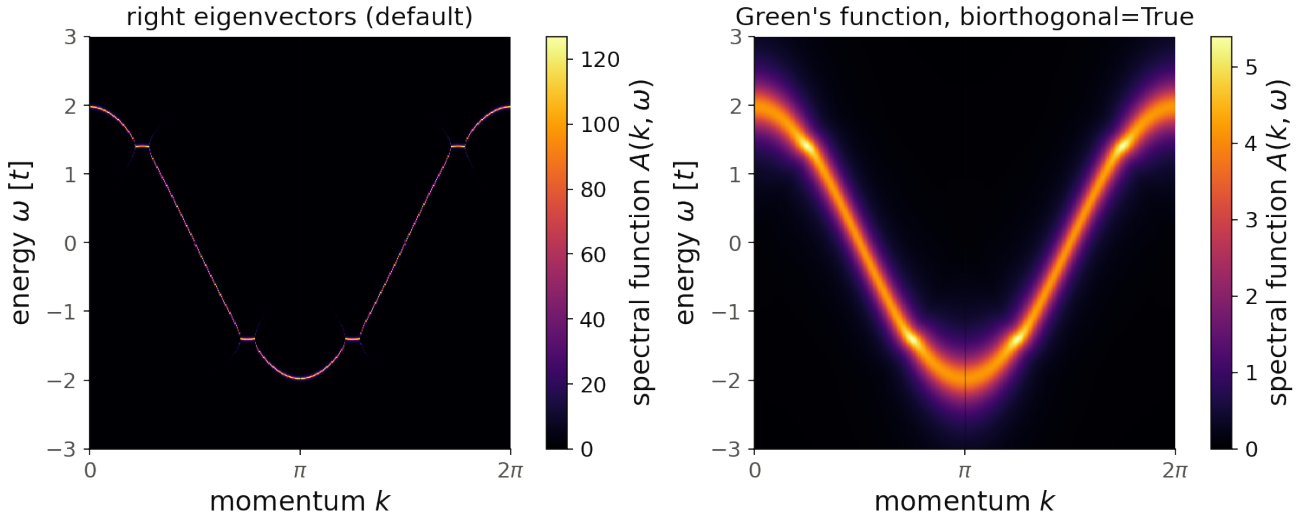


Figure 14: Unfolded spectral function of the chain with a modulated loss, with the right eigenvectors and with the Green's function.

For the figure we use a finer energy grid than the default, `energies=np.linspace(-3., 3., 1201)`, so that the sharp peaks of the left panel are not lost between two energies. The 400 points of the k -path are needed for the

same reason: with 100 points the band moves by more than δ from one k-point to the next, and the left panel shows a dotted line. In both panels you see the cosine band of the chain, from 2 at $k = 0$ to -2 at $k = \pi$. All the states have the same decay rate here, $\text{Im } E_n = -0.3$, the average loss of the four sites. In the left panel the peaks have the width $\delta = 0.01$ and a height of up to 127, so the band is a thin line. In the right panel each peak has a full width of 0.61, which is $2(0.3 + \delta)$, and a height of about 4.2, so the lifetime of the states is visible. Around $\omega = \pm 1.4$ two folded bands of the supercell meet, and there the weight goes up to 5.4.

The code of this figure is in `documentation/figures/nonhermitian_kdos.py`.

The same choice exists in other functions:

- `mode="green"` in `h.get_kdos_bands()` computes the Green's function spectral function directly, so it needs `biorthogonal=True` for a non-Hermitian Hamiltonian.
- `h.get_bands(operator=..., biorthogonal=True)` returns the weights O_n^{LR} state by state. They are complex in general, and at each k-point they add up to the trace of the operator.
- `h.get_dos(biorthogonal=True)` sums the Green's function spectral function over the Brillouin zone.

4.3 What is and is not available

These quantities have a real non-Hermitian version: the band structure, the density of states, the local density of states, the Berry curvature and the density matrix. The occupied states are those with the lowest real part of the energy. Each of them enters with its right and left eigenvectors, normalized so that $\langle L|R \rangle = 1$, and the density matrix is

$$\rho = \sum_{n \in \text{occ}} |R_n\rangle \langle L_n|.$$

An expectation value is then $\langle A \rangle = \text{Tr}(\rho A) = \sum_{\text{occ}} \langle L_n | A | R_n \rangle$, complex in general. `h.get_density_matrix()` gives this ρ , and `h.get_vew()` and `h.get_magnetization()` give the real part of the expectation value. The self-consistent mean field also uses this ρ , so the mean-field Hamiltonian is itself non-Hermitian.

A simple check is the open Hatano-Nelson chain, with hopping t_R to the right and t_L to the left. An imaginary gauge transformation turns it into a Hermitian chain with hopping $\sqrt{t_R t_L}$, without changing any density. So with a Hubbard U its self-consistent spectrum is the one of the Hermitian chain (see `jupyter-notebooks/functionalities/interacting_mean_field_hamiltonians/08_hermitian_nonhermitian.ipynb`).

This works with exact diagonalization, for a Hamiltonian without superconductivity. The kernel polynomial method, `use_jax=True` and `integration="qtci"` assume a real spectrum, and raise a `NotImplementedError`. In the same way, `h.get_dos()` accepts only `mode="ED"`. `h.get_ldos()` only works with its default mode, so do not pass another one. Other quantities use the Hermitian formula on a non-Hermitian Hamiltonian, which may or may not have a meaning for a complex spectrum, so check before you use them.

Operators work as usual, also `operator="unfold"`, so a supercell can be unfolded to the primitive Brillouin zone with a complex spectrum; see `examples/1d/unfolding_non_hermitian/main.py`. `num_bands` also works, and gives the few eigenvalues closest to `central_energy`.

See `examples/1d/NH_ldos/main.py` (the model above, state by state), `examples/0d/non_hermitian_aah/main.py` and `examples/0d/non_hermitian_aah_dos/main.py` (a non-Hermitian Aubry-André chain as a function of the phase of the modulation) for runnable scripts, and `jupyter-notebooks/functionalities/single_particle_hamiltonians/0` for an executed notebook.

5 Band structures and density of states

In this page we compute the two basic quantities of a periodic Hamiltonian: the band structure, the energies as a function of momentum, and the density of states, the number of states at each energy. The local, momentum-resolved and surface versions of the density of states are in `Local density of states` and `Spectral functions`.

5.1 Band structure

For a system that is periodic in space, the momentum k is a good quantum number, and the Hamiltonian is diagonal in the basis of the Bloch states,

$$H = \sum_{k,\alpha} \epsilon_{k,\alpha} \Psi_{k,\alpha}^\dagger \Psi_{k,\alpha},$$

where α is the band index and $\epsilon_{k,\alpha}$ is the energy of band α at momentum k . So at each k there is one energy per band. This is the band structure, the quantity that ARPES measures for the occupied states. We compute it with `h.get_bands()`:

```
from pyqula import geometry
g = geometry.honeycomb_lattice() # geometry of the 2D model
h = g.get_hamiltonian() # generate the Hamiltonian
(k,e) = h.get_bands() # compute band structure
```

`h.get_bands()` returns two arrays of the same length: the position along the k-path, and the energy. Each k-point appears once for each band, so the two arrays together are a scatter plot of the band structure. For the honeycomb lattice you see two bands that touch at zero energy at the K point, the Dirac point. Each band extends to $\pm 3t$.

The keywords are

- **kpath**: the k-path, as a list of high-symmetry points (["G","K","M"]) or of explicit k-vectors. By default, the standard path of the lattice.
- **nk**: number of k-points along the path.
- **operator**: an operator, or a list of operators, whose expectation value is computed in each eigenstate. The output is then (k,e,c).
- **num_bands**: compute only this many bands around **central_energy**, for large Hamiltonians.

With an operator A , the third array is $\langle \Psi_{k,\alpha} | A | \Psi_{k,\alpha} \rangle$ for each state, and it is the color of the bands in a plot. For example, the group velocity tells apart flat bands from dispersive ones, and the spin operator "sz" tells apart the two spin channels of a magnet, which we use in Mean-field Hubbard models:

```
from pyqula import geometry
g = geometry.honeycomb_lattice()
h = g.get_hamiltonian(has_spin=True)
(k,e,c) = h.get_bands(operator="velocity") # bands colored by group velocity
```

The two bands touch at zero energy at K and K', the Dirac points, and they reach $\pm 3t$ at Γ . The color is the group velocity. At the Dirac points it is $1.5ta/\hbar$, which is $3ta/2\hbar$, the Dirac velocity of graphene, with a the first-neighbor distance. It goes to zero at Γ and at M, where the bands are flat. The flat bands at M, at $\pm t$, are the saddle points that give the van Hove singularities of the density of states.

The code of this figure is in `documentation/figures/bands_honeycomb.py`.

With a list of operators, the output has one extra array per operator, (k,e,c1,c2,...):

```
(k,e,c_sz,c_site) = h.get_bands(operator=["sz","site"]) # two operators at once
```

The available operators are described in Operators.

For a very large unit cell, for example a moire supercell, we usually only want a few bands near the Fermi energy. With **num_bands**, only these bands are computed, which is much faster than computing all of them:

```
(k,e) = h.get_bands(num_bands=20) # only the 20 bands closest to central_energy
```

See `examples/2d/velocity_bands/main.py` and `examples/2d/strain_TBG/main.py` for runnable scripts, and `jupyter-notebooks/functionalities/single_particle_hamiltonians/04_state_resolved_bands.ipynb` for bands colored by an operator.

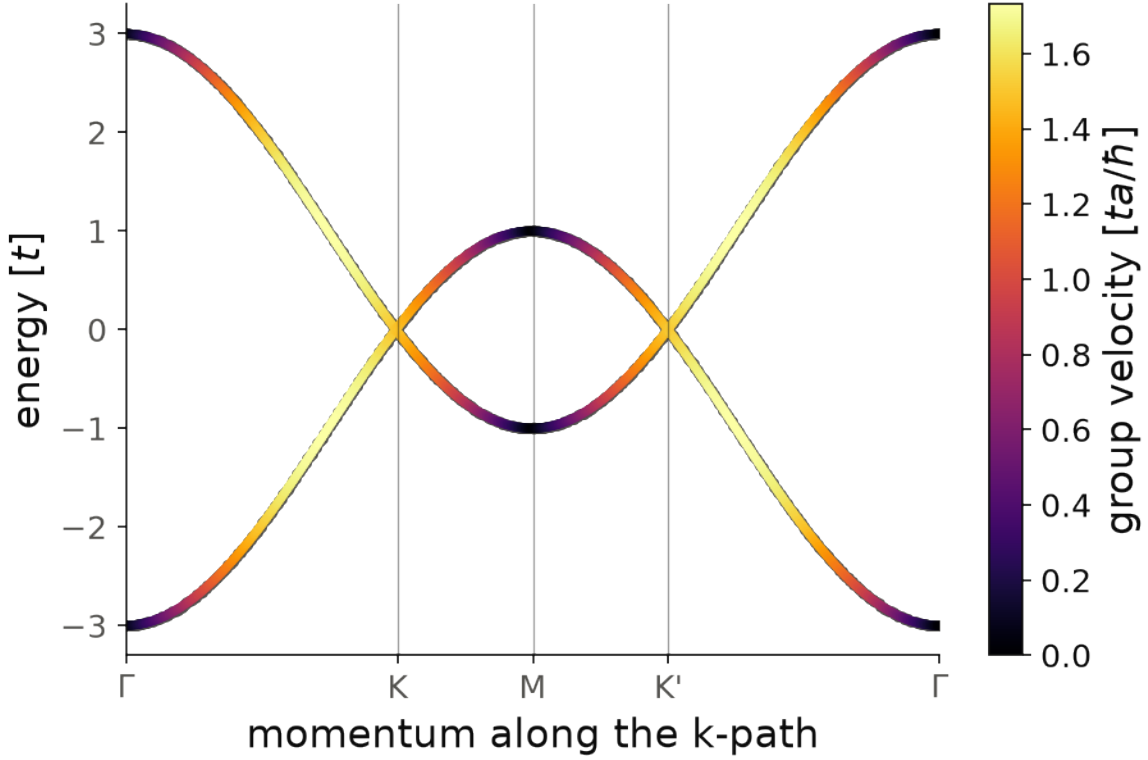


Figure 15: Band structure of the honeycomb lattice, colored by the group velocity of each state.

5.2 Density of states

The density of states counts how many states there are at energy ω ,

$$D(\omega) = \frac{1}{N_k} \sum_{k,\alpha} \delta(\omega - \epsilon_{k,\alpha}) = -\frac{1}{\pi N_k} \sum_k \text{Im Tr } G(k, \omega + i\delta),$$

where N_k is the number of k-points and $G(k, z) = (z - H_k)^{-1}$ is the Green's function. In practice the delta function is replaced by a Lorentzian of width δ , the broadening. With this normalization, the integral of $D(\omega)$ over all energies is the number of states per unit cell, for example 2 for a spinful chain. A flat part of a band gives a peak in $D(\omega)$, and a saddle point gives a logarithmic peak, a van Hove singularity. The density of states is what is measured, averaged over space, by tunneling spectroscopy, and it controls the specific heat and the Pauli susceptibility. We compute it with `h.get_dos()`:

```
from pyqula import geometry
g = geometry.triangular_lattice() # get the geometry
h = g.get_hamiltonian() # get the Hamiltonian
(es, ds) = h.get_dos()
```

`h.get_dos()` returns the energies and $D(\omega)$ at each energy. By default the energies go from -4 to 4 , in units of the hopping. For the triangular lattice the band goes from -3 to 6 , so the density of states is not symmetric. You see a van Hove singularity at $E = -2$, the energy of the saddle point at the M point. The top of the band, at $E = 6$, is outside the default window, so give `energies=` when you need the whole band.

The keywords are

- **energies**: the energies where $D(\omega)$ is computed.
- **delta**: the broadening δ . For `mode="ED"` the default is $5/nk$.
- **operator**: project the density of states on an operator.
- **mode**: how the integral is done: "ED" (default), "Green" (or "RG"), "KPM" or "adaptive".

- **nk**: number of k-points per direction of the k-mesh.

The broadening should be larger than the energy spacing that the finite k-mesh gives, otherwise you see the k-mesh and not the band structure. So a smaller **delta** needs a larger **nk**.

All modes compute the same quantity with the same normalization, and they differ only in cost. "ED" diagonalizes H_k on a k-mesh. "Green" computes the Green's function at each energy, which is cheaper when you need only a few energies. "KPM" uses an expansion in Chebyshev polynomials and is the choice for very large systems (see Kernel polynomial method). "adaptive" integrates over the Brillouin zone with a quadrature that refines the k-mesh where it is needed, with a target error **error=1e-1**. In the example below, a spinful chain with longer-range hopping and Rashba spin-orbit coupling, the ED and Green's function results fall on top of each other:

```
from pyqula import geometry
import numpy as np
g = geometry.chain()
h = g.get_hamiltonian(tij=[0.5,0.,0.,0.5],has_spin=True)
h.add_rashba(0.7)
energies = np.linspace(-4.,4.0,60)
(e1,d1) = h.get_dos(energies=energies,delta=1e-2,mode="ED",nk=1000)
(e2,d2) = h.get_dos(energies=energies,delta=1e-2,mode="Green")
```

With an operator A , the projected density of states is

$$D_A(\omega) = \frac{1}{N_k} \sum_{k,\alpha} \langle \Psi_{k,\alpha} | A | \Psi_{k,\alpha} \rangle \delta(\omega - \epsilon_{k,\alpha}).$$

For the sublattice operator, which is +1 on sublattice A and -1 on sublattice B, this is the difference between the density of states of the two sublattices. In a honeycomb lattice with a sublattice imbalance, the states below the gap live on one sublattice and the states above it on the other, so $D_A(\omega)$ has opposite signs on the two sides of the gap:

```
from pyqula import geometry
g = geometry.honeycomb_lattice()
h = g.get_hamiltonian(has_spin=True)
h.add_sublattice_imbalance(.4)
(es,ds) = h.get_dos(operator="sublattice",nk=200,delta=5e-2)
```

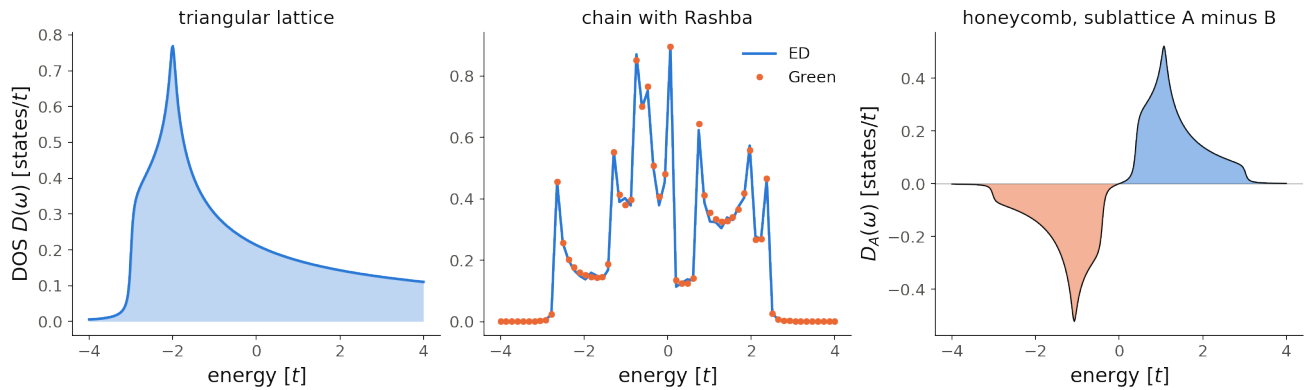


Figure 16: Density of states of the triangular lattice, of the chain with Rashba coupling in two modes, and the sublattice-projected density of states of the honeycomb lattice.

In the left panel, the density of states of the triangular lattice starts at $-3t$ and has its van Hove peak at $-2t$, with a height of 0.77. The window stops at $4t$, so the integral over the window is 1.78 and not 2: the states between $4t$ and $6t$ are missing. For this panel we take **nk=400** and **delta=0.05**, the same broadening as the default **nk=100**. With the default k-mesh the curve has small regular ripples on the right of the peak. Each

ripple is a group of k-points of the mesh, which the broadening does not merge. At `nk=400` they are gone, and `nk=800` changes the curve by less than 10^{-4} . In the middle panel, the ED (line) and the Green's function (dots) results fall on top of each other at every energy. In the right panel, the sublattice-projected density of states is zero for $|\omega| < 0.4$, the gap of the sublattice imbalance, up to the tails of the broadening. It is negative below the gap and positive above it, with its largest values, ± 0.52 , at ± 1.08 , the van Hove peaks $\pm\sqrt{t^2 + 0.4^2}$ moved by the gap. Here `nk=200` is converged: `nk=400` changes the curve by less than 10^{-5} .

The code of this figure is in `documentation/figures/bands_dos.py`.

See `examples/1d/dos_GF/main.py` and `examples/2d/operator_dos/main.py` for runnable scripts, and `jupyter-notebooks/functionality/single_particle_hamiltonians/06_operator_resolved_dos.ipynb` for an executed notebook.

6 Local density of states

In this page we resolve the density of states in space. This is the quantity that STM measures: the differential conductance dI/dV at bias V and tip position $\mathbf{r}$ is, in a first approximation, proportional to the local density of states at energy eV and position $\mathbf{r}$.

6.1 Local density of states at one energy

The local density of states on site n is

$$D(\omega, n) = \frac{1}{N_k} \sum_{k,\alpha} \delta(\omega - \epsilon_{k,\alpha}) |\langle n | \Psi_{k,\alpha} \rangle|^2,$$

where $\epsilon_{k,\alpha}$ are the eigenenergies, $\langle n | \Psi_{k,\alpha} \rangle$ is the amplitude of the eigenstate on site n , and the delta function is broadened with a Lorentzian of width δ . It counts the states at energy ω , each one weighted by its probability on site n . The sum over all the sites gives back the density of states of Band structures and DOS. We compute it with `h.get_ldos()`:

```
from pyqula import geometry
g = geometry.honeycomb_zigzag_ribbon() # get the geometry
h = g.get_hamiltonian() # get the Hamiltonian
(x,y,d) = h.get_ldos()
```

`h.get_ldos()` returns three arrays: the x and y coordinates of the sites, and the local density of states on each site, at the energy `e` (zero by default). For a periodic system the unit cell is repeated a few times, so the map is easier to see. For the zigzag ribbon at zero energy, about three quarters of the weight is on the outermost sites of the two edges. These are the zigzag edge states, and this profile is what STM measures at a zigzag edge of graphene.

The keywords are

- `e`: the energy ω .
- `delta`: the broadening δ .
- `operator`: weight each state with an operator, for example `"sz"` for a map of the spin density of states instead of the charge one.
- `mode`: `"arpack"` (default, diagonalization on a k-mesh), `"green"` (Green's function, 2D only) or `"KPM"` (Chebyshev expansion, for large systems, see Kernel polynomial method).
- `projection`: `"TB"` (default, one value per site), `"TBRs"` (the same values, interpolated on a continuous grid in real space) or `"atomic"` (each site replaced by an atomic orbital, which gives a smooth map closer to an STM image).
- `num_bands`: compute only this many states around `e`, for large Hamiltonians.

With an operator A , the two modes do not compute exactly the same thing. `mode="arpack"` weights each eigenstate with its expectation value, $\langle \Psi | A | \Psi \rangle |\Psi(n)|^2$. `mode="green"` and `mode="KPM"` use the local matrix element $\text{Re } \Psi^*(n)(A\Psi)(n)$. The two agree for a state that is an eigenstate of A , and summed over the sites

both give the same total. For a state whose spin, say, changes from site to site, only the local matrix element shows this change in the map. An operator that depends on momentum, such as "valley", only works with `mode="arpack"`, because the Green's function has already summed over the Brillouin zone.

For a smoother map than one value per site, use `projection="TBRs"`:

```
from pyqula import geometry
g = geometry.honeycomb_zigzag_ribbon() # get the geometry
h = g.get_hamiltonian() # get the Hamiltonian
(x,y,d) = h.get_ldos(e=0.0,projection="TBRs") # interpolated real-space map
```

To follow how the map changes with energy, as in a series of STM conductance maps at different biases, `h.get_multildos()` computes the local density of states at many energies at once:

```
import numpy as np
(x,y,es,ldos) = h.get_multildos(energies=np.linspace(-2.0,2.0,100),projection="atomic") # one map per ene
```

`x` and `y` are the positions, `es` the energies, and `ldos` has one row for each energy, so the rows are the frames of a stack $D(x,y,\omega)$. The maps are also written to the folder `MULTILDOS/`, one file per energy, together with the total density of states in `MULTILDOS/DOS.OUT`, unless `write=False` is given. `DOS.OUT` has the same normalization as `h.get_dos()`. With the default projection, `projection="TB"`, the maps have the same normalization as `h.get_ldos()`, and the sum of a map over the sites is the density of states at that energy. With `projection="atomic"` the map is a density on a continuous grid in real space. Each eigenstate is written in real space as $\Psi(\mathbf{r}) = \sum_n \Psi(n) \phi(\mathbf{r} - \mathbf{r}_n)$, with $\phi(\mathbf{r}) = e^{-|\mathbf{r}|/1.5}$ an atomic orbital, and the map is $|\Psi(\mathbf{r})|^2$. The orbitals of neighboring sites overlap, so $|\Psi(\mathbf{r})|^2$ also contains the interference between them. For this reason each state is normalized on the grid, before the broadening: the integral of its map, the sum over the grid times the area of one grid cell, is the weight the state has in the map of `projection="TB"`. The shape of each state does not change. As a result, the integral of an atomic map is the density of states at that energy, as the sum of a map with `projection="TB"`, and the intensities at different energies can be compared. For the zigzag ribbon with `delta=0.1` and `nk=200`, the integral of the atomic map is 13.3 at both $\omega = -2t$ and $\omega = 2t$, the same as the sum of the tight-binding map, because the density of states is the same at the two energies. The interference is still in the shape: the maximum of the map is 2.6 times higher at $\omega = -2t$ than at $\omega = 2t$. With an operator, `h.get_multildos()` uses the expectation value $\langle \Psi | A | \Psi \rangle$ times $|\Psi(n)|^2$, as `mode="arpack"` above. `projection="atomic"` does not accept an operator.

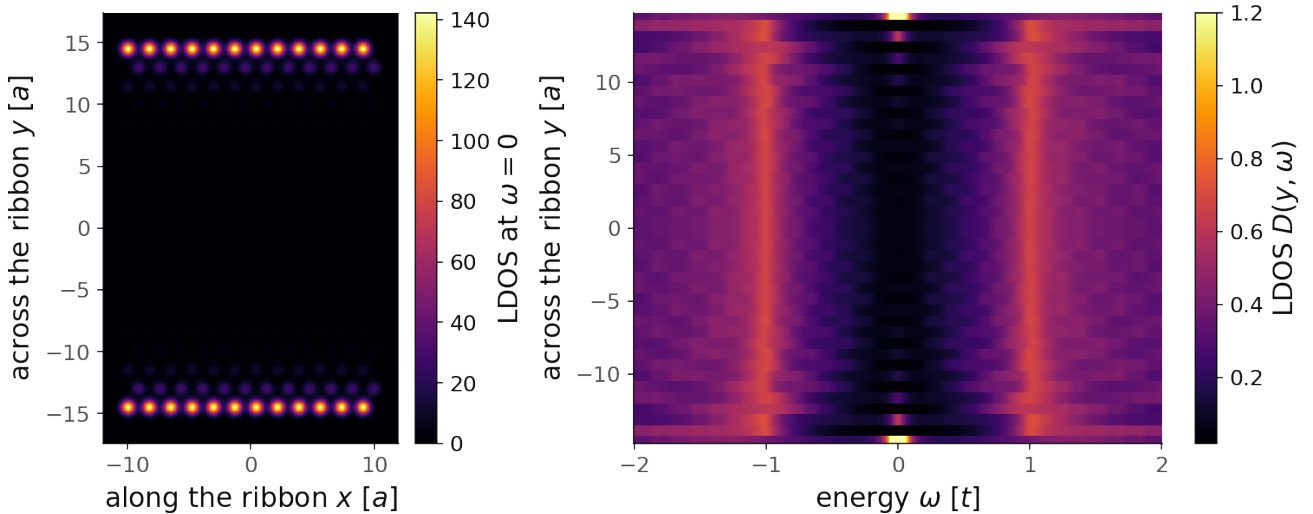


Figure 17: Local density of states of the zigzag ribbon at zero energy in real space, and resolved in position across the ribbon and energy.

In both panels the position across the ribbon is the vertical axis. On the left, the map at zero energy with `projection="TBRs"`, with the ribbon along the horizontal axis. We take `nrep=12`, twelve unit cells of the ribbon, so that the map is not a thin strip. Lengths are in units of the first-neighbor distance a . The weight is on the

outermost sites of the two edges, and it decays into the ribbon on every second site, on one sublattice only. On the right, the maps of `h.get_multildos()` at 401 energies, one column per energy, as a stack of STM maps. Here we use the default projection, one value per site, instead of `projection="atomic"`, so that each row is one site. We take `nk=1000` and `delta=0.05`. With the default `nk=100` and `delta=0.01` the map is speckled, for two reasons. The first is the k-mesh: the broadening is smaller than the energy step between neighboring k-points. At `nk=1000` this is gone, and `nk=3000` changes the map by less than 10^{-5} . The second is physical: the ribbon has 40 sites in its unit cell, so 40 subbands, and the edge of each subband gives a sharp peak in energy. The larger `delta=0.05` smooths them, as the finite energy resolution of an STM does. You see a bright line at $\omega = 0$ on the edge sites, with the value 2.8, against 0.60 on the next site of the same sublattice and almost zero on the other sublattice. Away from zero energy the weight spreads over the whole ribbon, with the bulk van Hove peaks at $\pm t$. The alternation from one row to the next is the sublattice.

The code of this figure is in `documentation/figures/ldos_zigzag.py`.

See `examples/0d/island/main.py` for a runnable script on a superconducting island with `projection="TBRS"`.

6.2 Local density of states of a finite island

On a finite island the states are discrete, and the map at each energy shows one or a few of them. For a triangular island of the honeycomb lattice, we compute the maps at many energies with `h.get_multildos()`:

```
from pyqula import islands
g = islands.get_geometry(name="honeycomb", n=3, nedges=3) # get an island
h = g.get_hamiltonian() # get the Hamiltonian
h.get_multildos(projection="atomic") # get the LDOS
```

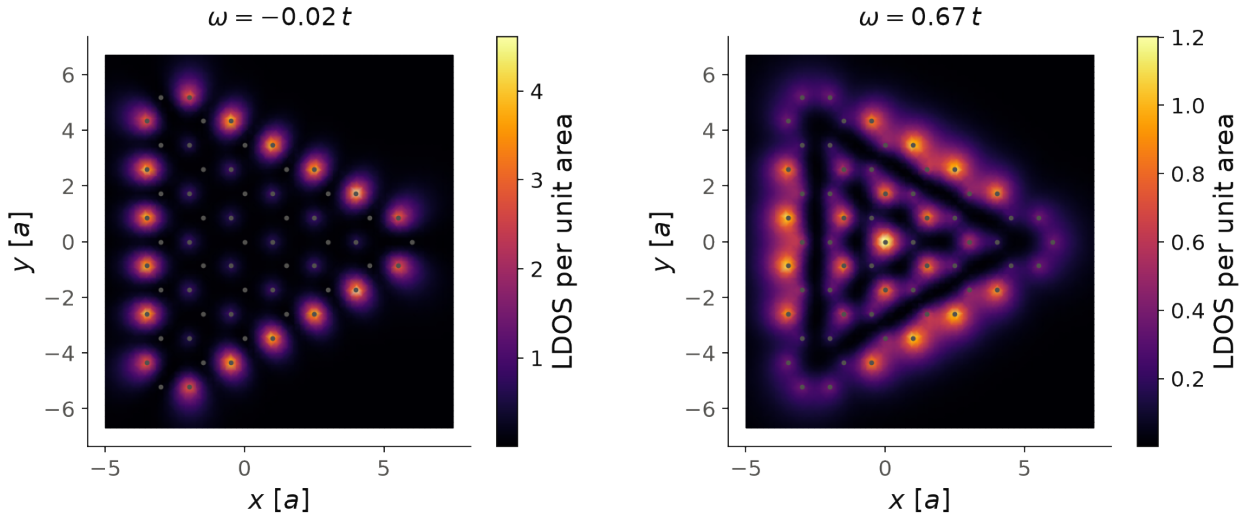


Figure 18: Local density of states of a triangular honeycomb island at zero energy and at the first level above it, with the atomic projection.

The island has 61 sites. Its Hamiltonian has five zero-energy states, the difference between the number of sites of the two sublattices. On the left, the map at $\omega = -0.02t$, the energy of the grid closest to zero, shows these states: the weight is on the outer sites of the three zigzag edges, on one sublattice, and it is almost zero inside. On the right, at the first level above them, $\omega = 0.67t$, the weight is spread over the inside of the island, with a node line between the edge and the center. The small dots are the sites. Here we assign the output of `h.get_multildos()` to arrays, which the snippet above does not do.

The code of this figure is in `documentation/figures/ldos_island.py`.

The map at each energy is what STM measures on the island at that bias, and `DOS.OUT` is the spectrum of the whole island. See `examples/readme_examples/ldos_island/main.py` for a runnable script that plots the maps.

7 Spectral functions and Fermi surfaces

In this page we compute the spectral function resolved in momentum and energy: along a k-path, on a constant-energy cut of the Brillouin zone (the Fermi surface), and at the surface of a semi-infinite crystal. These are the quantities that ARPES measures.

7.1 Momentum-resolved spectral function

The band structure gives the energies as sharp lines. The momentum-resolved spectral function gives the same information as a heatmap, where each band has a finite width and can be weighted by an operator O ,

$$A_O(k, \omega) = -\frac{1}{\pi} \text{Im Tr} [O G(k, \omega + i\delta)] = \sum_{\alpha} \langle \Psi_{k,\alpha} | O | \Psi_{k,\alpha} \rangle \delta(\omega - \epsilon_{k,\alpha}),$$

where $G(k, z) = (z - H_k)^{-1}$ is the Green's function, $\epsilon_{k,\alpha}$ and $\Psi_{k,\alpha}$ are the eigenenergies and eigenstates at momentum k , and the delta function is a Lorentzian of width δ . Without an operator, O is the identity and $A(k, \omega)$ is the total spectral weight. This is the band structure as ARPES sees it. The operator is useful to select a part of the states. For example, in a superconductor the projector on the electron sector gives the electron spectral function, without the hole copy of the bands. We compute it with `h.get_kdos_bands()`:

```
from pyqula import geometry
g = geometry.triangular_lattice() # get the geometry
h = g.get_hamiltonian() # get the Hamiltonian
h.get_kdos_bands()
```

`h.get_kdos_bands()` returns three arrays: the position along the k-path, the energy, and $A(k, \omega)$ at each pair. By default the energies go from -3 to 3 . The same three columns are written to `KDOS_BANDS.OUT`. The broadening `delta` is the width of each line in the heatmap.

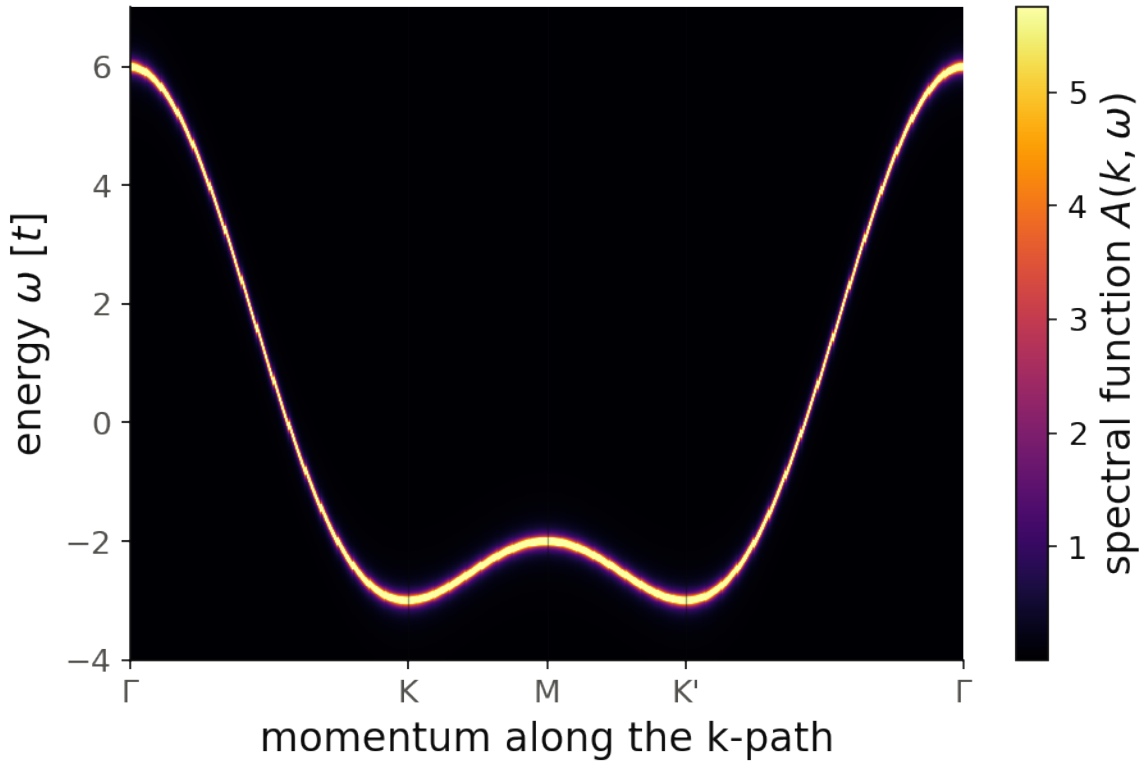


Figure 19: Momentum-resolved spectral function of the triangular lattice along the k-path.

For the figure we take `energies=np.linspace(-4,7,1101)`, because the default window from -3 to 3 cuts the band. We also take `nk=600` points on the k-path and `delta=0.05`. The energy step, 0.01 , and the change of the band energy between two k-points must both be smaller than `delta`, otherwise the line breaks into separate dots. You see the single band of the triangular lattice, from $-3t$ at K and K' to $6t$ at Γ , with a local maximum at $-2t$ at M, the saddle point. The width of the line is the broadening.

The code of this figure is in `documentation/figures/spectral_kdos.py`.

The keywords are

- `energies`: the energies ω .
- `delta`: the broadening δ .
- `operator`: the operator O .
- `kpath`, `nk`: the k-path and the number of k-points, as in `h.get_bands()`.
- `mode`: "ED" (default, diagonalization), "green" (Green's function) or "KPM" (Chebyshev expansion, for large systems).

The three modes give the same quantity with the same normalization, so a system that is too large to diagonalize can go from `mode="ED"` to `mode="KPM"` without changing the scale. Only the line shape changes: the Chebyshev expansion broadens each band with its kernel, not with a Lorentzian of width `delta`.

See `examples/1d/kdos_armchair/main.py` for a runnable script on a wide ribbon with `mode="KPM"`, and `jupyter-notebooks/functionalities/single_particle_hamiltonians/05_momentum_resolved_spectral_functions.ipynb` and `jupyter-notebooks/functionalities/spectral_functions/06_operator_momentum_resolved.ipynb` for executed notebooks.

7.2 Fermi surface

The Fermi surface is the set of momenta where a band crosses the Fermi energy. It is what a metal looks like in reciprocal space, and it controls the low-temperature transport and the nesting instabilities. For a 2D Hamiltonian, `h.get_fermi_surface()` computes the spectral weight $A(\mathbf{k}, \omega = E)$ on a (k_x, k_y) mesh, at one energy E (by default the Fermi energy, `e=0.0`). It is a constant-energy cut of the spectral function, as measured by ARPES:

```
from pyqula import geometry
g = geometry.triangular_lattice() # get the geometry
h = g.get_hamiltonian() # get the Hamiltonian
(kx,ky,fs) = h.get_fermi_surface(e=0.0,nk=50,delta=1e-1)
```

It returns the two coordinates of each point of the mesh and the spectral weight at that point. A scatter plot colored by the weight shows the Fermi surface. The broadening `delta` sets the thickness of the contour. If `delta` is smaller than the energy spacing of the k-mesh, you see the mesh and not the contour.

The keywords are

- `e`: the energy of the cut.
- `nk`: number of k-points per direction.
- `delta`: the broadening.
- `operator`: weight each state with an operator, which gives for example a spin or valley texture of the Fermi surface.

With an operator, each point is weighted with the expectation value of the operator. This is how we see the spin texture of a material with spin-orbit coupling. The NbSe2 Hamiltonian below is a multi-orbital model with spin-orbit coupling, and the weight $\langle s_z \rangle$ tells apart the two spin channels on each pocket:

```
from pyqula.specialhamiltonian import NbSe2
h = NbSe2(soc=0.9) # multi-orbital spin-orbit-coupled Hamiltonian
(kx,ky,fs) = h.get_fermi_surface(e=0.,nk=100,delta=3e-1,operator="sz")
```

The momenta are Cartesian, in units of $2\pi/a$. On the left, the Fermi surface of the triangular lattice at zero energy is a circle of radius 0.38 around Γ , repeated around the neighboring reciprocal lattice points. For the figure we take `nk=150` instead of `nk=50`: with `nk=50` and `delta=1e-1` the contour breaks into dots, which is the

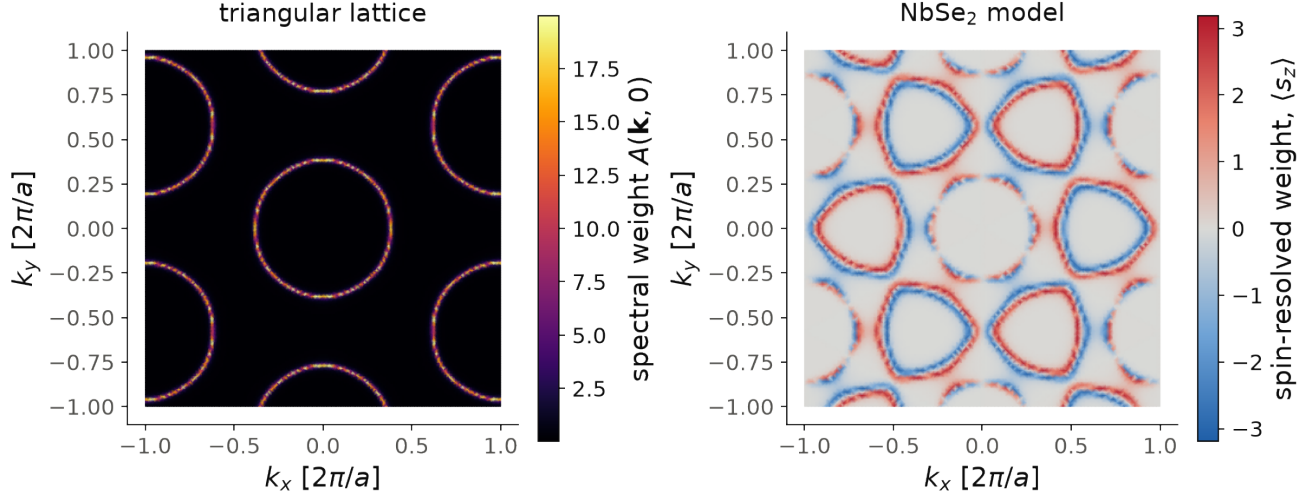


Figure 20: Fermi surface of the triangular lattice, and the spin-resolved Fermi surface of the NbSe2 model.

k -mesh and not the physics. On the right, the NbSe2 model has a pocket around Γ and pockets around K and K' , at $k_x = \pm 0.67$. Each pocket at K and K' is split in two by the spin-orbit coupling, with opposite $\langle s_z \rangle$, and the order of the two colors is reversed between K and K' . This is the Ising spin-orbit coupling of NbSe2. The pocket around Γ has a much weaker spin texture.

The code of this figure is in `documentation/figures/spectral_fermi.py`.

`h.get_multi_fermi_surface()` computes the same map at many energies, and writes one file per energy to the folder `MULTIFERMISURFACE/`. This shows how the constant-energy contours change away from the Fermi energy:

```
import numpy as np
h.get_multi_fermi_surface(energies=np.linspace(-4,4,100),delta=1e-1)
```

7.2.1 Unfolded Fermi surface of a supercell

A defect or a charge order needs a supercell, and the Fermi surface of a supercell is folded into its small Brillouin zone. With `operator="unfold"`, the spectral weight is projected back onto the Brillouin zone of the primitive cell, as ARPES sees it (see Band unfolding for the formula). The supercell must be built with `store_primal=True`, and `nsuper` stretches the mesh over the primitive Brillouin zone:

```
import numpy as np
from pyqula import geometry
g0 = geometry.triangular_lattice()
n = 3 # size of the supercell
g = g0.get_supercell(n,store_primal=True)
h = g.get_hamiltonian()
h.add_onsite(lambda r: 100.0 if np.linalg.norm(r-g.r[0])<1e-1 else 0.0) # a point defect
```

```
out = h.get_multi_fermi_surface(nk=50,energies=np.linspace(-4,4,100),delta=0.1,
                                nsuper=n,operator="unfold")
```

`nsuper=n` works for a supercell with the same shape as the primitive cell. For a supercell with another shape, the mesh comes out sheared. For any supercell, use `primal_mesh=True` instead. The mesh is then drawn in the Brillouin zone of the primitive cell, and each point is computed at the equivalent momentum of the supercell:

```
import numpy as np
from pyqula import geometry
g0 = geometry.triangular_lattice() # primitive geometry
g = g0.get_supercell(np.sqrt(3),store_primal=True) # sqrt(3) x sqrt(3) supercell
```

```

h = g.get_hamiltonian() # Hamiltonian of the supercell
h.add_onsite(lambda r: 0.6 if np.linalg.norm(r-g.r[0])<1e-1 else 0.0) # charge order
(kx,ky,fs) = h.get_fermi_surface(e=0.0,nk=50,delta=0.1,operator="unfold",
    primal_mesh=True) # unfolded Fermi surface, on the primitive mesh

```

The result is on the same mesh and in the same coordinates as the Fermi surface of the primitive cell, so the two can be compared point by point. For a supercell without any modulation, the two maps are the same, up to a factor equal to the number of primitive cells in the supercell. Note that a supercell with a non-integer size, as $\sqrt{3}$ here, is rotated so that its first lattice vector is along x , and its Fermi surface is drawn in this rotated frame.

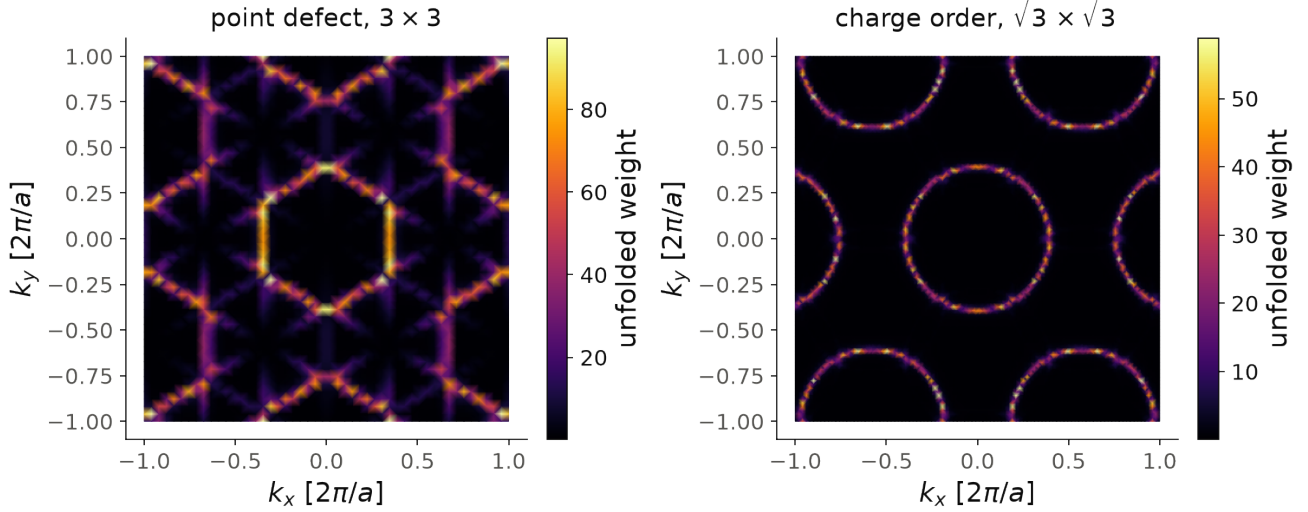


Figure 21: Unfolded Fermi surfaces at zero energy, of a point defect in a 3×3 supercell and of a three-sublattice charge order in a $\sqrt{3} \times \sqrt{3}$ supercell.

Both maps are at zero energy, in units of $2\pi/a$ of the primitive cell. On the left, the frame of `h.get_multi_fermi_surface()` closest to zero energy, $\omega = -0.04t$, with the mesh divided by `n` to have the same units. The point defect is strong, one site in nine, and the circle of the clean lattice becomes a hexagon of about the same size, with weak shadow lines of the folded copies around it. On the right, the charge order of 0.6 leaves the circle of radius 0.38 almost unchanged at this energy. The neighboring circles are around $(\pm 1.15, 0)$ and $(\pm 0.58, \pm 1)$, the reciprocal lattice of the rotated frame. For the figure we take `nk=100` instead of `nk=50`, so that the circle is a smooth line. The largest weight is 58, three times the one of the primitive cell, as expected for three primitive cells in the supercell.

The code of this figure is in `documentation/figures/spectral_unfoldfs.py`.

See `examples/readme_examples/fermi_surface/main.py`, `examples/2d/operator_fermi_surface/main.py` and `examples/readme_examples/unfolding_FS/main.py` for runnable scripts.

7.3 Surface spectral function

A ribbon has two edges, and a set of discrete bands that depends on its width. A cleaved crystal is different: it has one surface, and a continuum of bulk states behind it. To describe it we use a semi-infinite system, and compute the Green's function of its surface layer, $G_s(k, \omega)$, with `h.get_surface_kdos()`. Here k is the momentum parallel to the surface. G_s is obtained by a renormalization (decimation) method: the layers behind the surface are added one after another to the surface, until one more layer changes nothing. The surface spectral function is

$$A_s(k, \omega) = -\text{Im Tr } G_s(k, \omega + i\delta),$$

and in the same way the bulk one, A_b , is computed from the Green's function of a layer deep in the bulk. Note that there is no factor $1/\pi$ here, so these numbers are π times a spectral function. A_s is what ARPES measures on the surface of a crystal. For a honeycomb lattice with Kane-Mele spin-orbit coupling:

```

from pyqula import geometry
g = geometry.honeycomb_lattice() # create honeycomb lattice
h = g.get_hamiltonian() # create hamiltonian of the system
h.add_soc(0.05) # Kane-Mele spin-orbit coupling opens a topological gap
h.add_rashba(0.1) # break z-mirror symmetry to expose the edge states
(k,e,ds,db) = h.get_surface_kdos(delta=1e-2) # surface and bulk spectral functions

```

ds and db are the surface and bulk spectral functions at each $(\mathbf{k}, \mathbf{e})$. The spin-orbit coupling opens a gap in the bulk, so db is zero inside the gap. In ds you see states that cross this gap: the helical edge states of the quantum spin Hall insulator. They exist at the edge and not in the bulk, and they are protected by time-reversal symmetry and the Z_2 invariant (see Topological invariants).

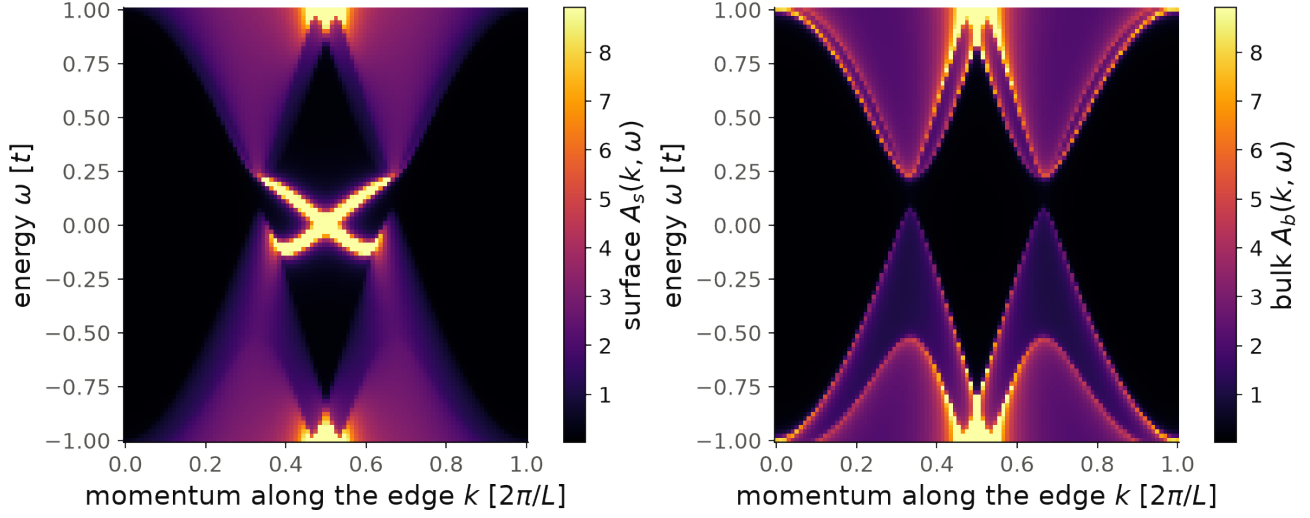


Figure 22: Surface and bulk spectral functions of the Kane-Mele model with Rashba coupling.

k is the momentum along the edge, in units of $2\pi/L$ with L the period of the edge. In the bulk (right), the bands have a gap at $k = 1/3$ and $k = 2/3$, the projections of K and K' . The gap goes from $0.075t$ to $0.22t$, the same as in the band structure of the two-dimensional crystal: the spin-orbit coupling opens it and the Rashba coupling makes it smaller. At the surface (left), two bright lines cross this gap and meet at $k = 1/2$ near $\omega = 0$. These are the helical edge states, which cross at the time-reversal invariant momentum. The color scale is the same in the two panels and saturates on the edge states.

The code of this figure is in `documentation/figures/spectral_surface.py`.

See `examples/readme_examples/surface_2dTI/main.py` for a runnable script, and `jupyter-notebooks/functionalities/surface_2dTI` for an executed notebook.

The surface density of states, integrated over k , is `dos.surface_dos(h, ...)`, which returns the surface and bulk densities of states at a set of energies, with the same normalization as A_s (without the $1/\pi$). It also takes an `operator=`, for example to get the spin-resolved surface density of states of a magnet. An operator that depends on momentum, such as "valley", is not accepted, because the Green's function has already been summed over k . The same decimation of a semi-infinite lead is used in Quantum transport.

8 Quasiparticle interference

In this page we compute what a single defect does to the local density of states of a two-dimensional system. A defect scatters an electron from a state with momentum $\mathbf{k}$ to a state with momentum $\mathbf{k}'$ on the same constant-energy contour. The incoming and the scattered waves interfere, so the local density of states around the defect oscillates with wavevector $\mathbf{q} = \mathbf{k}' - \mathbf{k}$. The Fourier transform of these oscillations at each energy is called the quasiparticle interference (QPI) pattern. In an experiment, it is obtained by STM as the Fourier transform of a

conductance map $dI/dV(\mathbf{r}, V)$. It shows the constant-energy contours of the band structure, and which pairs of states the defect connects.

8.1 QPI from the spectral function

The simplest estimate of the QPI pattern is the autocorrelation of the momentum-resolved spectral function,

$$P(\mathbf{q}, \omega) = \sum_{\mathbf{k}} A(\mathbf{k}, \omega) A(\mathbf{k} + \mathbf{q}, \omega),$$

where $A(\mathbf{k}, \omega)$ is the spectral weight at momentum $\mathbf{k}$ and energy ω , broadened with a Lorentzian of width δ (see Spectral functions). It counts all the pairs of states at energy ω separated by $\mathbf{q}$, so it is large for a $\mathbf{q}$ that connects two flat, parallel parts of the contour. This is `mode="pm"` of `h.get_qpi()`, for 2D Hamiltonians:

```
import numpy as np
from pyqula import geometry
g = geometry.triangular_lattice()
h = g.get_hamiltonian(has_spin=False)
(qs, es, qpi) = h.get_qpi(mode="pm", nk=200, delta=1e-1, energies=np.linspace(-6., 6., 100)) # QPI at each energy
```

`qs` contains the wavevectors $\mathbf{q}$, one per row, `es` the energies, and `qpi` the intensity, with one row per energy and one column per wavevector. So one row of `qpi`, plotted over `qs`, is the QPI map of an STM measurement at that energy. The band of the triangular lattice goes from $-3t$ at K and K' to $6t$ at Γ . So near the top of the band the contour is a small circle of radius k_F around Γ , and the QPI map is a disk of radius $2k_F$, since $2k_F$ is the largest momentum that connects two points of the circle. Near the bottom of the band the contour is two small pockets, around K and K'. The same maps are written to the folder `MULTIQPI/`, one file per energy, with the density of states in `DOS.OUT`, unless `write=False` is given.

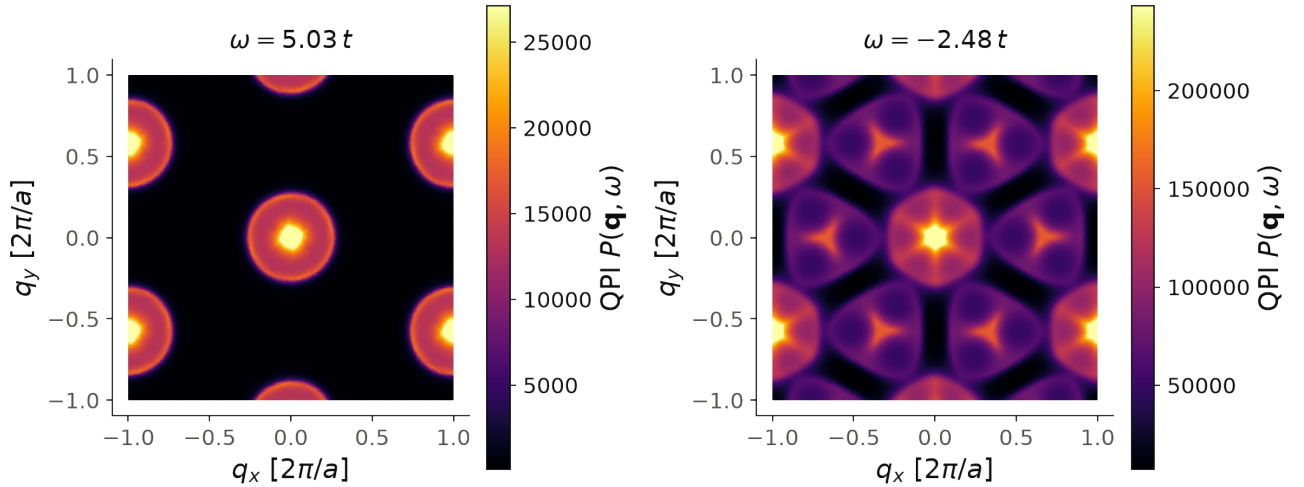


Figure 23: QPI maps of the triangular lattice, at an energy near the top of the band and at an energy near the bottom.

The wavevectors are Cartesian, in units of $2\pi/a$, and the color scale stops at the 99th percentile, because the peak at $\mathbf{q} = 0$ is much larger than the rest. With this sign of the hopping, the band of the triangular lattice goes from $-3t$ at K to $6t$ at Γ , so the small circle around Γ is at the top of the band. At $\omega = 5.03t$ (left) the contour is a circle of radius $k_F \approx 0.13$ and the QPI map is a disk of radius 0.26 , which is $2k_F$, repeated at every reciprocal lattice vector. At $\omega = -2.48t$ (right), near the bottom of the band, the contour is two small pockets around K and K'. The map has a disk at $\mathbf{q} = 0$, from the scattering inside one pocket, and bright features at the K points of $\mathbf{q}$, from the scattering between the two pockets. The pockets are small, so they need a dense $\mathbf{k}$ -mesh: with `nk=50` the right map is noisy, and the maps at `nk=200` and `nk=300` agree within one percent. With a smaller δ , increase `nk` again.

The code of this figure is in `documentation/figures/qpi_pm.py`.

The keywords are

- **energies**: the energies ω .
- **nk**: number of k-points per direction.
- **delta**: the broadening δ .
- **mode**: "pm", the autocorrelation above, or "response" (default), a nesting function of the clean bands, defined below.
- **operator**: with "unfold", the spectral function of a supercell is unfolded to the primitive Brillouin zone (mode="pm" only).
- **nunfold**: the linear size of the supercell for the unfolding, as in `g.get_supercell()`: n for an $n \times n$ supercell, `np.sqrt(3)` for a $\sqrt{3} \times \sqrt{3}$ one.
- **write**: False to return the maps without writing files.

`mode="response"` uses only the energies of the bands, not the wavefunctions,

$$R(\mathbf{q}, \omega) = \sum_{\mathbf{k}, \alpha, \beta} \Theta(\omega - \epsilon_{\mathbf{k}+\mathbf{q}, \alpha}) \Theta(\epsilon_{\mathbf{k}, \beta} - \omega) \frac{\delta}{(\epsilon_{\mathbf{k}+\mathbf{q}, \alpha} - \epsilon_{\mathbf{k}, \beta})^2 + \delta^2},$$

which counts pairs of states, one below and one above ω , with almost the same energy. It is a measure of the nesting of the contour at energy ω . It is cheaper, but it ignores the form factors of the wavefunctions, and it does not accept an operator or unfolding. `mode="pm"` uses the real spectral weight of the system, defect included, and it is the one to use for a real scatterer.

The realistic case is a single point defect in a supercell, with the QPI unfolded back to the Brillouin zone of the primitive cell. The supercell must be built with `store_primal=True`, so that the primitive cell is known for the unfolding:

```
import numpy as np
from pyqula import geometry
g0 = geometry.honeycomb_lattice()
ns = 2
g = g0.get_supercell(ns, store_primal=True)
h = g.get_hamiltonian(has_spin=False)
h.add_onsite(lambda r: 100.0 if np.linalg.norm(r-g.r[0])<1e-1 else 0.0) # a strong point defect

h.get_qpi(mode="pm", delta=1e-2, operator="unfold", nsuper=2, nk=140, nunfold=ns)
```

This calculation is expensive. `mode="pm"` diagonalizes the Hamiltonian on an `nk` x `nk` mesh and then computes the autocorrelation, so `nk=140` takes minutes. Use `nk=60` (about 40 seconds) to prepare the calculation, and a large `nk` only for the final result, since `nk` sets the resolution in $\mathbf{q}$.

See `examples/2d/multiqpi/main.py` (clean system, `mode="pm"`) and `examples/2d/multiqpi_unfold/main.py` (defect in a supercell, unfolded) for runnable scripts.

8.2 QPI from a real-space impurity

`h.get_qpi_impurity()` does the calculation in the same way as the experiment. It builds a supercell of `h`, puts one or more impurities in it, computes the local density of states $D(\mathbf{r}, \omega)$ around them (see Local density of states), and takes its Fourier transform,

$$\text{QPI}(\mathbf{q}, \omega) = \left| \sum_{\mathbf{r}} e^{-i\mathbf{q}\cdot\mathbf{r}} D(\mathbf{r}, \omega) \right|.$$

So here the interference of the scattered waves is included exactly, for the impurity potential you choose:

```

from pyqula import geometry
g = geometry.honeycomb_lattice()
h = g.get_hamiltonian(has_spin=False)
r,ldos_r,q,qpi_q = h.get_qpi_impurity(nsuper=10,
    impurities=[{"position": [0.,0.,0.], "onsite": 3.0}],
    energies=0.3,num_waves=60,nk=2,delta=0.2)

```

r and $ldos_r$ are the positions of the sites and the local density of states on them, the real-space map. q and qpi_q are the wavevectors and the QPI intensity, its Fourier transform.

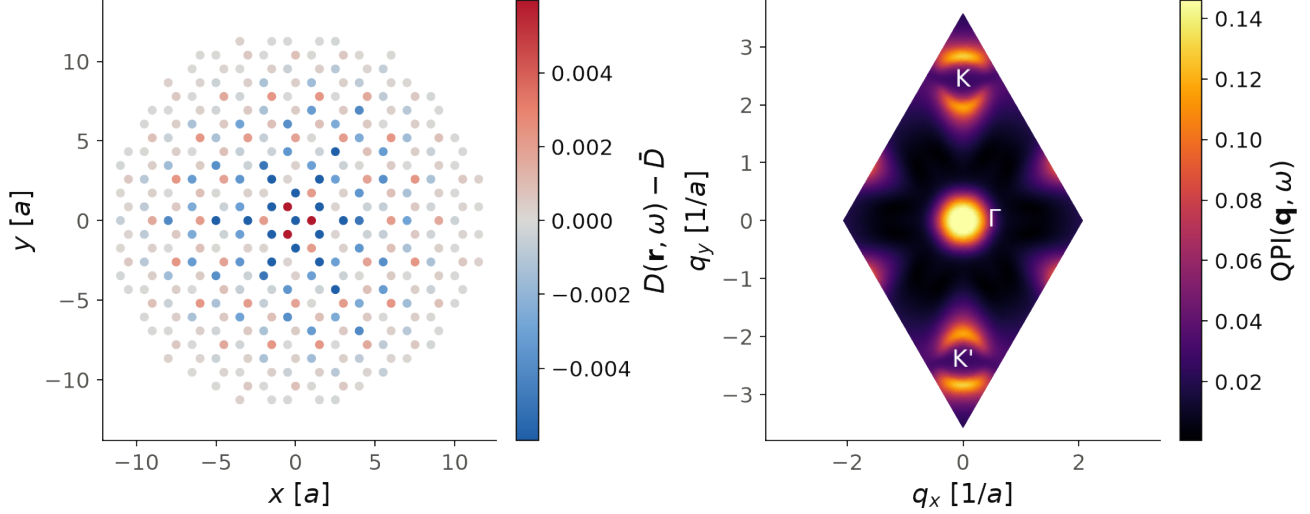


Figure 24: Change of the local density of states around an impurity in the honeycomb lattice, and its Fourier transform.

For the figure we take a much larger supercell, $nsuper=69$, so that the q mesh is fine and the oscillations decay well inside the supercell. The size is a multiple of three, because only then the K points are on the q mesh. With such a large supercell one k-point is enough ($nk=1$), and we can take a smaller broadening, $\delta = 0.1t$. We ask for all the states with num_waves , which is a full diagonalization of 9522 sites, and it takes about 3 minutes. With $nsuper=48$ the maps are the same within a few percent. Lengths are in units of the first-neighbor distance a . On the left, the change of the local density of states at $\omega = 0.3t$ with respect to its mean, within $12a$ of the impurity. The color scale stops at the 95th percentile, so that the impurity site, where the density drops, does not hide the rest. You see the Friedel oscillations around the impurity, with a period of $3a$ along x , and they decay with the distance. On the right, the QPI intensity over the 69×69 points of the mesh, which cover one Brillouin zone. The point $q = 0$ is left out, since it is the mean density and it is about 4200 times larger than the rest. The brightest feature is then the disk around Γ , from the scattering inside one valley. Around K and K', at $|q| \approx 2.4/a$, there is a ring of radius about $0.4/a$, which is $2k_F$ of the Dirac cone at this energy, brightest along the line to Γ . This is the scattering between the two valleys, and it gives the period of $3a$ in real space. For larger maps, or for a single impurity in an infinite system, use the T-matrix, described in QPI from the T-matrix: it gives the same result as this calculation with $nk=1$ and $nsuper$ equal to the nk of the T-matrix, at a much lower cost.

The code of this figure is in `documentation/figures/qpi_impurity.py`.

The keywords are

- **nsuper**: size of the supercell, a number or $(n1,n2)$.
- **impurities**: a list of impurities. Each one is an onsite potential, `{"position": [x,y,z], "onsite": v}` (or `{"index": i, "onsite": v}` for a site of the supercell), or a vacancy, `{"position": [x,y,z], "vacancy": True}`.
- **energies**: one energy or an array of energies.
- **num_waves**: a first guess for the number of eigenstates computed near each energy. It is increased automatically when needed.
- **nk, delta**: as in `h.get_ldos()`.

A vacancy is modeled as a very large onsite potential on the site, which in practice removes the site from the low-energy states.

The wavevectors $\mathbf{q}$ are the $n_1 \times n_2$ points that are commensurate with the supercell, and they cover the whole Brillouin zone of the primitive cell. So `nsuper` sets the resolution in $\mathbf{q}$, not the range. Only these points are free of finite-size artifacts: at any other $\mathbf{q}$, even a clean supercell would give a signal. No unfolding is needed here, because the Fourier transform is done directly on the real-space density.

See `examples/2d/qpi_realspace_impurity/main.py` for a runnable script that plots the real-space map and the QPI, and `jupyter-notebooks/functionalities/spectral_functions/07_qpi_maps.ipynb` for an executed notebook.

8.3 QPI from the T-matrix

We will now see how to compute the QPI of a single impurity in an infinite system, without a supercell. The way we do this is with the T-matrix. We take the clean Hamiltonian H_0 and one impurity V , which acts only inside the unit cell at the origin. The Green's function of the clean system is

$$G_0(\mathbf{k}, \omega) = [\omega + i\delta - H_0(\mathbf{k})]^{-1},$$

where $H_0(\mathbf{k})$ is the Bloch Hamiltonian and δ the broadening. The impurity scatters the electrons many times. The sum of all these scattering events is the T-matrix,

$$T(\omega) = V [1 - G_0(\omega) V]^{-1}, \quad G_0(\omega) = \frac{1}{N} \sum_{\mathbf{k}} G_0(\mathbf{k}, \omega),$$

where $G_0(\omega)$ is the local Green's function of the unit cell and N the number of $\mathbf{k}$ -points. Because the impurity is local, T does not depend on momentum. The Green's function with the impurity is $G = G_0 + G_0 T G_0$, so the change of the local density of states has the Fourier transform

$$\delta\rho(\mathbf{q}, \omega) = -\frac{1}{2\pi i} \sum_a e^{-i\mathbf{q}\cdot\tau_a} [\Lambda_{aa}(\mathbf{q}, \omega) - \Lambda_{aa}^*(-\mathbf{q}, \omega)], \quad \Lambda(\mathbf{q}, \omega) = \frac{1}{N} \sum_{\mathbf{k}} G_0(\mathbf{k}, \omega) T(\omega) G_0(\mathbf{k} - \mathbf{q}, \omega),$$

where a runs over the orbitals of the unit cell and τ_a is the position of the site of orbital a , so the result is the Fourier transform of a real-space map, as in STM. This is the formalism of the review by Rhodes, Kohsaka, Hanaguri, Marques and Wahl ([arXiv:2607.22487](https://arxiv.org/abs/2607.22487)). For a superconductor, only the electron part of the Nambu Green's function enters the trace. The QPI intensity is $|\delta\rho(\mathbf{q}, \omega)|$.

When the impurity is weak, $T \approx V$. This is called the Born approximation, and it is the first order in V . For a point impurity in a single-band model with inversion symmetry, the expression above becomes $\delta\rho = -\frac{1}{\pi} \text{Im}[T \Lambda_0(\mathbf{q})]$, where Λ_0 is the sum of $G_0(\mathbf{k}) G_0(\mathbf{k} - \mathbf{q})$. In the Born approximation $T = V$ is real, so only $\text{Im} \Lambda_0$ enters. A strong impurity has a complex T , so $\text{Re} \Lambda_0$ enters too, and the shape of the pattern changes, not only its size.

This is `mode="tmatrix"` of `h.get_qpi()`, and `mode="born"` for the Born approximation. The impurity is given with `impurity`:

```
from pyqula import geometry
g = geometry.square_lattice()
h = g.get_hamiltonian(has_spin=False)
(qs, es, qpi) = h.get_qpi(mode="tmatrix", impurity=5.0, nk=120, nsuper=2,
                          delta=1e-1, energies=[-1.0]) # a strong point impurity
```

As for the other modes, `qs` contains the wavevectors, one per row, in units of the reciprocal lattice vectors, `es` the energies, and `qpi` the intensity, with one row per energy and one column per wavevector. Here $\delta\rho$ contains only the change due to the impurity, so the point $\mathbf{q} = 0$ is the change of the total density of states, and it is not much larger than the rest.

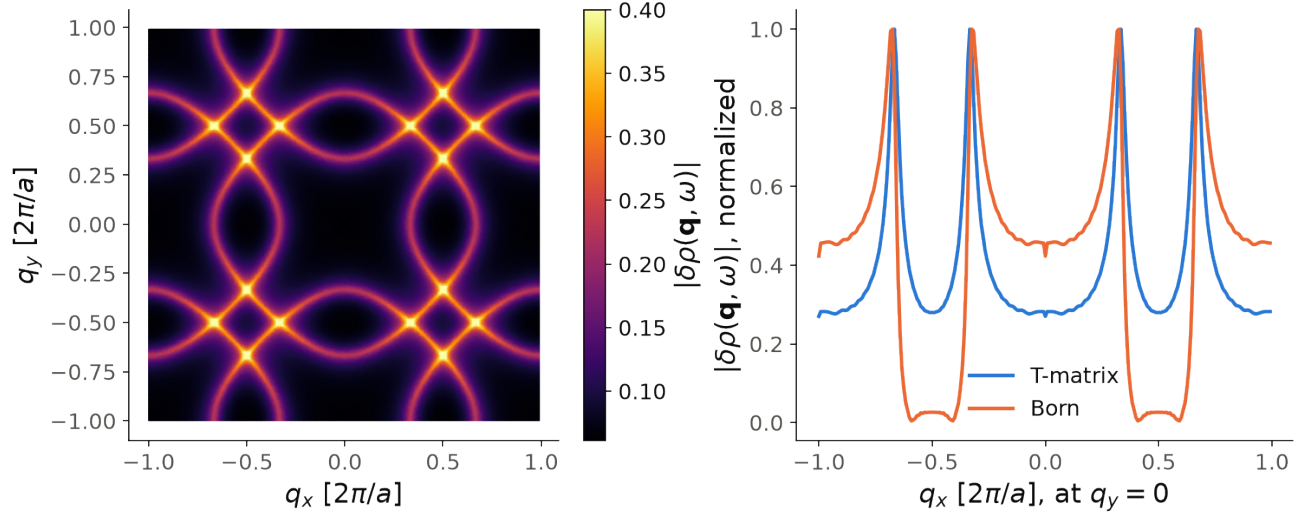


Figure 25: QPI map of a point impurity in the square lattice from the T-matrix, and a cut along the q_x axis compared with the Born approximation.

The band of the square lattice goes from $-4t$ at M to $4t$ at Γ . At $\omega = -t$ the contour is a rounded square around M, with a radius $k_F = 1/3$ along the axes, in units of $2\pi/a$. On the left, the QPI map of an impurity with $V = 5t$. You see a rounded square of radius $2k_F = 2/3$ around every reciprocal lattice vector, so the lines cross the q_x axis at $q_x = \pm 1/3$ and $\pm 2/3$. These are the vectors that connect the parallel sides of the contour, which give the strongest interference. On the right, the cut at $q_y = 0$, each curve divided by its maximum. The peaks are at the same $\mathbf{q}$ for the T-matrix and for the Born approximation, since they come from the contour. The background is different: between $q_x = 1/3$ and $2/3$ the Born result drops almost to zero, while the T-matrix stays at about 0.3 of the peak. For this impurity $T \approx (1.1 - 1.1i)t$, so the real part of Λ_0 enters with the same weight as the imaginary part.

The code of this figure is in `documentation/figures/qpi_tmatrix.py`.

The keywords are

- **impurity**: a number, an onsite potential on the first site of the unit cell; a list with one onsite potential per site of the cell; or a matrix with the dimension of the Hamiltonian of the cell, for example a magnetic impurity $V \mathbf{m} \cdot \boldsymbol{\sigma}$ in a spinful system.
- **nk**: number of k-points per direction. The wavevectors are on the same mesh, so **nk** also sets the resolution in $\mathbf{q}$.
- **nsuper**: number of Brillouin zones per direction covered by **qs**.
- **energies, delta, write**: as in `mode="pm"`.

The sum over k-points on an **nk** x **nk** mesh is the same as one impurity in a periodic system of **nk** x **nk** unit cells. So the result is exact for that system, and it is the same as `h.get_qpi_impurity()` with `nsuper=nk` and `nk=1`, at every $\mathbf{q}$ except $\mathbf{q} = 0$, where the real-space map also contains the density of states of the clean system. For a single impurity in an infinite system, **nk** must be larger than the length over which the Friedel oscillations decay, which is about v_F/δ in units of the lattice constant, where v_F is the Fermi velocity. In the figure $v_F \approx 1.7ta$ and $\delta = 0.1t$, so this is about 20 unit cells, and **nk**=120 is enough. With a smaller δ , increase **nk**.

The impurity matrix also lets you check a selection rule. With time-reversal symmetry, the two states of a Kramers pair give opposite contributions for a magnetic impurity. So in the Born approximation a magnetic impurity gives no QPI at all, even with a Rashba spin-orbit coupling. The T-matrix gives a signal at second order in V . In the square lattice with a Rashba coupling of $0.4t$, at $\omega = -t$, the largest QPI intensity of a magnetic impurity with $V = 0.3t$ is about 14 times weaker than the one of a potential impurity of the same strength, and about 3 times weaker for $V = t$.

See `examples/2d/qpi_tmatrix/main.py` for a runnable script.

9 Band unfolding

In this page we see how the band structure of a supercell is folded into its smaller Brillouin zone, and how to unfold it back to the Brillouin zone of the primitive cell, with `operator="unfold"`.

9.1 Folding and unfolding

A supercell does not change the physics, but it changes how the bands are drawn. A supercell of N_c primitive cells has a Brillouin zone N_c times smaller, and the bands of the primitive cell fold into it, so the supercell has N_c times more bands, all of them copies of the same dispersion. We use a supercell when there is a reason for it: a defect, a moire pattern, a charge density wave. To compare with ARPES, we want the opposite operation. ARPES measures the spectral weight on the Bloch states of the primitive cell, so when the perturbation is weak it sees the bands of the primitive cell, and only weak shadows of the folded copies.

Let $\mathbf{R}$ be the positions of the N_c copies of the primitive cell inside the supercell, and α the orbitals of the primitive cell. The Bloch state of the primitive cell with momentum $\mathbf{k}$ is

$$|\mathbf{k}, \alpha\rangle = \sum_{\mathbf{R}} e^{i\mathbf{k}\cdot\mathbf{R}} |\mathbf{R}, \alpha\rangle,$$

and the unfolded spectral function is the spectral weight on these states,

$$A(\mathbf{k}, \omega) = \sum_{\alpha} \langle \mathbf{k}, \alpha | \delta(\omega - H) | \mathbf{k}, \alpha \rangle,$$

where H is the Hamiltonian of the supercell at the momentum of the supercell that corresponds to $\mathbf{k}$. The Bloch states are not normalized, so for a supercell without any perturbation $A(\mathbf{k}, \omega)$ is N_c times the spectral function of the primitive cell. In pyqula this projection is the operator "unfold". It can be given to the observables that are resolved in momentum, `h.get_bands()`, `h.get_kdos_bands()` and `h.get_multi_fermi_surface()`. The supercell must be built with `store_primal=True`, so that the primitive cell is known:

```
from pyqula import geometry
import numpy as np
g = geometry.honeycomb_lattice() # primitive geometry
n = 3
gs = g.get_supercell(n, store_primal=True) # supercell, keeping the primitive cell info
h = gs.get_hamiltonian() # Hamiltonian of the supercell
kpath = gs.get_unfolded_kpath(nk=200) # primitive k-path, in supercell coordinates
(k, e, d) = h.get_kdos_bands(operator="unfold", delta=1e-1, kpath=kpath) # unfolded weight
```

`gs.get_unfolded_kpath()` gives the k-path of the primitive cell, written in the coordinates of the supercell (see Which k-path to unfold along below). `d` is the unfolded weight at each $(\mathbf{k}, \mathbf{e})$. A scatter plot of $\mathbf{k}, \mathbf{e}$ colored by `d` gives the band structure of the primitive cell: the folded copies that a plain `h.get_bands()` of the supercell shows have no weight, and the Dirac cones of the honeycomb lattice appear at K, where they belong. To unfold a constant-energy cut, give the same operator to `h.get_multi_fermi_surface()` (see Spectral functions).

On the left, `h.get_bands(kpath=kpath)` of the supercell along the same path, with nine times more bands. In a 3×3 supercell K folds onto Γ , so the folded bands also touch at zero energy at Γ . On the right, the unfolded weight shows only the two bands of the primitive cell, from $-3t$ to $3t$, with the Dirac points at K and K'. The weight under each band is nine times the one of the primitive cell, since the Bloch states are not normalized.

The code of this figure is in `documentation/figures/unfolding_honeycomb.py`.

See `examples/2d/unfolding/main.py`, `examples/1d/unfolding/main.py` and `examples/readme_examples/unfolding_FS/m` for runnable scripts, and `jupyter-notebooks/functionalities/single_particle_hamiltonians/08_unfolding_supercell` for an executed notebook.

Unfolding also works when sites are removed from the supercell, for example a vacancy or a flake cut out of the supercell, with `gs = gs.remove([...])` before `gs.get_hamiltonian()`. Each remaining site is matched to its

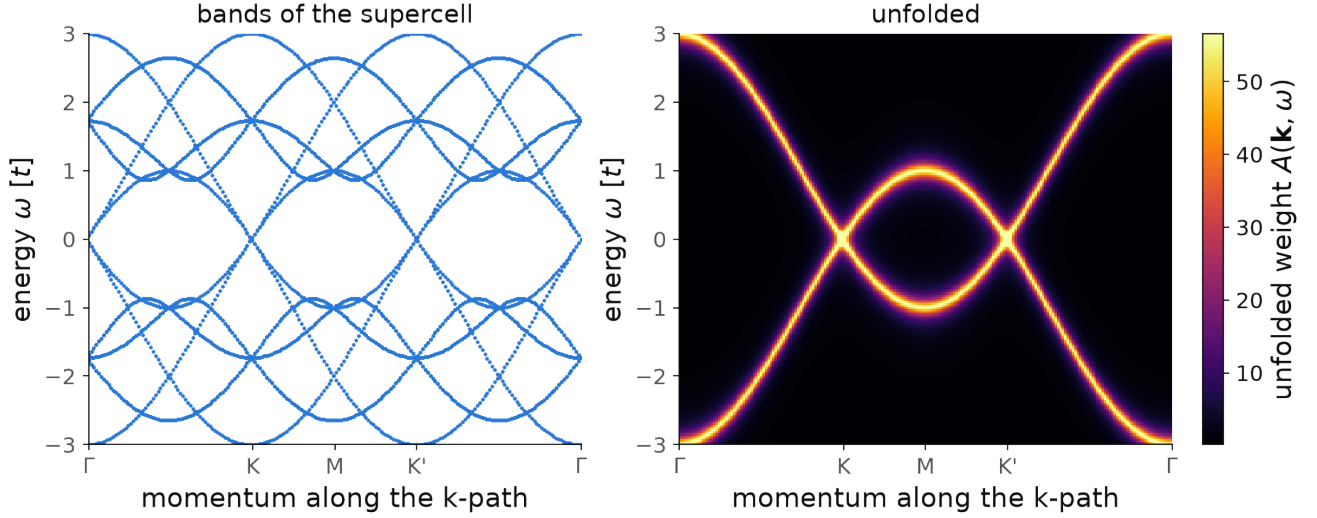


Figure 26: Band structure of a 3x3 supercell of the honeycomb lattice, and the unfolded spectral function along the primitive k-path.

copy of the primitive cell by its position, so the sites must not be moved (for example with `gs.center()` or a relaxation) between `gs.remove(...)` and `gs.get_hamiltonian()`. If they are moved, the unfolding raises a `ValueError`.

9.2 General supercells

A supercell does not need to be $n \times n$. Its lattice vectors can be any integer combinations of the primitive ones, $\mathbf{A} = M\mathbf{a}$, with M a 3x3 integer matrix, and $N_c = |\det M|$. Give M to `g.get_supercell()`, and the unfolding works in the same way:

```
from pyqula import geometry
g = geometry.honeycomb_lattice() # primitive geometry
M = [[2,1,0],[0,1,0],[0,0,1]] # non-diagonal supercell matrix, det(M)=2
gs = g.get_supercell(M,store_primal=True) # supercell, keeping the primitive cell info
h = gs.get_hamiltonian() # Hamiltonian of the supercell
kpath = gs.get_unfolded_kpath(nk=200) # primitive k-path, in supercell coordinates
(k,e,d) = h.get_kdos_bands(operator="unfold",delta=1e-1,kpath=kpath) # unfolded weight
```

Nothing in the formula above depends on the dimension, so it works in the same way for a chain, a two-dimensional lattice and a three-dimensional crystal:

```
from pyqula import geometry
g = geometry.cubic_lattice() # three dimensional primitive geometry
gs = g.get_supercell([2,2,2],store_primal=True) # supercell, keeping the primitive cell info
h = gs.get_hamiltonian() # Hamiltonian of the supercell
kpath = gs.get_unfolded_kpath(nk=200) # primitive k-path, in supercell coordinates
(k,e,d) = h.get_bands(operator="unfold",kpath=kpath) # unfolded band structure
```

On the left, the non-diagonal supercell with $\det M = 2$ unfolds to the same two bands of the honeycomb lattice as the 3×3 one above. On the right, the $2 \times 2 \times 2$ supercell of the cubic lattice has eight bands, and the color is the unfolded weight of each state. One band has the full weight $8 = N_c$: it goes from $6t$ at Γ to $2t$ at X, $-2t$ at M, back to $6t$ at Γ and to $-6t$ at R, which is the band $2t(\cos k_x + \cos k_y + \cos k_z)$ of the primitive cell. The other bands are the folded copies, with zero weight. Between X and M the band is degenerate with a folded copy, and the weight is shared between the two.

The code of this figure is in `documentation/figures/unfolding_general.py`.

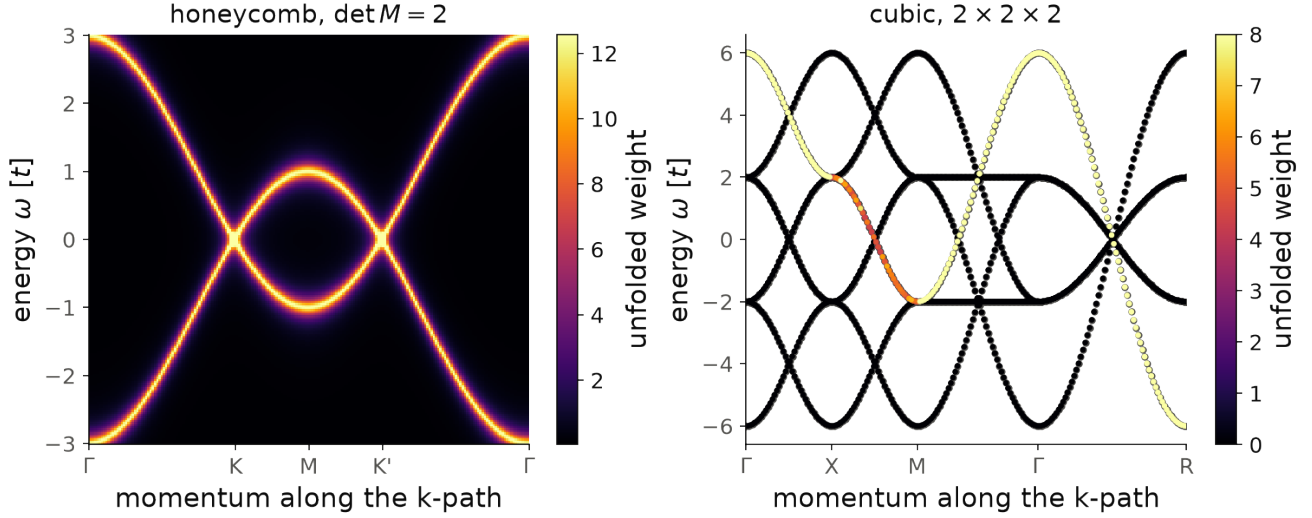


Figure 27: Unfolded spectral function of a non-diagonal supercell of the honeycomb lattice, and unfolded bands of a $2 \times 2 \times 2$ supercell of the cubic lattice.

See `examples/2d/unfolding_nonorthogonal/main.py` for a runnable script with a defect in a non-diagonal supercell, and `examples/3d/unfolding/main.py` for the three-dimensional case.

A supercell can also be asked for by its size only. The most common case is the $\sqrt{3} \times \sqrt{3}$ supercell of a triangular or honeycomb lattice, the unit cell of a charge order with three sublattices:

```
from pyqula import geometry
import numpy as np
g = geometry.triangular_lattice() # primitive geometry
gs = g.get_supercell(np.sqrt(3), store_primal=True) # sqrt(3) x sqrt(3) supercell
```

A non-integer size s means a cell s^2 times larger in area, here three primitive cells. `g.get_supercell()` returns the most compact cell of this size. For the triangular lattice it has $|\mathbf{A}_1| = |\mathbf{A}_2| = \sqrt{3}$ in units of the lattice constant, with 60 degrees between them, and it is rotated so that $\mathbf{A}_1$ is along x . The matrix that was used is `gs.supercell_matrix`. To choose the matrix and the orientation yourself, give an explicit M .

9.3 Which k-path to unfold along

The unfolded bands must be drawn along a path of the Brillouin zone of the primitive cell. A momentum $\mathbf{k}_0$ of the primitive cell corresponds to the momentum $\mathbf{k} = M\mathbf{k}_0$ in the reduced coordinates of the supercell. For an $n \times n$ supercell this is the primitive path multiplied by n . For the $\sqrt{3} \times \sqrt{3}$ supercell it is also a rotation, which no multiplication by a number can give. `gs.get_unfolded_kpath()` returns the right path for any supercell. Here we add a three-sublattice charge order to the $\sqrt{3} \times \sqrt{3}$ supercell of the triangular lattice:

```
from pyqula import geometry
import numpy as np
g = geometry.triangular_lattice() # primitive geometry
gs = g.get_supercell(np.sqrt(3), store_primal=True) # sqrt(3) x sqrt(3) supercell
h = gs.get_hamiltonian() # Hamiltonian of the supercell
fons = lambda r: (np.sum((r - gs.r[0])**2) < 1e-2) * 0.6 # onsite in one of the three sites
h.add_onsite(fons) # three-sublattice charge order
kpath = gs.get_unfolded_kpath(nk=200) # primitive k-path, in supercell coordinates
(k, e, d) = h.get_kdos_bands(operator="unfold", delta=1e-1, kpath=kpath) # unfolded weight
```

The k-points are in the reduced coordinates of the supercell, which is what `kpath` takes. `gs.get_unfolded_kpath()` takes the same arguments as `g.get_kpath()`, so the high-symmetry points can be named, `gs.get_unfolded_kpath(["G", "K", "R", "X", "M", "K'])`, and they are the points of the primitive cell. In the plot of `k, e` colored by `d` you see the single band of the

triangular lattice along Γ -K-M-K'- Γ , with a gap opened at K by the charge order, and a weak shadow band, the folded copy, with the small weight that the modulation gives it. Along the k-path of the supercell, `gs.get_kpath()`, the same calculation goes along a path of the primitive Brillouin zone that has no special meaning. See `examples/2d/unfolding_sqrt3/main.py` for a runnable script.

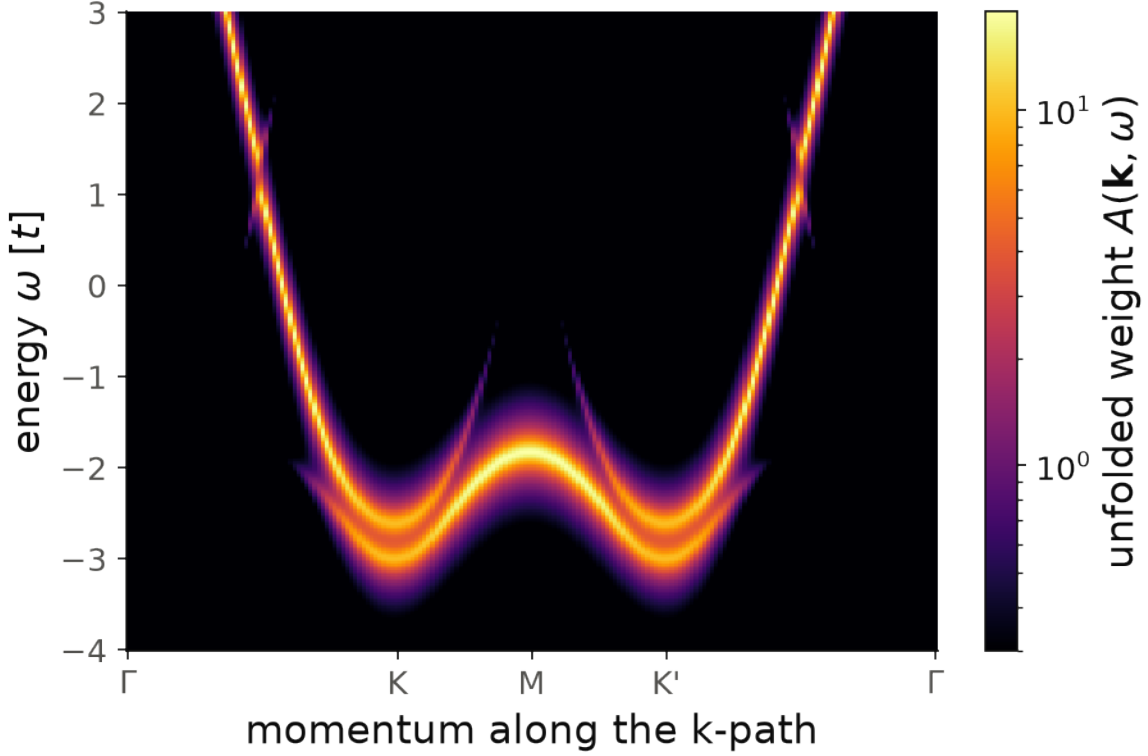


Figure 28: Unfolded spectral function of the triangular lattice with a three-sublattice charge order, on a logarithmic color scale.

For the figure we take `energies=np.linspace(-4,3,300)`, since the default window starts at -3 , the bottom of the band. The color scale is logarithmic, so that the weak shadow band is visible. At K and K' the charge order splits the bottom of the band into two states, at $-3.0t$ and $-2.6t$. The shadow band is the faint line that rises from M up to about $-0.5t$, and it crosses the main band at about $1.3t$, once between Γ and K and once between K' and Γ , where a small gap opens.

The code of this figure is in `documentation/figures/unfolding_sqrt3.py`.

9.4 Unfolding a large supercell with the kernel polynomial method

The cases where unfolding matters most, a moire pattern or a disordered region, have the largest supercells, and there diagonalizing the Hamiltonian at each k-point is too expensive. With `h.get_kdos_bands(operator="unfold",mode="KPM")` each term $\langle \mathbf{k}, \alpha | \delta(\omega - H) | \mathbf{k}, \alpha \rangle$ of the formula above is computed with a Chebyshev expansion (see Kernel polynomial method), started from the Bloch state $|\mathbf{k}, \alpha\rangle$ itself. So no random vectors are needed, and the result has no statistical noise:

```
import numpy as np
from pyqula import geometry
from pyqula import potentials
g0 = geometry.triangular_lattice() # primitive geometry
g = g0.get_supercell(12,store_primal=True) # supercell of 144 primitive cells
h = g.get_hamiltonian() # Hamiltonian of the supercell
fmoire = potentials.commensurate_potential(g,n=3,minmax=[0,1]) # moire potential
h.add_onsite(fmoire) # onsite energy following the moire
```

```
kpath = g.get_unfolded_kpath(nk=300) # primitive k-path, in supercell coordinates
(k,e,d) = h.get_kdos_bands(operator="unfold",mode="KPM",delta=5e-2,kpath=kpath,
    energies=np.linspace(-3,-1,200)) # unfolded weight, with the KPM
```

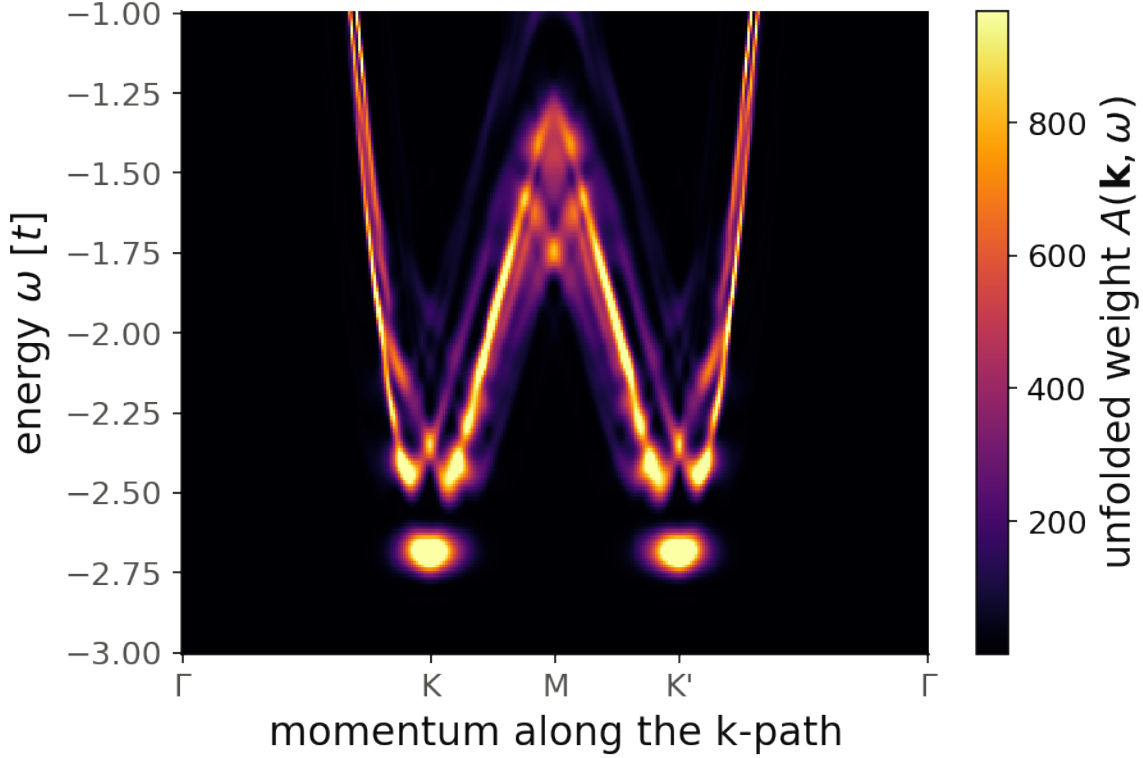


Figure 29: Unfolded spectral function of a triangular lattice with a moire potential, computed with the kernel polynomial method.

The moire potential splits the bottom of the band into minibands. The lowest state is at K and K', at about $-2.7t$, above the $-3t$ of the clean lattice because the potential is positive. It is separated by a gap from the next minibands, which start at about $-2.5t$. Around M you see several copies of the band, shifted by the moire wavevectors, each one with a part of the weight.

The k-path needs more points than for a band structure. Between two neighboring k-points a steep band moves in energy, and if this step is larger than `delta` the band shows up as a row of separate dots. With `nk=100` the steep branches of the map break into such dots. With `nk=300` they are continuous lines, and the map does not change any more when we go to `nk=600`. There is no other convergence parameter: the expansion has no random vectors, and the number of polynomials follows from `delta`.

The code of this figure is in `documentation/figures/unfolding_kpm.py`.

The output is the same as with `mode="ED"`, with the same weight under each band. Only the line shape is different: the Chebyshev expansion broadens each band with its kernel, with a number of polynomials set by `delta`, and not with a Lorentzian. The cost of each k-point grows linearly with the number of sites of the supercell, not with its cube, so supercells with thousands of sites are possible. A smaller `delta` needs proportionally more polynomials.

10 Altermagnets

An altermagnet is a collinear magnet with zero net magnetization, but with spin-up and spin-down bands that are different. The two spin channels are related by a rotation or a mirror of the lattice, not by a translation or by inversion, so the spin splitting changes sign in reciprocal space, with d-wave, g-wave or i-wave symmetry. In

this page we see how to measure how large the spin splitting is, and how to find its symmetry from the nonlinear spin current.

10.1 Spin splitting of an altermagnet

When spin is a good quantum number, the Bloch Hamiltonian has one block per spin, and we can diagonalize each block alone. This gives energies $E_n^\uparrow(\mathbf{k})$ and $E_n^\downarrow(\mathbf{k})$, sorted by band index n . For each $\mathbf{k}$ and n we define the spin splitting and its mean energy,

$$\Delta_n(\mathbf{k}) = E_n^\uparrow(\mathbf{k}) - E_n^\downarrow(\mathbf{k}), \quad \bar{E}_n(\mathbf{k}) = \frac{E_n^\uparrow(\mathbf{k}) + E_n^\downarrow(\mathbf{k})}{2}.$$

The splitting is what spin-resolved ARPES sees as the energy difference of the two spin bands. pyqula gives two functions of energy built from the pairs $(\bar{E}_n(\mathbf{k}), \Delta_n(\mathbf{k}))$ on a k-mesh of the whole Brillouin zone:

- `h.get_spin_splitting_vs_energy()`: the largest $|\Delta_n(\mathbf{k})|$ in each energy bin. Its maximum is the largest spin splitting anywhere in the Brillouin zone.
- `h.get_spin_splitting_density()`: each pair broadened with a width `delta`, giving a smooth weighted density, the typical splitting at each energy.

To quote one number for a material, use the first one. The maximum along a single cut in reciprocal space (a circle, say) depends on where you look; the maximum over the full mesh does not.

```
import numpy as np
from pyqula import specialhamiltonian

h = specialhamiltonian.square_altermagnet(am=1.)

# largest spin splitting in each energy bin, over the whole BZ
(es, ds) = h.get_spin_splitting_vs_energy(nk=800, nbins=400)
print("largest spin splitting anywhere in the BZ:", ds.max())

# the smooth counterpart, on the same axes
(xs, ys) = h.get_spin_splitting_density(nk=100, delta=1e-1, energies=es)
```

Both return (energies, values) with the same shape, so they can be plotted together. An energy bin without states gives 0.0.

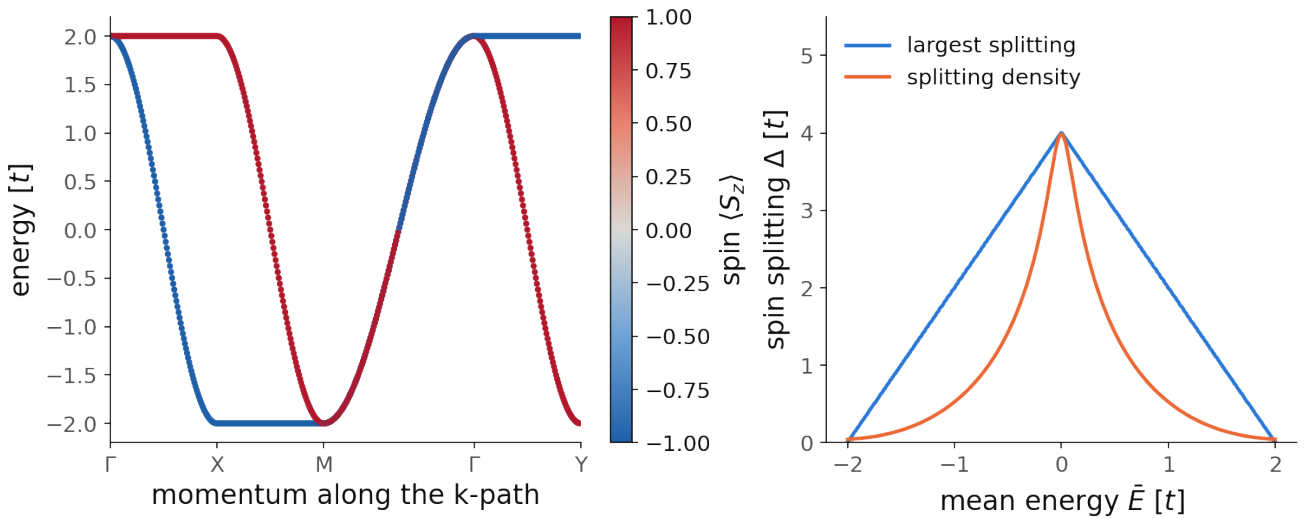


Figure 30: Bands of the square altermagnet colored by spin, and its spin splitting as a function of energy.

The left panel shows the bands of this model along the path Γ -X-M- Γ -Y, colored by the spin of each state with `h.get_bands(operator="sz")`. With `am=1` the spin-up electrons hop only along y and the spin-down electrons only along x , so $E^\uparrow = 2 \cos k_y$ and $E^\downarrow = 2 \cos k_x$. Along Γ -X the spin-up band (red) is flat at 2 and the spin-down band (blue) goes down to -2 . Along Γ -Y it is the opposite, and along M- Γ the two bands are the same. This sign change of the splitting between the two directions is the d-wave symmetry. The right panel is the output of the snippet. The largest splitting is 4.0, at mean energy 0, which is the X point, where $E^\uparrow = 2$ and $E^\downarrow = -2$. The largest splitting per bin (blue) is the triangle $\Delta_{\max} = 4 - 2|\bar{E}|$. At a fixed mean energy the splitting is largest when one spin sits at the top of its band, at 2, so $E^\uparrow - E^\downarrow = 2 - (2\bar{E} - 2) = 4 - 2\bar{E}$ for $\bar{E} > 0$. This curve needs a fine mesh, because each of the 400 bins must hold a k-point close to where its maximum is reached. With `nk=100` many bins fall far below the triangle, and some are empty and give zero. With `nk=800` every bin is within 0.005 of the triangle, and `nk=1600` changes nothing visible. This takes about a minute. The density (orange) is smooth, with the same maximum, and it is already converged at `nk=100`.

The code of this figure is in `documentation/figures/alttermagnets_splitting.py`.

The bands are paired by their index n , and you may ask if the n -th up band and the n -th down band are really partners. In an altermagnet this is safe. The two spin channels are related by a point-group operation R , so $E_n^\uparrow(\mathbf{k}) = E_n^\downarrow(R\mathbf{k})$ exactly, and $\Delta_n(\mathbf{k}) = E_n^\downarrow(R\mathbf{k}) - E_n^\downarrow(\mathbf{k})$ compares the same band at two momenta. This holds even when the splitting is larger than the distance between bands. For the square altermagnet R is the mirror $k_1 \leftrightarrow k_2$.

There is one case where the pairing gives a wrong number: a unit cell larger than the magnetic unit cell. In a supercell, the bands at one $\mathbf{k}$ come from several k-points of the primitive cell, and pairing them by index mixes them. The maximum then comes out too small. So use the true magnetic unit cell. If the magnetic order repeats with a shorter period than the cell you used, the result is only a lower bound.

Both functions need spin to be a good quantum number. `h.get_spin_splitting_vs_energy()` raises an error if the Bloch Hamiltonian mixes the two spins, as with Rashba coupling, other spin-orbit coupling, or non-collinear magnetic order. In these cases Δ_n is not defined.

See `examples/2d/spin_splitting_vs_energy/main.py` for a runnable script that plots both curves and shows the problem of a too large cell, `examples/2d/alttermagnetism_density/main.py` for the density alone, and `jupyter-notebooks/functionalities/single_particle_hamiltonians/11_altermagnet_spin_splitting.ipynb` for an executed notebook.

10.2 Nonlinear spin current as a measurement of altermagnetic order

The spin splitting tells how large the altermagnetic order is. We now ask which kind it is, d-wave, g-wave or i-wave. The answer can be found with a purely electrical measurement, without spin-orbit coupling: the lowest order in the electric field at which a spin current appears. This is computed with `h.get_nonlinear_drude_orders()`.

The spin splitting of an X-wave magnet is a lattice harmonic of order $l+1$ in $\mathbf{k}$: $k_x k_y$ for d-wave, $k_x k_y (k_x^2 - k_y^2)$ for g-wave, $k_x k_y (3k_x^2 - k_y^2)(k_x^2 - 3k_y^2)$ for i-wave. In the semiclassical Boltzmann theory, with relaxation time τ and frequency ω , the current of spin s at order $l = l_1 + l_2$ in the field, $j_b = \sigma_s^{x^{l_1} y^{l_2}; b} E_x^{l_1} E_y^{l_2}$, has the conductivity

$$\sigma_s^{x^{l_1} y^{l_2}; b} = \frac{(-e/\hbar)^{l+1}}{(i\omega + 1/\tau)^l} \int \frac{d^D k}{(2\pi)^D} f_s^{(0)}(\mathbf{k}) \frac{\partial^{l+1} \epsilon_s(\mathbf{k})}{\partial k_x^{l_1} \partial k_y^{l_2} \partial k_b},$$

where $\epsilon_s(\mathbf{k})$ is the band of spin s , $f_s^{(0)}$ its Fermi occupation, and b the direction of the current. The spin and charge conductivities are

$$\sigma_{\text{spin}} = \frac{\sigma_\uparrow - \sigma_\downarrow}{2}, \quad \sigma_{\text{charge}} = \sigma_\uparrow + \sigma_\downarrow.$$

A harmonic of order $l+1$ has a constant $(l+1)$ -th derivative, and all lower derivatives vanish after the integral over the zone. So the spin current appears for the first time at one order l , and is exactly zero below it:

wave	d	f	g	i
lowest order	1	2	3	5

The p-wave magnet, with splitting $\propto k_x$, would be at $l = 0$, but it has no spin current at any order. First, the order $l = 0$ is zero for every band: the integrand $f \partial \epsilon / \partial k_b$ is a total derivative, so there is no current in equilibrium. Second, on the lattice the p-wave splitting $J \sin k_x$ and the hopping $2t \cos k_x$ combine into a single cosine of amplitude $\sqrt{4t^2 + J^2}$, so the two spin bands are the same band shifted along k_x in opposite directions, and an integral over the zone does not see a shift. On a finite k-mesh these two zeros are small numbers instead of exact zeros: at **nk**=48 the p-wave magnet gives 3×10^{-4} at the even orders, and this goes down exponentially with **nk**, to 10^{-11} at **nk**=384. The zeros below the threshold of the d-, f-, g- and i-wave magnets come from symmetry and are exact on any mesh. So a number that becomes smaller when you increase **nk** is a zero, not the threshold.

So the lowest order with a nonzero spin current gives the wave index. This is the proposal of Ezawa, *Phys. Rev. B* **111**, 125420 ([arXiv:2411.16036](https://arxiv.org/abs/2411.16036)). For an i-wave altermagnet:

```
from pyqula import specialhamiltonian

h = specialhamiltonian.iwave_altermagnet(J=0.3)  # or dwave_/fwave_/gwave_

# largest |sigma_spin| over every component of each order l
orders = h.get_nonlinear_drude_orders(lmax=6, nk=48, T=0.02, mu=-5.5)
# -> zero to machine precision for l = 0..4, nonzero at l = 5

# one component, and every component of a given order
s = h.get_nonlinear_drude_conductivity(field="yyyyy", current="x", mu=-5.5)
c = h.get_nonlinear_drude_components(5, nk=48, mu=-5.5)  # {"x~l1 y~l2;b": value}
```

`h.get_nonlinear_drude_orders()` returns, for each order l up to `lmax`, the largest $|\sigma_{\text{spin}}|$ over all the components of that order. Here it is zero (to machine precision) for l from 0 to 4 and nonzero at $l = 5$, the fingerprint of i-wave order. The keywords are:

- **field**: one character per power of the electric field, so "yyyyy" is the fifth-order response to E_y .
- **current**: the direction b of the current.
- **channel**: "spin" (default), "charge", "up" or "dn".
- **mu**, **T**: the chemical potential and the temperature of $f^{(0)}$.
- **nk**: the k-mesh, per direction.

Units are $e = \hbar = 1$, and the integral is normalized per unit area (or volume), $\int d^D k / (2\pi)^D$.

The models are `specialhamiltonian.xwave_magnet(wave=...)`, with the names `pwave_magnet`, `dwave_altermagnet`, `fwave_magnet`, `gwave_altermagnet` and `iwave_altermagnet`. These are Ezawa's models, each on the lattice its symmetry needs: square for p, d and g, triangular for f and i.

Each row is one model of `specialhamiltonian.xwave_magnet()`, with $J = 0.3$ and the chemical potential 0.5 above the bottom of its band, as $\mu = -5.5$ in the snippet for the i-wave one. The color is $\log_{10}$ of the output of `h.get_nonlinear_drude_orders()`, and the zeros are drawn at 10^{-16} . In each row, the first bright cell is the threshold of the table: $l = 1$ for d-wave (0.011), $l = 2$ for f-wave (0.041), $l = 3$ for g-wave (0.053) and $l = 5$ for i-wave (0.56). Every cell below it is black, an exact zero. The d-, f- and g-wave rows also respond at higher orders, the higher harmonics. The p-wave row is the mesh error discussed above: 3×10^{-4} at the even orders at **nk**=48, and it goes down when you increase **nk**.

The code of this figure is in `documentation/figures/altermagnets_orders.py`.

Some things to keep in mind before using the result:

- The orders above the threshold are not zero. A lattice harmonic also contains higher harmonics, so a d-wave altermagnet responds at $l = 1$, and also at $l = 3$ and $l = 5$. The information is in the absence of all the orders below the threshold.

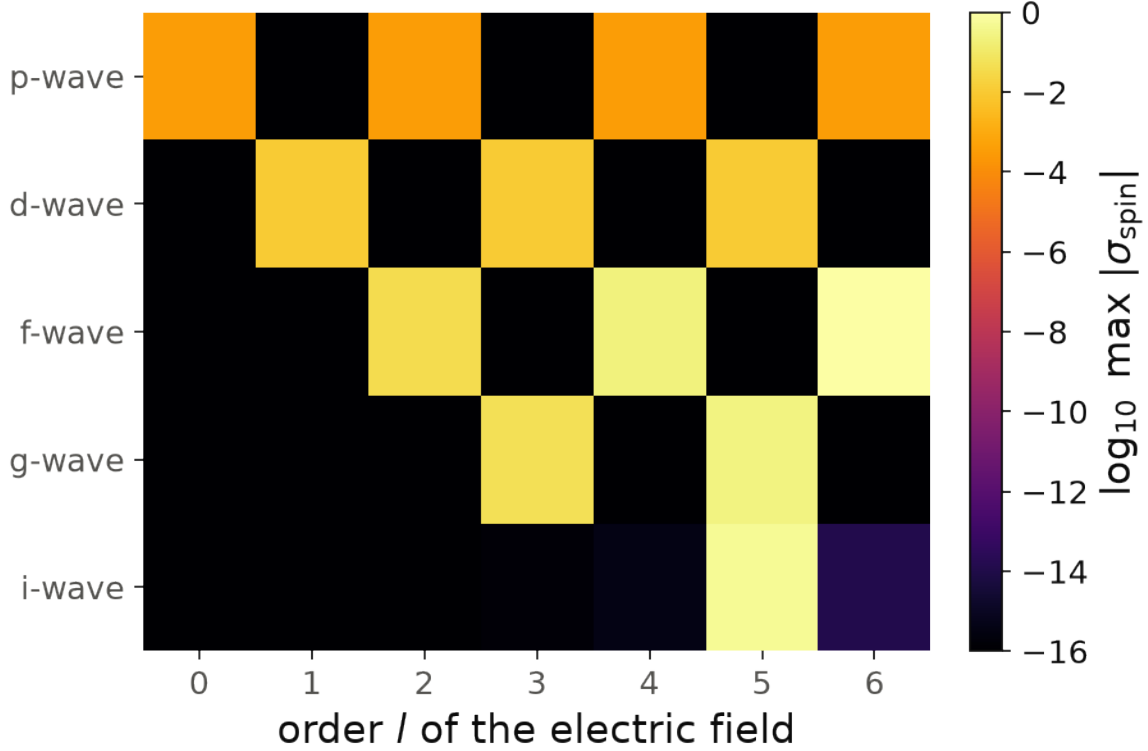


Figure 31: Largest nonlinear spin conductivity at each order of the electric field, for the p-, d-, f-, g- and i-wave magnets.

- An insulator gives zero at every order. With the chemical potential in the gap, every band is full or empty, and the integral vanishes, also at fifth order. So “nothing below fifth order” says i-wave only if the fifth order is nonzero, which needs a metal.
- A spin-degenerate state has no spin current. A compensated Néel state on a bipartite lattice has PT symmetry and its two spin channels are exactly degenerate: it is an antiferromagnet, not an altermagnet. In this case the result is numerical noise that does not look small, so a warning is raised when the spin splitting is below 10^{-10} of the bandwidth.

Real hoppings also give a selection rule, useful when building a model. With real hoppings and no spin-orbit coupling, $\epsilon_s(\mathbf{k}) = \epsilon_s(-\mathbf{k})$, so the splitting is even under a rotation by 180° and all odd harmonics vanish. A C_3 axis also removes every harmonic whose order is not a multiple of three. Only the orders 6, 12, ... survive, which is $l = 5, 11, \dots$. So a real, spin-orbit-free cell with C_3 symmetry is always i-wave, and no pattern of vacancies or supercell can make it f-wave. The models agree with this: p-wave and f-wave need complex hoppings, while d-, g- and i-wave are real.

Spin must be a good quantum number, because the formula treats each spin channel alone. A Hamiltonian with spin-orbit coupling is rejected: with Rashba or Kane-Mele terms the nonlinear response has extra contributions from the quantum metric and the Berry curvature dipole ([arXiv:2409.09241](https://arxiv.org/abs/2409.09241)), which are not in this formula.

Unit cells with many orbitals (superlattices, antidot lattices, multilayers) are supported, and they are the interesting case. The derivatives of the bands are computed analytically, also at the degeneracies that a C_3 or C_6 axis creates at high-symmetry points, so an order forbidden by symmetry comes out as an exact zero.

See `examples/2d/xwave_nonlinear_spin_current/main.py` for a runnable script that prints the whole table, and `jupyter-notebooks/functionalities/single_particle_hamiltonians/12_nonlinear_spin_conductivity.ipynb` for an executed notebook.

11 Superconductivity

So far all our Hamiltonians contained only normal terms,

$$H_0 = \sum_{ijss'} t_{ijss'} c_{i,s}^\dagger c_{j,s'},$$

where i, j run over the sites and s, s' over the spin. Each term destroys one electron and creates one, so the number of electrons is conserved. In a superconductor there is also an anomalous term,

$$H_{SC} = \sum_{ijss'} \Delta_{ij}^{ss'} c_{i,s} c_{j,s'} + h.c.,$$

which destroys or creates a pair of electrons, so the number of electrons is not conserved anymore. The pairing amplitude $\Delta_{ij}^{ss'}$ is the superconducting order parameter. In this page we add it by hand. In Mean-field Hubbard models we obtain it self-consistently from an attractive interaction.

11.1 The Nambu basis

To solve $H = H_0 + H_{SC}$ we use the Nambu spinor of site n ,

$$\Psi_n = \begin{pmatrix} c_{n,\uparrow} \\ c_{n,\downarrow} \\ c_{n,\downarrow}^\dagger \\ c_{n,\uparrow}^\dagger \\ -c_{n,\uparrow} \end{pmatrix}.$$

You can think of it as the two electron operators of the site, stacked on top of their conjugates. In this basis a term that creates two electrons becomes a hopping between the upper and the lower half of the spinor, so the problem is again a single-particle one,

$$H = \frac{1}{2} \Psi^\dagger \mathcal{H} \Psi + \text{const},$$

where $\mathcal{H}$ is the Bogoliubov-de Gennes (BdG) Hamiltonian, a matrix twice as large as the normal one. Its lower diagonal block is the hole copy of H_0 , and its off-diagonal blocks contain Δ . Diagonalizing it gives

$$H = \sum_{\alpha} \epsilon_{\alpha} \gamma_{\alpha}^\dagger \gamma_{\alpha} + \text{const},$$

with γ_{α} the Bogoliubov quasiparticles. Because the lower half of the spinor is the conjugate of the upper half, the eigenvalues of $\mathcal{H}$ come in pairs $\pm \epsilon_{\alpha}$: every state has an electron copy and a hole copy. The gap of the spectrum around zero energy is $2 \min |\epsilon_{\alpha}|$.

`h.setup_nambu_spinor()` builds this doubled basis. `h.add_swave()` and `h.add_pairing()` build it themselves if it is not there. After this, every observable (band structure, density of states, LDOS) is computed in the Nambu basis. A Nambu Hamiltonian in pyqula is always spinful. For a superconducting mean-field calculation we call `h.setup_nambu_spinor()` before starting, as shown in Mean-field Hubbard models.

11.2 s-wave superconductivity

The simplest superconductor has spin-singlet s-wave pairing,

$$H_{SC} = \Delta_0 \sum_n c_{n,\uparrow} c_{n,\downarrow} + h.c.,$$

meaning that an electron with spin up and an electron with spin down on the same site form a pair, with the same amplitude Δ_0 on every site. This is the pairing of a conventional superconductor. We take a triangular lattice,

$$H = H_0 + H_{SC}, \quad H_0 = \sum_{\langle ij \rangle} c_i^\dagger c_j + h.c.,$$

and add the pairing $\Delta_0 = 0.2$ with `h.add_swave()`:

```
from pyqula import geometry
g = geometry.triangular_lattice() # geometry of the 2D model
h = g.get_hamiltonian() # generate the Hamiltonian
h.add_swave(0.2) # add s-wave superconductivity
(k,e) = h.get_bands() # compute band structure
```

The BdG band structure shows both the electron and the hole bands. A gap opens at zero energy, from $-\Delta_0$ to $+\Delta_0$, so it is $2\Delta_0$ wide. It opens at every point of the Fermi surface, whatever its momentum: this is the signature of s-wave pairing, and it is the hard gap that tunneling spectroscopy (STM) measures in a conventional superconductor. With `operator="electron"` in `h.get_bands()` each state is colored by its electron weight, so you can tell the electron copy from the hole copy, and read the electron spectral function, which is what ARPES measures.

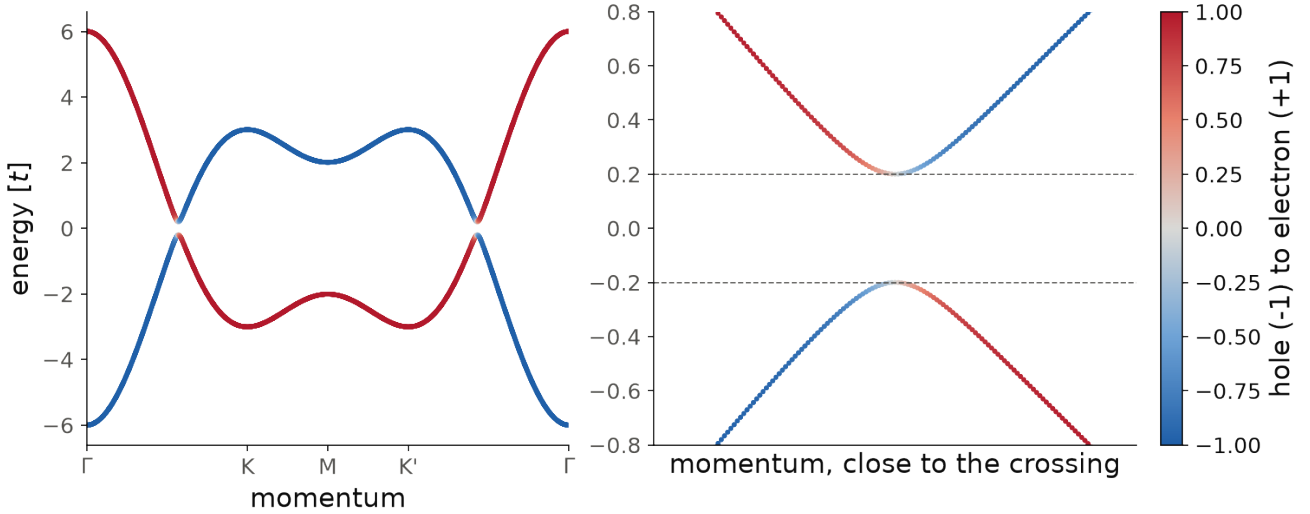


Figure 32: Bogoliubov-de Gennes bands of the triangular lattice with s-wave pairing, colored by the electron or hole character of each state.

The figure uses `nk=3000` points on the k-path, so that the zoom on the right is smooth. Red states are electrons and blue states are holes. The electron band goes from 6 at Γ to -3 at K, and the hole band is its mirror image. Where the two bands cross zero energy they mix, the color goes to the middle of the scale, and a gap opens. The zoom on the right shows the gap edges at ± 0.2 (dashed lines), so the gap is $2\Delta_0 = 0.4$.

The code of this figure is in `documentation/figures/superconductivity_swave.py`.

The argument of `h.add_swave()` can also be a function of the position, to have a pairing that changes in space. For other pairing symmetries (p-wave, d-wave, a general d-vector) use `h.add_pairing(delta=...,mode=...)`.

See `examples/2d/comparison_swave/main.py` for a runnable script, which adds the same gap with `h.add_swave()` and with `h.add_pairing(mode="swave")`. The notebooks `jupyter-notebooks/functionalities/single_particle_hamiltonians/03_magnetism_soc_superconductivity.ipynb` and `jupyter-notebooks/functionalities/single_particle_hamiltonians/03_magnetism_soc_superconductivity.ipynb` show the Nambu basis, and superconductivity together with magnetism and spin-orbit coupling.

11.3 Spin-triplet pairing and the d-vector

In the s-wave case the two electrons of a pair form a spin singlet. The other possibility is a spin triplet, and we now see how to characterize it with `h.get_dvector_non_unitarity()`. For a triplet the pairing is a symmetric 2×2 matrix in spin space, and we write it with a complex three-component vector $\vec{d}$, the d-vector,

$$\Delta = i(\vec{d} \cdot \vec{\sigma})\sigma_y = \begin{pmatrix} -d_x + id_y & d_z \\ d_z & d_x + id_y \end{pmatrix}.$$

So d_z is the opposite-spin pairing $\Delta_{\uparrow\downarrow}$, and d_x, d_y give the equal-spin pairings $\Delta_{\uparrow\uparrow}$ and $\Delta_{\downarrow\downarrow}$. The component $\Delta_{\uparrow\downarrow}$ can also contain a singlet part. The singlet is odd when we exchange the two spins and even when we exchange the two sites, the opposite of the triplet, and pyqula removes it before computing d_z . So a state with both singlet and triplet pairing gives the d-vector of the triplet part only.

Triplet states are of two kinds. We compute

$$\Delta\Delta^\dagger = |\vec{d}|^2\mathbb{1} + \vec{q} \cdot \vec{\sigma}, \quad \vec{q} = i(\vec{d} \times \vec{d}^*).$$

- $\vec{q} = 0$: the state is unitary. $\Delta\Delta^\dagger$ is proportional to the identity, the two quasiparticle branches have the same gap, and the Cooper pairs have no net spin.
- $\vec{q} \neq 0$: the state is non-unitary. The two spin branches have different gaps, and the pairs carry a spin moment $\vec{q}$, which is a real vector.

If all the components of $\vec{d}$ have the same phase, the state is always unitary. Non-unitarity needs a relative phase between the components, as in $\vec{d} \propto (1, i, 0)$, which is pure $\Delta_{\uparrow\uparrow}$ pairing with $\vec{q}$ along $+z$. So a non-unitary state breaks time-reversal symmetry. It appears naturally in ferromagnetic superconductors, where the pairs form in the majority band and $\vec{q}$ is parallel to the magnetization.

`h.get_dvector_non_unitarity()` returns $\vec{q}$ for each site, an array of shape (number of sites, 3). The d-vector is computed on a uniform k-mesh, $\vec{q}$ is averaged over the mesh, and the contributions of all the pairing partners of a site are summed. As an example, we add by hand a p-wave pairing with $\vec{d} \propto (1, i, 0)$:

```
from pyqula import geometry
g = geometry.chain() # geometry of the 1D model
h = g.get_hamiltonian() # generate the Hamiltonian
h.setup_nambu_spinor() # initialize the Nambu basis
h.add_pairing(delta=0.3, mode="pwave", d=[1., 1j, 0.]) # up-up pairing
q = h.get_dvector_non_unitarity(nk=20) # -> [[0., 0., 0.36]]
```

The array has one row, for the single site of the chain, and only its z component is not zero, because of the relative phase i between d_x and d_y . The same p-wave pairing with a real d-vector, $\mathbf{d}=[1., 0., 0.]$, is unitary and gives zero.

Here we compare the two d-vectors, with the same `delta=0.3`. With the real d-vector $(1, 0, 0)$, $\vec{q} = 0$ and both spins have the same gap, with edges at ± 0.6 . The two spin bands have the same energy, so the color of each state is a mix of the two. With $\vec{d} = (1, i, 0)$ only the spin-up electrons are paired. Their gap is twice as large, with edges at ± 1.2 , and the spin-down bands cross zero energy without any gap. These are the two different gaps of a non-unitary state.

The code of this figure is in `documentation/figures/superconductivity_dvector.py`.

A non-unitary state can also come out of a self-consistent calculation. With an attractive first-neighbor interaction and a large exchange field (as in the long-range interaction example of Mean-field Hubbard models), the ground state is a spin triplet whose $\vec{q}$ follows the magnetization:

```
from pyqula import geometry
g = geometry.triangular_lattice() # generate the geometry
h = g.get_hamiltonian() # create Hamiltonian of the system
h.add_exchange([0., 0., 3.]) # add exchange field
h.setup_nambu_spinor() # initialize the Nambu basis
```

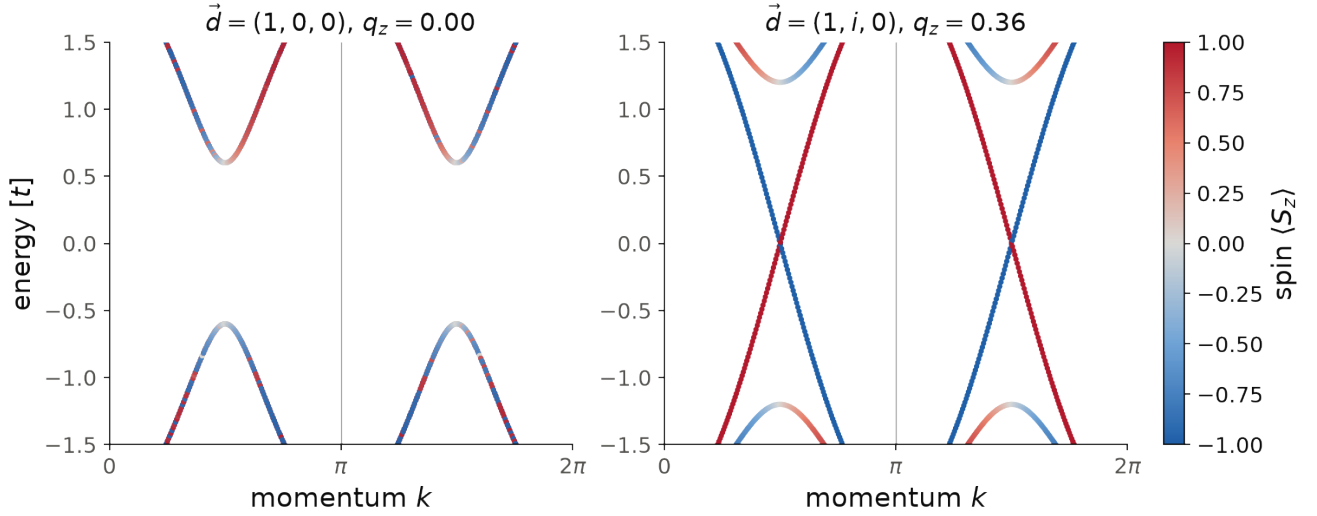


Figure 33: Bogoliubov-de Gennes bands of the chain with a unitary and a non-unitary p-wave pairing, colored by the spin of each state.

```
# perform a superconducting non-collinear mean-field calculation
h = h.get_mean_field_hamiltonian(V1=-1.5,filling=0.3,mf="random",nk=30)
q = h.get_dvector_non_unitarity(nk=30) # antiparallel to +z, as is h.get_magnetization()
```

Other related methods:

- `h.write_non_unitarity()`: writes $\vec{q}$ as a map in real space, for an inhomogeneous system, in `NON_UNITARITY_MAP.OUT` with columns (x, y, z, q_x, q_y, q_z) .
- `h.get_average_dvector()`: returns $(|d_x|^2, |d_y|^2, |d_z|^2)$ averaged over the Brillouin zone. With `non_unitarity=True` it returns the averaged squares of the components of $\vec{q}$, which say if the state is non-unitary but not in which direction.

See `examples/1d/dvector_map/main.py` (a self-consistent triplet state on a spin spiral, with its map of non-unitarity) and `examples/2d/SC_symmetry/main.py` (a mixed singlet and triplet pairing separated into its two parts) for runnable scripts. The notebook `jupyter-notebooks/functionalities/interacting_mean_field_hamiltonians/0` shows how to identify a self-consistent order parameter, and `jupyter-notebooks/functionalities/interacting_mean_field` the non-unitarity.

11.4 Superfluid weight and BKT temperature

A finite pairing amplitude alone does not make a superconductor. What carries a supercurrent is the stiffness of the state against a twist of the phase of the order parameter. We compute it with `h.get_superfluid_weight()`. This stiffness is the superfluid weight D_s ,

$$D_s^{ab} = \frac{1}{V} \frac{\partial^2 \Omega}{\partial Q_a \partial Q_b},$$

the second derivative of the grand potential Ω with respect to the wavevector $\mathbf{Q}$ of the phase twist, $\Delta \rightarrow \Delta e^{i\mathbf{Q} \cdot \mathbf{r}}$, at fixed $|\Delta|$, with V the volume (area in two dimensions). The superfluid weight is proportional to the inverse square of the London penetration depth, so it is what a measurement of the magnetic penetration depth or of the kinetic inductance gives. `pyqula` computes it with the general multiband BdG formula of Phys. Rev. B **95**, 024515.

The superfluid weight has two parts:

- the conventional part, from the band velocities, the only one in a single-band model.

- the quantum-geometric part, from the interband matrix elements. For an isolated flat band it is the integral of the quantum metric (Nat. Commun. **6**, 8944). A flat band has no velocity, so all its stiffness is geometric, and this is why a flat band can be superconducting at all.

This split is only defined for uniform onsite pairing and with time-reversal symmetry; otherwise asking for it raises a `ValueError`, and only the total is available. We compute both parts for a square lattice away from half filling:

```
from pyqula import geometry
g = geometry.square_lattice() # geometry of the 2D model
h = g.get_hamiltonian() # generate the Hamiltonian
h.add_onsite(-0.6) # move away from half filling
h.add_swave(0.3) # BdG Hamiltonian with an s-wave gap

D = h.get_superfluid_weight(nk=20) # Cartesian (dim,dim) tensor
out = h.get_superfluid_weight(nk=20,decompose=True)
print(out["total"],out["conventional"],out["geometric"])
print(h.get_bkt_temperature(nk=20)) # Nelson-Kosterlitz criterion
```

D is a 2×2 Cartesian tensor. On the square lattice it is isotropic, about 0.76 on the diagonal. With `decompose=True` the result is a dictionary, and its "geometric" entry is zero here: a single band has no other band to couple to, so all its stiffness is conventional. The geometric part appears only in a multiband model, mostly for a flat band.

The phase twist uses the full bond vector $\mathbf{R} + \mathbf{r}_j - \mathbf{r}_i$ between two orbitals, not only the lattice vector $\mathbf{R}$. This matters when the unit cell has more than one orbital. With the lattice vector alone, D_s is anisotropic on the honeycomb lattice, which the C_3 symmetry forbids, and it changes if we describe the same crystal with a supercell. With the full bond vector the result is isotropic and the same for any supercell. At fixed $|\Delta|$ the superfluid weight depends on where the orbitals sit (Phys. Rev. B **106**, 014518). `gauge="lattice"` uses the lattice vector only, which is the convention of Nat. Commun. **6**, 8944; `h.get_quantum_metric()` has the same keyword with the same default (see Quantum geometry).

In two dimensions the superfluid weight gives the Berezinskii-Kosterlitz-Thouless (BKT) temperature, through the Nelson-Kosterlitz criterion

$$T_{\text{BKT}} = \frac{\pi}{8} D_s(T_{\text{BKT}}),$$

which `h.get_bkt_temperature()` solves self-consistently in T . For the square lattice above it gives $T_{\text{BKT}} \approx 0.15$, in units of the hopping. $|\Delta|$ is kept fixed and does not decrease with temperature, so this is an upper bound.

For this figure we compute D_s^{xx} at 13 temperatures with the keyword `T`, instead of the single call at $T = 0$. The superfluid weight is 0.76 at low temperature and stays flat up to $T \approx 0.05$. Then it decreases, because thermal quasiparticles reduce the stiffness. The BKT temperature is where the curve crosses the line $8T/\pi$, at $T = 0.15$ with $D_s = 0.38$. Above it D_s does not go to zero, because $|\Delta|$ is kept fixed.

The code of this figure is in `documentation/figures/superconductivity_superfluid.py`.

Two keywords and one caveat:

- `mode="finite_difference"`: computes the second derivative of Ω numerically. It is much slower, but it does not use the assumptions of the analytic formula, so it is a good check.
- `T`: the temperature, in units of the hopping.

At $T = 0$, for a normal state without a gap and zero or very small pairing, the diamagnetic and paramagnetic parts should cancel. This cancellation comes from a term $-\partial f / \partial E$ that becomes a delta function at $T = 0$, and a finite k-mesh cannot resolve it. So D_s comes out equal to the Drude weight of the normal state instead of zero. To check that a state has no stiffness, use a temperature that the k-mesh resolves.

See `examples/2d/superfluid_weight/main.py` for a runnable script, and `jupyter-notebooks/functionalities/interacting` for an executed notebook.

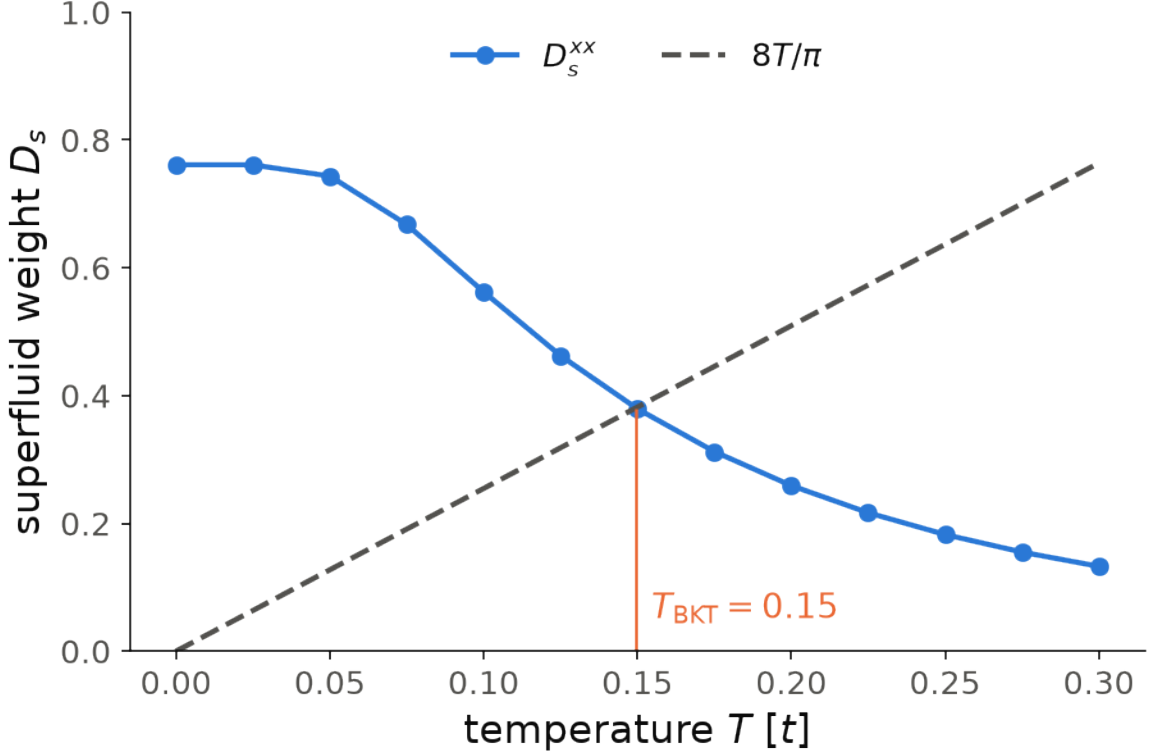


Figure 34: Superfluid weight of the square lattice as a function of temperature, with the Nelson-Kosterlitz line.

12 Mean-field Hubbard models

So far all our Hamiltonians were single-particle Hamiltonians: every term is bilinear, one creation and one annihilation operator. An interaction between electrons is a quartic term, with four fermion operators, and the problem is then a quantum many-body one. In this page we treat interactions at the mean-field level. The quartic term is replaced by bilinear terms that depend on expectation values in the ground state. These expectation values depend on the Hamiltonian, so the problem is solved self-consistently.

Mean field is an uncontrolled approximation. It does not capture quantum fluctuations, so it overestimates ordering, and in low dimensions it finds order where the exact ground state has none. Still, it is the practical way to describe magnetism, charge order and superconductivity driven by interactions. Everything in this page uses one method, `h.get_mean_field_hamiltonian()`, which returns a new Hamiltonian with the converged mean-field terms. The original `h` is not changed.

12.1 The collinear Hubbard model

We start with the simplest interaction, a local repulsion in a spinful system, the Hubbard model,

$$H = \sum_{\langle ij \rangle, s} c_{i,s}^\dagger c_{j,s} + h.c. + U \sum_i c_{i,\uparrow}^\dagger c_{i,\uparrow} c_{i,\downarrow}^\dagger c_{i,\downarrow}.$$

The first term is the first-neighbor hopping, and U is the energy cost of having a spin-up and a spin-down electron on the same site, in units of the hopping. The mean-field approximation replaces the quartic term by all the terms we get by taking the expectation value of two of the four operators. The simplest of them, the collinear (Hartree) terms, are

$$H_U^{MF} = U \sum_i \langle n_{i,\uparrow} \rangle n_{i,\downarrow} + n_{i,\uparrow} \langle n_{i,\downarrow} \rangle, \quad n_{i,s} = c_{i,s}^\dagger c_{i,s},$$

where $\langle \cdot \rangle$ is the expectation value in the ground state. So an electron with spin down feels an onsite energy proportional to the density of spin-up electrons on its site, and the other way around. If the two densities are different, the two spins see different onsite energies: this is an exchange field, and the state is magnetic.

The mean-field Hamiltonian depends on the ground state, and the ground state depends on the mean-field Hamiltonian. We solve this by iteration:

$$|GS_0\rangle \rightarrow H_0^{MF} \rightarrow |GS_1\rangle \rightarrow H_1^{MF} \rightarrow |GS_2\rangle \rightarrow \dots$$

We start from an initial guess $|GS_0\rangle$, compute H_0^{MF} , diagonalize it to get a new ground state $|GS_1\rangle$, and so on. The loop stops when two successive mean fields differ by less than `maxerror`. Two things matter in practice:

- The result can depend on the initial guess. If the energy has several local minima, different guesses can converge to different states. Try several guesses, and compare the total energies.
- The new mean field is mixed with the old one at each step. Close to a phase transition the loop can converge slowly or oscillate.

As an example, we take a square lattice, a 2×2 supercell, and a repulsive U at half filling. The ground state is an antiferromagnetic Neel state with a gap:

```
from pyqula import geometry
g = geometry.square_lattice() # geometry of a square lattice
g = g.get_supercell([2,2]) # generate a 2x2 supercell
h = g.get_hamiltonian() # create hamiltonian of the system
h = h.get_mean_field_hamiltonian(U=2.0,filling=0.5,
                                mf="random") # perform SCF
(k,e) = h.get_bands() # calculate band structure
m = h.get_magnetization() # get the magnetization
```

Without interactions the half-filled square lattice is a metal. With $U = 2$ the band structure has a gap at zero energy of about 0.8. The magnetization `m` has one vector per site. The vectors have the same length, about 0.38, and opposite sign on the two sublattices. Their direction is set by the random initial guess, because the Hamiltonian has spin rotation symmetry and nothing selects a direction. We need the supercell because a Neel state has two different sites, and a cell with one site can only hold a ferromagnet. A larger U gives larger moments and a larger gap. For large U the gap grows linearly with U , with one localized electron per site.

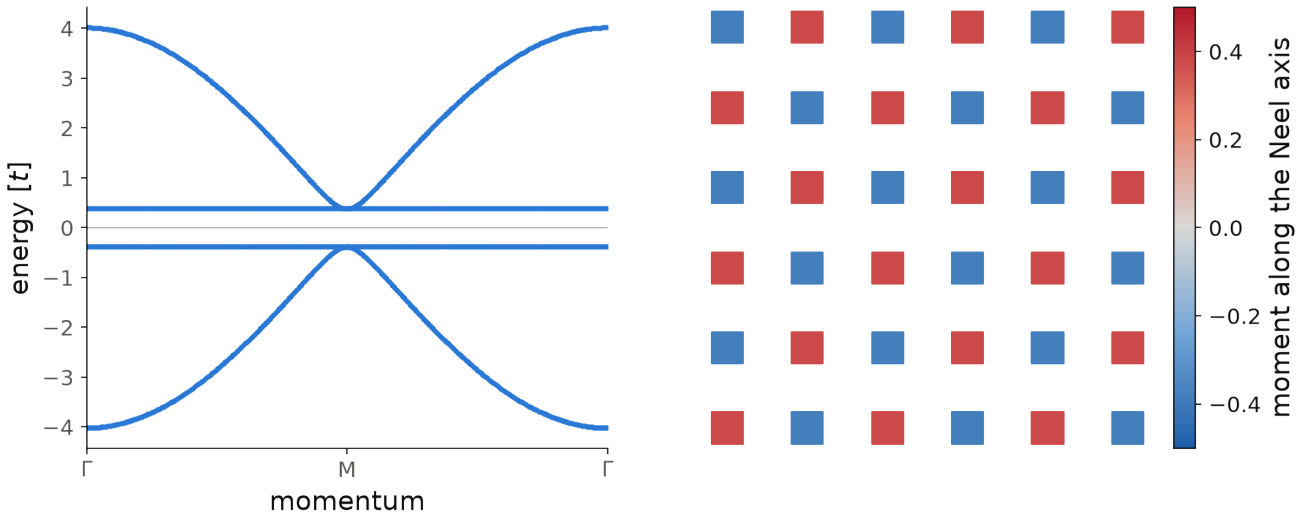


Figure 35: Band structure of the self-consistent Neel state of the square lattice, and its magnetization in real space.

The script fixes the random seed, so that the figure is reproducible. On the left, the band structure has a gap of 0.77 at zero energy. The exchange field on each site is $Um/2 = 0.38$, with $m = 0.38$ the moment, and the

gap is twice this field. The two flat bands at ± 0.38 are states of the k-path that lie on the Fermi surface of the non-interacting model. On the right, the 2×2 cell is repeated three times in each direction, and each site is colored by its moment along the moment of the first site. The moments are ± 0.38 , red on one sublattice and blue on the other. This checkerboard is the Neel state, which neutron diffraction sees as a magnetic Bragg peak at (π, π) .

The code of this figure is in `documentation/figures/meanfield_collinear.py`.

The main keywords are:

- **U**: the Hubbard interaction, a number or one value per site.
- **filling**: the filling, kept fixed at every iteration (see the filling section of Building a Hamiltonian).
- **mf**: the initial guess. A name such as "random", "ferro", "antiferro" or "swave", or a Hamiltonian whose mean-field terms are used as the starting point.
- **nk**: the number of k-points per direction used to compute the expectation values. A metal needs more k-points than an insulator.
- **maxerror, mix**: the convergence tolerance and the mixing.

If the loop does not converge, the method raises an error instead of returning a Hamiltonian. The page Exchange interactions discusses this case and the parameters that control it.

See `examples/2d/hubbard_honeycomb/main.py` for the gap as a function of U on the honeycomb lattice. There the density of states vanishes at the Dirac point, so a small U does nothing, and the gap opens only above a critical interaction, around $U \approx 2.2$ at the mean-field level. The notebook `jupyter-notebooks/functionalities/interacting_mean_field_hamiltonians/02_collinear_noncollinear.ipynb` shows the executed version.

12.2 Non-collinear Hubbard model

The collinear terms above only allow a magnetization along one axis. With frustration, an external field or spin-orbit coupling, the magnetization can point in any direction, and change direction from site to site. To describe this, the mean field also includes the spin-flip (Fock) term

$$H_U^{ncMF} = -U \sum_i \langle c_{i,\downarrow}^\dagger c_{i,\uparrow} \rangle c_{i,\uparrow}^\dagger c_{i,\downarrow} + h.c.,$$

set by the in-plane magnetization of each site. With this term the mean-field Hubbard Hamiltonian has the full SO(3) spin rotation symmetry of the Hubbard model. `pyqula` always includes it. This is also why in the previous example the direction of the Neel moments was free.

We now take the same square lattice and add a Zeeman field along z . The Zeeman energy competes with the antiferromagnetic order, and the result is a canted state:

```
from pyqula import geometry
g = geometry.square_lattice() # geometry of a square lattice
g = g.get_supercell([2,2]) # generate a 2x2 supercell
h = g.get_hamiltonian() # create hamiltonian of the system
h.add_zeeman([0.,0.,0.1]) # add out-of-plane Zeeman field
h = h.get_mean_field_hamiltonian(U=2.0,filling=0.5,
                                mf="random") # perform SCF
(k,e,c) = h.get_bands(operator="sz") # calculate band structure
m = h.get_magnetization() # get the magnetization
```

The magnetization now has two parts. The antiferromagnetic part alternates between the sublattices and lies in the plane perpendicular to the field. The uniform part is the same on every site, and it points opposite to the vector given to `h.add_zeeman()`, because the Zeeman term is $+\vec{B} \cdot \vec{\sigma}$ (see Building a Hamiltonian). The moments cant out of the plane instead of flipping, as an antiferromagnet does in a field that is weak compared to its exchange, and a larger field cants them more. The third array `c` returned by `h.get_bands(operator="sz")` is the spin projection $\langle S_z \rangle$ of each state, so the bands can be colored by it.

From a random guess this loop does not always converge. In 2 of 40 random seeds it stops after `maxite` iterations and raises an error. For both of them a larger mixing, `mix=0.5`, or more iterations, `maxite=5000`, gives a converged state. Calling `np.random.seed()` before the calculation makes a run reproducible.

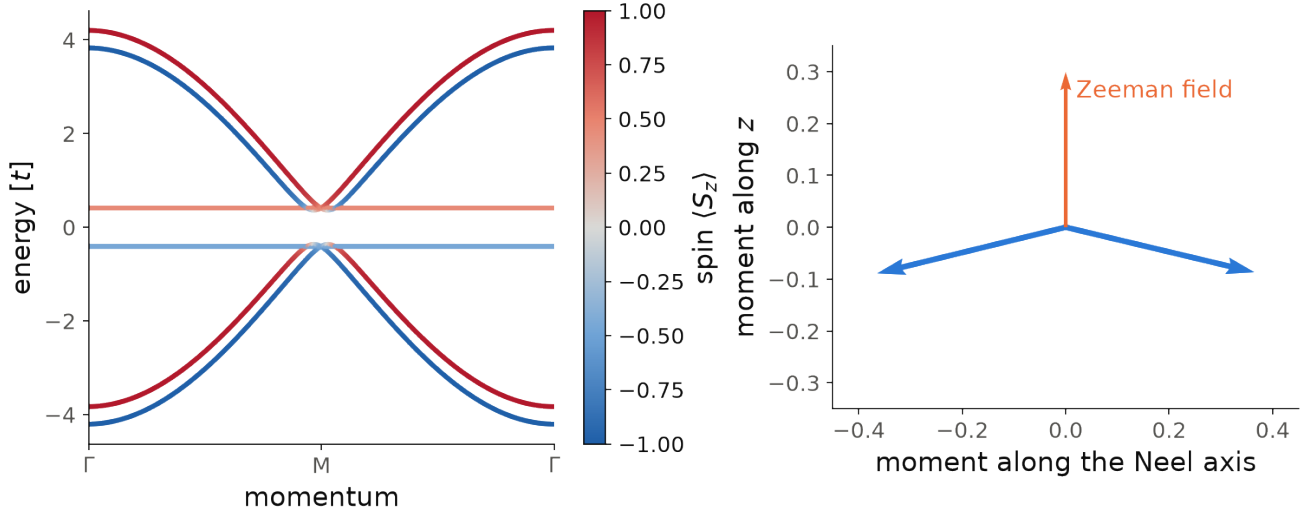


Figure 36: Bands of the canted antiferromagnet colored by the spin of each state, and the moments of the four sites.

The script fixes the random seed, so that the figure is reproducible. On the left, each band has a spin-up copy, higher in energy, and a spin-down copy, split by the Zeeman field. On the right are the moments of the four sites, in the plane of the Neel axis and of z . The antiferromagnetic part is 0.36, in the plane. The uniform part is -0.09 on every site: it points along $-z$, opposite to the vector given to `h.add_zeeman()`, because the term is $+\vec{B} \cdot \vec{\sigma}$ (see Building a Hamiltonian).

The code of this figure is in `documentation/figures/meanfield_noncollinear.py`.

See `examples/readme_examples/scf_square/main.py` for this calculation with a plot of the bands and of the moments, and the notebook `02_collinear_noncollinear.ipynb` in the folder above.

12.3 Superconducting mean field

A local interaction can also break a different symmetry, the $U(1)$ gauge symmetry, and give superconductivity. This comes from the anomalous contraction of the Hubbard term,

$$H_U^{aMF} = U \sum_i \langle c_{i,\uparrow} c_{i,\downarrow} \rangle c_{i,\downarrow}^\dagger c_{i,\uparrow}^\dagger + h.c.$$

The expectation value of two annihilation operators is the pairing amplitude. It plays the role of the Δ_0 of the s-wave term in Superconductivity, but now it comes from the interaction, with $|\Delta_0| = |U \langle c_{i,\uparrow} c_{i,\downarrow} \rangle|$, and is not added by hand. It can be different from zero for an attractive interaction, $U < 0$. The anomalous terms are included when the Hamiltonian has the Nambu basis, together with the collinear and non-collinear ones. So we first call `h.setup_nambu_spinor()`, then `h.get_mean_field_hamiltonian()`:

```
from pyqula import geometry
import numpy as np
g = geometry.triangular_lattice() # geometry of a triangular lattice
h = g.get_hamiltonian() # get the Hamiltonian
h.setup_nambu_spinor() # setup the Nambu form of the Hamiltonian
h = h.get_mean_field_hamiltonian(U=-2.0, filling=
                                0.45, mf="swave", nk=40) # perform SCF
# electron spectral-function
```

```
h.get_kdos_bands(operator="electron",nk=400,
                energies=np.linspace(-2.0,2.0,200),delta=0.03)
```

The guess `mf="swave"` starts from a uniform singlet pairing, and the loop finds its self-consistent value. The electron spectral function, with `operator="electron"` so that the hole copy of the bands is not shown, is the band of the triangular lattice with a gap at the Fermi energy. The gap grows with $|U|$. `h.get_kdos_bands()` is explained in Spectral functions.

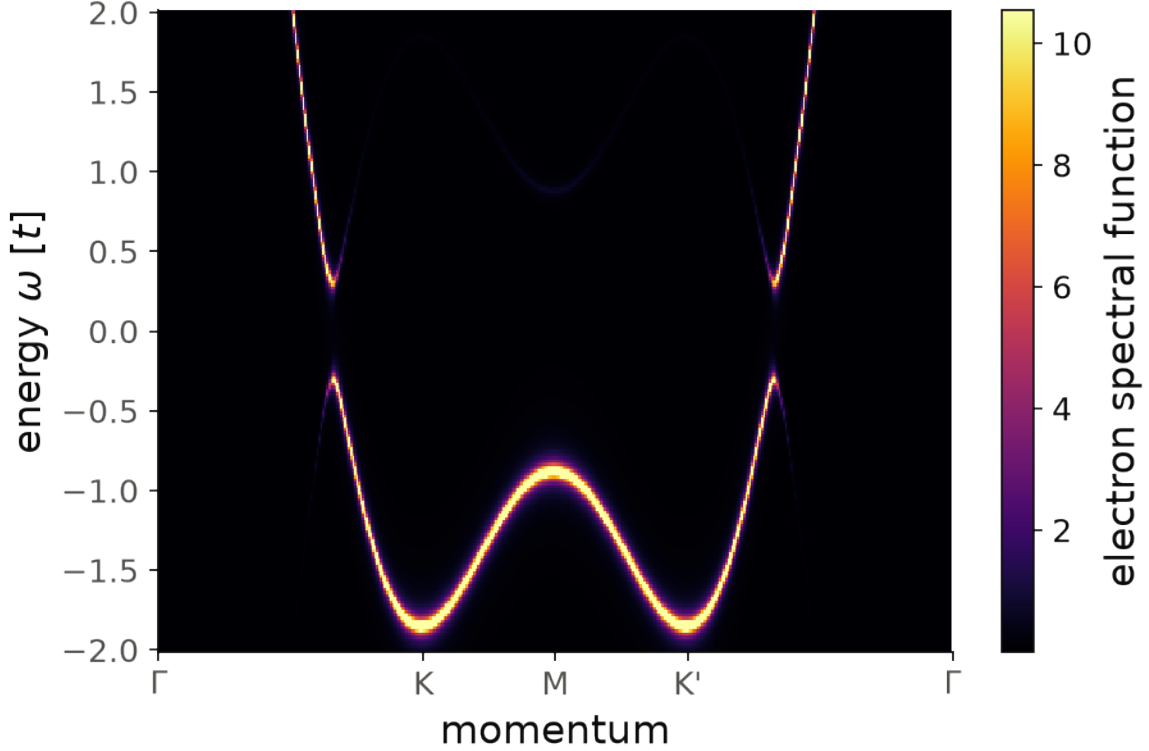


Figure 37: Electron spectral function of the self-consistent s-wave superconductor on the triangular lattice.

The self-consistent pairing is $\Delta_0 = 0.30$, so the gap at the Fermi energy goes from -0.30 to 0.30 . Away from the Fermi energy the spectral function follows the normal band, with its minimum at K at -1.85 . The faint band above, with its minimum at 0.9 at M, is the hole copy: its electron weight is small, but not zero. ARPES measures this spectral function below the Fermi energy.

The code of this figure is in `documentation/figures/meanfield_swave.py`.

See `examples/readme_examples/scf_SC/main.py` for a runnable script with a plot, and the notebook `04_anomalous_mf_sc.ipynb` in the folder above.

12.4 Long-range interactions

So far the interaction was only onsite. Interactions between neighbors are needed to get charge density waves, bond order (Peierls) instabilities and unconventional superconductivity. We now take

$$H = \sum_{\langle ij \rangle, s} c_{i,s}^\dagger c_{j,s} + h.c. + U \sum_i n_{i,\uparrow} n_{i,\downarrow} + V_1 \sum_{\langle ij \rangle} n_i n_j, \quad n_i = \sum_s n_{i,s},$$

where U is the onsite interaction and V_1 the interaction between first neighbors, with each bond counted once. The keywords of `h.get_mean_field_hamiltonian()` are:

- V_1, V_2, V_3 : the interaction between first, second and third neighbors.

- **Vr**: a function $Vr(\mathbf{r1}, \mathbf{r2})$ of the positions of two sites, the interaction of each pair, in the same convention as V_1 . A **Vr** equal to V_1 for first neighbors and zero beyond is the same calculation as **V1**. The pair of a site with itself is included, so $V_r(0)$ acts as an onsite $U = V_r(0)$; a function meant only for bonds must return zero at zero distance.
- **rcut**: the range of **Vr**, 5 by default in a periodic system. Whole shells of pairs at the same distance are kept or dropped, so the interaction keeps the symmetry of the lattice. In a finite system all pairs are kept.

pyqula does the full Hartree-Fock decoupling: it keeps all the contractions, and with a Nambu basis also all the anomalous ones. The first-neighbor interaction V_1 gives three new kinds of terms. The first is the charge (Hartree) term,

$$H^{MF} \sim \langle n_{i,s} \rangle n_{j,s'},$$

so the energy of a site depends on the charge of its neighbors. It gives charge density waves, with different charge on different sites. The second is the bond (Fock) term,

$$H^{MF} \sim \langle c_{i,s}^\dagger c_{j,s} \rangle c_{j,s}^\dagger c_{i,s} + h.c.,$$

which changes the hopping. If the change is not the same on all bonds, the state has a bond order, as in a Peierls distortion. The third kind are the anomalous terms,

$$H^{MF} \sim \langle c_{i,\uparrow}^\dagger c_{j,\uparrow}^\dagger \rangle c_{j,\downarrow} c_{i,\downarrow} + h.c., \quad \langle c_{i,\downarrow}^\dagger c_{j,\downarrow}^\dagger \rangle c_{j,\uparrow} c_{i,\uparrow} + h.c., \quad \langle c_{i,\uparrow}^\dagger c_{j,\downarrow}^\dagger \rangle c_{j,\downarrow} c_{i,\uparrow} + h.c.$$

The first two pair electrons with the same spin, so they are odd (spin triplet). The third pairs opposite spins, and can be even or odd. So an attractive V_1 can pair electrons on neighboring sites, with the same or with opposite spin, while the onsite U can only pair opposite spins on the same site.

In the example below an attractive V_1 together with a large exchange field gives a spin-triplet superconductor, with $\Delta_{\uparrow\uparrow}$ and $\Delta_{\downarrow\downarrow}$ different from zero:

```
import numpy as np
from pyqula import geometry
g = geometry.triangular_lattice() # generate the geometry
h = g.get_hamiltonian() # create Hamiltonian of the system
h.add_exchange([0.,0.,1.]) # add exchange field
h.setup_nambu_spinor() # initialize the Nambu basis
# perform a superconducting non-collinear mean-field calculation
h = h.get_mean_field_hamiltonian(V1=-1.5,
                                filling=0.3,mf="random",nk=40)
# electron spectral-function
h.get_kdos_bands(operator="electron",nk=400,
                 energies=np.linspace(-2.0,2.0,200),delta=0.03)
```

The exchange field splits the two spin bands so much that, at the Fermi energy, there are no pairs of opposite spin and opposite momentum to form a singlet. The attractive V_1 then pairs electrons of the same spin on neighboring sites. The converged state has an odd, spin-triplet order parameter, and the electron spectral function shows a gap at the Fermi energy in each of the two spin-split bands. The d-vector of this state, and whether it is unitary, are computed in Superconductivity.

For the figure the loop uses **nk=20** instead of **nk=40**, to keep it short, and the script fixes the random seed. The smallest gap is 0.42 with **nk=20** and about 0.40 with **nk=40**, so the k-mesh changes it by a few percent. The figure shows the two copies of the triangular band, split by the exchange field. Both have a gap at the Fermi energy, and the smallest one is on the band that crosses the Fermi energy between Γ and K. The moment is -0.43 along z , opposite to the exchange vector.

The code of this figure is in `documentation/figures/meanfield_triplet.py`.

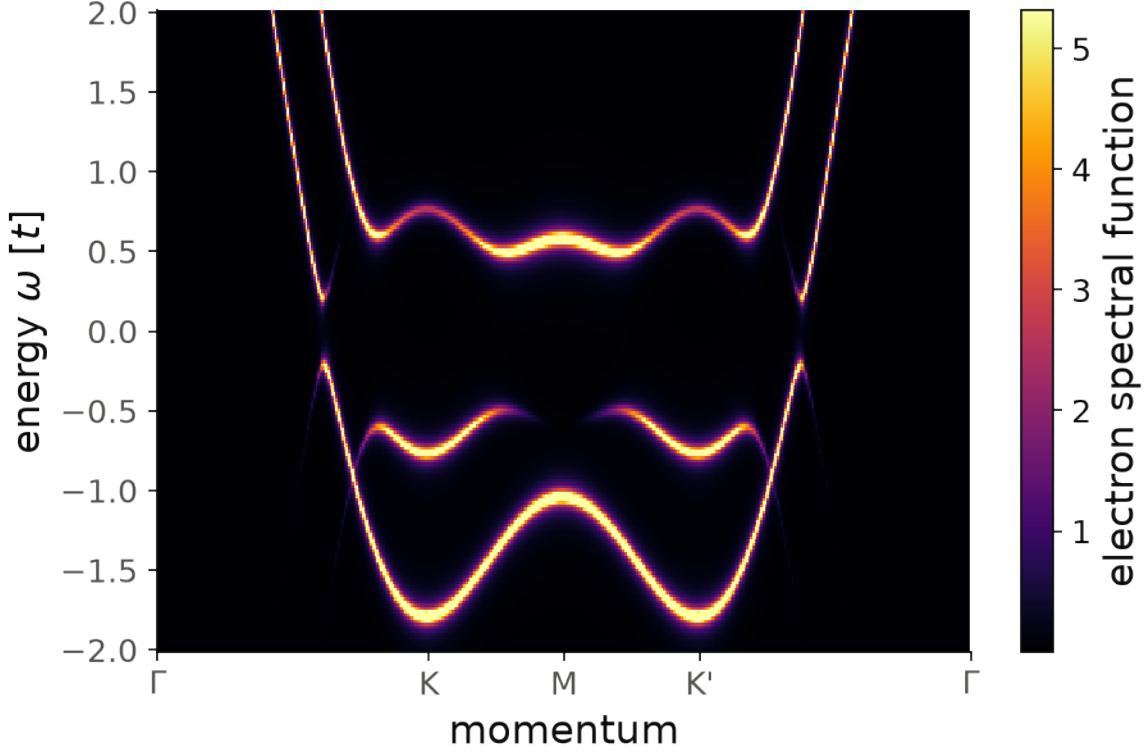


Figure 38: Electron spectral function of the self-consistent spin-triplet superconductor on the triangular lattice.

See `examples/readme_examples/scf_SC_triplet/main.py` for this calculation with a plot, `examples/2d/meanfield_V1V2V3/` for a charge density wave driven by V_1 , V_2 and V_3 in a spinless system, and the notebook `05_full_wick_noncollinear_sc.ipynb` in the folder above.

13 Exchange interactions

The interactions of Mean-field Hubbard models are of density-density type: they couple the charge on one site to the charge on another. In a magnet made of local moments the natural interaction is instead an exchange coupling between the spins of two sites. In this page we treat it at the mean-field level, alone or together with the density-density interactions. We also see what decides whether a self-consistent calculation converges, and how to treat large systems.

13.1 Spin-spin exchange

The exchange Hamiltonian is

$$H_J = J_z \sum_{\langle ij \rangle} S_i^z S_j^z + J_x \sum_{\langle ij \rangle} S_i^x S_j^x + J_y \sum_{\langle ij \rangle} S_i^y S_j^y, \quad \vec{S}_i = \frac{1}{2} \sum_{ss'} c_{i,s}^\dagger \vec{\sigma}_{ss'} c_{i,s'},$$

where $\vec{S}_i$ is the spin-1/2 operator of the electrons on site i , $\vec{\sigma}$ are the Pauli matrices, and each bond $\langle ij \rangle$ is counted once. $J > 0$ is antiferromagnetic and $J < 0$ is ferromagnetic. With $S_i^z = (n_{i,\uparrow} - n_{i,\downarrow})/2$ we get

$$S_i^z S_j^z = \frac{1}{4} (n_{i,\uparrow} n_{j,\uparrow} - n_{i,\uparrow} n_{j,\downarrow} - n_{i,\downarrow} n_{j,\uparrow} + n_{i,\downarrow} n_{j,\downarrow}),$$

so $S_i^z S_j^z$ is a density-density interaction between spin-orbitals, and it is decoupled in the same way as U and V_1 , with all the Hartree and Fock terms. The $S_i^x S_j^x$ and $S_i^y S_j^y$ terms are the same after a rotation of the spin axis.

The methods are

- `h.get_szsz_mean_field_hamiltonian()`: the $S^z S^z$ coupling only.
- `h.get_sxsx_mean_field_hamiltonian()`, `h.get_sysy_mean_field_hamiltonian()`: the same along x and y .
- `h.get_exchange_mean_field_hamiltonian()`: different couplings along the three axes, an anisotropic exchange.
- `h.get_combined_mean_field_hamiltonian()`: exchange and density-density interactions together.

The couplings are J_1 , J_2 , J_3 for first, second and third neighbors, and J_r for a general function of the two positions, in the same convention as V_1 , V_2 , V_3 and V_r . Like V_r , J_r is evaluated on every pair of sites up to `rcut`, including the pair of a site with itself. So a coupling meant only for bonds must return zero at zero distance; a value $J_r(0)$ is an onsite term $\frac{1}{2}J_r(0)\vec{S}_i \cdot \vec{S}_i$, which at the mean-field level acts as a Hubbard $U = -\frac{3}{4}J_r(0)$.

As an example, a ferromagnetic $S^z S^z$ coupling on a chain away from half filling:

```
from pyqula import geometry
g = geometry.chain() # a chain, prone to ferromagnetic order away from half filling
h = g.get_hamiltonian(has_spin=True)
h = h.get_szsz_mean_field_hamiltonian(J1=-2.0,filling=0.2,
                                     mf="ferroZ", # ferromagnetic Sz-Sz coupling
                                     nk=10,mix=0.3,maxite=300)
m = h.get_magnetization() # uniform moment along z
```

The loop converges, and every site gets the same moment along z , which `h.get_magnetization()` returns as one vector per site. But this moment is not a converged ferromagnet. It is -0.2 with `nk=10`, -0.1 with `nk=20`, -0.05 with `nk=40` and -0.02 with `nk=100`, so it scales as $2/n_k$: one electron on the k -mesh flips its spin, and the moment goes to zero for a fine mesh. So at $J_1 = -2$ the chain is not a ferromagnet at the mean-field level. A real ferromagnet needs a larger coupling and a stronger guess. With $J_1 = -6$, `nk=40` and a guess with an exchange field of 1 instead of the 0.1 of "ferroZ", the loop converges to a fully polarized state, with moment -0.4 , and its total energy is lower than the one of the unpolarized solution. On a finer mesh, `nk=60` or `nk=100`, this state needs a small temperature, $T=0.01$, for the reason given in the convergence section below. With the weak "ferroZ" guess the same $J_1 = -6$ still gives -0.05 , because the unpolarized state is also a self-consistent solution. In both cases the exchange field is found by the calculation, it is not added by hand with `h.add_exchange()`.

13.1.1 Convergence

All the mean-field methods of this page and of Mean-field Hubbard models raise an error, `MeanFieldConvergenceError`, when the loop does not converge. Its message gives the number of iterations, the error reached and `maxerror`. In a sweep over a parameter, catch it with `try` and `except` to skip the points that do not converge. It is imported with `from pyqula.meanfield import MeanFieldConvergenceError`. The functions of `pyqula.meanfield` that run the loop directly, such as `meanfield.VJinteraction()`, raise the same error. With `check_convergence=False` they return the unconverged result instead, with `.converged` set to `False`. Four keywords control the convergence:

- `nk` (default 8): the number of k -points per direction used to compute the expectation values.
- `mix` (default 0.1): the fraction of the new mean field mixed into the old one at each iteration.
- `maxite` (default 1000): the maximum number of iterations; `None` means no limit.
- `maxerror` (default 10^{-5}): the largest allowed change of the mean field between two iterations.

The k -mesh is usually the problem. If the mesh does not resolve the Fermi surface, the occupied states change from one iteration to the next, and no mixing helps. The chain above at `filling=0.2` never converges with `nk=8`, and converges at once with `nk=10`. The reason is that it converges when the mesh holds a whole number of electrons, $0.4n_k$, as for `nk=10` and `nk=15`, and not for `nk=8` or `nk=12`. So `nk` is the first thing to change. A smaller `mix` helps when the loop oscillates around a solution. A second problem appears when two states, at k and $-k$, sit exactly at the Fermi level, as for the polarized chain at $J_1 = -6$ with `nk=60` or `nk=100`. At the default temperature, $T = 10^{-7}$, a very small change of the mean field puts one of them below the Fermi level and the other one above. At the next iteration the two swap, so the loop oscillates forever and no `mix` helps. A small temperature, $T=0.01$ in units of the hopping, smooths the occupation of these two states, and the loop

converges to the same state as at `nk=40`, with moment -0.4 and the same energy per site within 10^{-3} . The same holds for `h.get_combined_mean_field_hamiltonian()`, for example with $U = 5$ and $J_1 = -1$ at `nk=40`. On a large system 1000 iterations take long, so a smaller `maxite` is useful while you explore parameters.

13.1.2 Anisotropic exchange

Different couplings along the three axes are given in one calculation with `h.get_exchange_mean_field_hamiltonian()`, here $J_z = -1$ and $J_x = -0.5$ on first neighbors:

```
h = g.get_hamiltonian(has_spin=True)
h = h.get_exchange_mean_field_hamiltonian(Jz1=-1.0, Jx1=-0.5,
                                          filling=0.2, mf="ferroZ",
                                          nk=10, mix=0.3, maxite=300)
```

As for $S^z S^z$ above, the moment of this chain is -0.2 with `nk=10` and -0.05 with `nk=40`, so it scales as $2/n_k$ and is not a converged ferromagnet.

13.1.3 Exchange and density-density interactions together

A model with both charge and magnetic correlations needs density-density interactions and exchange together. `h.get_combined_mean_field_hamiltonian()` solves them in one self-consistent calculation. Its exchange keywords are

- `J1, J2, J3, Jr`: isotropic Heisenberg couplings $J(S_i^x S_j^x + S_i^y S_j^y + S_i^z S_j^z)$ on the first, second and third neighbors, and as a function of the positions.
- `J1x, J1y, J1z`: an anisotropy added to `J1` for first neighbors only, so the first-neighbor coupling along z is $J_1 + J_{1z}$.

All of them are zero by default, and `U, V1, V2, V3, Vr` are as in Mean-field Hubbard models.

```
h = g.get_hamiltonian(has_spin=True)
h = h.get_combined_mean_field_hamiltonian(U=5.0, J1=-1.0,
                                          filling=0.2, mf="ferroZ")
```

This snippet runs with the default `nk=8` and gives a moment of -0.375 , and with `nk=40` it gives -0.05 , again $2/n_k$. So on this chain it shows how to call the method, not a converged ordered state. With `nk=100`, or with $U = 10$ or $J_1 = -3$ at `nk=40`, the loop does not converge.

13.1.4 Superconductivity from exchange

All the exchange methods also work on a Bogoliubov-de Gennes (BdG) Hamiltonian, after `h.turn_nambu()` or `h.setup_nambu_spinor()`. Then the exchange is decoupled in the normal and in the anomalous channels, as U and V_1 are. So exchange alone can give superconductivity. An antiferromagnetic isotropic J , without any U or V , can decouple into a singlet paired state: this is the resonating-valence-bond mechanism of exchange-driven superconductivity. A ferromagnetic J does not do this and stays magnetic. This paired state must be seeded: pass as `mf` a Hamiltonian with a small pairing, for example with `h.add_swave(0.1)`. A random guess has little overlap with it and usually goes back to zero pairing.

A state with both magnetic and superconducting order can come from an exchange field together with an attractive V_1 :

```
h = g.get_hamiltonian(has_spin=True)
h.add_exchange([0., 0., 0.3])
h.turn_nambu()
h = h.get_combined_mean_field_hamiltonian(V1=-1.0, J1z=-0.3,
                                          filling=0.3, mf="random")
```

With the default `nk=8` this result is not converged in the k -mesh: one run gave a gap of only 0.02 in the BdG bands, and another run did not converge. With `nk=20` and `nk=40` the gap is 0.14 and 0.15, and the moment is -0.14 and -0.15 , so here a k -mesh of at least `nk=20` gives a state with both magnetic and superconducting order.

With `return_total_energy=True` the method also returns the mean-field total energy, with the double counting of both the normal and the pairing mean field removed. Comparing this energy with the one of an unpaired solution of the same interaction tells you which of the two is the ground state at the mean-field level.

13.1.5 Nonlinear solvers

If a loop does not converge with plain mixing, it can be solved with a nonlinear solver instead, with `use_jax=True`. One iteration of the loop is a map $x \rightarrow f(x)$ of the mean field, and the solver looks for the root of $x - f(x) = 0$. `solver="error_gradient"` is the most robust on a generic Hamiltonian and the first one to try; `"newton"` is the default. The other options are `"newton_krylov"`, `"fsolve"`, `"lbfgs"`, `"broyden_mixing"` and `"linear_mixing"`. These solvers work only for the normal state, and do not accept `constrains`.

```
h = g.get_hamiltonian(has_spin=True)
h = h.get_combined_mean_field_hamiltonian(U=5.0,J1=-1.0,filling=0.2,
                                          mf="ferroZ",use_jax=True,
                                          solver="newton")
```

13.1.6 Expectation values from the kernel polynomial method

At each iteration the expectation values come from diagonalizing the Bloch Hamiltonian at every k-point, and for a large system this is the expensive step. With `integration="kpm"` the density matrix is computed instead with a Chebyshev expansion, the kernel polynomial method, without any diagonalization:

```
h = g.get_hamiltonian(has_spin=True)
h = h.get_combined_mean_field_hamiltonian(U=5.0,J1=-1.0,filling=0.2,
                                          mf="ferroZ",integration="kpm")
```

The expansion with `npol` polynomials resolves energies down to roughly the bandwidth divided by `npol`. A magnetic splitting or a superconducting gap smaller than this is smeared out. In practice the resolution should be several times smaller than the gap, and about ten times smaller close to a transition, where a gapped and a gapless solution have almost the same energy. When a model has several self-consistent states, this route and exact diagonalization can converge to different ones from the same guess. Keep the one with the lower total energy.

13.1.7 On a GPU

The same calculation can run on a GPU (see GPU). There `kpm_prec` sets the precision. Single precision, the default on the GPU, changes the density matrix by about 10^{-8} , much less than a finite `npol` does, but the change of the mean field between iterations then stops at about 10^{-6} . For a `maxerror` below this, use `kpm_prec="double"`.

```
from pyqula import gpu
gpu.set_gpu(True)
h = g.get_hamiltonian(has_spin=True)
h = h.get_combined_mean_field_hamiltonian(U=5.0,J1=-1.0,filling=0.2,
                                          mf="ferroZ",integration="kpm",
                                          npol=400,kpm_prec="single")
```

`gpu.set_gpu(False)` returns the rest of the script to the CPU.

13.2 Large islands

For a large finite island, say 10^5 sites, a dense Hamiltonian does not fit in memory. Build it sparse, with `is_sparse=True`. Then the interaction, the density matrix and the mean field stay sparse during the whole loop, and the memory grows linearly with the number of sites. This works with spin-orbit coupling, with exchange, and with the Nambu basis of a superconductor:

```
from pyqula import geometry
g = geometry.honeycomb_lattice().get_supercell(12) # a honeycomb island
g.dimensionality = 0 # make it finite
h = g.get_hamiltonian(has_spin=True,is_sparse=True) # keep the matrices sparse
```

```
h.add_rashba(0.2) # spin-orbit coupling, so the moments are not collinear
h = h.get_mean_field_hamiltonian(U=3.0,filling=0.5,mf="random",
                                integration="kpm",npol=150,mix=0.5,
                                maxerror=1e-3)
```

The result is the same self-consistent Hamiltonian as with a dense matrix, and it is still sparse, so `h.extract("mx")` and similar read the exchange field site by site without any diagonalization. Some points to keep in mind:

- `maxerror` is the change of the mean field per entry, for any size of the island. The direction of the moments, which the Rashba coupling selects only weakly, relaxes slowly. The snippet reaches a change of 10^{-3} in less than a hundred iterations, while 2×10^{-4} takes about 400.
- The total energy is the band energy of the KPM density matrix at zero temperature, $\text{Tr}(H\rho)$. It tends to the sum of the occupied levels as `npol` grows.
- On an island, `Vr` and `Jr` need an explicit `rcut`, otherwise every pair of sites interacts.
- On a small island, if the filling times the number of states is not an integer, KPM keeps the fractional number of electrons, while exact diagonalization rounds it to whole states. So the two converge to slightly different states.

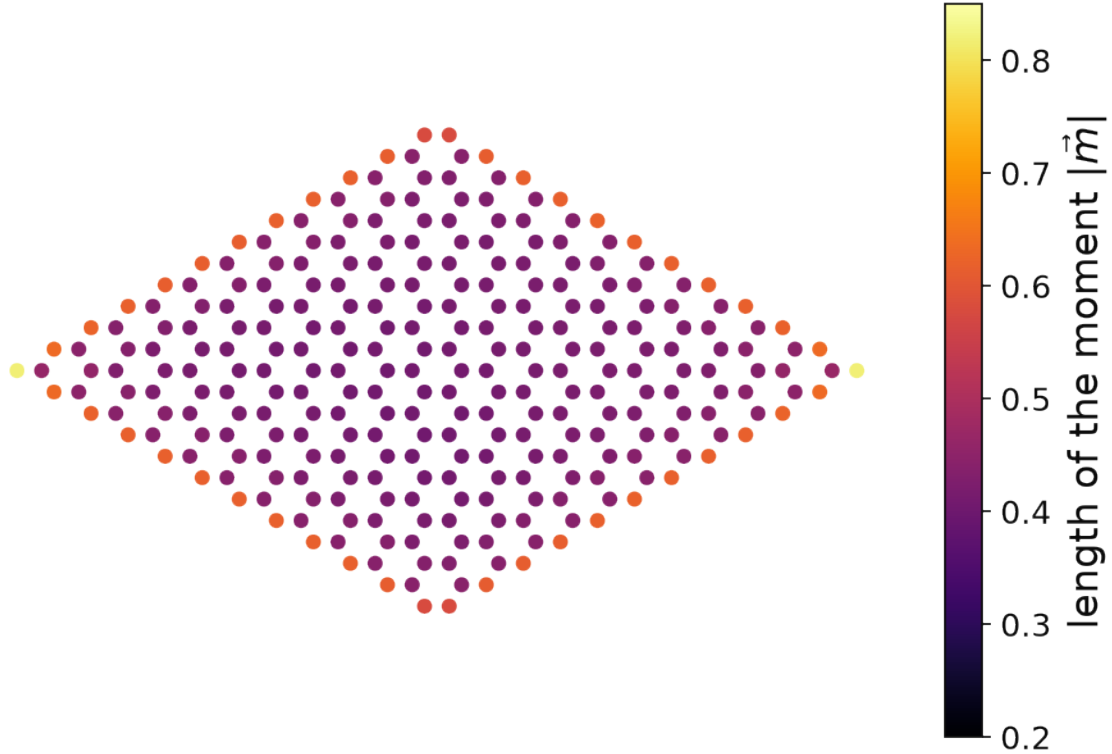


Figure 39: Length of the self-consistent magnetic moment on each site of a honeycomb island with Rashba coupling.

The map shows the length of the moment on each of the 288 sites of the island. In the bulk it is about 0.42. On the zigzag edges, where a site has two neighbors instead of three, it grows to about 0.62, and on the two corner sites, with one neighbor, it is 0.82. The moment of every site is antiparallel to the ones of its neighbors, so this is the antiferromagnet of the honeycomb lattice, and its direction changes across the island because of the Rashba coupling. Spin-polarized STM would see the larger moments at the edges.

The code of this figure is in `documentation/figures/exchange_island.py`.

See `examples/0d/kpm_scf_sparse/main.py` for a runnable script with an antiferromagnet and a superconductor.

13.2.1 Truncation to a radius

The time of one iteration grows as the square of the number of sites, because the occupation of each orbital comes from a Chebyshev recursion over the whole island. But after n steps of the recursion, a vector that starts on one orbital has only reached the sites within n hops. So we can compute the occupations around an orbital with the Hamiltonian restricted to the sites within R hops: this is exact up to the moment $2R$. Whether this is enough depends on the physics. In a gapped state the density matrix decays exponentially, over a length of the order of the velocity divided by the gap (or by the temperature at finite temperature), so the error falls exponentially with R . In a metal at zero temperature the density matrix decays as a power law, and the truncation is not controlled. `kpm_radius` sets R , in hops between sites:

```
h = g.get_hamiltonian(has_spin=True,is_sparse=True) # the same island
h.add_rashba(0.2)
h = h.get_mean_field_hamiltonian(U=3.0,filling=0.5,mf="random",
                                integration="kpm",npol=150,mix=0.5,
                                maxerror=1e-3,kpm_radius=8)
```

The cost of an iteration is now linear in the number of sites. Converge R as you converge a k-mesh, by comparing two values. For the antiferromagnet of a half-filled honeycomb island at $U = 3$, the exchange field differs from the one of the full recursion by 0.11, 0.023, 0.005 and 0.001 at 2, 4, 6 and 8 hops, so every two hops reduce the error by a factor of about five. On six desktop cores, one iteration of the Hubbard model with Rashba coupling on a square island of 10^5 sites takes about half a minute with `kpm_radius=10` and `npol=100`, and about two minutes with the Nambu basis of a superconductor.

See the notebooks `03_spin_spin_exchange.ipynb`, `11_kpm_scf.ipynb` and `12_jax_scf_solvers.ipynb` in `jupyter-notebooks/functionalities/interacting_mean_field_hamiltonians/` for executed versions.

14 Pseudofermion mean field

In this page we treat models of local spins with the same mean-field method as electrons. Each spin one half is written with an auxiliary fermion, the Abrikosov pseudofermion,

$$\vec{S}_i = \frac{1}{2} \sum_{ss'} f_{i,s}^\dagger \vec{\sigma}_{ss'} f_{i,s'}, \quad \sum_s f_{i,s}^\dagger f_{i,s} = 1,$$

with exactly one fermion on each site, because a spin one half has no charge fluctuations. You can think of the pseudofermion as the spin part of an electron, with the charge frozen. We use it first for the Heisenberg model, where it gives spinons, and then for the Kondo lattice, where it gives heavy fermions.

14.1 Spinon mean field for Heisenberg models

We take a spin- $\frac{1}{2}$ Heisenberg model,

$$H = J \sum_{\langle ij \rangle} \vec{S}_i \cdot \vec{S}_j,$$

where no electron hops. With the pseudofermions, the exchange term is quartic in f . Its mean-field decoupling gives a bond order parameter

$$\chi_{ij} = \langle f_i^\dagger f_j \rangle,$$

which acts as a hopping for the pseudofermions, so they get a dispersion. These dispersing fermions are the spinons, and this is the resonating-valence-bond ansatz. The decoupling is the same one that the J channel of `h.get_combined_mean_field_hamiltonian()` does (see Exchange interactions), with two differences: the bare hopping is zero, and the constraint of one fermion is imposed on every site, not only on average. Each site gets its own Lagrange multiplier λ_i , a local chemical potential, chosen so that $\langle n_i \rangle = 1$. The construction follows section 4 of arXiv:1601.03742. `SpinonHamiltonian` (in `pyqula.spinon`) builds it:

```

from pyqula import geometry
from pyqula.spinon import SpinonHamiltonian

g = geometry.triangular_lattice() # a canonical frustrated-Heisenberg lattice
h = SpinonHamiltonian(g) # zero bare hopping, couplings come from J1/J2/...
h2 = h.get_mean_field_hamiltonian(J1=1.0,nk=12,mix=0.1,maxerror=1e-4)

h2.local_occupation # <n_i> per site, exactly 1.0 at convergence
h2.constraint_lambda # converged per-site Lagrange multiplier (local chemical potential)
h2.get_bands()       # spinon dispersion

```

The result is a spinon Hamiltonian:

- `local_occupation`: the occupation of each site, equal to one at convergence.
- `constraint_lambda`: the Lagrange multiplier λ_i of each site.
- `h2.get_bands()`: the spinon dispersion. In a quantum spin liquid these spin- $\frac{1}{2}$ excitations replace the magnons of an ordered magnet, and they appear as a continuum in inelastic neutron scattering.

The keyword `filling` is not accepted here, because the representation is only valid with exactly one fermion per site. Only the U(1) ansatz, with the bond order χ_{ij} alone, is implemented. The Z2 ansatz, which also includes the pairing of spinons that J can induce, is not.

On a frustrated lattice (triangular, kagome) the converged state is not unique. Several self-consistent states, with different fluxes of χ_{ij} around a plaquette, can exist at the same J , and a random guess can land on any of them. This is part of the physics of the mean-field ansatz, not a problem of the loop. On a chain with one site per unit cell there is no frustration and the solution is unique, so repeated runs agree within `maxerror`. On a frustrated lattice, give an explicit `mf` to choose the ansatz you want.

A Zeeman field is already bilinear in f , so it enters exactly, without any decoupling. We add it with `h.add_zeeman()` or `h.add_exchange()` on the `SpinonHamiltonian`, before the mean-field calculation:

```

h = SpinonHamiltonian(g)
h.add_zeeman([0., 0., 0.3]) # or h.add_exchange([0., 0., 0.3])
h2 = h.get_mean_field_hamiltonian(J1=1.0, nk=12)
h2.get_magnetization()      # induced <sigma> per site, twice <S>

```

The vector $\vec{b}$ given to `h.add_zeeman()` multiplies the Pauli matrices, not $\vec{S} = \vec{\sigma}/2$. The term is $+\vec{b} \cdot \vec{\sigma} = 2\vec{b} \cdot \vec{S}_i$, so the induced $\langle \vec{S}_i \rangle$ points opposite to $\vec{b}$. This is the same convention as for electrons in Building a Hamiltonian. The constraint fixes the total occupation, not the spin, so it stays exact in a field, and $\langle \vec{S}_i \rangle$ grows with the field until it saturates when the field is large compared with J .

In a field the loop is fragile. With the snippet above, only 1 of 5 random seeds converges, and the converged state has a different bond order χ_{ij} from the state without field, so its spinon band has another shape.

The script fixes the random seed, so that the figure is reproducible. On the left, the spinon band without field goes from -0.47 to 0.57 in units of J . It is half filled, so the spinons have a Fermi surface. On the right, with $b_z = 0.3$, the two spin bands are split by 0.1 and $\langle \sigma_z \rangle = -1/6$, so $\langle S_z \rangle = -1/12$ per site, against the field vector. The splitting is much smaller than the bare $2b_z = 0.6$. The reason is the exchange with the six neighbors, which adds a field $6J\langle S_z \rangle = -0.5$ on S_z and cancels most of it. The band also changes its shape, because the bond order in the field is another one.

The code of this figure is in `documentation/figures/pseudofermions_spinon.py`.

See `examples/spinon/heisenberg_chain/main.py` and `examples/spinon/heisenberg_chain_zeeman_field/main.py` for runnable scripts, and the notebook `jupyter-notebooks/functionalities/interacting_mean_field_hamiltonians/13_s`

14.2 Kondo lattice

The Kondo lattice has a local moment on each site, coupled by exchange to a conduction electron on the same site. It is the minimal model of heavy-fermion compounds. We write it in the Coqblin-Schrieffer form of arXiv:cond-mat/0612006 (section II.C),

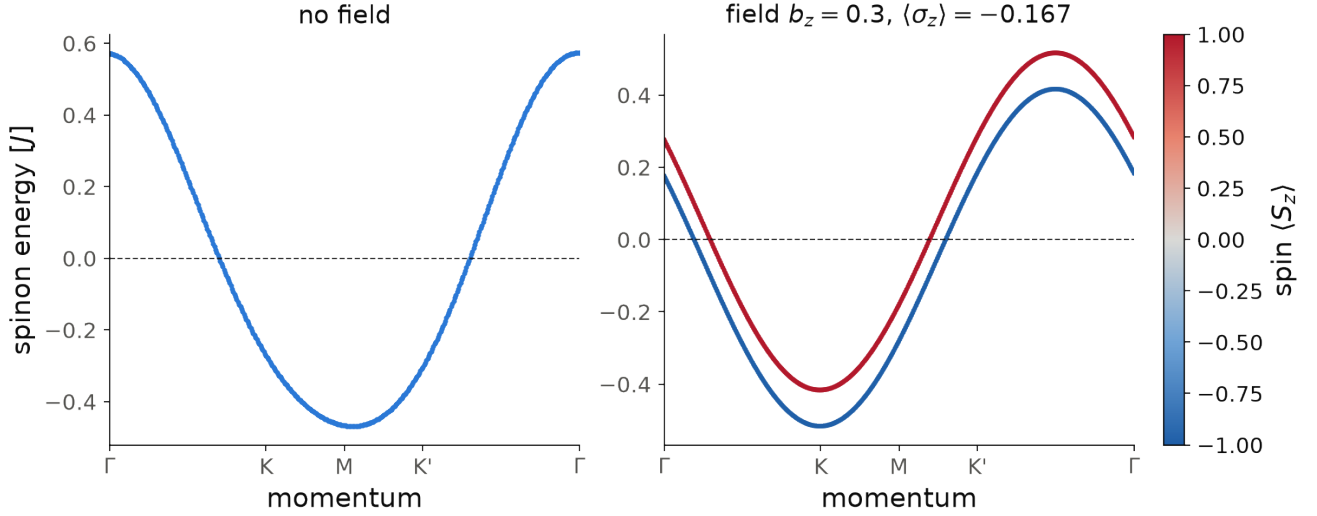


Figure 40: Spinon bands of the triangular lattice without and with a Zeeman field, colored by the spin of each state.

$$H = \sum_k \epsilon_k c_k^\dagger c_k + \frac{J}{N} \sum_j \sum_{ab} S_{ab}(j) c_{jb}^\dagger c_{ja}, \quad S_{ab}(j) = f_{ja}^\dagger f_{jb} - \frac{n_f}{N} \delta_{ab},$$

with $N = 2$ for a spin one half, a, b the spin indices, and ϵ_k the conduction band. Note that J here is the Coqblin-Schrieffer coupling, which enters as J/N ; it is not the coefficient of a Kondo term written as $J\vec{S}_j \cdot \vec{s}_j$, and the two differ by a numerical factor. The moments are written with pseudofermions as before, and the exchange is decoupled into a hybridization field

$$V_j = -\frac{J}{2} \sum_a \langle f_{ja}^\dagger c_{ja} \rangle,$$

together with a Lagrange multiplier λ_j that keeps one fermion per f-site. You can think of V_j as the amplitude for a conduction electron to turn into a spin flip of the local moment and back. This is the Read-Newns (large- N) mean field of the Kondo lattice.

`KondoLatticeHamiltonian` (in `pyqula.kondolattice`) takes a conduction-electron Hamiltonian and adds one localized f-site per conduction site, with zero bare hopping. Then it solves for V_j and λ_j self-consistently:

```
from pyqula import geometry
from pyqula.kondolattice import KondoLatticeHamiltonian

gc = geometry.chain()
hc = gc.get_hamiltonian(has_spin=True) # conduction electrons
h = KondoLatticeHamiltonian(hc)

seed = ([0.3+0.0j],[0.0]) # (V,lam), see the caveat below for why
h2 = h.get_mean_field_hamiltonian(J=1.5,filling=0.15,nk=200,mf=seed)

h2.local_occupation # <n_f> per localized site, exactly 1.0 at convergence
h2.hybridization    # converged V per localized site
h2.constraint_lambda # converged per-site Lagrange multiplier
```

The result is the heavy-fermion band structure. The conduction band hybridizes with a flat f-level at an energy set by λ_j , and V_j opens a hybridization gap there. The bands on both sides of the gap are flat, heavy, because most of their weight is on the f-sites. This is the heavy-fermion band seen in ARPES, and the hybridization gap

is seen in STM and optical spectroscopy. The three attributes are the occupation of each f-site, the hybridization V_j and the multiplier λ_j .

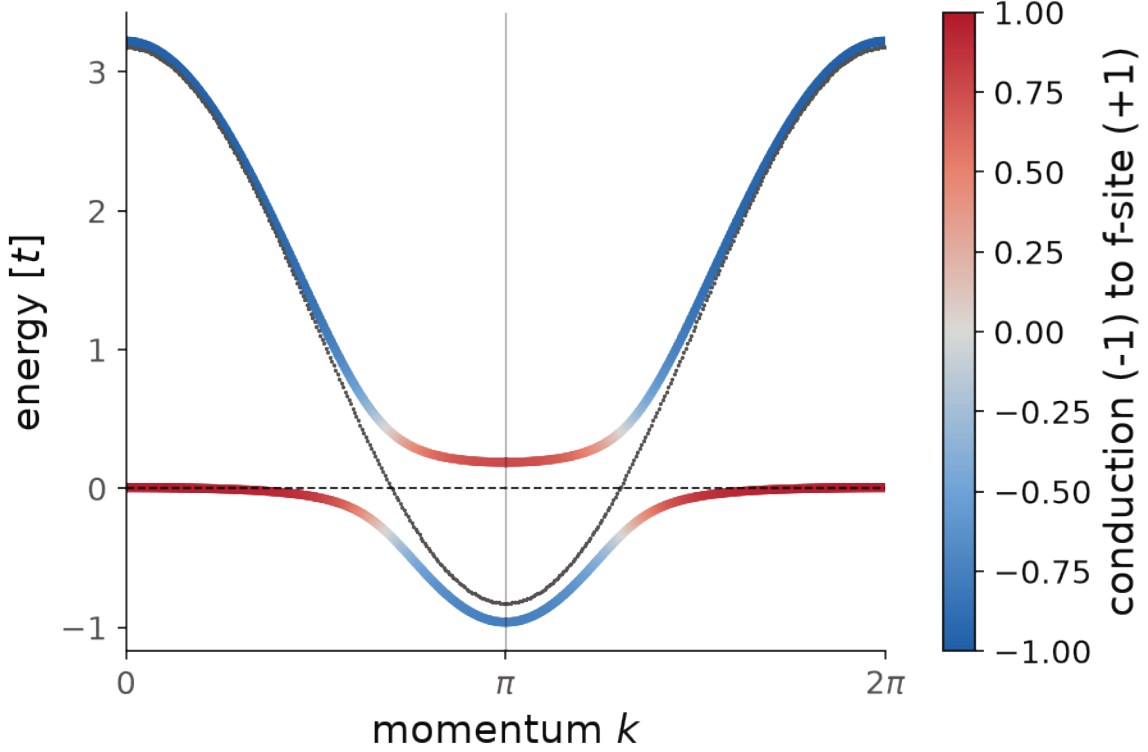


Figure 41: Heavy-fermion bands of the Kondo chain, colored by the weight of each state on the conduction and on the f-sites.

Energies are measured from the chemical potential (dashed line). Blue states live on the conduction sites and red ones on the f-sites. The gray line is the bare conduction band, $2t \cos k$, as a reference. The f-level sits 0.05 above the chemical potential. The converged hybridization $V_j = 0.37$ mixes it with the conduction band, and a gap opens between the top of the lower band, at 0.007, and the bottom of the upper band, at 0.19. Close to the gap the bands are flat and mostly red: these are the heavy quasiparticles.

The code of this figure is in `documentation/figures/pseudofermions_kondo.py`.

The keywords and caveats are:

- **J**: the Coqblin-Schrieffer coupling above.
- **filling**: sets the chemical potential once, from the bands with $V = 0$, and then keeps it fixed. The number of electrons can change while V and λ converge. The constraint $\langle n_f \rangle = 1$ is imposed by λ_j , not by **filling**.
- **mf**: the initial (V, λ) . $V = 0$ is always a self-consistent solution, like the trivial solution of the BCS gap equation. A run without a seed starts there and stays there, even when a hybridized solution exists. So a nonzero seed, as above, is needed. When both exist, the hybridized state has the lower energy.
- **T**: the temperature of the Fermi-Dirac distribution, 0.02 by default.

Do not choose a filling that puts the chemical potential inside the flat f-level of the problem with $V = 0$. All the f-orbitals are then at the same energy λ , and a wide range of fillings, roughly 0.25 to 0.75 for one conduction orbital per site, give the same ill-defined starting point. The **filling=0.15** above keeps the chemical potential in the conduction band.

At zero temperature the large- N theory gives a hybridization for any J , with a Kondo scale $T_K = D e^{-1/(J\rho)}$, where D is the bandwidth and ρ the density of states. At the finite temperature T of the loop this becomes a crossover: below a threshold in J , which depends on T , $V = 0$ is the only solution, and at the threshold V jumps to a value of order one instead of growing continuously from zero.

A Zeeman field acts exactly on both the conduction electrons and the local moments, since both terms are bilinear. As for spinons, we add it with `h.add_zeeman()` or `h.add_exchange()` on the `KondoLatticeHamiltonian`, before the mean-field calculation:

```
h = KondoLatticeHamiltonian(hc)
h.add_zeeman([0., 0., 0.05])
h2 = h.get_mean_field_hamiltonian(J=1.5, filling=0.15, nk=150, mf=seed)
```

The field acts on every site, conduction and f. To put it on only one of the two, give a function of the position instead of a vector (the f-sites are shifted in z). The constraint $\langle n_f \rangle = 1$ stays exact in a field. The field competes with the Kondo singlet: at fixed J , $|V|$ decreases as the field grows, and a large enough field destroys the hybridized state. There the loop does not converge and the call raises an error, instead of giving a small nonzero V . This is physics, not a numerical failure: it is the same decay to the $V = 0$ solution that a small J gives.

See `examples/kondolattice/chain/main.py` for a runnable script, and the notebook `jupyter-notebooks/functionalities/`

15 Topological invariants

A topological insulator has a gap in the bulk and states at the edges that no perturbation can remove, as long as the gap stays open. A topological invariant is an integer computed from the occupied bands over the whole Brillouin zone. An integer cannot change continuously, so the only way to change it is to close the gap. This is why the edge states it counts are protected. In this page we see the Chern number, the Z_2 invariant, the spin and mirror Chern numbers, the quadrupole moment, the three-dimensional Z_2 indices and the one-dimensional invariants.

15.1 Chern number

The Chern number is the invariant of a two-dimensional insulator with broken time-reversal symmetry. It is computed with `h.get_chern()`. For the occupied Bloch states $|u_n(\mathbf{k})\rangle$, the Berry connection is $\mathbf{A}(\mathbf{k}) = i \sum_n \langle u_n | \nabla_{\mathbf{k}} u_n \rangle$, the Berry curvature is $\Omega(\mathbf{k}) = \partial_{k_x} A_y - \partial_{k_y} A_x$, and the Chern number is

$$C = \frac{1}{2\pi} \int_{\text{BZ}} \Omega(\mathbf{k}) d^2\mathbf{k},$$

the total Berry flux of the occupied bands through the Brillouin zone, in units of 2π . The occupied bands are those with energy below zero, the Fermi energy. C is an integer. It is the Hall conductivity of the insulator in units of e^2/h , and it is the number of chiral states that cross the gap at each edge, so it is measured in a Hall experiment.

We take the Haldane model: a honeycomb lattice with a complex second-neighbor hopping, which breaks time-reversal symmetry and opens a topological gap at the Dirac points.

```
from pyqula import geometry
from pyqula import kdos
g = geometry.honeycomb_lattice() # create honeycomb lattice
h = g.get_hamiltonian() # create hamiltonian of the system
h.add_haldane(0.05) # Add Haldane coupling
C = h.get_chern() # Chern number
```

The result is $C = 2$. The Hamiltonian is spinful by default, and the Haldane term is the same for both spins, so each spin gives one. The same model with `has_spin=False` gives $C = 1$.

The map is the Berry curvature $\Omega(\mathbf{k})$ of this snippet, computed with `h.get_berry_curvature(nk=100)`. It covers a rectangle of momenta centered at Γ , from -0.577 to 0.577 in k_x and from -1 to 1 in k_y , and this rectangle holds exactly two Brillouin zones. The curvature is periodic, so the figure shows the map repeated once along k_x , with one zone drawn as the gray hexagon. The curvature is in the units of the reduced Brillouin zone, whose area is one. It is concentrated at the six corners of the zone, the K and K' points, where the gap

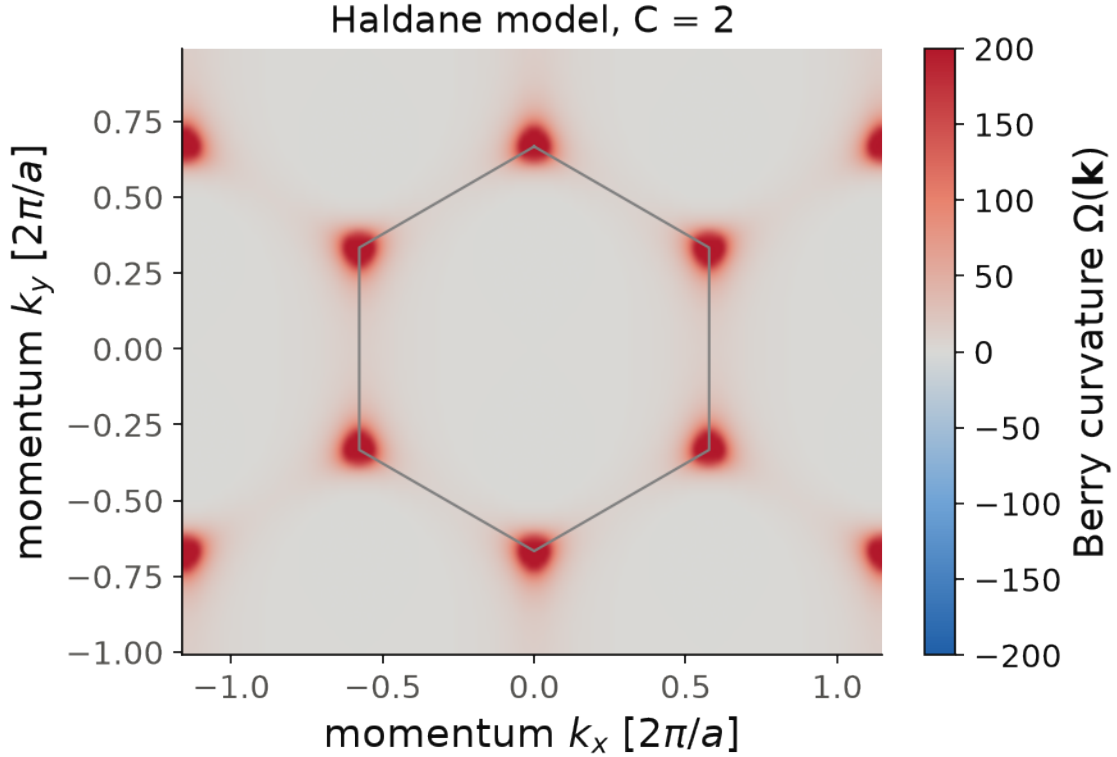


Figure 42: Berry curvature of the Haldane model over the Brillouin zone, with the zone drawn as a hexagon.

is smallest, from -0.225 to 0.225 . Its peak is about 620, so the color scale is cut at 200 to show the tails. The curvature has the same sign at K and at K' , so the two valleys add up instead of cancelling. These sharp peaks are what the k -mesh has to resolve.

Because the map holds a whole number of zones, the Chern number comes back from it as the mean of the curvature divided by 2π . Here `np.mean(b)/(2*np.pi)` is 2.000, the same as `h.get_chern()`. The window is chosen in the same way for every two-dimensional lattice. For a square lattice it is one zone, a square of side $|\mathbf{b}_1|$, and for a hexagonal lattice it is the rectangle of two zones of the figure, of sides $|\mathbf{b}_1|$ and $\sqrt{3}|\mathbf{b}_1|$. A lattice without such a rectangle, for example an oblique one, gets the parallelogram spanned by its two reciprocal lattice vectors, one zone, and then the points of the map are not on a rectangular grid. With `reciprocal=False` the map uses the reduced momenta from -1 to 1 , four zones, and the mean gives the Chern number too.

The code of this figure is in `documentation/figures/topology_chern.py`.

The integral is a sum over a k -mesh of `nk` points per direction, ten by default. The default sum counts the Berry phase around each small plaquette of the mesh, so it gives an integer at every `nk`. But the integer is only the right one if the mesh resolves the Berry curvature. Close to a gap closing, the curvature is sharply peaked near the gap minimum, and then `nk` must be larger.

Keywords:

- **nk**: number of k -points per direction of the mesh.
- **operator**: weight the curvature with an operator, for example a spin or a valley. This gives a spin or a valley Chern number (see Operators).
- **integration**: how the integral is done, "grid" (default), "qtci" or "wannier", explained below.

`h.get_berry_curvature()` returns the curvature $\Omega(\mathbf{k})$ itself, as a map over the Brillouin zone. `h.get_berry_curvature_path()` returns it along a k -path, by default the k -path of `h.get_bands()`, so that the peaks of the curvature can be put next to the band structure.

See `examples/2d/chern/main.py` (a Chern insulator from Rashba coupling and an exchange field),

examples/2d/haldane_model/main.py and examples/2d/valley_chern/main.py for runnable scripts, and jupyter-notebooks/functionalities/topological_characterization/01_chern_z2.ipynb and 02_operator_resolved_chern.ipynb for the curvature maps.

15.1.1 Adaptive integration with tensor cross interpolation

With `integration="qtci"`, the Brillouin-zone integral is done by quantics tensor cross interpolation. The integrand is learned on an adaptive grid, with more points where it changes fast, and then integrated with a quadrature rule. The number of evaluations grows roughly as the logarithm of the resolution, instead of as nk squared.

```
from pyqula import geometry
g = geometry.honeycomb_lattice()
h = g.get_hamiltonian()
h.add_haldane(0.05)
C = h.get_chern(integration="qtci", nk=20) # tensor-cross-interpolated BZ integral
```

Use it only for a smooth Berry curvature, so for a large gap. There it gives the integer to between 10^{-5} and 10^{-9} with far fewer evaluations than a dense mesh. For a small gap it is the wrong choice: the Haldane model above, with its small gap, gives $C = 2.015$ at $nk=20$, and the error grows with nk , because the quadrature nodes miss the peak of the curvature. For a small gap, use the default grid, which is exactly quantized.

The same method can replace the k -mesh sum of the density matrix in a mean-field calculation (see Mean-field Hubbard models):

```
from pyqula import geometry
g = geometry.honeycomb_lattice()
h = g.get_hamiltonian()
hscf, e = h.get_mean_field_hamiltonian(U=2.0, filling=0.5, mf="antiferro",
                                       nk=8, maxerror=1e-4, return_total_energy=True, integration="qtci")
```

For an insulator, "qtci" and the k -mesh sum agree. For a metal, the density matrix has a step at the Fermi surface, and a quadrature rule converges slowly on a step. For an attractive $U = -2$ on the square lattice at `filling=0.3`, the pairing gap is 16% too small at $nk=8$ and within 3% at $nk=16$. This mode works for two-dimensional Hamiltonians only, and in two dimensions it is not cheaper than the k -mesh sum. See examples/2d/chern_qtci/main.py and examples/2d/mean_field_qtci/main.py.

15.1.2 Winding of the hybrid Wannier centers

With `integration="wannier"`, the Chern number is obtained without the Berry curvature. We fix the momentum k_2 along the second reciprocal direction. The system is then a one-dimensional chain along the first direction. For this chain we compute the Wilson loop of the occupied bands, the product of the overlaps $\langle u_n(\mathbf{k}) | u_m(\mathbf{k} + \delta\mathbf{k}) \rangle$ around the closed loop in k_1 . The phases of its eigenvalues are the positions of the hybrid Wannier centers: the centers of charge of states that are localized along the first direction and extended along the second. When k_2 goes once around the Brillouin zone, the sum of the centers moves by $-C$ lattice constants, with C the Chern number of `h.get_chern()`. So for $C = 2$ the centers move by two lattice constants towards negative positions. The Chern number is the charge pumped across the unit cell in one cycle, and its sign is the direction of the pumping.

```
from pyqula import geometry
g = geometry.honeycomb_lattice() # create honeycomb lattice
h = g.get_hamiltonian() # create hamiltonian of the system
h.add_haldane(0.05) # Add Haldane coupling
C = h.get_chern(integration="wannier") # winding of the hybrid Wannier centers
```

The result is $C = 2$, as with the grid. The winding is always an integer. It is the right integer if the sum of the centers moves by less than half a lattice constant between two consecutive values of k_2 . There are `nt` values of k_2 (100 by default) and `nk` k -points along each Wilson loop (30 by default). The centers move fastest where the gap is smallest, so the gap decides how large `nt` must be: with a Haldane coupling of 0.02, `nt=20` gives 0 and `nt=50` gives 2.

The flow of the centers, the curve one plots to see the pumping, is returned by `topology.z2_wannier_centers(h,full=True)`. Its first row is k_2 , and there is one row per occupied band after it, each center given as the phase $2\pi x$ of its fractional coordinate x along the loop.

- `loop, pump`: the two reciprocal directions, of the Wilson loop and of the pumping.
- `gauge="atomic"`: put each orbital at its real position in the cell, instead of at the origin of the cell. The windings do not change, but the centers move to where the charge really is. Use it when the positions themselves matter, for example for a polarization.

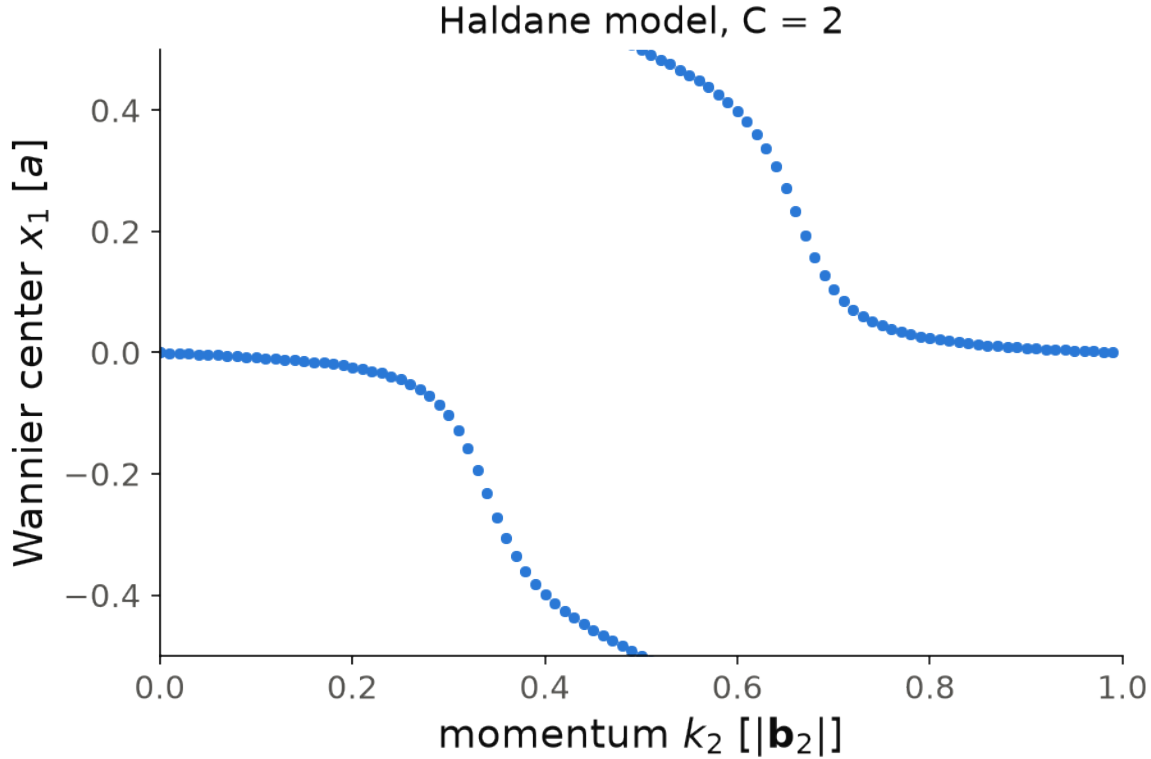


Figure 43: Flow of the hybrid Wannier center of the Haldane model as the momentum k_2 goes once around the Brillouin zone.

The figure shows this flow for the Haldane model of the snippet, with the center written as x_1 , in units of the lattice constant. The two spins have the same center, so the two rows lie on top of each other. At $k_2 = 0$ the center is at 0. It moves slowly, then fast near $k_2 = 1/3$, where the gap is smallest, and reaches the border of the cell, $x_1 = -1/2$, at $k_2 = 1/2$. It comes back in at $+1/2$ and goes back to 0 at $k_2 = 1$. So in one cycle each spin moves by one lattice constant towards negative x_1 . The two spins together move by -2 lattice constants, so $C = 2$. The plot repeats the center at $x_1 \pm 1$, so that the line is continuous across the border of the cell.

The code of this figure is in `documentation/figures/topology_wannier.py`.

See `examples/2d/wannier_centers/main.py` for a script that plots the flow.

15.2 Z2 invariant

With time-reversal symmetry the Chern number is zero. The invariant that remains is the Z_2 index. It has two values: trivial, or quantum spin Hall insulator, with a helical pair of states at each edge. We compute it for the Kane-Mele model, a honeycomb lattice with intrinsic spin-orbit coupling, with `topology.z2_invariant()`:

```
from pyqula import geometry
from pyqula import kdos
g = geometry.honeycomb_lattice() # create honeycomb lattice
h = g.get_hamiltonian() # create hamiltonian of the system
```

```

h.add_soc(0.05) # Add spin-orbit coupling
from pyqula import topology
z2 = topology.z2_invariant(h) # Z2 invariant

```

The result is a parity: -1 for a topological insulator and $+1$ for a trivial one. Here it is -1 . The honeycomb lattice gapped by a sublattice imbalance instead gives $+1$. The invariant is the parity of the number of helical pairs at each edge. An odd number of Kramers pairs crossing the gap cannot be removed by any perturbation that keeps time-reversal symmetry, but an even number can. Two helical pairs can hybridize and open a gap.

It is computed from the flow of the hybrid Wannier centers over half of the Brillouin zone, from $k_2 = 0$ to $k_2 = 1/2$, the two time-reversal invariant lines. On this half, the Kramers partners either switch or come back to themselves, and the parity of the number of times the centers cross a fixed line tells which. The keywords are `nk` momenta along the loop and `nt` pumping steps, 60 each by default. `h.add_soc()` is the same term as `h.add_kane_mele()`.

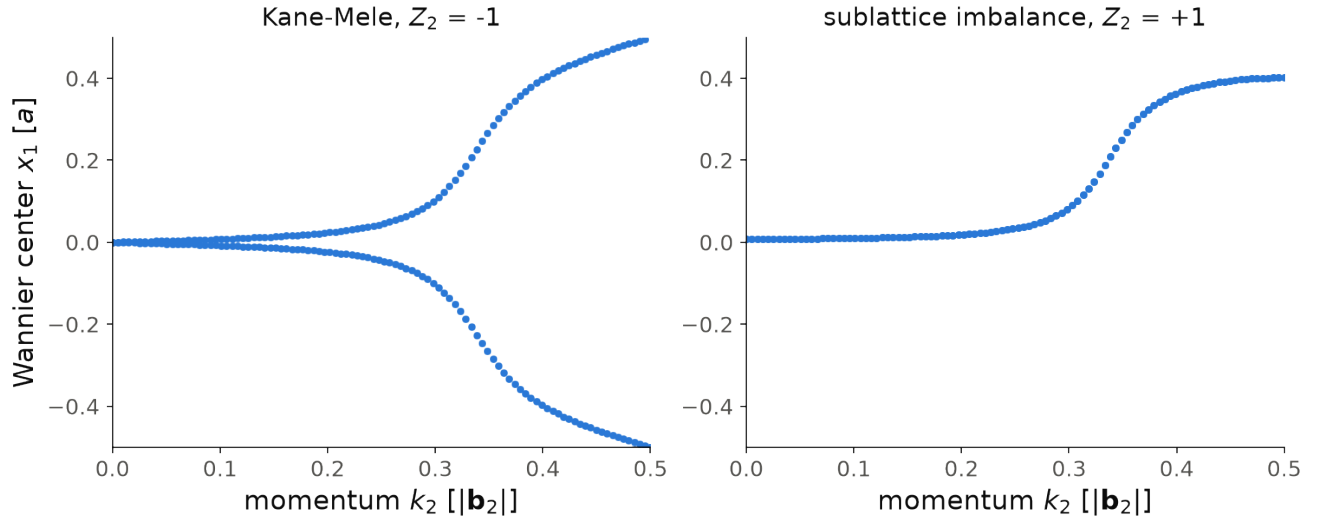


Figure 44: Flow of the hybrid Wannier centers over half of the Brillouin zone, for the Kane-Mele model and for a trivial insulator.

On the left is the flow of the Kane-Mele model of the snippet. On the right is the same lattice with a sublattice imbalance of 0.2 instead of the spin-orbit coupling, a trivial insulator. In the Kane-Mele model the two Kramers partners start together at $x_1 = 0$ for $k_2 = 0$. They separate, and at $k_2 = 1/2$ they arrive at $x_1 = +1/2$ and $x_1 = -1/2$, which is the same point of the periodic cell. So each partner ends with a new partner: they have switched. In the trivial insulator the pair starts at $x_1 = 0.008$ and ends at $x_1 = 0.40$, still together. A horizontal line at any x_1 is crossed once by the Kane-Mele flow, and an even number of times, zero or two, by the trivial one. This parity is the invariant.

The code of this figure is in `documentation/figures/topology_z2.py`.

`h.get_topological_invariant()` chooses the invariant for you. In two dimensions it returns the Z_2 invariant when the Hamiltonian is time-reversal symmetric, and the Chern number otherwise. In one dimension it returns the Berry phase, and in three dimensions the indices of the section on three dimensions below.

15.2.1 Time-reversal symmetric superconductors

The same index classifies a superconductor with time-reversal symmetry, a class DIII superconductor. Here it is the parity of the number of helical pairs of Majorana states at each edge, in-gap states that a tunneling probe at the edge would see. For a Bogoliubov-de Gennes (BdG) Hamiltonian, the invariant is computed from the negative-energy states. Time-reversal symmetry is checked up to a global phase of the pairing, since a uniform phase of the order parameter is only a gauge choice.

We take a helical p-wave superconductor on the square lattice. Spin up pairs as $p_x - ip_y$ and spin down as $p_x + ip_y$, so the two spins have opposite Chern numbers, and time reversal exchanges them.

```

import numpy as np
from pyqula import geometry
g = geometry.square_lattice() # square lattice, band between -4 and 4
h = g.get_hamiltonian() # spinful first-neighbor Hamiltonian
h.add_onsite(2.0) # chemical potential at -2, above the band bottom
def helical(r1,r2): # helical p-wave pairing, d-vector along the bond
    dr = r1-r2 # bond vector
    if abs(dr.dot(dr)-1.)>1e-4: return np.zeros((2,2)) # first neighbors only
    return 1j*np.array([[0.,dr[0]-1j*dr[1]],[dr[0]+1j*dr[1],0.]]) # i d.sigma
h.add_pairing(mode=helical,delta=0.3) # add the pairing, h is now a BdG Hamiltonian
z2 = h.get_topological_invariant() # Z2 invariant of the superconductor

```

The result is -1 when the chemical potential is inside the band. It is $+1$ when the chemical potential is below the band bottom, with `h.add_onsite(5.0)` instead of 2, because then there is no Fermi surface to pair. At half filling, `h.add_onsite(0.0)`, the Fermi surface goes through the X points, where this pairing is zero, so the gap closes and the invariant is not defined.

Two remarks. First, the Kane-Mele insulator above gives $+1$ once a small s-wave pairing is added, `h.add_swave(0.01)`, although its normal state gives -1 . The Nambu doubling contains its helical edge twice, and the pairing gaps the two copies, so a quantum spin Hall insulator with s-wave pairing is a trivial superconductor. Second, the Chern number of each spin cannot be obtained with `h.get_chern(operator="sz")` in the helical superconductor. For a Nambu Hamiltonian "sz" is the operator $\sigma_z\tau_0$, which this pairing does not conserve, and the weighted curvature integrates to zero. The operator that separates the two decoupled sectors is $\sigma_z\tau_z$, `Operator(np.diag([1.,-1.,-1.,1.]))`, and with it the weighted Chern number is $C_\uparrow - C_\downarrow = -2$.

See `examples/2d/z2_helical_superconductor/main.py` for the invariant across the band, and `examples/2d/z2_kane_mele/main.py` and `examples/2d/z2_transition/main.py` (a transition driven by a sublattice imbalance) for the insulator.

15.3 Spin Chern number and mirror Chern number

The Z_2 invariant needs time-reversal symmetry, and it is only a parity. Two other invariants count the helical pairs themselves, and they also exist when time reversal is broken: `h.get_spin_chern()` and `h.get_mirror_chern()`. Both are built in the same way. At every k-point, the occupied states are divided into two halves, by the sign of an operator projected on them. Each half has its own Chern number.

The simplest case is the spin. When s_z is conserved, the two halves are the two spin sectors, and the spin Chern number is

$$C_s = \frac{C_+ - C_-}{2},$$

half the difference of their Chern numbers. It counts the helical pairs at each edge, where the two spins move in opposite directions. When s_z is not conserved, for example with a Rashba coupling, the two halves are the eigenstates of Ps_zP , with P the projector on the occupied states. C_s is still an integer as long as the spectrum of Ps_zP has a gap around zero, and the code checks this. With time-reversal symmetry, the parity of C_s is the Z_2 invariant.

```

from pyqula import geometry
g = geometry.honeycomb_lattice() # create honeycomb lattice
h = g.get_hamiltonian() # create hamiltonian of the system
h.add_kane_mele(0.1) # Add spin-orbit coupling
h.add_rashba(0.1) # Add Rashba coupling, so that the spin is not conserved
Cs = h.get_spin_chern() # spin Chern number

```

The result is $C_s = 1$. Note that `topology.spin_chern` is a different quantity: the integral of the Berry curvature weighted by s_z . It is $C_\uparrow - C_\downarrow = 2C_s$ when s_z is conserved, and it is not an integer here (2.097 with `nk=24`).

The mirror Chern number divides the occupied states by the eigenvalues $\pm i$ of the mirror $z \rightarrow -z$, which maps every momentum of a two-dimensional system to itself:

$$C_M = \frac{C_{+i} - C_{-i}}{2}.$$

It is the invariant of a topological crystalline insulator, where the mirror, not time reversal, protects the edge states. The mirror is found from the geometry and the Hamiltonian, and the call raises an error when there is no mirror, for example with the Rashba coupling above, which breaks it.

```
from pyqula import geometry
g = geometry.honeycomb_lattice() # create honeycomb lattice
h = g.get_hamiltonian() # create hamiltonian of the system
h.add_kane_mele(0.1) # Add spin-orbit coupling
h.add_exchange([0.,0.,0.3]) # out-of-plane exchange: breaks time reversal, keeps the mirror
CM = h.get_mirror_chern() # mirror Chern number
```

The result is $C_M = -1$. For a single layer the mirror is $-i\sigma_z$, so $C_M = -C_s$. It stays quantized with an out-of-plane exchange field, which keeps the mirror but breaks time reversal, so here the Z_2 invariant is not even defined. For two coupled layers, the AA bilayer `specialhamiltonian.multilayer_graphene(l=[0,0])` with the same spin-orbit coupling, we find $C_s = 2$ with a trivial Z_2 invariant (two copies of the quantum spin Hall state), and $C_M = 0$, because the two mirror sectors have opposite Chern numbers.

- `nk`: number of k-points per direction of the mesh, 40 by default, for both calls.

See `examples/2d/spin_mirror_chern/main.py` for the two invariants as the Rashba coupling and the exchange field grow, and `jupyter-notebooks/functionalities/topological_characterization/08_spin_mirror_chern.ipynb`.

15.4 Quadrupole moment and the nested Wilson loop

A quadrupole insulator is a two-dimensional insulator with zero polarization and zero Chern number, but with a zero mode at each corner of a flake. STM would see these as peaks in the local density of states at the corners only. Its invariant is the quadrupole moment, computed with `h.get_quadrupole_moment()`.

The construction uses the hybrid Wannier centers of the previous sections. The Wilson loop along $\vec{b}_1$ gives, at each momentum k_2 , the centers $\nu_1(k_2)$, the positions along $\vec{a}_1$ of the occupied states. Suppose these Wannier bands form two groups separated by a gap, one with $0 < \nu_1 < 1/2$ and one with $-1/2 < \nu_1 < 0$. Each group is a sector of electrons on one side of the unit cell along $\vec{a}_1$. Now we ask where the electrons of one sector are along $\vec{a}_2$. The answer is a second Wilson loop, along $\vec{b}_2$, over the states of this sector only. This is called the nested Wilson loop. The phase of its determinant, averaged over k_1 , is the polarization of the sector along $\vec{a}_2$, written $p_2^{\nu_1^\pm}$. With a mirror symmetry along each direction, each of these polarizations is 0 or 1/2, and the quadrupole moment is

$$q_{xy} = p_2^{\nu_1^-} p_1^{\nu_2^-} + p_2^{\nu_1^+} p_1^{\nu_2^+} \mod 1.$$

So $q_{xy} = 1/2$ when the Wannier sectors are displaced by half a lattice constant along both directions, and 0 otherwise. We take the Benalcazar-Bernevig-Hughes model: a square lattice with flux π through each plaquette, four sites per unit cell, hopping $1 - \delta$ inside the cell and $1 + \delta$ between cells.

```
from pyqula import specialhamiltonian
h = specialhamiltonian.square_20TI(delta=0.3) # quadrupole insulator, four sites per cell with pi flux
q = h.get_quadrupole_moment() # quadrupole moment from the nested Wilson loops
p = h.get_wannier_sector_polarization(loop=0,sector="+") # polarization along a2 of one Wannier sector
```

The result is $q_{xy} = 1/2$, and each of the four sector polarizations is 1/2. With the stronger hopping inside the cell, `delta=-0.3`, all of them are 0, a trivial insulator. This is the dimerization of the Su-Schrieffer-Heeger chain, along both directions at the same time. An open flake of the first model has a zero mode at each corner, and the second has none. When the two directions are dimerized differently, only one pair of sector polarizations is 1/2, and $q_{xy} = 0$.

This invariant can change without closing the bulk gap: it can change by closing the gap between the Wannier bands instead. For `delta=-1.0` the plaquettes are decoupled, every Wannier center is at 0, and the two sectors

cannot be separated. The call raises an error when a Wannier band comes closer than `tol` to 0 or to 1/2, and `topology.wannier_gap(h)` gives this distance. The quantization of q_{xy} needs the two mirrors; without them the formula gives a number with no meaning.

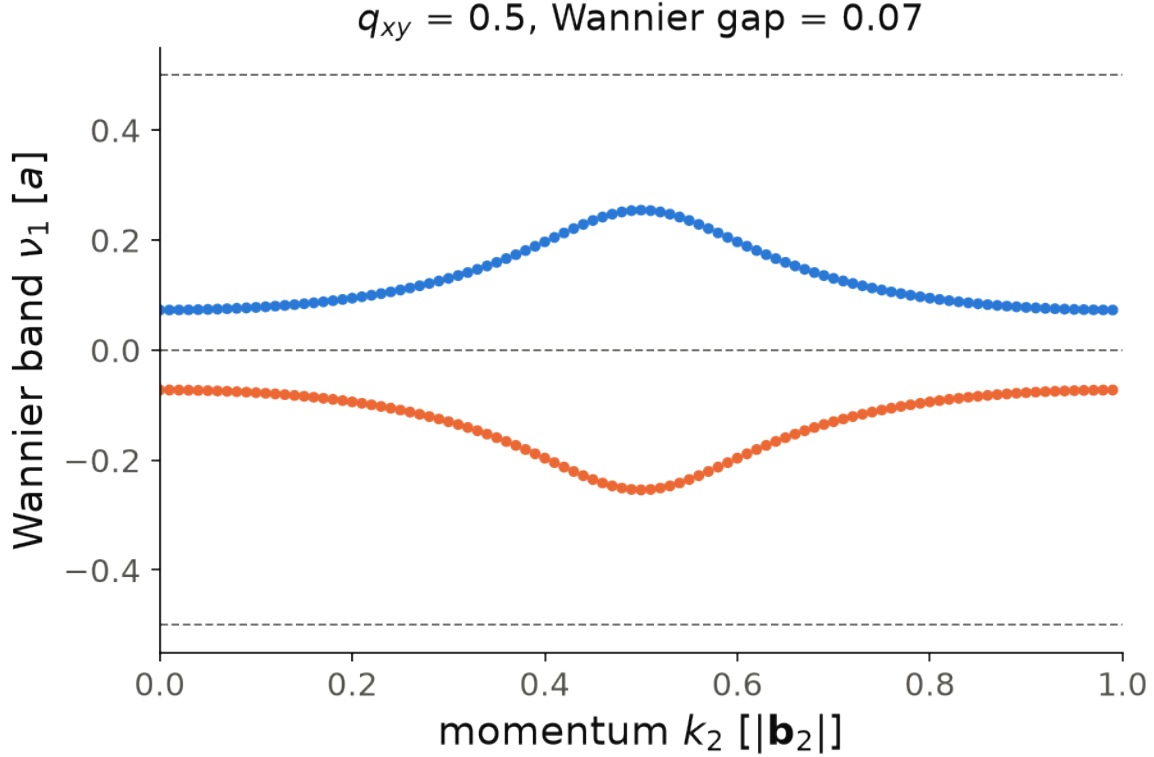


Figure 45: Wannier bands of the quadrupole insulator as a function of the momentum k_2 .

The figure shows the Wannier bands $\nu_1(k_2)$ of the snippet, the hybrid Wannier centers along $\vec{a}_1$ from `topology.z2_wannier_centers(h,full=True)`. There are two, one per occupied band. They are mirror images of each other, and they form the two sectors: blue for $\nu_1 > 0$ and orange for $\nu_1 < 0$. They come closest to 0 at $k_2 = 0$, at ± 0.073 , which is the value of `topology.wannier_gap(h)`, and they reach ± 0.25 at $k_2 = 1/2$. They never touch 0 or 1/2, so the two sectors are well defined. The value of q_{xy} is not visible in this picture: it comes from the nested Wilson loop over each sector, which gives the four polarizations 1/2.

The code of this figure is in `documentation/figures/topology_quadrupole.py`.

- `nk`: k-points per direction, 40 by default.
- `gauge="atomic"`: put each orbital at its position in the cell. By default every orbital is at the origin of its cell, which is the convention that defines the quadrupole moment. With "atomic" the sector polarizations are positions in the unit cell.

See `examples/2d/quadrupole/main.py` for the invariant across the transition, `examples/0d/cornermodes/main.py` for the corner modes of the flake, and `jupyter-notebooks/functionalities/topological_characterization/09_quadrupole`

15.5 Strong and weak Z_2 indices in three dimensions

In three dimensions, `h.get_topological_invariant()` returns four Z_2 indices. Take a plane of the Brillouin zone at a fixed momentum k_i along one reciprocal lattice vector. At $k_i = 0$ and at $k_i = 1/2$, time reversal maps the plane to itself, so each of these six planes is a two-dimensional time-reversal symmetric insulator, with its own Z_2 invariant. Four combinations of them are independent, written $\nu_0; (\nu_1\nu_2\nu_3)$:

- the weak index ν_i is the invariant of the plane $k_i = 1/2$;
- the strong index ν_0 is the product of the invariants of the planes $k_i = 0$ and $k_i = 1/2$, which is the same for the three directions.

You can think of a weak topological insulator, $\nu_0 = 0$ with some $\nu_i = 1$, as quantum spin Hall layers stacked along $\nu_1 \vec{b}_1 + \nu_2 \vec{b}_2 + \nu_3 \vec{b}_3$. It has Dirac cones only on the surfaces that cut the layers. A strong topological insulator, $\nu_0 = 1$, has an odd number of Dirac cones on every surface, which is what ARPES sees.

We take the Fu-Kane-Mele model: the diamond lattice with spin-orbit coupling between second neighbors. The gap opens when one of the four first-neighbor bonds is different from the other three.

```
from pyqula import geometry
g = geometry.diamond_lattice_minimal() # diamond lattice, two sites per cell
h = g.get_hamiltonian() # first-neighbor hopping
h.add_kane_mele(0.05) # spin-orbit coupling between second neighbors
h.intra = 1.3*h.intra # make the bond inside the unit cell stronger
z2 = h.get_topological_invariant() # strong and weak Z2 indices
```

The result is (1, (1, 1, 1)), a strong topological insulator. With this bond weaker than the other three, $h.intra = 0.7*h.intra$, it is (0, (1, 1, 1)), a weak topological insulator of layers stacked along the bond. When the bond dominates, $h.intra = 3.1*h.intra$, it is (0, (0, 0, 0)), a band insulator of decoupled dimers.

Note that here the indices are 0 or 1, not the ± 1 of the two-dimensional invariant, because the weak ones are the components of a vector. They are given in the reciprocal lattice vectors of the geometry, so the weak vector depends on the choice of unit cell, but ν_0 does not. Each plane needs one two-dimensional invariant, with **nk** and **nt** as there, 60 by default. The call raises an error if the strong index is different for the three directions, which means that the flow of some plane was not resolved, or that the gap closes.

When time reversal is broken, the same call returns the Chern vector (C_1, C_2, C_3): the Chern numbers of the planes at fixed k_1, k_2 and k_3 . For Chern insulator layers stacked along $\vec{a}_3$ it is (0, 0, C), with C the Chern number of one layer, the Hall conductance of each layer in units of e^2/h . Both are also available as `topology.z2_invariant_3d(h)` and `topology.chern_vector(h)`. See `examples/3d/z2_diamond/main.py` for the three phases of the diamond model, and `jupyter-notebooks/functionalities/topological_characterization/10_z2_t`.

15.6 Winding number of a chiral chain

In one dimension `h.get_topological_invariant()` returns the Berry (Zak) phase. It is 0 or π , so it only tells whether the number of zero modes at each end is even or odd. To count them we use `h.get_winding_number()`.

The simplest case is the Su-Schrieffer-Heeger chain: two sites per cell, an intracell hopping t_1 and an intercell hopping t_2 . The Hamiltonian only connects the two sublattices, so it anticommutes with the sublattice operator S . This is called a chiral symmetry. In the eigenbasis of S , the Bloch Hamiltonian is off-diagonal, with a block $q(k)$ between the two sublattices. The winding number is

$$W = \frac{1}{2\pi i} \oint dk \partial_k \log \det q(k),$$

the number of times the phase of $\det q(k)$ turns around when k crosses the Brillouin zone. It is the number of zero modes at each end of an open chain, all on one sublattice. STM sees them as a zero-bias peak at the end.

```
from pyqula import geometry
g = geometry.bichain() # two-site chain with a sublattice
def fun(r1,r2): # intracell hopping 0.5, intercell hopping 1
    dr = r1-r2
    if abs(dr.dot(dr)-1.)>1e-4: return 0. # first neighbors only
    return 0.5 if abs((r1[0]+r2[0])/2.)<1e-4 else 1.0 # the intracell bond is centered at x=0
h = g.get_hamiltonian(fun=fun,has_spin=False) # SSH chain
W = h.get_winding_number() # winding number, with the sublattice operator
```

The result is $W = -1$, one zero mode at each end. It becomes 0 when the intracell hopping is the stronger one. The sign depends on which sublattice S counts as positive.

The figure shows $q(k)$ of the snippet in the complex plane, as k crosses the Brillouin zone once. Each sublattice has one site, so $q(k)$ is a single number, and here it is $q(k) = 0.5 + e^{-ik}$, with k in units of the inverse lattice

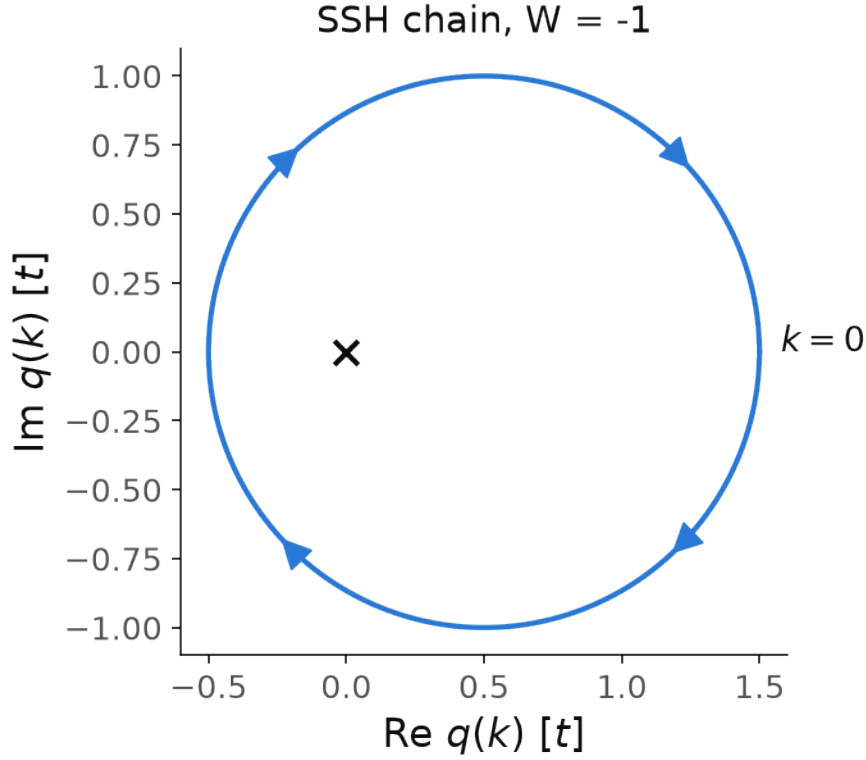


Figure 46: The off-diagonal block $q(k)$ of the SSH chain in the complex plane, as k crosses the Brillouin zone.

constant. This is a circle: its center is the intracell hopping 0.5, and its radius is the intercell hopping 1. So the circle goes around zero, the black cross, once and clockwise, and this is $W = -1$. With the intracell hopping larger than the intercell one, the circle does not contain zero, and $W = 0$.

The code of this figure is in `documentation/figures/topology_winding.py`.

For a normal Hamiltonian the default chiral operator is the sublattice operator. For a Nambu Hamiltonian it is $\sigma_y \tau_y$ on every site, the chiral symmetry of a BdG Hamiltonian that is real in real space. So the same call counts the Majorana zero modes of a Kitaev chain:

```
from pyqula import geometry
h = geometry.chain().get_hamiltonian() # spinful chain
h.add_on_site(20.) # shift both spin bands up
h.add_zeeman([0.,0.,20.]) # and bring one of them back to the Fermi energy
h.add_pairing(mode="pwave",delta=0.3,d=[1.,0.,0.]) # p-wave pairing: a Kitaev chain
W = h.get_winding_number() # sigma_y tau_y is the chiral operator by default
```

This gives $W = 1$ when the chemical potential is inside the band, and 0 outside. The difference with the Zak phase appears for two such chains coupled by a weak rung hopping: there are two Majorana zero modes at each end, $W = 2$, while the Zak phase gives 0, the same as a trivial chain.

- **chiral**: another chiral operator, as a name, a matrix or an `Operator`. The call raises an error if it does not anticommute with the Hamiltonian, or if the gap closes.

See `examples/1d/winding_kitaev_ladder/main.py` for the single chain and the ladder across the band, and `jupyter-notebooks/functionalities/topological_characterization/11_one_dimensional_invariants.ipynb`.

15.7 Z2 invariant of a helical superconducting wire

A wire with time-reversal symmetry, a class DIII superconductor, has a Kramers pair of Majorana zero modes at each end. This is a zero-bias peak that a magnetic field splits. The two invariants above do not see it:

the Zak phase adds the two Kramers partners and is always trivial, and the winding with $\sigma_y \tau_y$ counts them with opposite signs and gives 0. For a time-reversal symmetric Nambu chain, `h.get_topological_invariant()` returns instead the Z_2 invariant, the Kramers polarization

$$\nu = \det U \frac{\text{Pf} \theta(0)}{\text{Pf} \theta(\pi)},$$

where U is the product of the overlaps of the negative-energy states from $k = 0$ to $k = \pi$, and θ is the time-reversal operator restricted to these states at the two time-reversal invariant momenta. So ν compares how the Kramers partners are paired at the two ends of half of the Brillouin zone, and it is -1 when they switch.

We take a helical p-wave wire, where the two spins pair with opposite chirality, with Rashba spin-orbit coupling so that spin is not conserved, and with a competing s-wave pairing:

```
from pyqula import geometry
h = geometry.chain().get_hamiltonian() # spinful chain
h.add_onsite(0.5) # shift the band, keeping the Fermi energy inside it
h.add_rashba(0.2) # Rashba spin-orbit coupling, so that spin is not conserved
h.add_pairing(mode="pwave", delta=0.5j, d=[1., 0., 0.]) # helical p-wave pairing
h.add_swave(0.2) # a competing s-wave pairing
nu = h.get_topological_invariant() # Z2 invariant of the wire
```

The result is -1 , a topological wire. When the s-wave pairing is larger than the p-wave one, `h.add_swave(1.3)`, it is $+1$, and the gap closes in between, near 0.9. When spin is conserved, without the Rashba coupling, ν is the Zak parity of one spin block, a Kitaev chain.

- `nk`: number of k-points from 0 to π , 200 by default. A value of ν far from ± 1 raises an error, which asks for a larger `nk`.

It is also available as `topology.z2_invariant_1d(h)`. See `examples/1d/z2_helical_wire/main.py` for the invariant as the s-wave pairing grows, and the `11_one_dimensional_invariants.ipynb` notebook above.

16 Quantum geometry and Berry curvature

The Berry curvature is only one part of the geometry of the Bloch states. In this page we compute the full quantum geometric tensor, whose other part is the quantum metric, and then we resolve the Berry curvature in energy and in real space.

16.1 Quantum geometric tensor

The quantum geometric tensor is computed with `h.get_quantum_geometric_tensor()`. Its antisymmetric part is the Berry curvature. Its symmetric part is the quantum metric, which measures the distance between neighboring Bloch states. In a flat band the metric is the important quantity: it says how spread the states are, and it controls, for example, the superfluid weight of a flat-band superconductor (see Superconductivity).

For a chosen subspace S of bands, for example the occupied bands, the tensor is

$$Q_{ij}^{mn}(\mathbf{k}) = \sum_{l \notin S} \frac{\langle u_m | \partial_{k_i} H | u_l \rangle \langle u_l | \partial_{k_j} H | u_n \rangle}{(E_m - E_l)(E_n - E_l)}, \quad m, n \in S,$$

where m and n are bands of the subspace, l runs over the bands outside it, $|u\rangle$ are the Bloch states and E their energies. The quantum metric and the Berry curvature are

$$g_{ij} = \text{Re} \sum_{m \in S} Q_{ij}^{mm}, \quad \Omega_{ij} = -2 \text{Im} \sum_{m \in S} Q_{ij}^{mm}.$$

Only states outside S appear in the energy denominators. So the formula is well defined when S contains degenerate bands: a spin-degenerate pair, the Kramers pairs of a model with inversion and time-reversal symmetry, or several orbitals that meet at a high-symmetry point. The only case that fails is a degeneracy between S and a band outside it, and then the call raises a `ValueError`. The derivative $\partial_{k_i} H$ is computed analytically from the hoppings, without finite differences.

16.1.1 The tensor of each pair of bands

The individual entries Q_{ij}^{mn} are not physical. A rotation of the states of S among themselves, which is a free choice inside a degenerate multiplet, also rotates the tensor. Only its trace and its invariants are physical. For this reason `non_abelian=True` returns the tensor in the orbital basis,

$$Q_{ij}(\mathbf{k}) = \sum_{m,n \in S} |u_m\rangle Q_{ij}^{mn} \langle u_n| = P \partial_{k_i} P \partial_{k_j} P P,$$

where P is the projector on S . This depends only on the subspace, not on the basis chosen inside it. Its trace is the tensor above, and in any basis $|v_m\rangle$ of S the tensor of each pair of bands is $\langle v_m | Q_{ij} | v_n \rangle$.

16.1.2 Coordinates

The momenta are in reduced coordinates, $\mathbf{k} = \sum_i k_i \mathbf{b}_i$, with $\mathbf{b}_i$ the reciprocal lattice vectors and k_i of period 1, as everywhere in pyqula. The tensor in Cartesian momenta, the one to compare with a continuum calculation, is JQJ^T with $J_{\alpha i} = (\mathbf{a}_i)_\alpha / 2\pi$. `coordinates="cartesian"` returns it, as a 3×3 tensor in x, y and z . Only this tensor has the symmetry of the crystal, because the reciprocal lattice vectors are in general not orthogonal. For example, on the honeycomb lattice the trace of the reduced metric is not $g_{xx} + g_{yy}$, and it is different at the three M points.

16.1.3 Positions of the orbitals

The band structure does not depend on where the orbitals are inside the unit cell, but the Bloch states do, and so does their geometry. We can write the Bloch Hamiltonian with the phase $e^{i\mathbf{k} \cdot \mathbf{R}}$ of the lattice vector only, as if every orbital were at the origin of its cell, or with the full bond vector, $e^{i\mathbf{k} \cdot (\mathbf{R} + \mathbf{r}_j - \mathbf{r}_i)}$. The two choices give the same bands and the same Chern number, but a different Berry curvature and quantum metric at each $\mathbf{k}$, and a different Brillouin-zone integral of the metric. Only the second one is the geometry of the crystal. It respects the point group: on the honeycomb lattice with a Haldane flux and a sublattice imbalance, the Berry curvature is the same at three $\mathbf{k}$ -points related by a C_3 rotation, while with the first choice it even changes sign between them. And it does not depend on the choice of unit cell: the Brillouin-zone average of $\text{Tr } g$, which is the gauge-invariant spread of the Wannier functions of the subspace, doubles exactly when the cell is doubled.

- `gauge="atomic"` (default): every orbital at its position in the geometry.
- `gauge="lattice"`: every orbital at the origin of its cell. This is the convention of part of the work on flat-band superconductivity, and the one to use to reproduce numbers from it. The dependence on the orbital positions is discussed by Simon and Rudner, Phys. Rev. B **102**, 165148.

16.1.4 Code

```
from pyqula import geometry
from pyqula import topology
g = geometry.honeycomb_lattice() # create honeycomb lattice
h = g.get_hamiltonian() # create hamiltonian of the system (spinful by default)
h.add_haldane(0.2) # Add Haldane coupling
h.shift_fermi(0.3) # put the Fermi level safely mid-gap (gap is [-0.9,0.9])

Q = h.get_quantum_geometric_tensor(k=[0.1,0.2,0.],occ_idxs=[0,1]) # at a k-point
g_metric = h.get_quantum_metric(k=[0.1,0.2,0.],occ_idxs=[0,1]) # quantum metric only

# non-Abelian tensor of the two occupied bands, in the orbital basis
```

```
Qna = topology.quantum_geometric_tensor(h,k=[0.1,0.2,0.],occ_idx=[0,1],
    non_abelian=True)
```

```
# along a k-path, and integrated over the BZ: the integral of the Berry
# curvature part reproduces h.get_chern()
```

```
inds,gpath,omegapath = topology.quantum_geometric_tensor_path(h,occ_idx=[0,1])
C = topology.chern_from_qgt(h,nk=20,occ_idx=[0,1])
```

In two dimensions Q is a complex 2×2 array, one entry per pair of directions, and the metric is its real part. With `non_abelian=True` it is a $2 \times 2 \times 4 \times 4$ array: one 4×4 matrix in the spin-orbital basis for each pair of directions. There is no spin-orbit coupling here, so this matrix is block diagonal in spin, and the trace over the spin-up orbitals is the contribution of the spin-up electrons. Along the k-path the function returns the position along the path, and the metric and the curvature at each point, one 2×2 tensor per k-point. The integral of the curvature over the Brillouin zone gives the Chern number, here $C = 2$ for the two spin-degenerate occupied bands, the same as `h.get_chern()`.

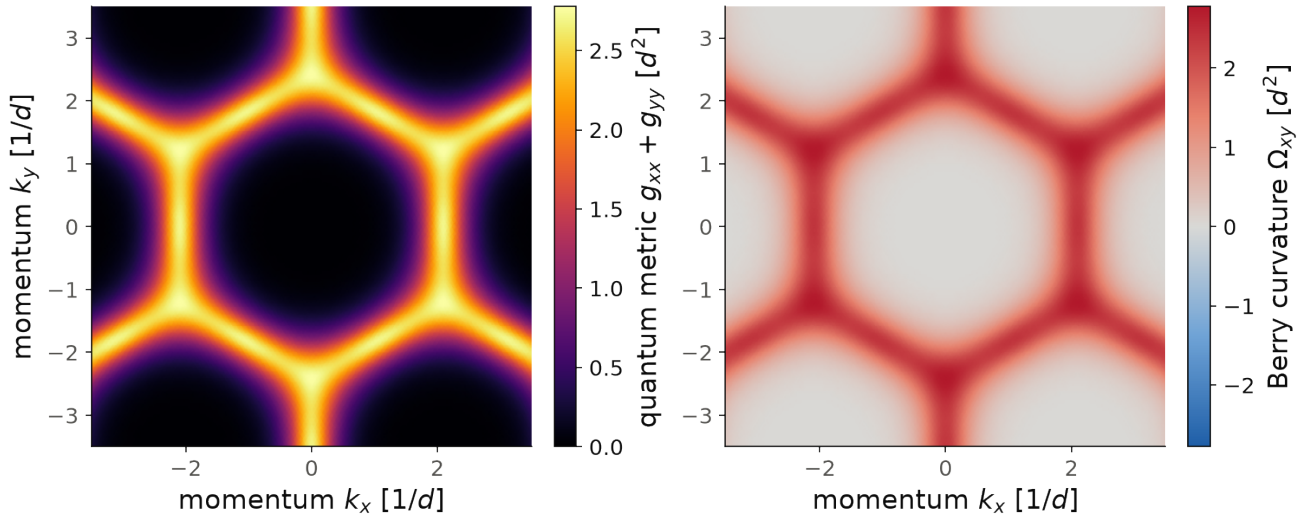


Figure 47: Trace of the quantum metric and Berry curvature of the two occupied bands of the Haldane model over the Brillouin zone.

The figure shows the same tensor on a 60×60 k-mesh, from `topology.quantum_geometric_tensor_mesh(h,nk=60,occ_idx=[0,1])`. We use Cartesian momenta, so that the maps have the hexagonal symmetry of the crystal. Momenta are in units of $1/d$, with d the first-neighbor distance, and the tensor is in units of d^2 . The trace of the metric, $g_{xx} + g_{yy}$, and the Berry curvature are both largest at the K and K' points, the corners of the hexagons, with 2.78, and both are zero at Γ . The curvature is positive everywhere. At every k-point the trace of the metric is larger than or equal to the curvature, by at most 0.32, and the two are equal at K and K'. Averaged over the Brillouin zone, the trace of the metric is 0.89 and the curvature 0.83.

The code of this figure is in `documentation/figures/geometry_qgt.py`.

- `occ_idx`: the indices of the bands in S . By default, the bands with $E < 0$, as in `h.get_chern()`, so it follows `h.shift_fermi()`.
- `k`: the k-point, in reduced coordinates.
- `non_abelian`, `coordinates`, `gauge`: as explained above.

The same functions work in one and three dimensions, where the tensor has one and nine components, and `topology.quantum_geometric_tensor_mesh(h,nk=...)` returns it on a uniform k-mesh.

See `examples/2d/quantum_geometric_tensor/main.py` for the spin-resolved tensor above, `examples/2d/quantum_geometry/main.py` for the Berry curvature and the metric of the occupied bands along a k-path, and `jupyter-notebooks/functionalities/topology` for more.

16.2 Berry curvature density in energy

The Chern number is one integer for all the occupied bands. To see from which energies it comes, we use `topology.chern_density()`. The Berry curvature of the occupied states is written as an integral over energy,

$$\Omega(\mathbf{k}) = \int_{-\infty}^{\epsilon_F} \Xi(\mathbf{k}, \omega) d\omega,$$

where $\Xi(\mathbf{k}, \omega)$ is the energy-resolved Berry curvature: how much curvature the states at energy ω carry. It is built from the Green's function with a broadening `delta`. `topology.chern_density()` integrates $\Xi(\mathbf{k}, \omega)$ over the Brillouin zone at a set of energies, and also returns its cumulative integral in energy.

```
from pyqula import geometry
from pyqula import topology
import numpy as np
g = geometry.honeycomb_lattice() # create honeycomb lattice
h = g.get_hamiltonian() # create hamiltonian of the system
h.add_haldane(0.05) # Add Haldane coupling
es = np.arange(-3.5, 3.5+1e-9, 0.05) # from below the band to above it
(es, cs, csi, cex) = topology.chern_density(h, nk=12, delta=0.1, es=es, exact_at=[0.] )
```

`es` are the energies, `cs` the Berry curvature density at each energy, and `csi` its cumulative integral, which starts at the first energy of `es`. `cex` is the cumulative integral without broadening at the energies of `exact_at`, explained below. Inside the gap `csi` has a plateau, and when the integral goes over all the bands it comes back to zero, because the Berry curvatures of all the bands of a Hamiltonian add up to zero. The normalization is the one of `h.get_chnern()`, but the plateau of `csi` is the Chern number only in a limit that is expensive to reach. The snippet takes about 20 seconds, and its plateau at zero energy is 1.63, not $C = 2$. Four things decide how close the plateau is to the Chern number, and the snippet is chosen to respect the first three; the fourth is the broadening, which `exact_at` removes.

- The integral starts at the first energy of `es`, so `es` must start below the bottom of the band, here -3 . A window that starts at -1 misses the curvature of the lower states: with `es` from -1 to 1 , `delta`=0.02 and `dk`=0.02, the plateau is 3.97 at `nk`=10, 3.18 at `nk`=20 and 2.91 at `nk`=40, and none of these is the Chern number.
- The energy step must be at most `delta`/2, otherwise the peak of each state is not resolved. At `delta`=0.05 and `nk`=12 a step of `delta`/4 changes the plateau by 10^{-3} , but a step of `delta` changes it by 0.12. The snippet has a step of 0.05 for `delta`=0.1.
- The derivatives in momentum are finite differences with a step `dk`, in units of the reciprocal lattice vectors. Over `dk` the bands must move by much less than `delta`, so the default is `dk`=`delta`/25. A fixed step is too large at a small `delta`: at `delta`=0.05 and `nk`=20 the plateau is 1.62 with `dk`=0.02, 1.69 with `dk`=0.005 and the same with `dk`=0.001.
- The broadening `delta` spreads the curvature of the states above the gap into the gap. With `es` from -3.5 to 0 , a step of `delta`/2 or smaller and a small enough `dk`, the plateau at zero energy is 1.70 for `delta`=0.05 (1.69, 1.71 and 1.70 at `nk`=20, 24 and 40), 1.86 for `delta`=0.02 at `nk`=20, and between 1.92 and 1.98 for `delta`=0.01 at `nk`=20, 21 and 30. It goes up towards $C = 2$ as `delta` goes down, slowly, and a smaller `delta` needs a finer `nk`, a finer energy step and a smaller `dk`.

The broadening can be removed exactly at energies inside a gap. The cumulative integral is an integral of an analytic function of the complex energy $\omega + i\delta$, along the line at height `delta`. Inside a gap that function is smooth down to the real axis, so we can close the path with a vertical segment from $\omega + i\delta$ down to ω , and another one at the first energy of `es`, below the band. The result is the integral with no broadening at all, meaning that it does not depend on `delta`. This is what `exact_at` does, for the energies it lists. In the snippet `cex` is 2.56, and 2.58 with `delta`=0.05, while `csi` moves from 1.63 to 2.08. What is left is the k-mesh, since the curvature is a sum over the k-points of the mesh: `cex` is 2.01 at `nk`=24 and 1.99 at `nk`=36, the Chern number. Each energy of `exact_at` costs about as much as 9 energies of `es`, and inside a band the vertical segment is not smooth, so the result there has no meaning.

So read `csi` as a profile in energy, and `cex` as the number. The density also needs a fine enough `nk` for its `delta`, otherwise it shows spurious peaks near k-points where bands come close.

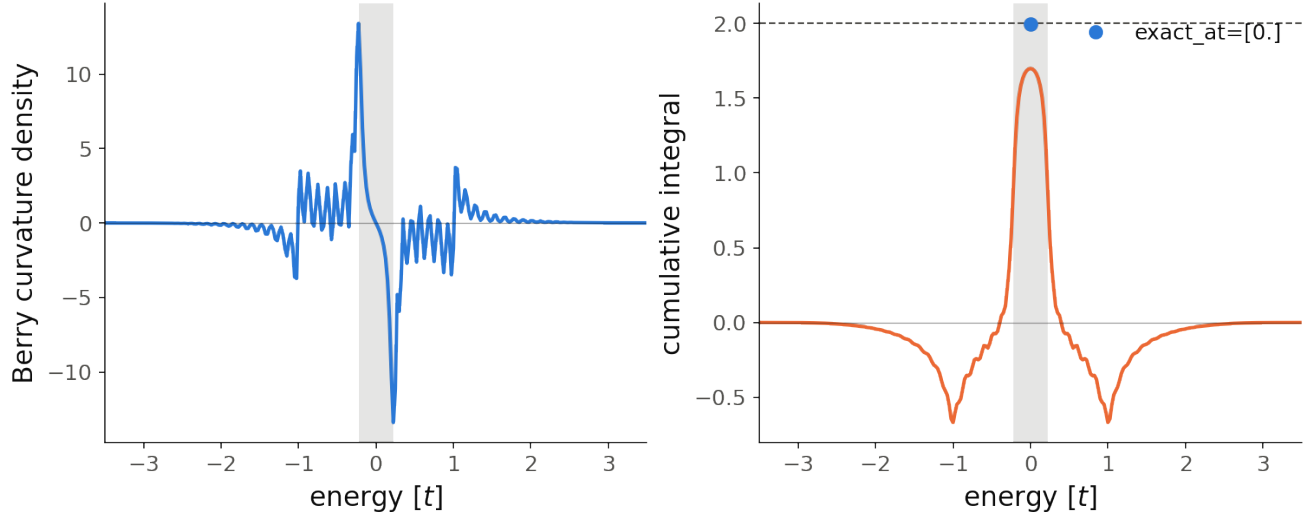


Figure 48: Berry curvature density in energy of the Haldane model, and its cumulative integral.

The figure shows `cs` on the left and `csi` on the right, for the snippet with a smaller broadening and a finer mesh: `es` from -3.5 to 3.5 with a step of `delta/2`, `delta=0.05`, `nk=48` and `exact_at=[0.]`. This takes about 15 minutes. A multiple of 3 for `nk` puts the K points on the mesh, so the band edges are where they should be. The gray band is the bulk gap, from -0.225 to 0.225 . The density has its two largest peaks at the band edges, of opposite sign, and the cumulative integral has its plateau in the gap, 1.70 at zero energy, the same as at `nk=24` within 0.01. The dashed line is the Chern number, $C = 2$: the plateau is below it by 0.30, the curvature that `delta=0.05` moves into the gap. The dot is `cex` at zero energy, 1.998, the same integral without the broadening. The density is odd in energy, so the cumulative integral comes back to zero at the end of the window. Away from the gap the density still oscillates. Each oscillation is a group of k-points of the mesh, which `delta=0.05` does not merge; from the energy step between neighboring k-points we estimate that a smooth density at this `delta` needs `nk` above about 100. The integral averages the oscillations out, so the plateau is converged while the density is not.

The code of this figure is in `documentation/figures/geometry_chern_density.py`.

- `nk`: number of k-points per direction of the mesh.
- `es`: energies at which the density is computed.
- `delta`: broadening of the Green's function.
- `exact_at=None`: energies inside gaps where the cumulative integral is also returned without broadening, as a fourth array.
- `nodes=8`: number of points of the vertical segments of `exact_at`.
- `dk`: step of the finite differences in momentum, in units of the reciprocal lattice vectors, `delta/25` by default.
- `operator`: weight the curvature with an operator, as in `h.get_chern()`.
- `write`: if True, also write `CHERN_DENSITY.OUT` and `CHERN_DENSITY_INTEGRATED.OUT`.

The map of $\Xi(\mathbf{k}, \omega)$ over the Brillouin zone at one energy is given by `topology.d0omega_dE_kmap(h, nk=40, e=0.)`, which writes it to `BERRY_DENSITY_KMAP.OUT`. It covers the same window of whole Brillouin zones as `h.get_berry_curvature()`, so the mean of the map divided by 2π is the value of `cs` at that energy, for the same `delta` and `dk`. See `examples/2d/berry_density/main.py`, `examples/2d/chern_density/main.py` and `examples/2d/berry_density_kmap/main.py`, and `jupyter-notebooks/functionalities/topological_characterization/`

16.3 Berry curvature density in real space

A Chern number needs a Brillouin zone. The Berry curvature can also be resolved in real space, to see where in the sample the topological flux is. This also works for an island, a disordered sample, or a supercell too large for a k-mesh. It is computed with `topology.real_space_chern()`. The idea is

$$\Omega(\mathbf{k}) = \int \Gamma(\mathbf{k}, \mathbf{r}) d^2\mathbf{r},$$

where $\Gamma(\mathbf{k}, \mathbf{r})$ is the Berry curvature resolved in position. In practice, for a finite system, pyqula computes a local Chern marker of the Bianco-Resta type,

$$C(\mathbf{r}) = \frac{2\pi}{A_{\mathbf{r}}} \text{Im} \langle \mathbf{r} | [PxP, PyP] | \mathbf{r} \rangle,$$

with P the projector on the occupied states, x and y the position operators, the diagonal element summed over the orbitals of the site $\mathbf{r}$, and $A_{\mathbf{r}}$ the area of the site. The occupied states are those below zero energy, so a Hamiltonian whose gap is not at zero must be shifted first, for example with `h.set_filling(0.5)`. Deep in the bulk this local quantity is the Chern number.

```
from pyqula import islands
from pyqula import topology
g = islands.get_geometry(name="honeycomb", n=8, nedges=3) # a honeycomb island
h = g.get_hamiltonian() # create hamiltonian of the system
h.add_haldane(0.05) # Add Haldane coupling
(r, c) = topology.real_space_chern(h) # spatially-resolved Berry curvature
```

`r` are the positions of the sites and `c` the local marker at each site. The call also writes `REAL_SPACE_CHERN.OUT`, one value per site, to plot as a map over the island. The Hamiltonian must be finite; a periodic one raises a `ValueError`. For a periodic system, build a large supercell and make it finite first. How to read the marker, and how to get the Chern number from it, is explained in Topology in real space.

The figure shows `c` at each site of the island of the snippet, a triangle of 573 sites, with the color scale cut at ± 3 . Within a distance of 8 from the center, the marker is between 1.75 and 2.02, close to $C = 2$. The sites of the outermost row are strongly negative, down to -17.6 , and they compensate the bulk.

The code of this figure is in `documentation/figures/geometry_realspace.py`.

See `examples/0d/real_space_chern/main.py` and `jupyter-notebooks/functionalities/topological_characterization`

17 Topology in real space

A topological invariant is defined in the Brillouin zone, but its consequences are local: the bulk carries a quantized Berry flux and the edges carry the states that cross the gap. In this page we compute the Chern number from the local marker of a finite sample, also without any lattice, and the spectral function at the edge of a semi-infinite system.

17.1 Chern number in real space

The local marker $C(\mathbf{r})$ of Berry curvature density in real space is a number without units: the marker of a site divided by the area of the site. So in principle

$$C = \frac{1}{A_{\text{cell}}} \int_{\text{cell}} C(\mathbf{r}) d^2\mathbf{r} = \frac{\sum_i C_i A_i}{\sum_i A_i},$$

the average of the marker over one unit cell of the bulk, with C_i the marker and A_i the area of the sites i of the cell. In a crystal every site of the cell has the same area, so this is the plain average over the sites of the cell, and

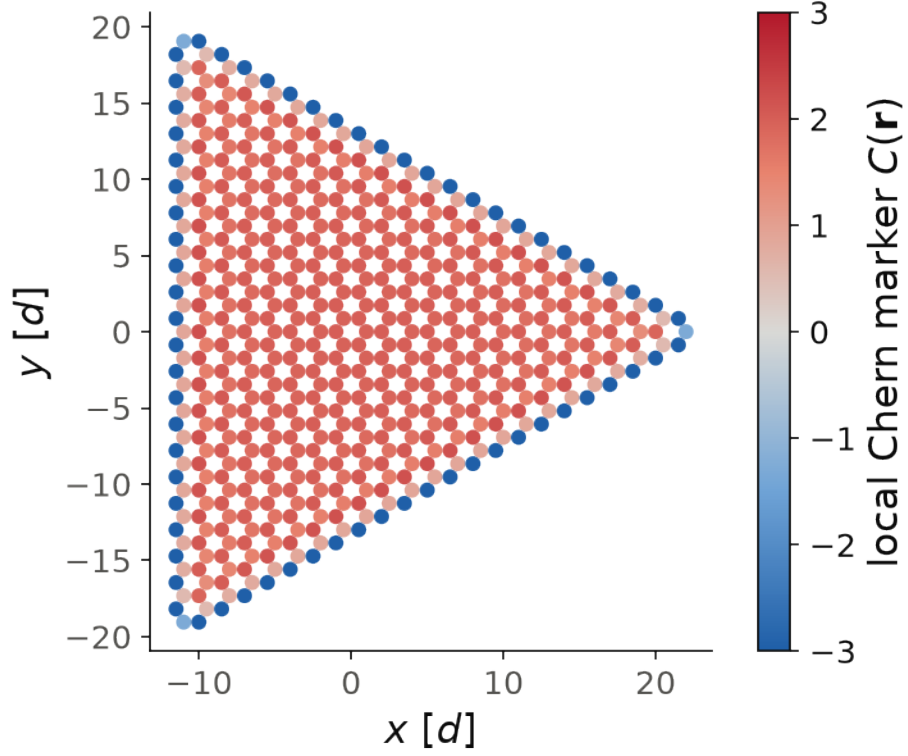


Figure 49: Local Chern marker at each site of a triangular honeycomb island with the Haldane coupling.

every site of the bulk gives about C . On a finite sample this has to be read with care. The sum of the marker, times the area of each site, over the whole island is exactly zero. So summing over all the sites does not give the invariant. What happens is the following. Deep inside a large enough island, the surroundings of a site look like the infinite bulk, and there the marker has a plateau at the bulk Chern number. The sites at the edge carry the opposite weight, which cancels it. This cancellation is the bulk-boundary correspondence in real space: the flux carried by the bulk is given back by the edge states.

```
from pyqula import islands
from pyqula import topology
import numpy as np
g = islands.get_geometry(name="honeycomb",n=8,nedges=3) # a honeycomb island
h = g.get_hamiltonian() # create hamiltonian of the system
h.add_haldane(0.05) # Add Haldane coupling
(r,c) = topology.real_space_chern(h) # local marker, per site
r = np.array(r)
bulk = np.argsort(np.linalg.norm(r-r.mean(axis=0),axis=1))[:len(r)//4] # innermost sites
C = np.mean(c[bulk]) # bulk plateau value approximates the total Chern number
```

The marker of each site is divided by the area of the site. This makes it a number without units, whose value in the bulk is the Chern number. The area of a site is the area of its Voronoi cell, the region of the plane closer to this site than to any other. In a crystal this is the area of the unit cell divided by the number of sites in it. The sites at the edge have open Voronoi cells, and they take the mean area of the others.

For this island the innermost quarter of the sites gives about 1.95, close to the $C = 2$ of the spinful Haldane model. The difference of a few percent comes from the small gap of $t_2 = 0.05$. A small gap means a long decay length of the edge into the bulk, and the island with $n=8$ is not much larger than it, so its innermost quarter still feels the edge. A larger gap or a larger island removes the difference: with $t_2 = 0.2$ the innermost sites give 2.00, and with $t_2 = 0.05$ the islands with $n=12$ and $n=16$ give 1.985 and 1.995. The outermost sites are strongly negative, and the marker times the area adds up to zero over the whole island.

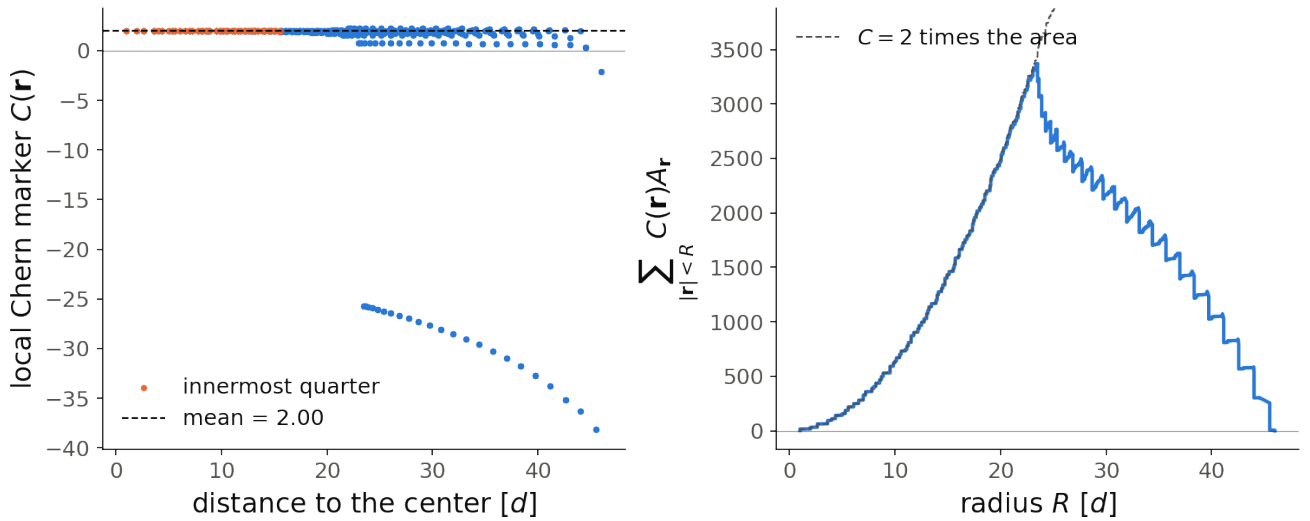


Figure 50: Local Chern marker of each site against its distance to the center of the island, and its sum inside a radius R .

The figure uses the larger island, $n=16$, with 2301 sites, which takes about two minutes. On the left is the marker of each site against its distance to the center of the island. The innermost quarter of the sites, in orange, sits on the plateau, with mean 1.995. Further out the marker spreads, between about 0.7 and 2.3, and the sites at the edge go down to -38 . On the right we add up the marker times the area of each site, for the sites closer than R to the center. In the bulk every site gives about 2 times its area, so the sum follows 2 times the area inside R (dashed line). It grows up to $R \approx 23.5$, the distance from the center to the middle of an edge of the triangle, where it is 3371, against 3440 for 2 times the area. Then the edge sites come in, with their large negative marker, and the sum goes down to zero, within 10^{-11} , at the last site. This zero is exact for any finite island: the marker of a site is a diagonal element of the commutator of PxP and PyP , with P the projector on the occupied states, and the sum of the diagonal elements, the trace, of a commutator of two finite matrices is zero. So the edge always gives back what the bulk carries.

The code of this figure is in `documentation/figures/realspace_marker.py`.

17.1.1 An amorphous Chern insulator

The marker also works when there is no lattice at all. We take the amorphous Chern insulator of Agarwala and Shenoy. The sites are placed at random, uniformly and without correlations, in a square; this is what `geometry.amorphous_lattice()` builds. Each site has two orbitals. Every pair of sites closer than $R = 4$ is connected by a hopping that decays exponentially with the distance and depends on the direction of the bond. This dependence on the direction makes the model topological in a window of the mass M .

```
from pyqula import geometry
from pyqula import topology
from pyqula.geometrytk import amorphous
from pyqula.topologytk.realspace import site_areas
import numpy as np

g = geometry.amorphous_lattice(L=36.,density=0.6,seed=0) # 778 sites at random positions
h = amorphous.amorphous_chern_hamiltonian(g,M=-0.5) # Agarwala-Shenoy model, half filling
(r,c) = topology.real_space_chern(h) # local marker, per site
A = site_areas(g) # area of the Voronoi cell of each site
bulk = np.max(np.abs(g.r[:,0:2]),axis=1)<6. # central square, far from the edges
C = np.sum(c[bulk]*A[bulk])/np.sum(A[bulk]) # area-weighted average of the marker
```

The two orbitals of each site play the role of the spin, so for pyqula h is spinful. The Hamiltonian is shifted so that the Fermi energy of half filling is at zero, which is where `topology.real_space_chern()` puts the Fermi energy. The marker now changes a lot from site to site, because no two sites have the same surroundings. And

the sites do not have the same area, so the Chern number of a region is not the plain average of the marker, but its average weighted by the area of each site,

$$C = \frac{\sum_i C_i A_i}{\sum_i A_i}.$$

This gives $C \approx -1$. Agarwala and Shenoy find a Bott index of -1 at this density in the window $-2 \lesssim M \lesssim 1.2$, and the bulk marker goes to zero on both sides of it: sharply at $M \approx -2$, and slowly at positive M , where it is still about $-1/2$ at $M = 1$, because the gap of a sample of this size closes slowly there.

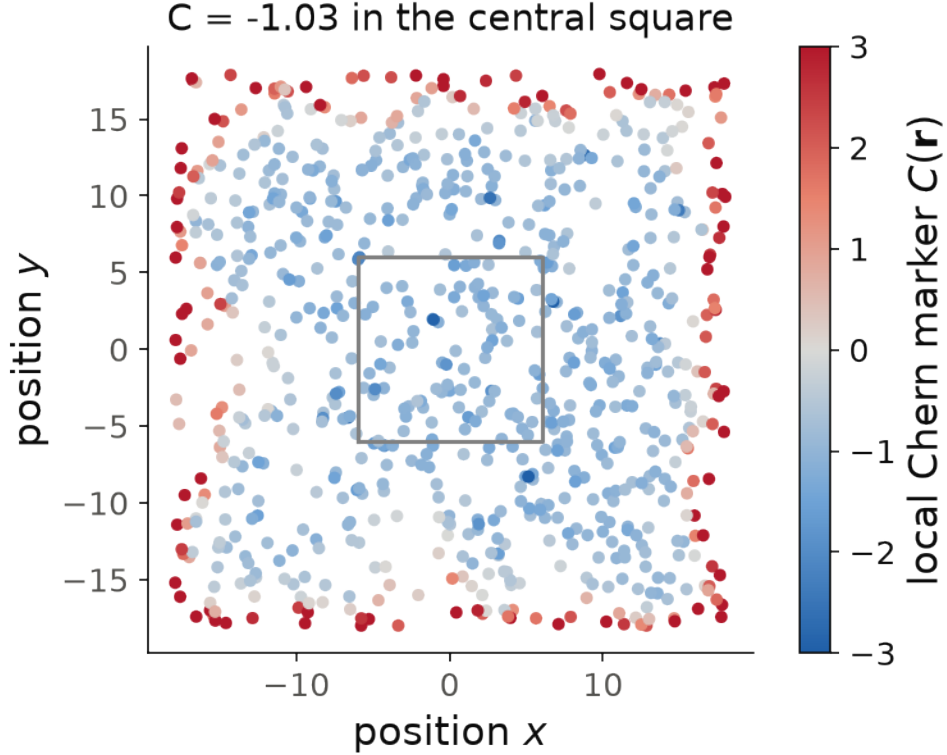


Figure 51: Local Chern marker at each site of the amorphous Chern insulator, with the central square drawn.

The figure shows the marker at each site of the snippet, with the central square of the average drawn in gray. Inside the sample the marker is negative at almost every site, around -1 , but it changes a lot from site to site. The area-weighted average over the central square is -1.03 . The sites at the edge of the sample are strongly positive, and they compensate the bulk, as on the island above.

The code of this figure is in `documentation/figures/realspace_amorphous.py`.

See `jupyter-notebooks/functionalities/topological_characterization/05_real_space_chern_amorphous.ipynb` for the maps of the Chern density and the two transitions.

17.1.2 Where local markers are useful

The real-space Berry curvature and the Chern marker are examples of a topological marker: a local quantity, computed from the ground-state projector only, that gives a bulk invariant without translational symmetry. So they are the tool for disordered systems, finite flakes, amorphous solids, or systems where the parameters change in space, for example a Haldane mass that changes sign across a boundary, or a topological insulator with dilute vacancies. The map of the marker shows where the invariant is carried, and where it changes.

17.2 Topological surface states

The invariant predicts states at the edge. The direct way to see them is the spectral function of a semi-infinite system: a half-plane, periodic along the edge and infinite away from it. The momentum along the edge is still a good quantum number, and the states localized at the edge appear as branches that cross the bulk gap. This is what ARPES sees on a surface, and what STM sees as in-gap density of states at an edge. For the Haldane model we use `kdos.surface()`:

```
from pyqula import geometry
from pyqula import kdos
g = geometry.honeycomb_lattice() # create honeycomb lattice
h = g.get_hamiltonian() # create hamiltonian of the system
h.add_haldane(0.05) # Add Haldane coupling
kdos.surface(h) # surface spectral function
```

The call writes `KDOS.OUT`, with four columns: the momentum along the edge, the energy, the spectral weight at the surface and the spectral weight in the bulk. It also returns these four arrays, flat, with one entry per pair of momentum and energy. By default it uses a hundred energies between -1 and 1 and a broadening `delta=0.01`. If you plot the surface weight as a function of momentum and energy, you see the chiral edge states crossing the gap, one for each unit of the Chern number. They are absent from the bulk weight: inside the gap the bulk weight is almost zero and the surface weight is not.

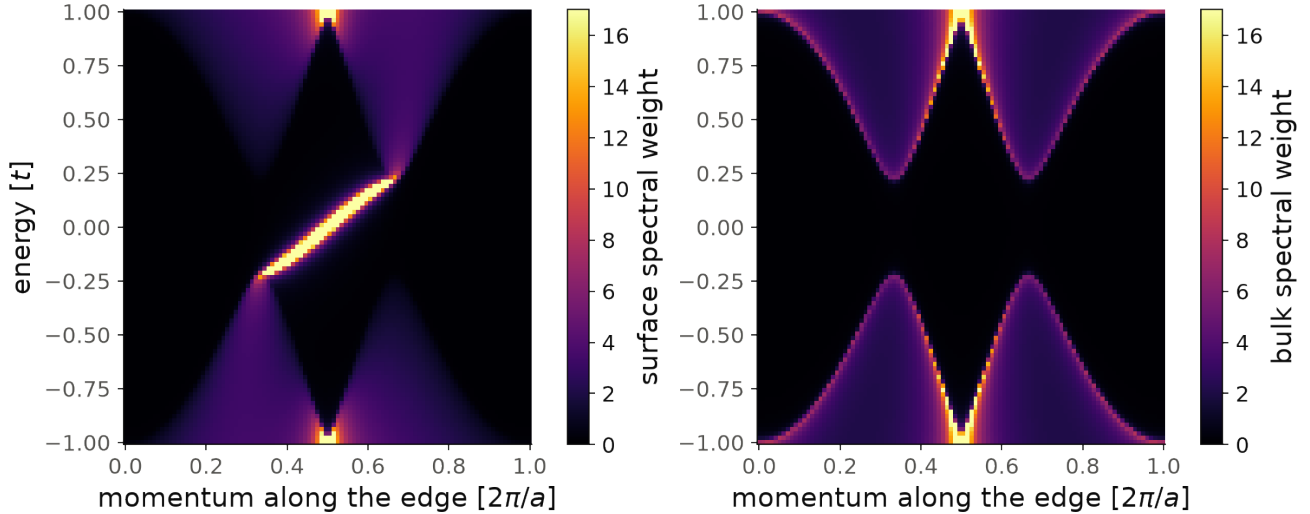


Figure 52: Surface and bulk spectral functions of the Haldane model as a function of the momentum along the edge and of the energy.

The figure shows the four arrays of the snippet, reshaped to 100 momenta times 100 energies: the surface weight on the left and the bulk weight on the right, with the same color scale, cut at the 99th percentile of the surface weight. The bulk weight shows the gap, from -0.225 to 0.225 , at the momenta $1/3$ and $2/3$, the projections of K and K' . In the surface weight one chiral branch crosses the gap, from -0.225 at momentum $1/3$ to 0.225 at momentum $2/3$, through zero energy at momentum $1/2$. The two spins give the same branch, on top of each other, so this one line carries the two units of $C = 2$.

The code of this figure is in `documentation/figures/realspace_surface.py`.

The same calculation is available as `h.get_surface_kdos()`, with the arguments `energies`, `delta`, `kpath` and `operator`, described in Spectral functions, together with the quantum spin Hall case. See `examples/readme_examples/chern_surface_states/main.py` and `examples/2d/kdos_edge/main.py` for scripts that make the plot.

18 Entanglement

In this page we compute the entanglement entropy of a region of a ground state, with `h.get_entanglement_entropy()`, and its entanglement spectrum, with `h.get_entanglement_spectrum()`.

18.1 Entanglement entropy and entanglement spectrum

The ground state of a non-interacting or mean-field Hamiltonian is a Slater determinant. For a Slater determinant, the reduced density matrix of a region A is the exponential of a single-particle operator, so it is fixed by the correlations inside A only. We compute the correlation matrix restricted to the region,

$$C_{ij} = \langle c_i^\dagger c_j \rangle, \quad i, j \in A,$$

and diagonalize it. Its eigenvalues ζ_n are between 0 and 1, and they give

$$S = - \sum_n [\zeta_n \ln \zeta_n + (1 - \zeta_n) \ln(1 - \zeta_n)], \quad \xi_n = \ln \frac{1 - \zeta_n}{\zeta_n},$$

the entanglement entropy S and the single-particle entanglement spectrum ξ_n . An eigenvalue at 0 or 1 is a state fully outside or fully inside the region, and it does not contribute to the entropy. An eigenvalue at $1/2$, $\xi_n = 0$, is a state shared equally between the region and the rest, and it contributes the most, $\ln 2$. Only a matrix of the size of the region is diagonalized, never the reduced density matrix itself, so large regions are cheap.

The region is given as a list of site indices, a boolean mask, a function of the position, or a fraction of the cells. Spin, sublattice and Nambu components are orbitals of a site, so a region is always a set of sites. A periodic Hamiltonian is first folded into a ring of `nsuper` unit cells, so the region A has two boundaries, not one. Its entropy is then twice the entropy of a single cut. Keep this in mind when comparing with a formula written for one boundary.

18.1.1 The entanglement spectrum as a topological probe

In a two-dimensional system cut along one direction, the momentum $k_{\parallel}$ parallel to the cut is still a good quantum number, and $\xi_n(k_{\parallel})$ is the Li-Haldane entanglement spectrum. For a Chern insulator, it has branches that cross $\xi = 0$ inside its gap, $2|C|$ of them: $|C|$ chiral modes for each of the two boundaries. This is the same as the edge spectrum of the model. For a trivial insulator, the entanglement spectrum has a gap around $\xi = 0$. This is a calculation of the bulk ground state only: no ribbon and no open boundary is built, and the edge states are read from the bulk.

```
from pyqula import geometry
g = geometry.honeycomb_lattice() # create a honeycomb lattice
h = g.get_hamiltonian(has_spin=False) # get the Hamiltonian
h.add_haldane(0.1) # Chern insulator, C = 1
print(h.get_chern(nk=20)) # 1.0

# Li-Haldane entanglement spectrum xi_n(k_par) across the BZ
(ks,xis) = h.get_entanglement_spectrum(nsuper=10,nk=101)

# entanglement entropy, per parallel unit cell, averaged over the BZ
print(h.get_entanglement_entropy(nsuper=10,nk=20))
```

`ks` are the `nk` parallel momenta across the Brillouin zone. `xis` has one row per momentum and one column per orbital of the region, ten here (half of a ring of ten cells, with two sites per cell). If you plot every column against `ks`, you see the levels of the Chern insulator crossing $\xi = 0$. The same plot for the honeycomb lattice gapped by a sublattice imbalance shows a spectrum that never comes close to zero. The entropy is given per parallel unit cell, averaged over `nk` momenta, and it counts both boundaries of the region.

On the left is the spectrum of the snippet. On the right is the same lattice with a sublattice imbalance of 0.3 instead of the Haldane coupling, a trivial insulator. In the Chern insulator two levels cross $\xi = 0$ at $k_{\parallel} = 1/2$, in

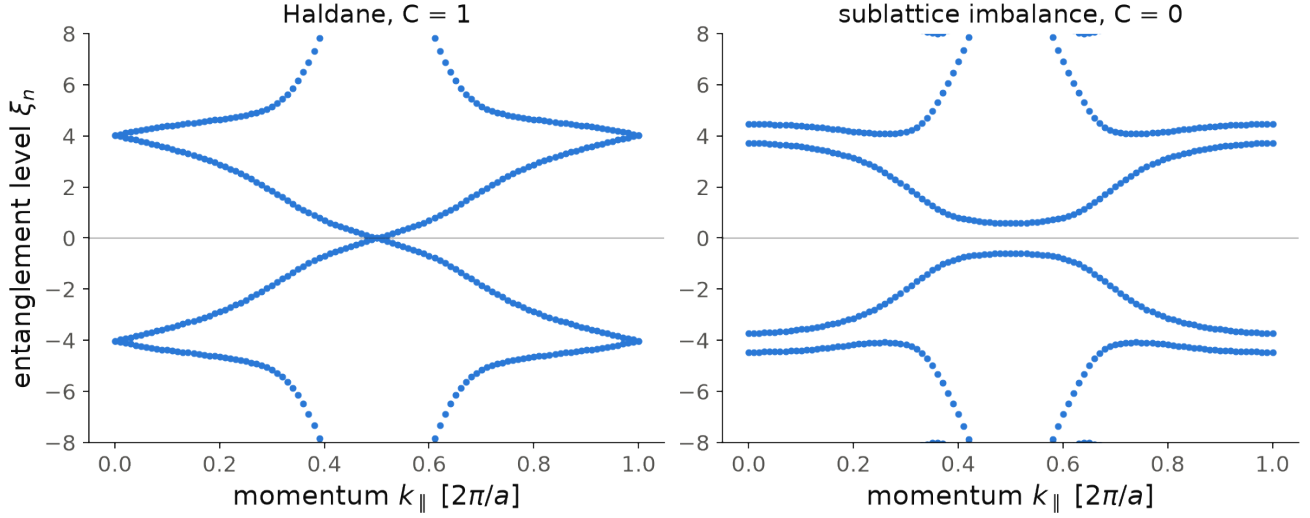


Figure 53: Li-Haldane entanglement spectrum of the Haldane model and of a trivial insulator, against the momentum parallel to the cut.

opposite directions: one for each boundary of the region, so $2|C| = 2$ crossings. In the trivial insulator no level comes closer to zero than 0.59. The plot is cut at ± 8 ; the levels beyond are states almost fully inside or fully outside the region, which do not contribute to the entropy.

The code of this figure is in `documentation/figures/entanglement_spectrum.py`.

Keywords:

- **region**: the region A , in one of the four forms above. **None** takes half of the system.
- **nsuper**: number of unit cells of the ring, for a periodic Hamiltonian.
- **direction**: the lattice direction normal to the cut, by default the last periodic one.
- **kpar**: a single parallel momentum. With **None**, a two-dimensional Hamiltonian is swept or averaged over **nk** momenta.
- **nk**: number of parallel momenta.
- **fermi**: the states with $E < \text{fermi}$ are occupied, zero by default.

18.1.2 Remarks

For a Bogoliubov-de Gennes Hamiltonian, the correlation matrix includes the anomalous correlations $\langle c_i c_j \rangle$, and the doubling of the Nambu basis is divided out. **fermi** must stay at zero in this case, because the chemical potential is already in the Hamiltonian.

The occupation is taken at zero temperature. A level exactly at the Fermi energy makes the ground state degenerate, and then the entropy depends on which state is chosen, so the call raises an error. On a ring this happens for some values of **nsuper**, for example for a half-filled chain with **nsuper** divisible by four. Changing the size of the ring by one solves it.

As a reference, the critical chain gives the conformal field theory result with central charge $c = 1$,

$$S = \frac{c}{3} \ln \left[\frac{L}{\pi} \sin \frac{\pi l}{L} \right],$$

for a region of l cells in a ring of L cells. A gapped two-dimensional insulator follows the area law instead, the entropy proportional to the length of the boundary.

See `examples/1d/entanglement_entropy_chain/main.py` (the central charge from a fit of the entropy against the size of the region) and `examples/2d/entanglement_spectrum_haldane/main.py` (the spec-

trum of the Haldane model next to the one of a trivial insulator) for scripts that make the plots, and `jupyter-notebooks/functionalities/topological_characterization/12_entanglement.ipynb`.

19 Optical conductivity

A response function tells how a system reacts to a weak perturbation. A field couples to an operator B , we measure the change of an operator A , and the ratio, as a function of frequency ω and momentum transfer q , is the response function $\chi_{AB}(q, \omega)$. Its imaginary part is what a spectroscopy measures. The response pages go in three steps: the bare response of a single-particle Hamiltonian (this page and Charge response and RKKY), the same response dressed with interactions in the random phase approximation (RPA), whose poles are magnons and plasmons (Spin response and magnons), and the Bethe-Salpeter equation for excitons (Excitons).

19.1 The Kubo formula

We start with the response to an electric field, the optical conductivity tensor $\sigma_{ab}(\omega)$ of a periodic Hamiltonian, computed with `h.get_optical_conductivity()`. It is given by the Kubo-Greenwood formula,

$$\sigma_{ab}(\omega) = \frac{ie^2\hbar}{N_k V_{\text{cell}}} \sum_{\mathbf{k}} \sum_{n,m} \frac{f_n - f_m}{E_m - E_n} \frac{v_{nm}^a v_{mn}^b}{\hbar\omega + i\eta - (E_m - E_n)},$$

where n, m are the bands at momentum $\mathbf{k}$, E_n their energies, f_n their occupations at temperature T , v_{nm}^a the matrix element of the velocity along direction a , N_k the number of $\mathbf{k}$ -points, V_{cell} the volume (area, length) of the unit cell, and η a broadening. The physical meaning is simple: light of frequency ω is absorbed when it can move an electron from an occupied band to an empty band at energy $\hbar\omega$ above, and the weight of this transition is given by the velocity matrix element.

The terms with $E_n = E_m$ give the intraband (Drude) response. For them the factor $(f_n - f_m)/(E_m - E_n)$ is replaced by its limit $-\partial f/\partial E$, and the result is a peak at $\omega = 0$,

$$\sigma_{ab}^{\text{intra}}(\omega) = \frac{iD_{ab}}{\omega + i\eta}, \quad D_{ab} = \frac{1}{N_k V_{\text{cell}}} \sum_{\mathbf{k}, n} \left(-\frac{\partial f}{\partial E} \right)_{E_n} v_{nn}^a v_{nn}^b,$$

with D_{ab} the Drude weight, available as `h.get_drude_weight()`. The velocity is the current operator of the tight-binding model, $v_a = i[H, r_a]$, built from the full vector of each bond, also the bonds inside the unit cell. Because of this the tensor has the symmetry of the lattice: on the honeycomb lattice $\sigma_{xx} = \sigma_{yy}$ exactly, and graphene gives its universal absorption $\pi e^2/4h$ per spin.

The full complex tensor is returned. $\text{Re } \sigma_{xx}$ is the optical absorption. σ_{xy} is the magneto-optical (Kerr and Faraday) response. Its $\omega \rightarrow 0$ limit is the anomalous Hall conductivity, which for a Chern insulator with Chern number C is quantized to

$$\sigma_{xy}(0) = -C \frac{e^2}{h},$$

with C as returned by `h.get_chern()`.

Units: $\hbar = e = 1$, energies in units of the hopping t , lengths in units of the lattice constant. The conductivity is returned in units of $e^2/\hbar$ (times a^{2-d} in d dimensions), so one conductance quantum $e^2/h = e^2/(2\pi\hbar)$ is the value $1/(2\pi)$. The Fermi energy is at zero, as everywhere in pyqula.

The absolute normalization is fixed by the f-sum rule,

$$\int_{-\infty}^{\infty} \text{Re } \sigma_{aa}(\omega) d\omega = \pi W_{aa},$$

where W_{ab} is the diamagnetic weight, the occupied average of $\partial^2 H / \partial k_a \partial k_b$, available as `h.get_sum_rule_weight()`. The weight is shared between the Drude peak and the interband absorption.

19.2 A Chern insulator

We take the honeycomb lattice with a Haldane term, a Chern insulator, and compute its absorption and its Hall response:

```
import numpy as np
from pyqula import geometry
g = geometry.honeycomb_lattice() # create a honeycomb lattice
h = g.get_hamiltonian(has_spin=False) # get the Hamiltonian
h.add_haldane(0.2) # Chern insulator
h.shift_fermi(0.3) # put the Fermi energy in the gap

(ws,s) = h.get_optical_conductivity(energies=np.linspace(0.,4.,401),
                                   nk=200,T=0.02,delta=0.05)

absorption = s[:,0,0].real # Re sigma_xx, the optical absorption

# the DC Hall response is quantized to the Chern number
(w0,s0) = h.get_optical_conductivity(energies=[0.],nk=40,T=0.01,delta=1e-3)
print("sigma_xy(0) in e^2/h:",2.*np.pi*s0[0,0,1].real) # -1
```

`ws` contains the frequencies and `s` the 3×3 tensor at each frequency, so `s` has shape $(\text{len}(ws), 3, 3)$. The absorption is zero below the gap and starts at the gap. The second call computes the tensor at zero frequency with a small broadening, and $2\pi\sigma_{xy}$ comes out as -1 , one conductance quantum with the sign of $-C$.

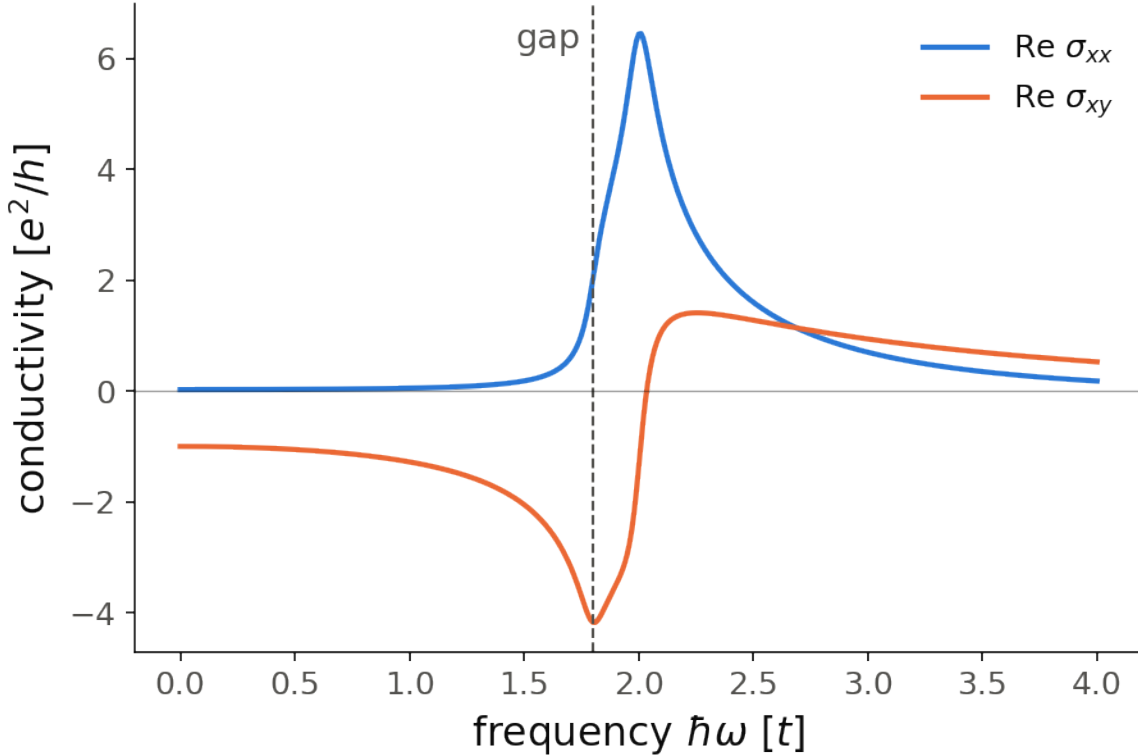


Figure 54: Real part of the longitudinal and Hall optical conductivity of the Haldane Chern insulator against frequency, with the band gap marked.

The dashed line is the band gap, $1.8t$, from `h.get_gap()`. Below the gap the absorption $\text{Re } \sigma_{xx}$ is almost zero; the small tail comes from the broadening $\text{delta}=0.05$. It rises at the gap and has its maximum, $6.45 e^2/h$, at

$\omega = 2.0t$. The Hall part $\text{Re} \sigma_{xy}$ starts at $-1.00 e^2/h$ at $\omega = 0$, the quantized value, goes down to $-4.18 e^2/h$ at the gap, and changes sign at $\omega = 2.03t$, next to the maximum of the absorption. With `delta=0.05` the k-mesh has to be fine: at `nk=40` both curves have wiggles above the gap, which are the separate k-points of the mesh. At `nk=200` they are gone, and `nk=300` changes the curves by less than $2 \times 10^{-4} e^2/h$.

The code of this figure is in `documentation/figures/optical_chern.py`.

The keywords are

- **energies**: the frequencies $\hbar\omega$.
- **nk**: number of k-points per direction. A small **delta** needs a large **nk**, otherwise the absorption shows the discrete k-mesh.
- **T**: temperature of the Fermi function.
- **delta**: the broadening η .
- **intraband**, **interband**: switch on or off the Drude part and the interband part.

The formula works for one- and two-dimensional Hamiltonians. Superconducting (Nambu) Hamiltonians and three-dimensional ones raise `NotImplementedError`; for a Nambu Hamiltonian the hole block carries the opposite charge, so the current is not $i[H, r]$.

See `examples/2d/optical_conductivity/main.py` and `examples/1d/optical_conductivity_chain/main.py` for runnable scripts, and `01_optical_conductivity.ipynb` and `02_drude_weight_f_sum_rule.ipynb` in `jupyter-notebooks/functionalities/response_functions/` for executed notebooks.

20 Charge and generic response functions

In this page we compute the bare response function of a single-particle Hamiltonian: first the charge response, then the response of any pair of operators, and finally the RKKY coupling between two magnetic moments, which is a static response.

20.1 Charge-charge response function

We put a small potential on site j and look at the change of the density on site i . This is the charge-charge response function, computed with `h.get_chi()`. For a periodic system at momentum transfer q it is the Lindhard function

$$\chi_{ij}(q, \omega) = \frac{1}{N_k} \sum_{\mathbf{k}} \sum_{n,m} [f(\epsilon_{\mathbf{k},n}) - f(\epsilon_{\mathbf{k}+q,m})] \frac{\Psi_{\mathbf{k},n}^*(i) \Psi_{\mathbf{k}+q,m}(i) \Psi_{\mathbf{k}+q,m}^*(j) \Psi_{\mathbf{k},n}(j)}{\epsilon_{\mathbf{k},n} - \epsilon_{\mathbf{k}+q,m} - \omega + i\delta},$$

where $\Psi_{\mathbf{k},n}(i)$ is the amplitude of the eigenstate n at momentum $\mathbf{k}$ on site i (summed over spin), $\epsilon_{\mathbf{k},n}$ its energy, f the Fermi-Dirac distribution and δ a broadening. A pair of states contributes only if one is occupied and the other is empty, and it resonates at the energy difference of the two states. A transition from an occupied state to an empty state at energy ω above gives a positive $\text{Im} \chi_{ii}$ at $\omega > 0$, and a negative one at $\omega < 0$. The static limit is negative: a positive potential pushes electrons away.

`h.get_chi()` returns $\chi_{ii}(q, \omega)$ averaged over the sites i of the unit cell. For the chain at $q = 0$:

```
from pyqula import geometry
g = geometry.chain() # create a chain
h = g.get_hamiltonian() # create hamiltonian of the system
(es, chis) = h.get_chi(q=[0.,0.,0.]) # get response function
```

`es` contains the frequencies and `chis` one complex number per frequency. The imaginary part is what inelastic scattering (electron energy loss, inelastic x-rays) measures. At $q = 0$ the chain has a single band, so there is no other state at the same momentum to scatter into, and the response vanishes at any finite frequency. Only the static limit survives,

$$\chi(\omega \rightarrow 0) = -N(0),$$

with $N(0) = 1/(\pi t)$ the density of states per site at the Fermi energy, counting both spins, so -0.32 here. With a finite broadening this appears as a peak of width δ around $\omega = 0$. This static limit comes from pairs of states with the same energy, which are treated with $-\partial f/\partial \epsilon$, the limit $\delta \rightarrow 0$ of the occupation factor over the energy difference. When δ is larger than the spacing of the energies on the k-mesh, these pairs give an error of order $1/n_k$, so for a converged calculation at finite δ use an odd n_k .

The keywords are

- **q**: the momentum transfer, in units of the reciprocal lattice vectors.
- **energies**: array of frequencies ω .
- **delta**: the broadening δ . The temperature T is equal to **delta** unless given.
- **nk**: number of k-points per direction for the sum over the Brillouin zone.

Scanning over q gives a $\chi(q, \omega)$ map:

```
import numpy as np
g = geometry.chain()
h = g.get_hamiltonian()
qs = np.linspace(-1., 1., 401)
es = np.linspace(-3., 3., 301) # frequencies
chis = [h.get_chi(q=[q, 0., 0.], energies=es, nk=401, delta=0.05)[1] for q in qs]
```

In this map you see the particle-hole continuum of the band: all the energies $\epsilon_{k+q} - \epsilon_k$ that it costs to move an electron from an occupied state k to an empty state $k + q$.

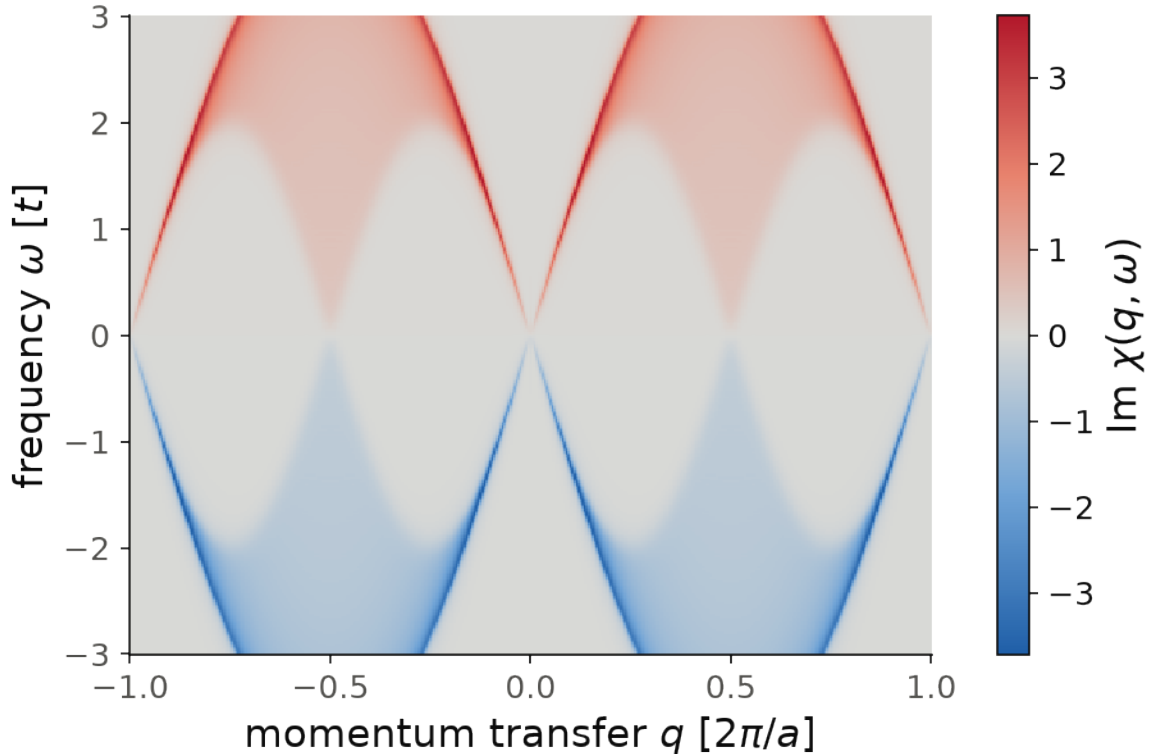


Figure 55: Imaginary part of the charge response of the chain over momentum transfer and frequency.

For the chain $\epsilon_k = 2t \cos k$, and at half filling the continuum at momentum transfer q goes from $2t|\sin 2\pi q|$ up to $4t|\sin \pi q|$. The weight is largest at the upper edge, the dark red lines. At small q the continuum is a narrow line, $\omega \approx 4\pi t q$, set by the Fermi velocity. At $q = 1/2$ it goes from zero up to $4t$, above the window of the plot, which stops at $3t$. The sign follows the frequency: red for $\omega > 0$, blue for $\omega < 0$. The k-mesh is converged: with $n_k=801$ instead of 401 the map changes by less than 0.01, for a maximum of 3.7. The width of the edges is the broadening $\text{delta}=0.05$. A k-mesh that is too coarse for this broadening gives horizontal stripes inside the

continuum, so when you make `delta` smaller, make `nk` larger too. The map takes about 20 seconds.

The code of this figure is in `documentation/figures/charge_map.py`.

See `examples/1d/charge_response/main.py` for a runnable script, and `jupyter-notebooks/functionalities/response_fur` for an executed notebook.

20.2 Response of any pair of operators

The same formula works for any two operators A and B ,

$$\chi_{AB}(q, \omega) = \frac{1}{N_k} \sum_{\mathbf{k}, n, m} [f(\epsilon_{\mathbf{k}, n}) - f(\epsilon_{\mathbf{k}+q, m})] \frac{\langle \Psi_{\mathbf{k}, n} | A | \Psi_{\mathbf{k}+q, m} \rangle \langle \Psi_{\mathbf{k}+q, m} | B | \Psi_{\mathbf{k}, n} \rangle}{\epsilon_{\mathbf{k}, n} - \epsilon_{\mathbf{k}+q, m} - \omega + i\delta},$$

where the perturbation couples to B and we measure A . Taking $A = B = S_x$ gives the transverse spin susceptibility. We put the chain in a Zeeman field along z , so that the spin and the charge responses are different:

```
from pyqula import geometry
import numpy as np
g = geometry.chain()
h = g.get_hamiltonian(has_spin=True)
h.add_zeeman([0., 0., 0.5]) # Zeeman field along z, splits the bands by 1.0
sx = h.get_operator("sx") # any of the operators from the "Operators" section
es = np.linspace(0., 2., 401)
(es, chi) = h.get_chi(A=sx, B=sx, q=[0.02, 0., 0.], energies=es, nk=2001, delta=0.02)
```

By default $A = B$ is the identity, which gives the charge response above. Any operator of the Operators page (spin, valley, sublattice, position, Nambu) can be used, which gives the corresponding susceptibility. Note that the operator "sx" is the Pauli matrix σ_x , without the factor 1/2. The operator "sz" would not show anything new here: it does not flip the spin, so with the field along z its response is exactly the charge response.

The field moves the two spin bands apart in energy by $1.0t$, so flipping a spin costs the Zeeman splitting $1.0t$. The charge response, in orange, does not flip the spin. An electron goes from k to $k+q$ in the same band, which costs the Fermi velocity times q , so there is one line at $0.245t$. The spin response, in blue, has nothing there. A S_x perturbation moves an electron from one spin band to the other, so it costs the Zeeman splitting plus or minus the Fermi velocity times q , and there are two lines, at $0.75t$ and $1.245t$, one for the electrons that move to the right and one for those that move to the left. At $q = 0$ the two lines merge into one at exactly $1.0t$: this is the precession of the spins in the field, the line that electron spin resonance measures. The lines are sharp because at small q all the electron-hole pairs at this q have about the same energy. The k-mesh is converged: with `nk=4001` the positions and the heights of the lines do not change. A finer `delta` makes the lines narrower and higher, and then `nk` has to grow with it.

The code of this figure is in `documentation/figures/charge_operator.py`.

See `jupyter-notebooks/functionalities/response_functions/04_operator_response.ipynb` for an executed notebook.

20.3 RKKY interaction

Now we use the static response to couple two magnetic impurities. A magnetic moment in a metal polarizes the conduction electrons around it, and a second moment at some distance feels this polarization. The result is an effective exchange coupling between the two moments, the Ruderman-Kittel-Kasuya-Yosida (RKKY) interaction,

$$H_{\text{RKKY}} = -J_{\text{RKKY}}(\mathbf{r}) \vec{S}_i \cdot \vec{S}_j,$$

where $J_{\text{RKKY}}(\mathbf{r})$ is proportional to J_K^2 , with J_K the local exchange between each moment and the electrons, times the static spin response $\chi_{ij}(\omega = 0)$ of the electrons between the two sites at distance $\mathbf{r}$. `rkky.rkky_map` computes it between a reference site and every other site up to some distance:

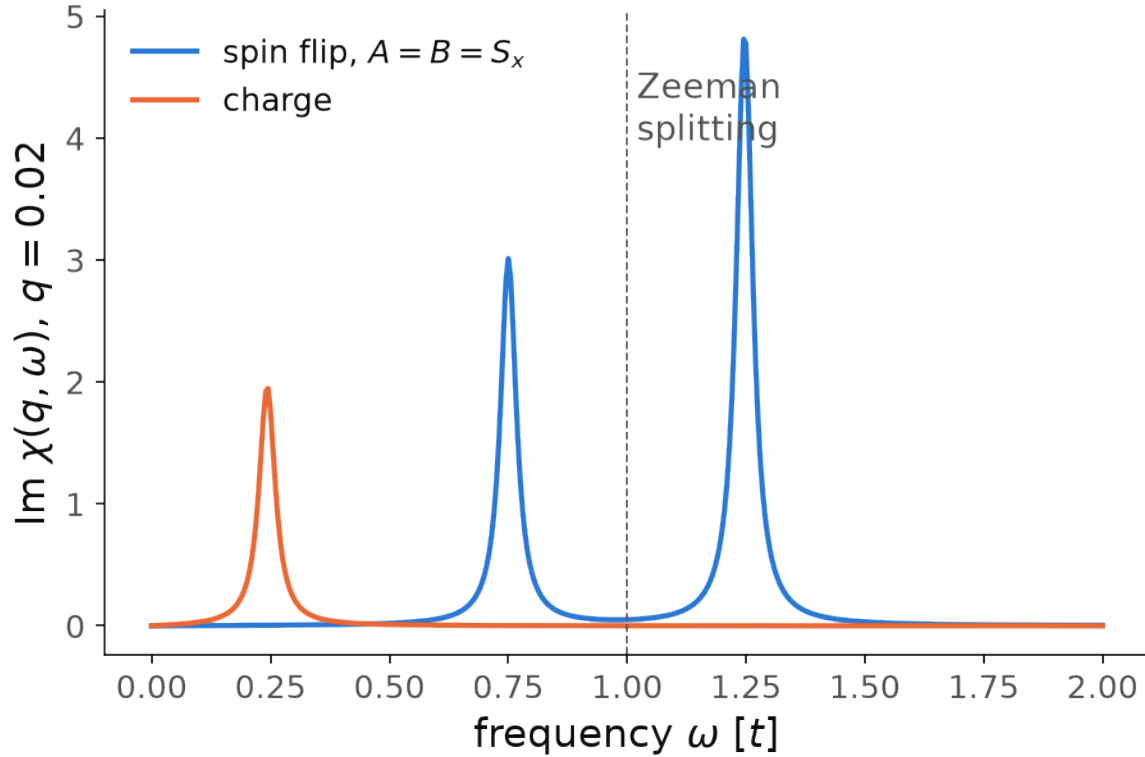


Figure 56: Imaginary part of the transverse spin response and of the charge response of the chain in a Zeeman field, at one momentum transfer, against frequency.

```
from pyqula import geometry
from pyqula import rkky
g = geometry.chain()
h = g.get_hamiltonian()
h.add_onsite(1.)
m = rkky.rkky_map(h,n=10,mode="LR",nk=200) # linear-response RKKY vs distance
```

The keywords are

- **mode**: "LR" computes the static response function above. "pm" puts a small magnetic field on each site and computes the change of the total energy, which is slower but needs no response function.
- **n**: up to how many unit cells away the coupling is computed.
- **nk**: number of k-points per direction for the sum over the Brillouin zone.

`m` has one row per site. The first three columns are the position (x, y, z) of the site relative to the reference site, and the last column is the RKKY coupling:

```
x,e = m[:,0],m[:,3]
```

The coupling alternates in sign with the distance and decays, with a period π/k_F set by the Fermi wavevector k_F . This is why the code moves the Fermi energy away from half filling with an onsite energy first. With the sign of H_{RKKY} above, a positive value is a ferromagnetic coupling and a negative value an antiferromagnetic one. As a reference: for the chain at half filling, without the onsite energy, the coupling is negative (antiferromagnetic) at odd distances, between the two sublattices, and positive (ferromagnetic) at even distances, as expected for a bipartite lattice at half filling.

The onsite energy moves the band to $2t \cos k + 1$, so the Fermi energy is at $\cos k_F = -1/2$, $k_F = 2\pi/3$, and the period π/k_F is 1.5 sites. On the lattice this gives a pattern that repeats every three sites: two antiferromagnetic values and one ferromagnetic value. The coupling is -0.027 , -0.025 and $+0.028$ at distances 1, 2 and 3, and it decays to -0.006 at distance 10. Only the sites on one side are plotted, the other side is the same.

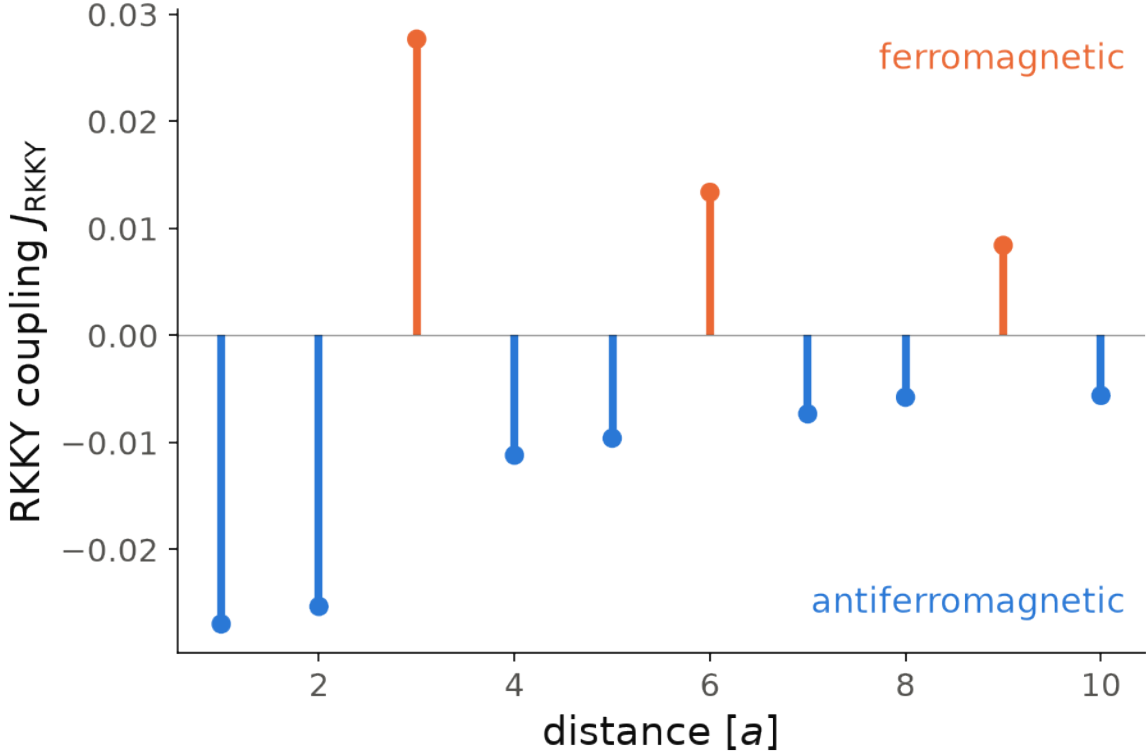


Figure 57: RKKY coupling of the chain against the distance from the reference site, ferromagnetic and antiferromagnetic values in two colors.

The code of this figure is in `documentation/figures/charge_rkky.py`.

See `examples/1d/RKKY/main.py` and `examples/1d/rkky_minimal/main.py`, which compares "pm" and "LR" on the same system, for runnable scripts, and `jupyter-notebooks/functionalities/response_functions/05_rkky.ipynb` for an executed notebook.

21 Spin susceptibility, RPA and magnons

In Charge response and RKKY we computed the bare response of non-interacting electrons. Now we add the interactions, in the random phase approximation (RPA), and we find the collective modes of an interacting state: magnons in a magnet and plasmons in the charge channel. The interacting state is a mean-field Hamiltonian, obtained as in Mean-field Hubbard models.

21.1 Spin susceptibility in the RPA

When we perturb an interacting system, the perturbation moves some electrons, this changes the mean field that the other electrons feel, and this moves more electrons. Summing this series gives the RPA response,

$$\chi_{\text{RPA}}(q, \omega) = \chi_0(q, \omega) [1 - V(q) \chi_0(q, \omega)]^{-1},$$

where χ_0 is the bare response matrix of the mean-field Hamiltonian, with one entry per site and per operator, and $V(q)$ is the interaction vertex in the same channel. For the spin response the vertex is built from the Hubbard U stored on the mean-field Hamiltonian. In a magnetically ordered state, the RPA response has the magnons, which the bare response does not have. It is computed with

- `h.get_spinchi_ladder()`: the transverse response, between S^+ and S^- , where the magnons appear.
- `h.get_spinchi_full()`: the full tensor between S_x , S_y and S_z .

We converge an antiferromagnetic chain and compute its transverse response at a small momentum transfer:

```
from pyqula import geometry
import numpy as np
g = geometry.bichain() # two sites per cell, so Neel order fits in the cell
h = g.get_hamiltonian(has_spin=True)
seed = h.copy() ; seed.add_antiferromagnetism(0.5) # symmetry-breaking seed
hmf = h.get_mean_field_hamiltonian(U=3.0,filling=0.5,mf=seed,nk=100) # magnetic state
(es,chis) = hmf.get_spinchi_ladder(energies=np.linspace(0.,2.,400),q=[0.1,0.,0.],nk=40,delta=2e-2)
```

The imaginary part of `chis` has a sharp peak at the magnon energy at this q . This is what inelastic neutron scattering measures in a magnet. Without spin-orbit coupling the peak goes to zero energy at $q = 0$: it is the Goldstone mode of the broken spin-rotation symmetry.

The keywords are

- `q`: the momentum transfer, in units of the reciprocal lattice vectors. Without `q`, the response is dressed at every q of the k -mesh and then averaged, which gives the local response, the one a probe on a single site sees.
- `RPA: True` (default) dresses the response with the interaction; `False` gives the bare response.
- `energies, delta, nk`: frequencies, broadening and k -mesh, as for `h.get_chi()`.

Two related methods use the same response:

- `h.get_qdos_iets()` scans `get_spinchi_full` along a q -path, so it gives the dispersion of the spin excitations along high-symmetry directions.
- `h.get_iets_ldos()` computes the local response at one energy on every site, which is the map of the spin-flip signal that inelastic tunneling spectroscopy (IETS) measures with an STM. Together with `h.get_ldos()` it gives the elastic and inelastic STM maps.

The q -path labels "G", "K", "M" need a two-dimensional lattice. On the antiferromagnetic honeycomb lattice, the dispersion along the path is

```
g2 = geometry.honeycomb_lattice() # already two sublattices per cell
h2 = g2.get_hamiltonian(has_spin=True)
hmf2 = h2.get_mean_field_hamiltonian(U=3.0,filling=0.5,mf="antiferro")
qdisp = hmf2.get_qdos_iets(energies=np.linspace(0.,2.,250),qpath=["G", "K", "M"],nq=120,nk=60,delta=1e-2)
```

Before reading the spectrum, check that the mean-field state is really ordered, with `hmf.get_vev("sz")`. An RPA calculation on a non-magnetic state runs without problems, but it has no magnons. Two things decide whether the state is ordered. First, the unit cell must be able to hold the order: a `geometry.chain()` has one site per cell and cannot have Néel order, while `bichain` and `honeycomb_lattice` have two sublattices. Second, U must be larger than the critical value: on the honeycomb lattice at half filling $U = 2$ gives a moment of 0.002, so no order, and $U = 3$ gives 0.44. The cost of `get_qdos_iets` grows with nq , with the number of energies and with the number of k -points, n_k^2 in two dimensions: the grid above takes about five minutes.

On the left is the first snippet, the chain at $q = 0.1$, with the bare response (`RPA=False`) added as a reference. Both are summed over the two sites of the cell and plotted on a log scale. The RPA response has one sharp peak, the magnon, at $\omega = 0.39t$. The bare response has no peak there: it has only the electron-hole continuum, which starts at the mean-field gap $1.86t$ and peaks at $2t$, with a tail below the gap from the broadening. With `nk=80` instead of 40 the two curves do not change. On the right is `qdisp` of the honeycomb lattice, also on a log color scale. The bright line is the magnon. It starts at zero at Γ , the Goldstone mode, rises to $1.25t$ along Γ -K, is at $1.16t$ at K and at $1.45t$ at M. The weaker signal above it is the electron-hole continuum. It starts at about $1.33t$ at Γ and at K, and its lower edge is a bright line. The magnon is the same with `nk=30, 40` and 60; the larger `nk` only makes the continuum smoother, so we use `nk=60`.

The code of this figure is in `documentation/figures/spin_rpa.py`.

See `examples/1d/rpa/main.py` (RPA spin response against q for an antiferromagnetic chain), `examples/2d/rpa_triangular/main.py` and `examples/2d/rpa_honeycomb/main.py` (dispersion along a q -path), and `examples/0d/rpa_island/main.py`

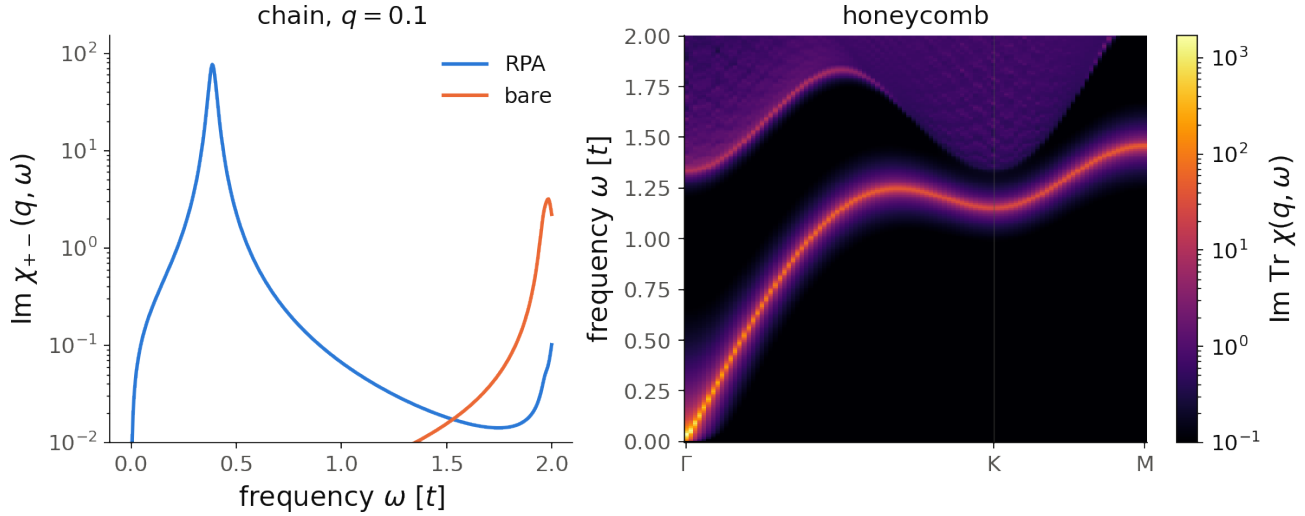


Figure 58: Left: transverse spin response of the antiferromagnetic chain at one momentum, with and without RPA. Right: spin response of the antiferromagnetic honeycomb lattice along a q-path, as a heat map.

and `examples/0d/rpa_finite_chain/main.py` (real-space IETS maps) for runnable scripts, and the notebooks `09_rpa_beyond_on-site.ipynb` and `10_rpa_collective_modes.ipynb` in `jupyter-notebooks/functionalities/interactin`

21.1.1 Precision on a GPU

The RPA functions can run on a GPU, after `gpu.set_gpu(True)` (see GPU). On a GPU the sum of the bare response is done in single precision by default, which changes the bare response by about 10^{-7} . Close to a pole, where $1 - V\chi_0$ is almost singular, the RPA amplifies this error: next to the Goldstone mode of a Néel state the dressed response changes by a few parts in 10^4 . If you need the response at a pole with more digits, use `chi_prec="double"`:

```
from pyqula import geometry
from pyqula import gpu
import numpy as np
gpu.set_gpu(True) # run the GPU kernels of the package on the device
g = geometry.honeycomb_lattice() # two sites per cell
h = g.get_hamiltonian() # first-neighbor hopping
hmf = h.get_mean_field_hamiltonian(U=3.0,filling=0.5,mf="antiferro",nk=10) # Neel mean field
es = np.linspace(0.01,1.0,50) # frequencies
(es,chi) = hmf.get_spinchi_full(q=[0.2,0.,0.],nk=10,energies=es,delta=0.05,
                                chi_prec="double") # the response in double precision
```

On the CPU the default is already double precision.

21.2 Collective modes and magnon bands

The RPA response diverges when the matrix $1 - V(q)\chi_0(q,\omega)$ becomes singular. The frequencies where this happens are the collective modes: magnons in the spin channel, plasmons in the charge channel. If an eigenvalue of the kernel crosses zero already at $\omega = 0$, the state is unstable (a Stoner or charge-order instability). `h.get_rpa_kernel_poles()` scans a frequency window at fixed q and returns all the poles, for any pair of operators A, B and any interaction matrix V. By default it uses the charge channel and $q = 0$:

```
from pyqula import geometry
import numpy as np
g = geometry.bichain() # two sites per cell, so Neel order fits in the cell
h = g.get_hamiltonian()
seed = h.copy() ; seed.add_antiferromagnetism(0.5) # symmetry-breaking seed
```

```

hmf = h.get_mean_field_hamiltonian(U=3.0,nk=100,mf=seed,filling=0.5)
N = len(g.r) # the response matrix is one entry per site
V = 3.0*np.identity(N) # charge-channel interaction, in site space
poles = hmf.get_rpa_kernel_poles(V=V,q=[0.1,0.,0.],
                                energies=np.linspace(0.,4.,200),delta=2e-2,nk=40)

```

V must have the size of the response matrix. In the charge channel there is one entry per site, so V is $N \times N$. It is not hmf.V , the mean-field interaction in the space of sites and spins ($2N \times 2N$), which gives a dimension error here. For the spin channel use `h.get_magnon_bands()` below, which builds the vertex from hmf.V itself.

`poles` is an array of shape `(npoles,2)`, one row per mode: the frequency of the pole and the imaginary part of the kernel eigenvalue at the crossing. This imaginary part can have either sign, so look at its absolute value: a small value means a sharp mode, a large value means a damped mode or numerical noise. The snippet above gives three poles, between $3.57t$ and $3.88t$, all with an absolute value between 2.3 and 2.7, so they are all damped: in the charge channel this state has no sharp collective mode at $q = 0.1$.

21.2.1 Magnon bands

For the magnons, `h.get_magnon_bands()` finds the poles in the spin channel, between S_x , S_y and S_z , with the interaction of the converged mean field, along a q -path. The result is the magnon dispersion of the ordered state:

```

qs,ws,gammas = hmf.get_magnon_bands(nq=200,energies=np.linspace(0.01,3.,600),delta=2e-2,nk=80)

```

Each q -point can have a different number of poles, so `qs`, `ws` and `gammas` are flat arrays of the same length, ready for a scatter plot. `qs` is the index of the q -point along the path, as in `h.get_bands()`, `ws` the frequency of the pole and `gammas` its imaginary part; keep `np.abs(gammas) < threshold` to see only the sharp branches. For the antiferromagnetic chain you see an acoustic branch that goes to zero at $q = 0$, the Goldstone mode, and an optical branch above it.

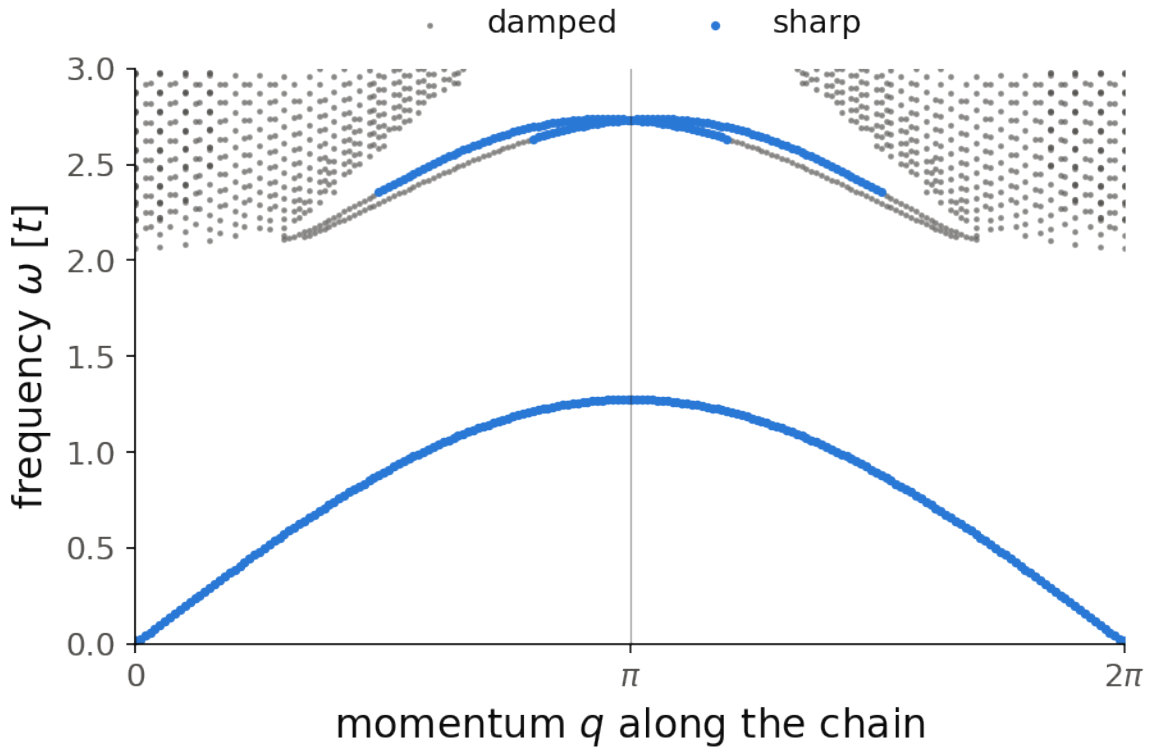


Figure 59: Magnon bands of the antiferromagnetic chain: sharp poles and damped poles of the RPA kernel against momentum.

Here we keep the poles with $|\gamma| < 0.05$ as sharp. The acoustic branch goes from zero at $q = 0$ up to $1.28t$ at

$q = \pi$, and back to zero at 2π . Above $2.06t$ the gray points are damped poles, the electron-hole continuum. On a finite k -mesh the continuum is a set of discrete lines, and the poles between them make the regular pattern of gray points; a finer nk makes it denser, not smoother. The optical branch is the line of sharp points from $2.35t$ to $2.74t$, between $q = \pi/2$ and $3\pi/2$, inside the continuum. The two branches are converged: with $nk=40$ instead of 80 they move by less than $10^{-3}t$. The scan takes about half a minute.

The code of this figure is in `documentation/figures/spin_magnons.py`.

See `examples/1d/magnon_bands/main.py` for a runnable script.

21.2.2 Interactions beyond onsite

The interaction V of `get_rpa_kernel_poles` can also have a range. It is then given as a dictionary `{(n1,n2,n3): matrix}`, with the key the lattice vector between the two unit cells, as for the hoppings in `h.get_hopping_dict()`. It is Fourier transformed in the same way as a hopping,

$$V(q) = \sum_{\mathbf{R}} V(\mathbf{R}) e^{2\pi i q \cdot \mathbf{R}},$$

with q in units of the reciprocal lattice vectors. For a first-neighbor V_1 on a chain this gives $V(q) = 2V_1 \cos 2\pi q$, repulsive at $q = 0$ and attractive at the zone boundary $q = 1/2$:

```
import numpy as np
from pyqula import geometry
g = geometry.chain()
h = g.get_hamiltonian(has_spin=False)
N = h.intra.shape[0] # one site per cell here
I = np.identity(N)
V = {(0,0,0): 0.0*I, (1,0,0): 0.6*I, (-1,0,0): 0.6*I} # nearest-neighbor V1
poles = h.get_rpa_kernel_poles(V=V,q=[0.5,0.,0.],energies=np.linspace(0.,2.,60),
                               delta=2e-2,nk=200)
```

By default this is the charge channel, so V here is a density-density interaction, one entry per site. At $q = 1/2$ the snippet gives three poles, at $0.011t$, $0.081t$ and $0.106t$. The lowest one, almost at zero frequency and with a small imaginary part (0.03), is the soft charge mode: the half-filled chain with this V_1 is close to a charge-density-wave instability.

21.2.3 Which magnon method

In the spin channel, an interaction between different sites needs more care. We write the interaction as $H_{\text{int}} = \frac{1}{2} \sum_{ij} V_{ij} n_i n_j$. In the transverse channel, the electron-hole pair (i, j) with the electron on site i and the hole on site j is scattered by $-V_{ij}$, the Fock (exchange) term, so the vertex is diagonal in the pair index (i, j) , not in the site index. An onsite U only has pairs with $i = j$, and then a vertex with one index per site is exact. A neighbor interaction V_{ij} acts on pairs with $i \neq j$, which a vertex with one index per site does not contain. For example, a ferromagnetic chain ordered only by V_1 would have no magnon at all with a site vertex. For this reason the spin response functions (`get_spinchi_full`, `get_spinchi_ladder`, `get_iets_ldos`, `get_magnon_bands`) use the site vertex only for an onsite U , and sum the series in the pair basis for any interaction between different sites. The call is the same in both cases. The pair basis does not support Nambu Hamiltonians.

`h.get_magnon_bands()` takes a `method` keyword with three options:

interaction	"rpa"	"pair"	"tdhf"
onsite Hubbard U	yes	yes	yes
neighbour-shell	through the pair basis	yes	yes
density-density $V_1, V_2, \dots$			
exchange $J_1, J_2, \dots$	through the pair basis	yes	yes
(isotropic or anisotropic)			
metallic reference	yes	yes	with <code>metal=True</code>

interaction	"rpa"	"pair"	"tdhf"
non-collinear (canted, spiral) state	yes	yes	yes
frequency-resolved $\chi(\omega)$	yes	yes	no (an eigenproblem)

- **"rpa"** (default): poles of the RPA kernel on a frequency grid. Exact for an onsite U ; for anything else it uses the pair basis.
- **"pair"**: the same series in the pair basis. Only the pairs that the interaction connects enter, so a short-range interaction is cheap. It works for metals, and for any spin structure.
- **"tdhf"**: time-dependent Hartree-Fock, an eigenvalue problem with no frequency grid and no broadening (see below).

"pair" and **"tdhf"** agree for every interaction in the table, and for an onsite U **"rpa"** gives the same magnons. This is true also at $Q = \Gamma$. There the Goldstone mode has zero energy, and **"pair"** and **"rpa"** find it at an energy equal to the broadening `delta`, because the broadening moves it there. The list of **"rpa"** also has poles of the longitudinal S_z response, because it uses the full (S_x, S_y, S_z) response. They are close to the ones that **"pair"** gives with `channel="all"`. For a collinear state **"pair"** keeps only the spin-flip pairs, so it gives the magnons and not the charge and longitudinal excitations. With `channel="all"` it keeps all the pairs and gives those excitations too. As an example, the honeycomb lattice with an onsite U and a first-neighbor V_1 , its magnons and its transverse response:

```
import numpy as np
from pyqula import geometry
from pyqula.meanfield import VJinteraction
nk = 24 # the SCF and the magnon share this mesh
g = geometry.honeycomb_lattice()
hmf = VJinteraction(g.get_hamiltonian(), U=3.0, V1=0.5, filling=0.5,
                    mf="antiferro", nk=nk, mix=0.5, maxerror=1e-10).hamiltonian
qs, ws, gammas = hmf.get_magnon_bands(method="pair", nq=90,
                                       energies=np.linspace(1e-3, 3., 400), delta=1e-3, nk=nk)
es, chi = hmf.get_transverse_spinchi(energies=np.linspace(0., 2., 300),
                                     q=[0.1, 0., 0.], delta=1e-2, nk=nk) # (Sx, Sy, Sz) x site chi(w)
```

The kernel has an exchange (Fock) term and a Hartree term. For a collinear state the Hartree term does not enter the transverse response. For a non-collinear state the spin sectors mix, and without the Hartree term the spin-rotation symmetry is broken and the acoustic magnon gets a wrong gap. Both terms are kept, so the Goldstone mode is at zero also for canted and spiral states.

An exchange interaction $J \vec{S}_i \cdot \vec{S}_j$ is not a density-density interaction. Its Ising part $JS_i^z S_j^z$ is, but its transverse part $\frac{J}{2}(S_i^+ S_j^- + \text{h.c.})$ flips a spin on each site. The mean field writes $S_i^x S_j^x$ and $S_i^y S_j^y$ as Ising terms in two rotated spin frames, and records the three spin channels on the Hamiltonian it returns. **"pair"** and **"tdhf"** read these channels, so they include the transverse part, and for an isotropic J the Goldstone mode is at zero. This needs a Hamiltonian from `VJinteraction`, or from `h.get_mean_field_hamiltonian()` with J1, J2, J3 or Jr. The single-axis couplings `SzSz`, `SxSx` and `SySy` are recorded as an anisotropic exchange. A global spin rotation keeps an isotropic exchange, but would turn an anisotropic one into cross terms like $S_i^x S_j^y$, so `h.global_spin_rotation()` refuses it (unless the rotation is around the axis of a uniaxial exchange); rotate the Hamiltonian before the mean-field calculation instead.

For the map we call `hmf.get_transverse_spinchi()` at 90 q-points of the default path, which goes from next to Γ through K, M and K' back to Γ , and plot the trace of its imaginary part on a log color scale. The circles are the lowest pole from `get_magnon_bands` at each q-point, and the small gray dots are all the other poles. The lowest bright line is the magnon: $0.11t$ at the first point of the path, $0.89t$ at K and $1.67t$ at M, and the circles sit on it. At $q = (0.1, 0, 0)$, the call of the snippet, the magnon peak of the response is at $0.57t$. At Γ , on the right edge, the response is large at $\omega \rightarrow 0$, the Goldstone mode, and the pole scan lists it at $\omega = 10^{-3}$, the broadening `delta`. Above the magnon is the electron-hole continuum, which starts at about $0.9t$ to $1t$ near Γ and K. On a finite k-mesh it is a set of discrete lines, and the interaction shifts each line that it couples into a pole, so the scan lists between 60 and 130 poles at each q-point, the gray dots. They are not other collective modes. A bare

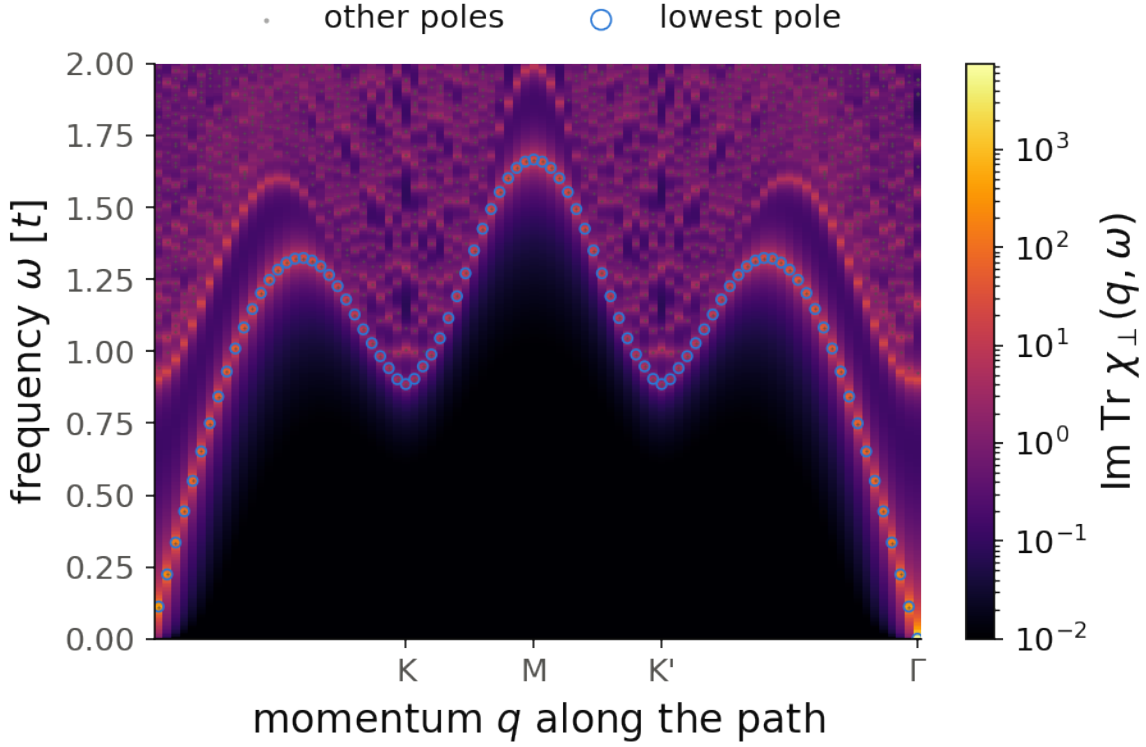


Figure 60: Transverse spin response of the honeycomb antiferromagnet with U and V_1 along the q -path, as a heat map, with the poles of the pair-basis magnon bands on top.

line that the interaction does not couple is not a pole: there the kernel goes through infinity, not through zero, so it is not listed.

The mean field and the response use $nk=24$. For this state the error of the self-consistent loop goes down slowly with the default mixing $mix=0.3$, so we take $mix=0.5$. Then $maxerror=1e-10$ is reached in about 200 iterations. With $nk=18$ the magnon moves by at most $0.015t$, near K , and with $nk=30$ it is at $0.89t$ at K too. The coarse mesh $nk=6$ gives a magnon that is wrong by up to $0.4t$ and a continuum of a few separate lines. The figure takes about five minutes.

The code of this figure is in `documentation/figures/spin_pair.py`.

See `jupyter-notebooks/functionalities/interacting_mean_field_hamiltonians/19_magnons_pair_ladder.ipynb` for an executed notebook that compares the pair basis with time-dependent Hartree-Fock.

21.2.4 Magnons from time-dependent Hartree-Fock

You can think of a magnon as an exciton of a mean-field state: a bound electron-hole pair, where the electron and the hole have opposite spin. So the Bethe-Salpeter equation of Excitons, restricted to the spin-flip pairs, describes it. This is `method="tdhf"`, time-dependent Hartree-Fock. It works in the pair basis, so it includes interactions between neighbors:

```
import numpy as np
from pyqula import geometry
from pyqula.meanfield import VJinteraction
nk = 24 # the SCF and the magnon share this mesh
g = geometry.honeycomb_lattice()
scf = VJinteraction(g.get_hamiltonian(), U=3.0, V1=0.5, filling=0.5,
                    mf="antiferro", nk=nk, mix=0.5, maxerror=1e-10)
hmf = scf.hamiltonian
```

```
print(hmf.get_goldstone_residual(nk=nk))          # 3e-10: zero to the SCF tolerance
qs,es = hmf.get_magnon_bands(method="tdhf",nk=nk,nq=90,n=2)
```

`qs` and `es` are flat arrays, as for "rpa", with `qs` the index along the q-path. Without `qpath`, the path on the honeycomb lattice is the line from (0,0) to (1,1) in units of the reciprocal lattice vectors, so it goes Γ , K, M, K', Γ , and its first point is next to Γ , not Γ itself. `es` is complex when the mean-field state is unstable against some excitation. `h.get_magnon_energies(Q=...)` gives the magnon energies at a single momentum.

The Goldstone theorem is the check. A magnetic state without spin-orbit coupling breaks the spin-rotation symmetry, so a uniform rotation of all the spins costs no energy, and there is a magnon at exactly zero energy at $Q = 0$. `h.get_goldstone_residual()` measures the distance from this. It is proportional to the tolerance of the mean-field calculation, about 3×10^{-10} for `maxerror=1e-10`, so check it before reading any dispersion. A small imaginary part of the same size on the acoustic branch is expected.

Three things matter:

- The mean field and the magnons must use the same `nk`. A mean field converged at `nk=20` is not self-consistent on a mesh of `nk=4`, and then the acoustic magnon gets a large wrong gap. The Hamiltonian does not keep the mesh it was converged on, so check with `get_goldstone_residual`.
- A metal needs `metal=True`. By default the same number of bands is taken as occupied at every k-point, which is wrong for a metal. With `metal=True` the occupied states are chosen at each k-point, which is what an itinerant magnet needs; for an insulator it changes nothing. In a metal the magnon is inside the Stoner continuum and is not the lowest mode, so use `by="weight"` to select the branches by their spin weight instead of by energy. Also, $E(q) = E(-q)$ only if the occupied states are symmetric under $k \rightarrow -k$ on the mesh, so choose a mesh with this symmetry.
- An isotropic exchange (`J1`, `J2`, `J3`, `Jr`) from the mean field has its transverse part and its Goldstone mode. A hand-built exchange matrix has only the Ising part and no record of its channels, so it raises `ValueError` (with it, the magnon would get a wrong gap of order J). An anisotropic exchange (`J1x`, `J1y`, `J1z` not equal, or `SzSz`, `SxSx`, `SySy`) breaks the spin-rotation symmetry, so its magnon gap is physical and there is no Goldstone mode to check; it also raises `ValueError`, and `check_su2=False` computes its spectrum.

For a collinear state only the spin-flip pairs are needed, which is exact and cheaper, and this is the default. For a canted or spiral state all the pairs are used automatically (`channel="all"` forces it). This includes the 120-degree spiral of the triangular-lattice Hubbard model, whose Goldstone mode is at zero to the tolerance of the mean field. For an onsite U , where both are exact, "tdhf" and "rpa" give the same magnons.

The Goldstone residual is 3×10^{-8} , so the dispersion can be read. The path starts next to Γ , not at it, and ends at Γ . There the lowest mode is at zero, the Goldstone mode, and the second one is at $0.91t$. At K the lowest mode is at $0.89t$, and at M the two modes are at $1.67t$ and $2.41t$. The lowest mode has the same energies as the lowest poles of the pair-basis map above, to $10^{-3}t$, as it should be. Both branches are smooth along the path with `nk=24`. On a coarse mesh, such as `nk=6`, they zigzag between neighboring Q , because the electron-hole pairs of the mesh at k and $k + Q$ change from one Q to the next. The lowest mode is converged: it moves by at most $0.015t$ from `nk=18` to 24, and by $0.005t$ at K from `nk=24` to 30. The second mode near K and K' is the bottom of the electron-hole continuum, not a second magnon, and it converges more slowly: at K it is at $1.09t$, $1.01t$ and $0.97t$ for `nk=18`, 24 and 30. Its limit is the bottom of the continuum at K, the smallest $E_c(\mathbf{k} + \mathbf{Q}) - E_v(\mathbf{k})$ of the mean-field bands, which is $0.91t$, the mean-field gap, since the valence band has its maximum at K and the conduction band its minimum at K'. On a finite mesh the lowest pair state sits above this edge and comes down to it only slowly as `nk` grows, so the second mode is a discrete level of the continuum and not a converged branch. The magnon at K, $0.89t$, is just below the edge. The figure takes about eight minutes.

The code of this figure is in `documentation/figures/spin_tdhf.py`.

See `examples/2d/magnon_bands_tdhf/main.py` for a runnable script, and `jupyter-notebooks/functionalities/interacti` for an executed notebook.

21.3 Charge response and plasmons

The same RPA can be done in the charge channel, with `h.get_densitychi_RPA()` and `h.get_plasmon_bands()`. These are the density-density versions of `get_spinchi_full` and `get_magnon_bands`. The collective modes are now plasmons, or the modes of a charge order. The interaction is a density-density interaction with an onsite U ,

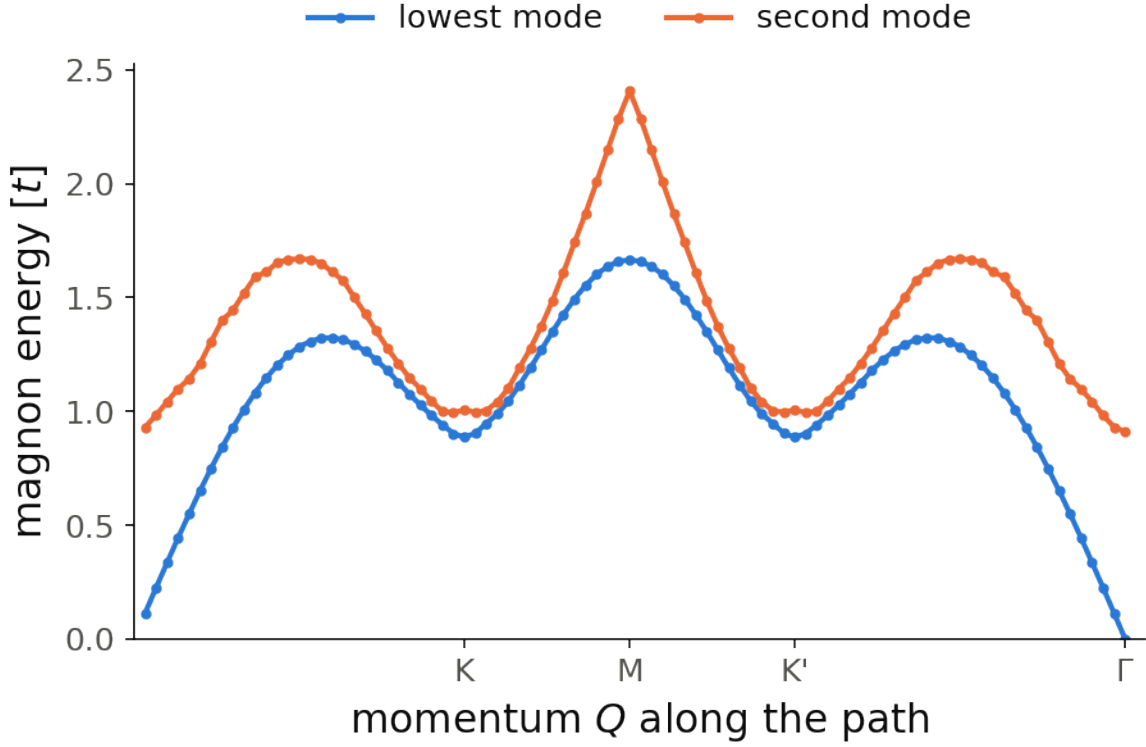


Figure 61: The two lowest magnon branches of the honeycomb antiferromagnet with U and V_1 , from time-dependent Hartree-Fock, along the q -path.

neighbor shells V_1 , V_2 , V_3 , and a general $V_r(r)$, as in `Vinteraction` and `VJinteraction`. Different from the spin channel, the interaction is given as keywords and is not read from `h.V`, so no mean-field calculation is needed: the bare response of the Hamiltonian you give is dressed with this interaction. This Hamiltonian can also be a converged mean-field one.

```
import numpy as np
from pyqula import geometry
g = geometry.chain()
h = g.get_hamiltonian(has_spin=True) # 1D chain at half filling
qs,ws,gammas = h.get_plasmon_bands(V1=0.6,qpath=[[0.3,0.,0.],[0.4,0.,0.],[0.5,0.,0.]],nq=3,
                                   energies=np.linspace(0.,1.,100),delta=2e-2,nk=2000)
```

`qs`, `ws` and `gammas` are as in `get_magnon_bands`. What matters here is the wavevector. At half filling the chain has perfect nesting at $q = 1/2$ ($k = \pi$), so the static charge response is large there. A repulsive V_1 gives $V(q) = 2V_1 \cos 2\pi q < 0$ at the zone boundary, so it pushes the charge channel toward an instability at $q = 1/2$, a charge-density wave. This is the charge version of the Stoner instability of the spin channel.

The map is not in the snippet: we call `h.get_densitychi_RPA()` with the same $V_1=0.6$ at 201 values of q with `nk=1000`, and plot it on a log color scale. With `nk=3000` the map is the same. The dashed lines are the edges of the free electron-hole continuum, $2t \sin 2\pi q$ and $4t \sin \pi q$. The circles are the poles of `get_plasmon_bands` at 50 values of q , with `energies` from 0 to $4t$. At small q , where $V(q) > 0$, the RPA moves the weight to the upper edge and a little above it: this is the plasmon. Its poles go from $0.28t$ at $q = 0.02$, just above the edge, to $2.49t$ at $q = 0.21$, where it enters the continuum and the poles stop. Their $|\gamma|$ is between 0.11 and 0.3 from $q = 0.04$ on, and 0.7 at $q = 0.02$, so the plasmon is damped most at the smallest q . Near $q = 1/2$, where $V(q) < 0$, the weight moves to the lower edge and below it, toward zero frequency: this is the soft charge mode of the charge-density wave. Its poles go from $1.26t$ at $q = 0.39$ down to $0.32t$ at $q = 0.45$, below the edge, and they get sharper as the frequency goes down, $|\gamma| = 0.01$ at $q = 0.45$. Closer to $q = 1/2$ there is no pole. At $q = 1/2$ the real part of the kernel eigenvalue changes sign at $0.60t$, but the eigenvalue stays at about 0.6 in absolute value, so the kernel

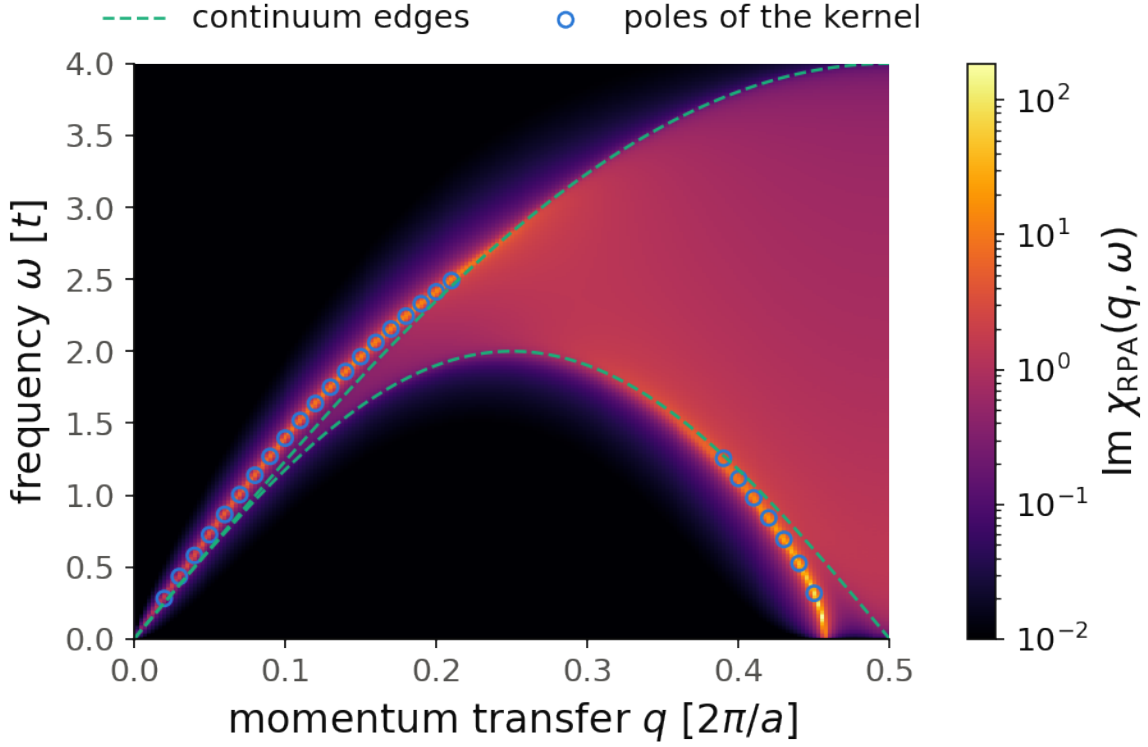


Figure 62: Imaginary part of the RPA charge response of the chain with a first-neighbor V1, over momentum transfer and frequency, with the edges of the electron-hole continuum.

never comes close to zero. The window of the snippet, `energies` from 0 to 1, misses the mode at $q = 0.4$, at $1.115t$, just below the edge of the continuum at $1.18t$; with `energies` up to $2t$ the same call finds it. The figure takes about three and a half minutes.

The code of this figure is in `documentation/figures/spin_plasmon.py`.

22 Excitons

When a semiconductor absorbs a photon, an electron goes to the conduction band and leaves a hole in the valence band. The electron and the hole attract each other, and they can form a bound state, the exciton. Its energy is below the band gap by the binding energy, so in an optical absorption measurement it is a peak below the absorption edge. A single-particle band structure cannot give this peak. We compute it here by solving the Bethe-Salpeter equation (BSE), the eigenvalue problem of an electron-hole pair, with `h.get_bse()` and `h.get_exciton_energies()`.

22.1 Excitons and the Bethe-Salpeter equation

An exciton with center-of-mass momentum Q is a superposition of electron-hole pairs of the mean-field ground state $|MF\rangle$,

$$|X\rangle_Q = \sum_{v,c,k} A_{vc}(k) c_{c,k+Q}^\dagger c_{v,k} |MF\rangle,$$

where v runs over the valence bands, c over the conduction bands, and k over a k-mesh of N points. The amplitudes $A_{vc}(k)$ are the wavefunction of the exciton in momentum space. They are the eigenvectors of the BSE Hamiltonian

$$H_{vck,v'c'k'}^{\text{BSE}} = (\epsilon_{c,k+Q} - \epsilon_{v,k}) \delta_{vv'} \delta_{cc'} \delta_{kk'} + \frac{1}{N} (X_{vck,v'c'k'} - D_{vck,v'c'k'}).$$

The first term is the energy of a free electron-hole pair. The kernel has two terms. With $C_a^{n,k}$ the coefficient of band n at k on orbital a , the direct term is

$$D = \sum_{ab} C_a^{c,k+Q} C_a^{c',k'+Q} W_{ab}(k-k') C_b^{v,k} C_b^{v',k'}^*,$$

the attraction between the electron and the hole through the interaction W . It binds the exciton. The exchange term is

$$X = \sum_{ab} C_a^{c,k+Q} C_a^{v,k} V_{ab}(Q) C_b^{c',k'+Q} C_b^{v',k'}^*,$$

the interaction between the charge densities of the two pairs. It is repulsive, it splits singlet and triplet excitons, and alone it gives the RPA. The orbitals are treated as point-like, as in the localized-orbital BSE of the Xatu code, [arXiv:2307.01572](https://arxiv.org/abs/2307.01572). By default pyqula solves the full problem, which also couples the absorption pairs to the emission pairs (beyond the Tamm-Dancoff approximation).

By default the interaction is the one of a converged mean-field Hamiltonian, `h.V`. So the same interaction gives the Fock self-energy in `h` and the BSE kernel, and nothing is counted twice: this is time-dependent Hartree-Fock on top of Hartree-Fock. As an example, we converge a mean field on the gapped honeycomb lattice and compute its lowest excitons. Note that with these parameters the converged state is not magnetic: `hmf.get_vev("sz")` is of order 10^{-5} , so it is the band insulator of the sublattice imbalance. At this imbalance, 0.6, and with `nk=6`, the staggered moment appears only at about $U = 3.2$, where it is 0.06, and it is 0.46 at $U = 3.3$:

```
from pyqula import geometry
g = geometry.honeycomb_lattice()
h = g.get_hamiltonian(has_spin=True)
h.add_sublattice_imbalance(0.6)
hmf = h.get_mean_field_hamiltonian(U=1.5, filling=0.5, mf="antiferro", nk=6) # converge a mean field
es = hmf.get_exciton_energies(nk=6, n=8) # the eight lowest excitons
```

`es` contains the eight lowest exciton energies at $Q = 0$.

An onsite Hubbard U has a too short range to bind a Wannier-Mott exciton in two dimensions, so a realistic calculation needs a long-range interaction. `bsetk.interaction.density_interaction` builds a density-density interaction with the same parameters as the mean-field calculation (`U`, `V1`, `V2`, `V3`, or a function `Vr` of the two positions), and it is given to every exciton method with `V=`. Here it is a Coulomb tail $V(r) = 0.8/\sqrt{r^2 + 0.25}$ on a gapped honeycomb lattice:

```
import numpy as np
from pyqula.bsetk.interaction import density_interaction
h2 = g.get_hamiltonian()
h2.add_sublattice_imbalance(1.0) # a gapped semiconductor
W = density_interaction(h2, Vr=lambda r1,r2: 0.8/np.sqrt((r1-r2).dot(r1-r2)+0.25)) # Coulomb tail
bse = h2.get_bse(V=W, nk=21)
print(bse.get_energies()[0], bse.get_binding_energies()[0]) # lowest exciton, and how far below the gap
```

The binding energy is the lowest free electron-hole transition on the mesh minus the exciton energy, so a positive value means a bound exciton. With the Coulomb tail the lowest exciton separates from the band edge.

The figure is for the second snippet, with the Coulomb tail. On the left, each line is one energy and its height the number of states there. With `nk=21` the mesh contains K and K' , because 21 is a multiple of 3, so the lowest free pair is at the true gap $2t$, with eight states: K and K' , times four spin states. The next free pairs are at $2.087t$ (48 states) and $2.207t$ (24 states). The lowest exciton is at $1.8445t$, so its binding energy is $0.1555t$. It has eight states, two valleys times four spin states. The next excitons are at $2.015t$ (eight states), $2.016t$ (one state)

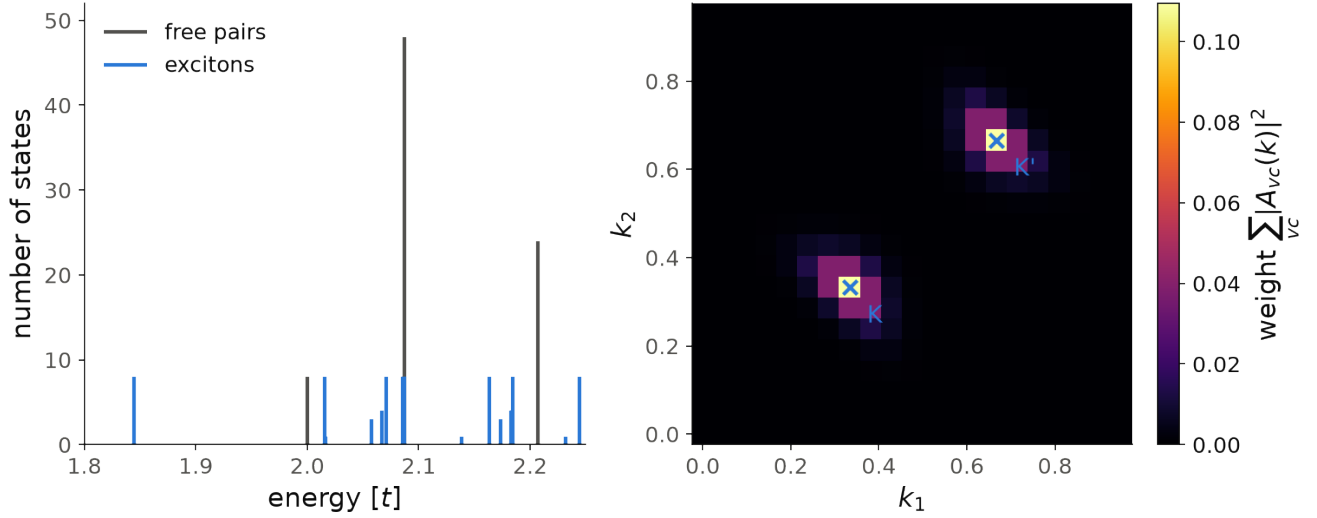


Figure 63: Left: exciton levels and free electron-hole pair levels of the gapped honeycomb lattice with a Coulomb tail. Right: weight of the lowest exciton over the Brillouin zone.

and $2.058t$ (three states), already above the gap. On the right is the weight $\sum_{vc} |A_{vc}(k)|^2$ of the lowest exciton on the k -mesh, averaged over its eight states, with k_1, k_2 in units of the reciprocal lattice vectors. The weight is a peak at K and K' , where the gap is smallest, and it falls to almost zero within three or four points of the mesh. The binding energy is converged: it is $0.1558t$ with $\mathbf{nk}=18$, and the Large k -meshes section below finds $0.1553t$ on meshes up to 30×30 in the Tamm-Dancoff approximation. A coarse mesh such as $\mathbf{nk}=8$, which does not contain K , gives $0.309t$, measured from its own band edge at $2.165t$. The calculation takes about a minute.

The code of this figure is in `documentation/figures/excitons_bse.py`.

- `h.get_exciton_energies()`: the exciton energies.
- `h.get_exciton_binding_energies()`: the binding energies (positive means bound).
- `h.get_exciton_states()`: the energies and the amplitudes $A_{vc}(k)$.
- `h.get_bse()`: the solved problem; its `pairs` attribute gives the k -point and the two bands of each pair, to resolve the amplitudes in momentum or by band.
- `Q=[qx,qy,qz]`: excitons with finite center-of-mass momentum.
- `nv, nc`: keep only the `nv` highest valence and the `nc` lowest conduction bands.
- `kernel`: "full" (default) uses both terms; "direct" only the attraction, so no singlet-triplet splitting; "exchange" only the exchange, which is exactly the RPA; "none" no interaction, so the energies are the free transitions.
- `tda=True`: the Tamm-Dancoff approximation, which keeps only the absorption pairs. The matrix is four times smaller and Hermitian, and it is a good approximation at weak coupling.

The Hamiltonian is spinful, so the spin is part of the orbital index, and singlet and triplet excitons come from the same calculation. In a system with spin-rotation symmetry each free electron-hole transition is four times degenerate in spin. The full kernel can split it into a triplet, three times degenerate, and a singlet; neither of the two terms alone gives this 3+1 pattern. Not every level splits, and the singlet is not always the higher one. For the gapped honeycomb lattice with the Coulomb tail above, the lowest exciton stays eight times degenerate at $1.8445t$, two valleys times four spin states. The 3+1 pattern appears higher up, with a singlet at $2.016t$ below a triplet at $2.058t$. With only an onsite term, `density_interaction(h2,U=1.0)` at $\mathbf{nk}=6$, the singlet is above the triplet, at $2.917t$ against $2.704t$.

The default `V=None` reads `h.V`, which is only the z channel of an exchange interaction. It misses the x and y channels of an anisotropic exchange (`J1x`, `J1y` together with `J1z`), and it is zero after `SxSx/SySy`. In these cases build the interaction yourself and give it with `V=`.

The reference state must have a gap: a metal has no well-defined electron-hole pairs, and it is refused. Nambu (BdG) Hamiltonians are refused too. The problem has $N_{\text{pair}} = n_v n_c N$ pairs and a dense matrix of size $2N_{\text{pair}}$,

so the k-mesh sets the cost. A calculation that needs more memory than `max_memory` (default 2 GB) is refused before it starts.

See `jupyter-notebooks/functionalities/interacting_mean_field_hamiltonians/17_excitons_bse.ipynb` for an executed notebook.

22.1.1 Large k-meshes

The k-mesh that an exciton needs depends on how strongly it is bound. A strongly bound exciton is small in real space, so it is spread over the Brillouin zone, and a coarse mesh is enough. A weakly bound Wannier-Mott exciton is large in real space, so its amplitude $A(k)$ is a sharp peak at the band edge, and only a fine mesh resolves it. `solver=` chooses how to reach a fine mesh:

```
import numpy as np
from pyqula import geometry
from pyqula.bsetk.interaction import density_interaction
g = geometry.honeycomb_lattice()
h = g.get_hamiltonian()
h.add_sublattice_imbalance(1.0) # a gapped semiconductor
W = density_interaction(h, Vr=lambda r1,r2: 0.8/np.sqrt((r1-r2).dot(r1-r2)+0.25))
h.get_bse(V=W,nk=1024,tda=True,solver="iterative",neig=4)
h.get_bse(V=W,nk=65536,tda=True,solver="qtt",neig=1)
```

- `solver="dense"` (default): the whole exciton spectrum. The only one that works without the Tamm-Dancoff approximation. Good up to a few thousand pairs.
- `solver="iterative"`: the same energies on much finer meshes, but only the `neig` lowest excitons.
- `solver="qtt"`: for the very fine meshes of a weakly bound exciton. Only the lowest exciton.

"iterative" and "qtt" need `tda=True`, which is accurate for the weakly coupled systems where bound excitons appear. "qtt" works best on a fine mesh with few bands, which is the case of a Wannier-Mott exciton, and it is the best choice in 1D. In 2D use it on a primitive cell with small `nv` and `nc`; on a supercell, "iterative" is better. It needs `nk` to be a power of two, and it cannot take an RPA-screened interaction directly; give it one with `ScreenedInteraction.get_dict()`.

See `examples/1d/exciton_qtt/main.py` and `examples/2d/exciton_qtt/main.py` for runnable comparisons of the three solvers, the second one plotting $|A(k)|^2$ over the Brillouin zone, and `examples/2d/excitons_bse/main.py` for the lowest exciton of a gapped honeycomb lattice as the Coulomb tail grows. See `jupyter-notebooks/functionalities/interacting_mean_field_hamiltonians/17_excitons_bse.ipynb` for an executed notebook.

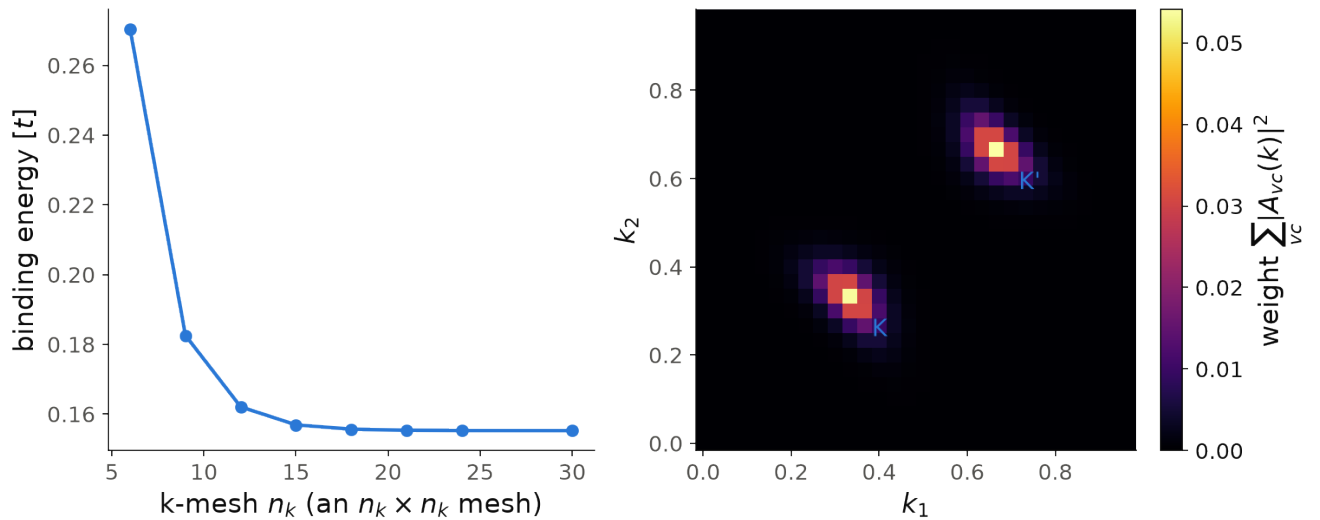


Figure 64: Left: binding energy of the lowest exciton of the gapped honeycomb lattice against the k-mesh. Right: weight of the lowest exciton over the Brillouin zone on the 30x30 mesh.

The figure uses the first solver of the snippet, `solver="iterative"`, on meshes from 6x6 to 30x30 instead of `nk=1024`, so that it runs in about three minutes. Every mesh is a multiple of 3, so it contains K and K', the lowest free pair is at the true gap $2t$, and the binding energy is measured from it. On the left, the binding energy is $0.270t$ on the 6x6 mesh, $0.162t$ on 12x12 and $0.1553t$ on 30x30. From 21x21 on it changes by less than $10^{-4}t$. On a coarse mesh the attraction is sampled on too few points, and the binding is overestimated. On the right is the weight $\sum_{vc} |A_{vc}(k)|^2$ of the lowest exciton on the 30x30 mesh, averaged over the four states that the solver returns. It is a peak at K and K', the same in the two valleys, and it is close to zero at about 0.15 reciprocal lattice vectors from them, four or five points of the mesh. This exciton is strongly bound, so about 20x20 is enough. A weakly bound exciton has a narrower peak, and it needs the much finer meshes of the snippet.

The code of this figure is in `documentation/figures/excitons_largek.py`.

22.1.2 Exciton band structure

An exciton moves through the crystal with center-of-mass momentum Q , so it has its own dispersion $E_X(Q)$, the exciton band structure. It is not the difference of two band energies, because the electron-hole interaction changes it. Its curvature gives the effective mass of the exciton, and a flat exciton band means a strongly bound, compact exciton. `h.get_exciton_bands()` solves one BSE for each Q on a path, and returns the result in the same form as `h.get_bands()`:

```
# h2 and W as in the section above
import numpy as np
from pyqula import geometry
from pyqula.bsetk.interaction import density_interaction
g = geometry.honeycomb_lattice()
h2 = g.get_hamiltonian()
h2.add_sublattice_imbalance(1.0) # a gapped semiconductor
W = density_interaction(h2, Vr=lambda r1,r2: 0.8/np.sqrt((r1-r2).dot(r1-r2)+0.25))
# nv=nc=2 keeps both members of each spin-degenerate pair
opts = dict(V=W, nq=20, nk=8, nv=2, nc=2, n=4)
qs, es = h2.get_exciton_bands(**opts) # the four lowest excitons along the path
qs0, es0 = h2.get_exciton_bands(kernel="none", **opts) # the bare continuum
```

`qs` is the index of the q-point along the path and `es` the exciton energy, both flat arrays for a scatter plot. `n` keeps the `n` lowest excitons at each q-point, and the other arguments go to `h.get_bse()`. `qpath` takes the same input as in `h.get_bands()`, and Q does not have to be on the k-mesh. The second call, with `kernel="none"`, gives the bottom of the free electron-hole continuum. The exciton band runs below this continuum along the whole path, and the distance between them is the binding energy at each Q . If the mean-field state is unstable at some Q , `es` is complex there. Each q-point is one full BSE, so small `nv`, `nc` and `tda=True` make a long path affordable.

The window `nv`, `nc` must not cut a degenerate pair of bands in half. In a spinful Hamiltonian without spin-orbit coupling or magnetic order, every band is two times degenerate. With `nv=1` the calculation keeps an arbitrary state of the pair, and the exciton energies become arbitrary too: in a time-reversal symmetric model $E_X(Q)$ is then not even in Q . `pyqula` gives a warning in this case. Use even `nv` and `nc` for a spin-degenerate Hamiltonian.

The default path goes from next to Γ through K, M and K' back to Γ . The excitons (blue) are below the free pairs (gray) at every Q . At Γ the lowest exciton is at $1.856t$ and the lowest free pair at $2.165t$; at M they are at $2.42t$ and $2.83t$. So the binding energy is between $0.3t$ and $0.4t$ along the whole path. Near K and K' the exciton band has a second minimum, $1.864t$, almost as low as at Γ : this is the exciton made of an electron in one valley and a hole in the other. Each blue point is close to a second one, the members of the spin multiplets.

The code of this figure is in `documentation/figures/excitons_bands.py`.

See `examples/2d/exciton_bands/main.py` for a runnable script, with the exciton band of a gapped honeycomb lattice below its electron-hole continuum.

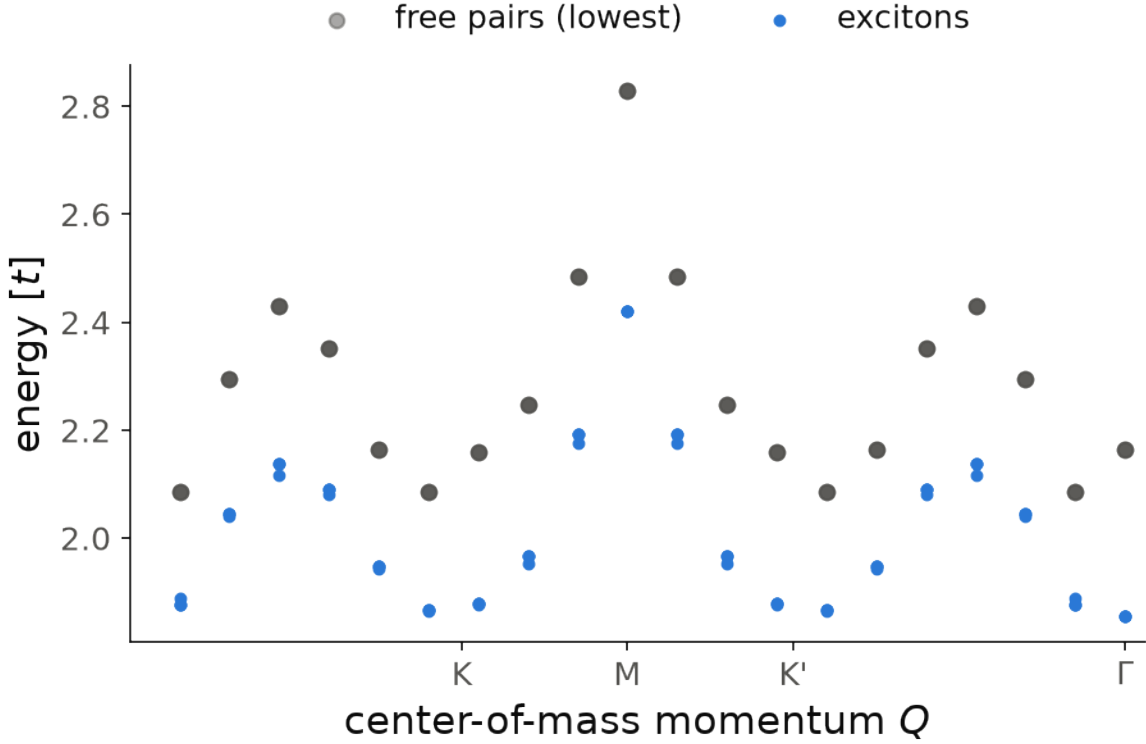


Figure 65: Exciton band structure of the gapped honeycomb lattice along the q -path, below the lowest free electron-hole pairs.

22.1.3 The screened interaction

So far the interaction in the kernel is the bare one. But an electron and a hole in a solid do not feel the bare interaction: the other electrons move around them and screen it. The mean-field calculation gives the bands ϵ_{nk} and the coefficients $C_a^{n,k}$ on a k -mesh, so the static RPA screening can be computed from them. The bare polarizability is

$$\chi_{ab}^0(q) = \frac{1}{N} \sum_k \sum_{n,m} (f_{nk} - f_{m,k+q}) \frac{\rho_a^{nm}(k, q) \rho_b^{nm*}(k, q)}{\epsilon_{nk} - \epsilon_{m,k+q}}, \quad \rho_a^{nm}(k, q) = C_a^{n,k*} C_a^{m,k+q},$$

where f is the occupation and a, b run over the spin-orbitals of the unit cell. The dielectric function and the screened interaction are

$$\epsilon(q) = 1 - v(q)\chi^0(q), \quad W(q) = \epsilon^{-1}(q) v(q),$$

with $v(q)$ the bare interaction. `screening="rpa"` puts W in the direct term:

```
import numpy as np
from pyqula import geometry
from pyqula.bsetk.interaction import density_interaction

h = geometry.honeycomb_lattice().get_hamiltonian()
h.add_sublattice_imbalance(1.0) # a gapped semiconductor

# a BARE interaction: an onsite term plus a soft-cutoff Coulomb tail
coulomb = lambda r1,r2: 0.6/np.sqrt((r1-r2).dot(r1-r2)+0.25)
V = density_interaction(h,U=1.0,Vr=coulomb)
```

```
bare = h.get_bse(V=V,nk=8) # time-dependent Hartree-Fock
screened = h.get_bse(V=V,nk=8,screening="rpa") # GW-BSE style
```

Compare the binding energies of `bare` and `screened`. With screening the binding is weaker, and the exciton is closer to the band edge, because the other electrons partly cancel the attraction between the electron and the hole.

The exchange term keeps the bare interaction for any `screening`. This is the usual GW-BSE choice: screening the exchange term too would count the same polarization bubbles twice. So `kernel="exchange"` does not change with `screening`, and `pyqula` gives a warning if both are set.

Do not screen a fitted Hubbard U . A U chosen to describe a material is already a screened interaction, and screening it again makes it too weak. Screening is for a bare interaction: a Coulomb tail from `density_interaction(Vr=...)`, or bare V_1 , V_2 , V_3 . The dangerous case is the default $V=h.V$ of a Hubbard calculation.

`screening="crpa"` is the constrained RPA ([arXiv:0710.4013](https://arxiv.org/abs/0710.4013)): the transitions inside the `nv`, `nc` window are left out of the polarizability. Use it when the bands of the window are a downfolded model that will be solved exactly later. It screens less than the full RPA. It needs a real subset of the bands: with the default `nv=nc=None` the window is the whole spectrum, nothing is left to screen, and the call is refused.

The screened interaction is also available alone, as a `ScreenedInteraction`:

```
W = h.get_screened_interaction(V=V,nk=8)
print(W.epsmin) # smallest dielectric eigenvalue over the mesh
Wq = W.at(W.qs[3]) # the screened interaction at one q-point
d = W.get_dict() # ... and back in real space, usable at any q
```

$W(q)$ exists only on the q -points of the mesh where it was computed, which are the ones the kernel needs; asking for another q raises an error. `W.get_dict()` gives W in real space, which can be evaluated at any q , used to see how far the screened interaction reaches, or given to `h.get_mean_field_hamiltonian(V=...)` for a screened-exchange mean field. `nkW` sets a finer mesh for the screening than for the exciton, and must be an integer multiple of `nk`.

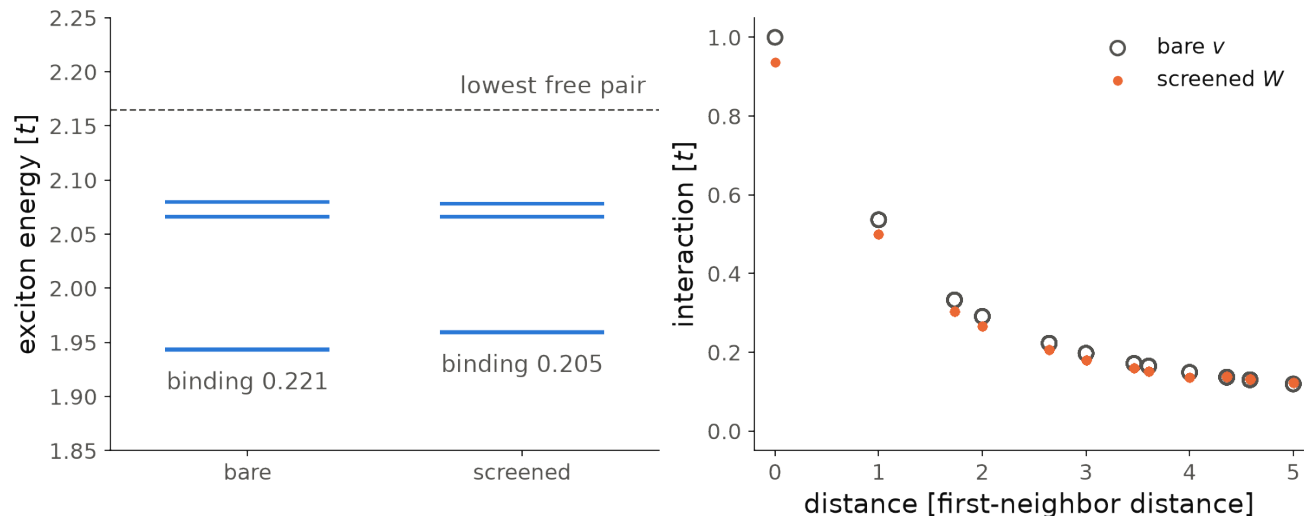


Figure 66: Left: exciton levels with the bare and with the screened interaction. Right: bare and screened interaction between opposite spins against distance.

The figure uses the two snippets above. On the left, the lowest exciton moves up from $1.943t$ to $1.959t$ with screening, so the binding energy goes from $0.221t$ to $0.205t$. The higher excitons hardly move. On the right we take the interaction between opposite spins from V and from `W.get_dict()`, against the distance between the two sites; the first-neighbor distance is 1. The onsite term goes from 1.0 to 0.94, the first-neighbor term from

0.537 to 0.50, and the reduction becomes smaller at larger distance. So the screening is weak, a few percent, because the gap is large. `W.epsmin` is 0.90, below 1: along one direction the interaction becomes a little stronger, the case explained at the end of this section. The tail of V stops at distance 5, and W from an 8x8 mesh repeats every 8 unit cells, so we do not plot it further.

The code of this figure is in `documentation/figures/excitons_screening.py`.

Screening acts on the charge. The total density polarizes the medium, and the induced charge acts back on the total density. So with `channel="charge"` (default) the dielectric matrix is built on the sites,

$$\varepsilon_{ij}(q) = \delta_{ij} - \sum_k v_{ik}^c(q) \chi_{kj}^c(q), \quad \chi_{ij}^c = \sum_{\sigma\sigma'} \chi_{(i\sigma)(j\sigma')}^0, \quad v_{ij}^c = \frac{1}{4} \sum_{\sigma\sigma'} v_{(i\sigma)(j\sigma')},$$

and the correction, which does not depend on spin, is added to the bare interaction,

$$W_{(i\sigma)(j\sigma')} = v_{(i\sigma)(j\sigma')} + [v^c \chi v^c]_{ij}, \quad \chi = \chi^c (1 - v^c \chi^c)^{-1}.$$

Note one convention: between two sites $v_{ij}^c = V_{ij}$, but on the same site $v_{ii}^c = U/2$ and not U , because a Hubbard term couples only opposite spins, so only half of the density of a site acts on an electron there.

In the charge channel, spin-rotation symmetry is kept exactly. The other choice, `channel="orbital"`, uses $W = \varepsilon^{-1}v$ on the full spin-orbital matrix. It screens charge and spin with the same kernel, so it breaks spin-rotation symmetry, and an exciton multiplet that should be degenerate splits. Use the default. For a spinless Hamiltonian the two are the same.

Screening does not always make the interaction weaker. It does when the charge interaction $v^c(q)$ is positive definite at every q , as a physical repulsion is: the static charge response is negative, so $\varepsilon = 1 - v^c \chi^c$ only reduces it. Without an onsite term, the interaction matrix has zero trace and has negative eigenvalues, and along them screening makes the interaction stronger. So include a realistic U . $U > 0$ is not enough: the eigenvalues of $v^c(q)$ are $U/2$ plus those of the Fourier transform of the intersite interaction, which averages to zero over the Brillouin zone and so is negative somewhere. For example, on the square lattice a first-neighbor V_1 gives $U/2 - 4V_1$ at the corner of the zone, and $U/2$ has to be larger than the most negative one.

If an eigenvalue of $\varepsilon(q)$ reaches zero, the RPA diverges. This is a charge or spin instability of the mean-field state at this q , the same condition $1 - V\chi = 0$ that `h.get_rpa_kernel_poles()` gives as a collective mode (see Spin response and magnons), and the call raises an error.

- **screening**: the interaction of the direct term. `None` for the bare one, `"rpa"` for the screened W , `"crpa"` for the constrained RPA.
- **channel**: where the dielectric matrix is built. Keep `"charge"`, which keeps spin-rotation symmetry.
- **nkW**: the k-mesh of the screening, an integer multiple of `nk`. Increase it when the screening needs more k-points than the exciton.

See `examples/2d/screened_bse/main.py` for a runnable script, and `jupyter-notebooks/functionalities/interacting_mea` for an executed notebook.

23 Quantum transport

In this page we compute the conductance of junctions. We start with two normal leads, then a normal lead and a superconductor, and we end with the current through a junction between two superconductors at finite voltage.

A transport calculation has a scattering region between two semi-infinite leads. Each lead is a periodic Hamiltonian, of the kind built in `Building a Hamiltonian`. The leads enter through their self-energies $\Sigma_L(E)$ and $\Sigma_R(E)$, computed from the surface Green's function of each lead, and the Green's function of the scattering region is

$$G(E) = [E + i\delta - H_C - \Sigma_L(E) - \Sigma_R(E)]^{-1}.$$

For normal leads, the zero-temperature conductance is the Landauer formula,

$$G = \frac{e^2}{h} T(E), \quad T(E) = \text{Tr} [\Gamma_L G \Gamma_R G^\dagger], \quad \Gamma_{L,R} = i (\Sigma_{L,R} - \Sigma_{L,R}^\dagger),$$

where $T(E)$ is the transmission, the sum of the transmission probabilities of all the channels. `pyqula` gives conductances in units of e^2/h , and counts the two spins as two channels, so a perfect spinful chain has $G = 2$. This is the differential conductance dI/dV at bias $V = E/e$ that a two-terminal measurement gives.

When one lead is superconducting, the Hamiltonian is written in the Nambu basis (see Superconductivity), and the conductance is computed from the scattering matrix with the Blonder-Tinkham-Klapwijk formula,

$$G(E) = \frac{e^2}{h} [N - R_{ee}(E) + R_{he}(E)],$$

where N is the number of channels of the normal lead, $R_{ee} = \text{Tr} r_{ee}^\dagger r_{ee}$ is the probability that an electron is reflected as an electron, and $R_{he} = \text{Tr} r_{he}^\dagger r_{he}$ the probability that it is reflected as a hole. The second process is Andreev reflection: an electron from the normal side is reflected as a hole, and a Cooper pair enters the superconductor. It transfers a charge $2e$, so it adds to the conductance.

23.1 Magnetoresistance in metal-metal transport

We now compute the conductance of a spin valve, a junction between two magnetic metals, with `HT.didv()`. We make two copies of the same chain and add an exchange field J to each of them. The first lead is magnetized along z . In the second lead we rotate the magnetization in the xz plane, by an angle θ from the first one, so $\theta = 0$ is the parallel and $\theta = \pi$ the antiparallel configuration:

```
from pyqula import geometry
from pyqula import heterostructures
import numpy as np
g = geometry.chain() # create the geometry
h = g.get_hamiltonian() # create the Hamiltonian
h.add_onsite(1.0) # the band is now [-1,3], and E=0 is close to its bottom
thetas = np.linspace(0., np.pi, 61) # angles between the two magnetizations
def conductance(J, theta): # conductance at E=0 for exchange J and angle theta
    h1 = h.copy() ; h1.add_exchange([0., 0., J]) # first lead, magnetization along z
    h2 = h.copy() ; h2.add_exchange([J*np.sin(theta), 0., J*np.cos(theta)]) # rotated in the xz plane
    HT = heterostructures.create_leads_and_central(h1, h2, h1) # create the junction
    return HT.didv(energy=0.0) # calculate conductance
Js = [0.3, 0.6, 0.9, 1.5] # exchange strengths, the last one is a half metal
Gs = {J: np.array([conductance(J, t) for t in thetas]) for J in Js}
```

`Gs[J]` is the conductance $G(\theta)$ at $E = 0$ for each angle, in units of e^2/h . The onsite energy moves the Fermi energy close to the bottom of the band, so that a moderate exchange field changes the bands a lot. The exchange field splits the band into a majority band, $[-1 - J, 3 - J]$, and a minority band, $[-1 + J, 3 + J]$. For $J < 1$ both cross the Fermi energy, and the chain is a magnetic metal. For $J > 1$ the minority band is above the Fermi energy, and only one spin is a metal: this is a half metal.

In the parallel configuration each spin sees the same band on both sides, so it is transmitted perfectly. Then $G(0)$ is 2 for $J < 1$, one per spin, and 1 for a half metal. When the second magnetization rotates, an electron in the majority band of the first lead has to enter the majority band of the second one, with its spin along the new direction. The overlap of the two spinors is $\cos(\theta/2)$, so the transmission goes down with the angle. The magnetoresistance is

$$\text{MR} = \frac{G_P - G_{AP}}{G_{AP}},$$

with $G_P = G(0)$ and $G_{AP} = G(\pi)$. This is what a spin valve measures: its resistance depends on the angle between the two magnetizations.

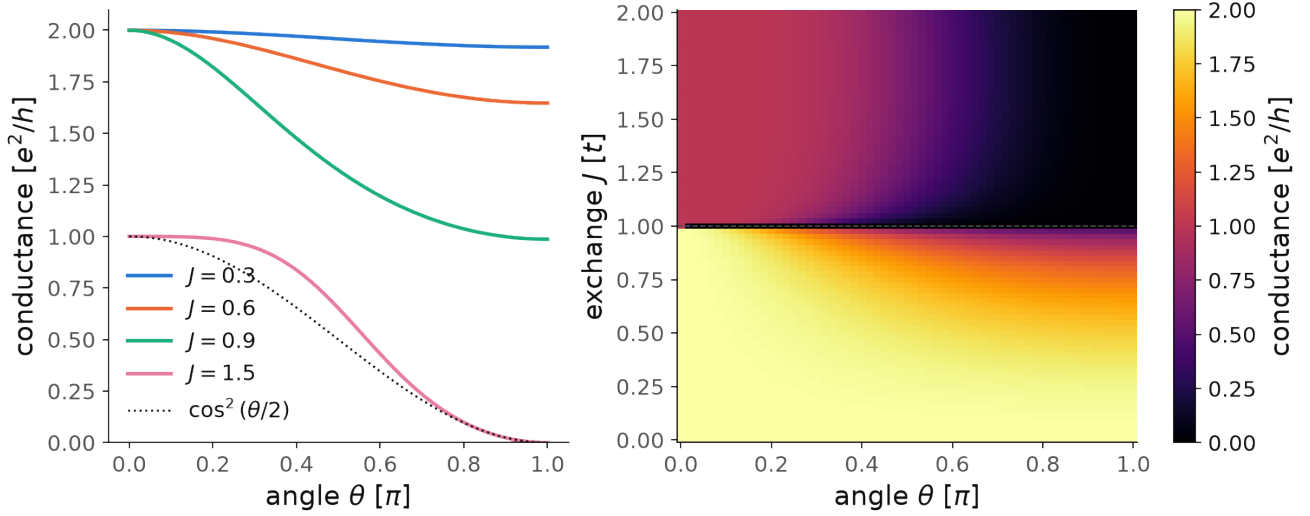


Figure 67: Conductance of a spin valve against the angle between the magnetizations of the two leads, for several exchange fields, and as a map over angle and exchange field.

On the left, every curve starts at its parallel value and goes down smoothly to its minimum at $\theta = \pi$. For $J = 0.3$ the antiparallel conductance is 1.92, so the magnetoresistance is only 4%, because the two spin bands are almost the same. It grows with the exchange field: G_{AP} is 1.65 for $J = 0.6$ and 0.99 for $J = 0.9$, where the minority band is close to leaving the Fermi energy, and the magnetoresistance is 103%. For the half metal, $J = 1.5$, the conductance goes from 1 at $\theta = 0$ to exactly zero at $\theta = \pi$, so the magnetoresistance is infinite. An electron of the first lead has no state with its spin in the second lead, and the minority band of the second lead is above the Fermi energy. The curve is close to $\cos^2(\theta/2)$ (dotted), the square of the overlap of the two spinors, but it is flatter near $\theta = 0$. The overlap alone is the answer of a tunnel contact, and this junction is a transparent one, so it does not have to follow it. The map on the right shows the conductance for all the exchange fields from 0 to 2. The dashed line, $J = 1$, separates the magnetic metal (below), where $G(0) = 2$, from the half metal (above), where $G(0) = 1$ and $G(\pi) = 0$.

The code of this figure is in `documentation/figures/transport_magnetoresistance.py`.

See `jupyter-notebooks/functionalities/quantum_transport/01_metal_metal_transport.ipynb` for an executed notebook.

23.2 Superconductor-metal transport

We now compute the conductance between a normal metal and a superconductor, with `heterostructures.build()` and `HT.set_coupling()`. There are two limits. At a transparent contact, an electron with energy inside the gap cannot enter the superconductor as a quasiparticle, so it is Andreev reflected as a hole with probability one, and the conductance below the gap is twice the normal one. At a tunneling contact Andreev reflection is suppressed, because it needs two electrons to tunnel. Then the conductance follows the density of states of the superconductor: zero inside the gap, and coherence peaks at $\pm\Delta$.

`HT.set_coupling(T)` multiplies the hopping between the two sides of the junction by T , so $T = 1$ is a perfect contact and $T \ll 1$ is a tunnel contact. The loop goes from one limit to the other:

```
from pyqula import geometry
from pyqula import heterostructures
import numpy as np
g = geometry.chain() # create the geometry
h = g.get_hamiltonian() # create the Hamiltonian
h1 = h.copy() # first lead
h2 = h.copy() # second lead
h2.add_swave(.01) # the second lead is superconducting
```

```

es = np.linspace(-.03,.03,100) # set of energies for dIdV
for T in np.linspace(1e-3,1.0,6): # loop over transparencies
    HT = heterostructures.build(h1,h2) # create the junction
    HT.set_coupling(T) # set the coupling between the leads
    Gs = [HT.didv(energy=e) for e in es] # calculate conductance

```

G_s is the conductance at each energy, for the last transparency of the loop. Compare it with its value outside the gap, $|E| > \Delta = 0.01$. At $T=1.0$ the conductance inside the gap is twice the one outside. At the smallest transparencies it goes to zero inside the gap and has peaks at the gap edges. This is the spectroscopy of a normal STM tip on a superconductor, and the transparency is what the experiment changes with the distance between the tip and the sample.

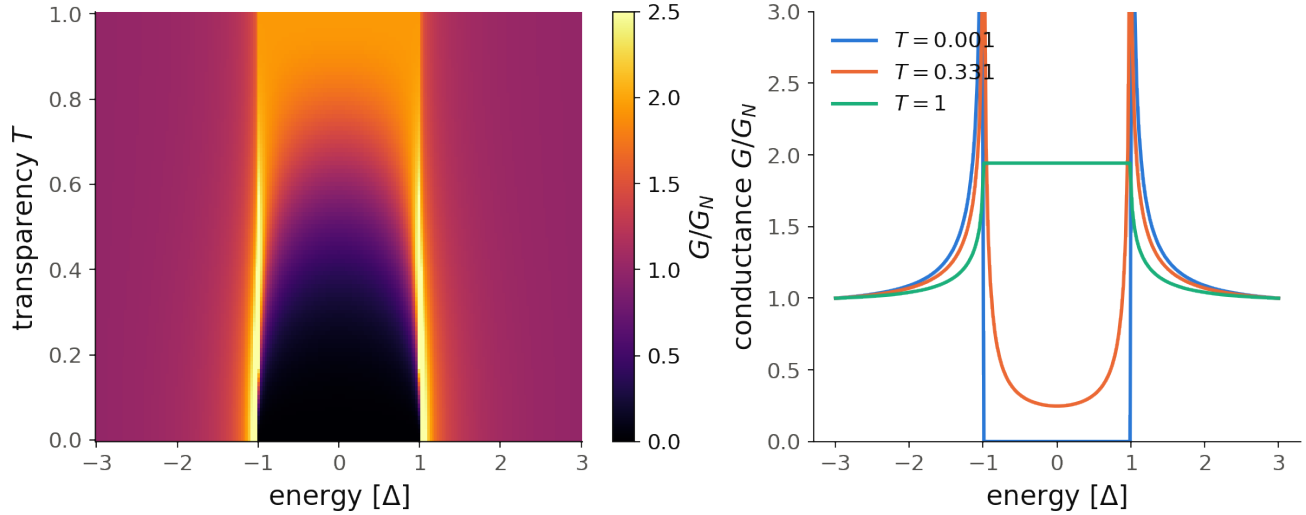


Figure 68: Conductance of a normal-superconductor junction against energy and transparency, normalized to its value outside the gap.

For the map we take 101 transparencies instead of 6, in steps of 0.01, and 201 energies, and we divide each curve by its value at $E = -3\Delta$, outside the gap. With 16 transparencies the map had visible steps along T . At $T = 1$ (top of the map, green curve) the conductance inside the gap is 1.94 times the one outside, close to the doubling of perfect Andreev reflection. When T goes down, the inside of the gap becomes dark: at $T = 0.331$ the conductance at $E = 0$ is 0.25, and at $T = 0.001$ it is 2×10^{-6} . At the same time the coherence peaks at $E = \pm\Delta$ appear, so the small- T curve (blue) is the BCS density of states. The color scale is cut at 2.5, so the peaks are saturated.

The code of this figure is in `documentation/figures/transport_andreev.py`.

See `examples/transport/andreev_reflection/main.py` for a runnable script, and `jupyter-notebooks/functionalities/q` for an executed notebook.

23.3 Transport through an arbitrary finite region

In the two examples above, the scattering region is made of copies of the unit cell of the leads. We now use any finite (0d) Hamiltonian as the scattering region, for example a molecule or a quantum dot, with two semi-infinite chains attached at sites i and j , with `h.get_central_heterostructure(i,j,left=None,right=None)`:

```

from pyqula import geometry

g = geometry.chain()
gc = g.get_supercell(5)
gc.dimensionality = 0 # a finite, 5-site cluster (no periodicity)
hc = gc.get_hamiltonian() # the central region, can be any 0d Hamiltonian

```

```
h_normal = g.get_hamiltonian() # normal lead
h_sc = g.get_hamiltonian(); h_sc.add_swave(0.05) # superconducting lead
```

```
ht = hc.get_central_heterostructure(0,4,left=h_normal,right=h_sc)
G = ht.didv(energy=0.02) # Andreev conductance, via the BdG scattering-matrix formula
```

G is the conductance at $E = 0.02$, inside the gap $\Delta = 0.05$ of the right lead, so it comes only from Andreev reflection. The result is a normal junction object, so `didv`, `get_dos` and `get_kappa` all work on it.

- `left`, `right`: the two leads. By default each is a spinless chain.
- `j`: the site of the second contact. By default the last site.
- At most one of `hc`, `left` and `right` can be superconducting. With two superconductors there is no normal lead to define the reflection, and the call raises a `ValueError`; use `get_dc_current` (below) instead.
- `ht.landauer()` only works for normal junctions, since the Landauer formula counts single electrons. With a superconductor use `ht.didv()`.

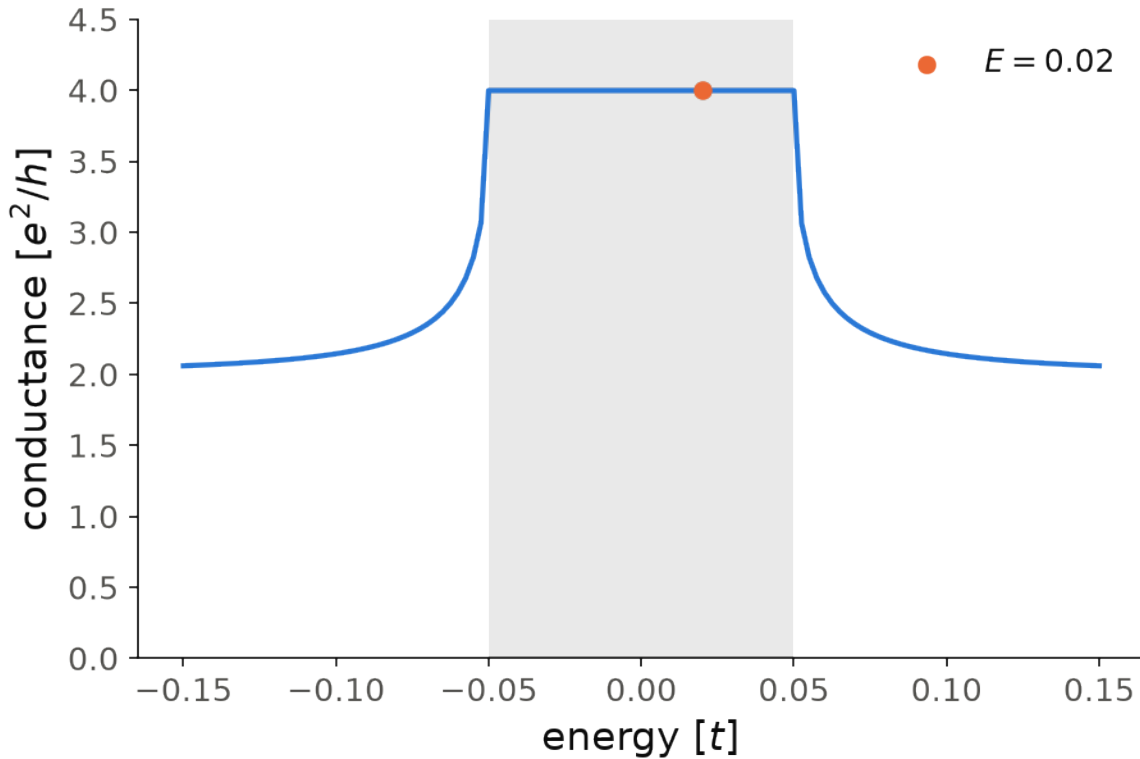


Figure 69: Conductance of the normal lead, five-site cluster and superconducting lead against energy, with the gap shaded.

For the figure we compute the conductance on a grid of energies, besides the single value G at $E = 0.02$ (orange dot). Here the normal lead is spinful, so it has two channels, and the conductance far from the gap goes to 2. Inside the gap (shaded, $|E| < 0.05$) it is 4.00, exactly two times 2: the cluster is a piece of the same chain, so nothing reflects the electrons at the contacts, and Andreev reflection is perfect. Just outside the gap the conductance decays slowly back to 2, it is still 2.06 at $E = 0.15$.

The code of this figure is in `documentation/figures/transport_central.py`.

Only finite (0d) scattering regions are possible for now. See `examples/transport/central_region_ij/main.py` for a runnable script, and `jupyter-notebooks/functionalities/quantum_transport/03_finite_central_region_two_leads/main.py` for an executed notebook.

23.4 Landauer transmission through a real-space device

There is a second way to build a junction, with `multiterminal.Device`. Here the leads and the scattering region are given as geometries, and the hoppings between them come from the real distances between the atoms. The transmission is the Landauer formula above, $T_{ij} = \text{Tr}[\Gamma_i G \Gamma_j G^\dagger]$ between leads i and j . This is useful for a disordered or irregular conductor, which has no unit cell to repeat.

```
import numpy as np
from pyqula import geometry
from pyqula import multiterminal

def lead_cell(x): # a one-site chain unit cell sitting at x
    g = geometry.chain() ; g.r = np.array([[x,0.,0.]]) ; g.r2xyz()
    return g

gc = geometry.chain().supercell(4) # the scattering region
xs = gc.r[:,0]
d = multiterminal.Device()
d.biterminal(left_g=lead_cell(min(xs)-1.),right_g=lead_cell(max(xs)+1.),
             central_g=gc,disorder=0.0)
print("ballistic transmission:",d.transmission(energy=0.0)[0])
```

A perfect chain has one open channel and nothing to scatter, so the transmission is 1 for every energy inside the band. With `disorder` larger than zero, random onsite energies are added in the scattering region, and the transmission goes below 1. Its value depends on the random configuration, so a disorder average needs several configurations. `d.transmission(energy=e)` returns a list with one value per pair of leads, by default only the pair (0,1). `multiterminal.landauer(d,energy,ij=[(i,j)])` gives the same with the pairs written explicitly. The same disorder physics, with a chain of disordered cells between two clean leads, is in `examples/transport/disordered_transport/main.py`.

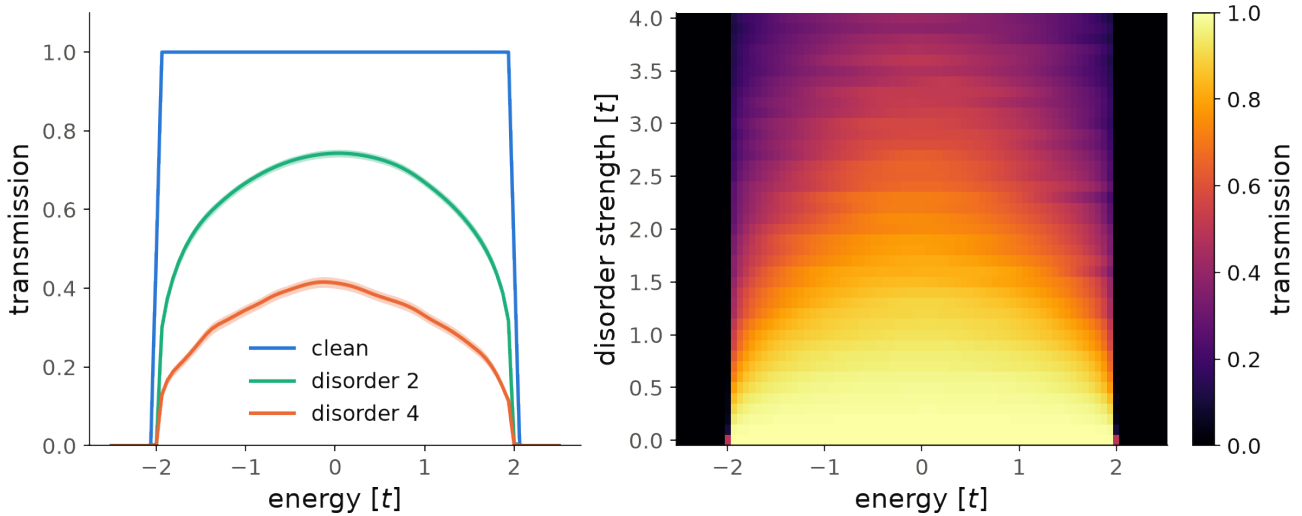


Figure 70: Transmission of the four-site chain against energy, clean and with disorder, and its disorder average against energy and disorder strength.

For the figure we compute the transmission at 81 energies, and we repeat it for 41 disorder strengths from 0 to 4, in steps of 0.1. Each point is the average over 400 random configurations, and the script takes about 17 minutes. The clean chain (blue) transmits 1 inside the band, $|E| < 2$, and 0 outside it; at the band edge $E = \pm 2$ it is 0.49. With disorder the transmission goes down, faster near the band edges than at the centre of the band. At $E = 0$ the average is 0.74 for a disorder 2 (green) and 0.41 for a disorder 4 (orange). The shaded bands are the statistical error of the average, the standard deviation over the configurations divided by $\sqrt{400}$. It is at most 0.015, and it is what you still see as horizontal stripes in the map at strong disorder. With 100 configurations

and another seed the error is twice as large, and the values at $E = 0$ are 0.75 ± 0.02 and 0.38 ± 0.03 , the same within the error.

The code of this figure is in `documentation/figures/transport_device.py`.

23.5 Multiple Andreev reflection and AC-Josephson current

So far all the conductances are at zero temperature and in linear response. We now apply a finite voltage V to a junction between two superconductors, with `HT.get_dc_current()`. With a voltage, the phase difference of the two superconductors grows in time, $\dot{\phi} = 2eV/\hbar$, so the problem is periodic in time with the Josephson frequency. Two things appear. The first is the AC Josephson effect. The second is multiple Andreev reflection (MAR): a quasiparticle is Andreev reflected many times between the two superconductors, and it gains an energy eV at each crossing, until it can leave above the gap. The process of order n opens when $neV = 2\Delta$, so the current shows features at $eV = 2\Delta/n$. What a measurement sees is the time-averaged current $I_{dc}(V)$, and this is what is computed.

The voltage is moved by a gauge transformation into a time-dependent phase of the hopping between the two leads, $t \rightarrow t e^{ieVt/\hbar}$, and the problem is solved in Floquet space, with sidebands at energies $\epsilon + neV$. The DC current is

$$I_{dc}(V) = \frac{e}{h} \int_0^{eV} d\epsilon \sum_n \text{Re Tr} \left\{ [G^r \Sigma_L^< + G^< \Sigma_L^a]_{nn} \tau_z \right\},$$

where G^r and $G^<$ are the retarded and lesser Floquet Green's functions, $\Sigma_L^<$ and Σ_L^a the lesser and advanced self-energies of the left lead, and τ_z is $+1$ for electrons and -1 for holes. This is the formalism of San-Jose, Cayao, Prada and Aguado, [arXiv:1301.4408](https://arxiv.org/abs/1301.4408). The current is in units of e/h times energy (in units of t), so that for two normal leads dI_{dc}/dV is the conductance in units of e^2/h . It works for any pair of normal and superconducting leads, also for two superconductors, where the scattering formula of `didv` above cannot be used.

```
from pyqula import geometry
from pyqula import heterostructures
import numpy as np
g = geometry.chain() # create the geometry
h = g.get_hamiltonian() # create the Hamiltonian
h1 = h.copy() ; h1.add_swave(0.1) # left superconducting lead
h2 = h.copy() ; h2.add_swave(0.1) # right superconducting lead
HT = heterostructures.build(h1,h2) # junction between the two superconductors
HT.set_coupling(0.5) # set the normal transparency of the weak link
vs = np.linspace(0.02,1.5,40)*0.1 # bias voltages
Is = HT.get_iv_curve(vs) # MAR/AC-Josephson dc current
```

`Is` is the DC current at each voltage in `vs`, from 0.02Δ to 1.5Δ , so all the voltages are below $2\Delta = 0.2$. There the current is carried by multiple Andreev reflections, and it shows steps at $eV = 2\Delta/n$, sharper for a small transparency; the steps $n = 2, 3, 4$ are inside the window. Above 2Δ , outside this window, the current grows roughly linearly with the voltage, as in a normal junction.

The figure is the snippet, with its 40 voltages from 0.02Δ to 1.5Δ , and it runs in about two minutes. Below 0.2Δ the current is smaller than 10^{-3} . Then it grows in steps. It rises fast just below each voltage $2\Delta/n$ and becomes flatter just above it. Across $2\Delta/3$ it goes from 0.050 at 0.63Δ to 0.063 at 0.67Δ , and then only to 0.065 at 0.70Δ . Across $2\Delta/2 = \Delta$ it goes from 0.127 at 0.97Δ to 0.142 at 1.01Δ , and above Δ it grows almost linearly, to 0.210 at 1.5Δ . The step at $2\Delta/4$ is weaker, a kink at 0.48Δ . The current is in units of et/h . At the two lowest voltages the sidebands reach `nmax_max` and a warning says that these points are not converged.

The code of this figure is in `documentation/figures/transport_mar.py`.

This is one of the most expensive calculations in the guide. A call first builds the self-energy of the leads up to the largest voltage, in about ten seconds, and then each voltage takes from a fraction of a second to a few seconds, so give all the voltages in one call: the 40 voltages of the snippet take about two minutes. The number of Andreev reflections that matter is about $2\Delta/eV$, and each one needs a Floquet sideband, so small voltages

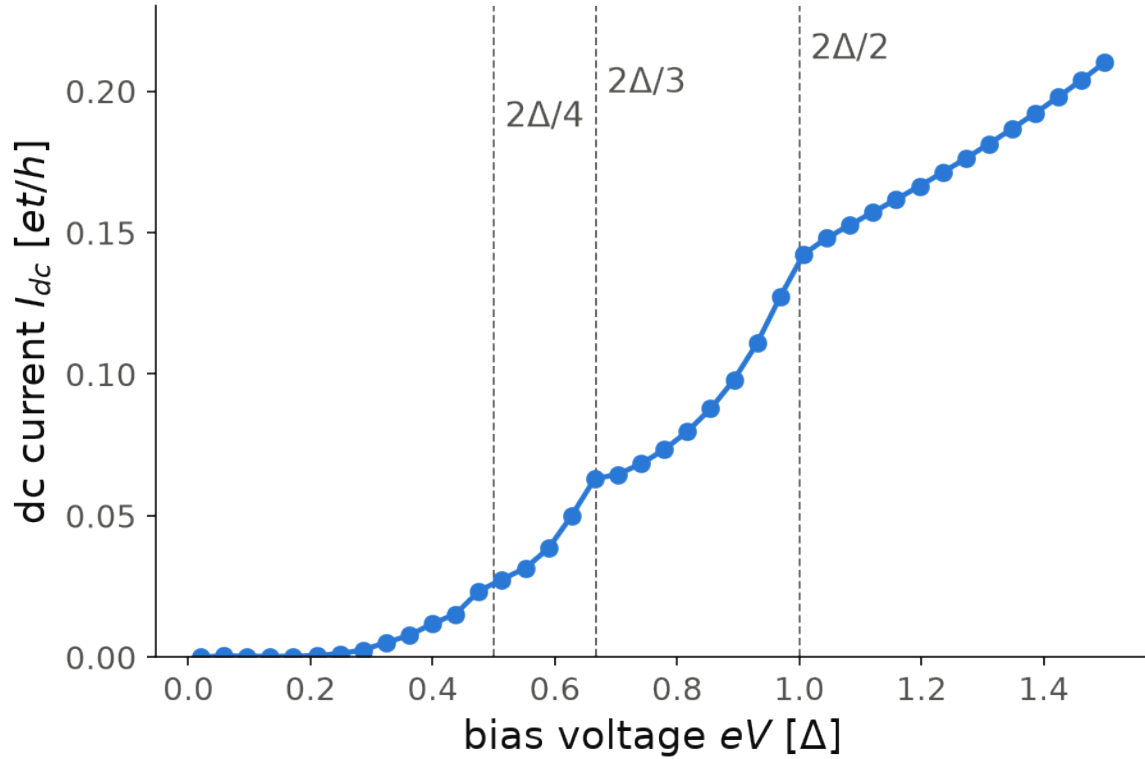


Figure 71: DC current of the junction between two superconductors against the bias voltage, with the voltages $2\Delta/n$ of multiple Andreev reflection.

need more sidebands. The sidebands are added until I_{dc} converges. In the snippet this gives `nmax=14` above 0.5Δ , and the cap `nmax_max=40` at 0.13Δ and below. This does not make small voltages slower, because the energy integral runs only from 0 to eV , and a short window needs fewer energy points. In the snippet every voltage costs less than one second, at low and at high voltage. If a voltage is very small, the number of sidebands can reach `nmax_max`, and a warning says that this point is not converged. Starting the voltages a bit higher is the cheap solution; a larger `nmax_max` is the expensive one.

Only 1D leads are possible. A central region between the leads, for example a quantum dot, can be added with `heterostructures.build(h1,h2,central=[hc])`, and it is slower. The voltage drops at the last bond on the right, so the central region is at the potential of the **left** lead. To move its levels, use `hc.shift_fermi(eps)`, which keeps it a valid BdG Hamiltonian, and do not add to its diagonal by hand. See `examples/transport/floquet_keldysh_mar/main.py` for a runnable script.

23.5.1 Conductance at finite voltage

`HT.didv()` chooses between the two formalisms with `method`:

- `method="smatrix"`: the zero-temperature scattering formula above. The normal lead is grounded and all the voltage drops on the other lead.
- `method="keldysh"`: the derivative of I_{dc} at voltage **energy**. Both leads are biased rigidly.
- `method="auto"` (default): `"keldysh"` if both leads are superconducting, and `"smatrix"` otherwise.

The two methods apply the voltage in a different way, so on a junction with one normal lead they do not give the same number, even when the pairing of the other lead goes to zero. Leave the default.

`didv` takes a single **energy** and returns a number, or an array `energies=[...]` and returns an array, but not both. For the junction between two superconductors above, this is the dI/dV curve in the MAR regime:

```
es = np.linspace(0.15,0.25,40)*0.1 # bias energies
Gs = HT.didv(energies=es, nmax_max=40) # dI/dV curve
```

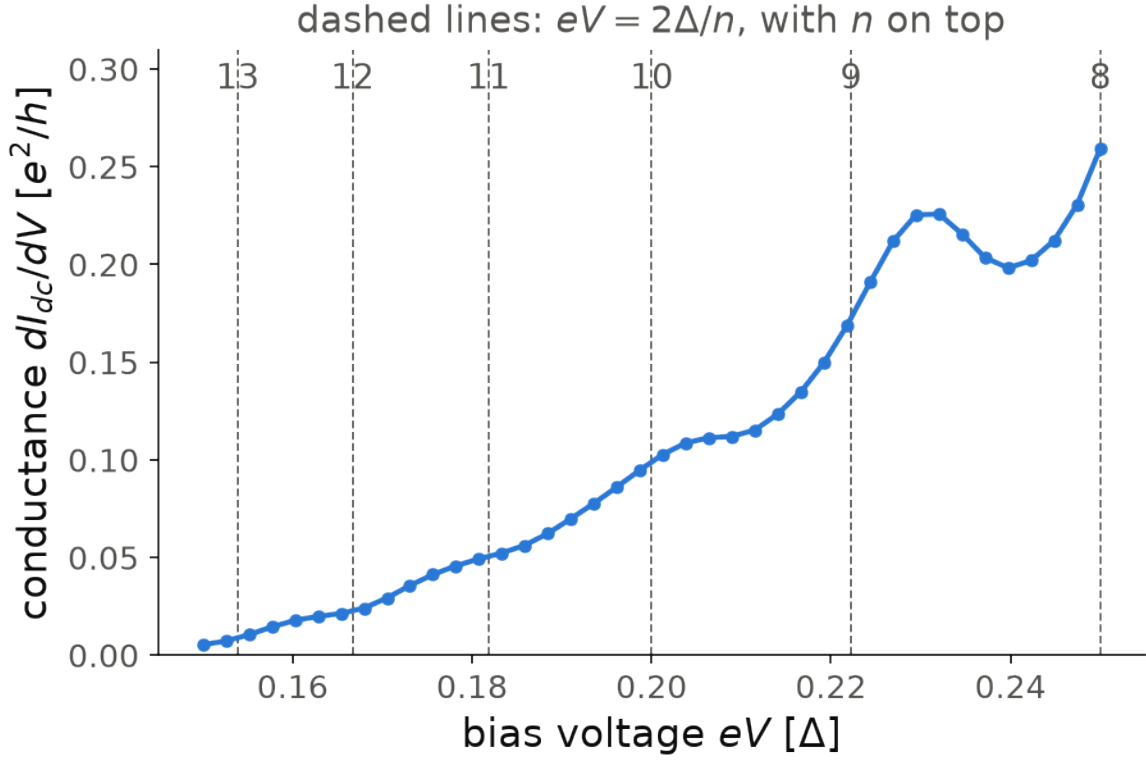


Figure 72: Conductance dI/dV of the junction between two superconductors at small bias voltage, with the voltages $2\Delta/n$ of multiple Andreev reflection of order 8 to 13.

The figure is the snippet, with its 40 voltages, and it runs in about three minutes. The window from 0.15Δ to 0.25Δ is deep inside the gap, where the current is carried by Andreev processes of order $n = 8$ to 13. The conductance is small, from 0.005 to 0.26 in units of e^2/h , and it is not smooth. It has a peak of 0.23 at 0.23Δ , just above $2\Delta/9$, then a minimum of 0.20 at 0.24Δ , and it rises again towards $2\Delta/8 = 0.25\Delta$. There is a shoulder at 0.21Δ , just above $2\Delta/10$, and weaker ones near $2\Delta/11$ and $2\Delta/12$. So each order n gives a feature close to $2\Delta/n$, and the features are broad because the transparency, 0.5, is not small.

The code of this figure is in `documentation/figures/transport_finitev.py`.

23.5.2 A superconducting STM tip

A `LocalProbe` is an STM tip, a semi-infinite chain coupled to one site of an infinite sample. The tip and the sample play the role of the two leads, and T is the hopping between them. The choice of method is the same as above: the scattering formula by default, and the Floquet-Keldysh current when both tip and sample are superconducting. For a normal tip the two methods agree, because the tip is grounded in both.

Besides the conductance, we compute the decay rate κ . In STM, the conductance changes with the tip height through the transparency T , and the exponent of this power law depends on how many electrons are transferred in each process. `pyqula` computes

$$\kappa = \frac{d \ln G_S / d \ln T}{d \ln G_N / d \ln T},$$

the ratio between the exponent with the superconductor and the exponent of the same junction with the pairing removed. In the tunnel limit $G_N \propto T^2$. A single quasiparticle also gives $G_S \propto T^2$, so $\kappa = 1$, while Andreev reflection transfers two electrons, $G_S \propto T^4$, and $\kappa = 2$. So κ tells you which process carries the current at each energy, and it is measured by changing the tip height.

```

from pyqula import geometry
from pyqula.transport.localprobe import LocalProbe

g = geometry.chain()
h = g.get_hamiltonian() ; h.shift_fermi(1.) ; h.add_swave(0.1) # SC sample
lead = geometry.chain().get_hamiltonian() ; lead.shift_fermi(1.) ; lead.add_swave(0.1) # SC probe
lp = LocalProbe(h,lead=lead,delta=1e-3)
lp.T = 0.3 # reference transparency
G = lp.didv(energy=0.25,nmax=4,nmax_max=60,tol=5e-4) # routed through Keldysh automatically
k = lp.get_kappa(energy=0.25,nmax=4,nmax_max=60,tol=5e-4)

```

G is the conductance and k the decay rate at voltage 0.25, which is above the combined gap $2\Delta = 0.2$ of tip and sample. The snippet takes about a minute, much more than with a normal tip, because each `didv` or `get_kappa` call solves several Floquet-Keldysh problems, and it is slowest deep inside the gap at small transparency. See `examples/transport/decay_constant_keldysh/main.py` for a runnable script with a coarse energy grid and few sidebands, and the notebooks `08_tunneling_contact_probe.ipynb` (normal tip) and `09_mar_ac_josephson_sc_probe.ipynb` (superconducting tip) in `jupyter-notebooks/functionalities/quantum_transport`.

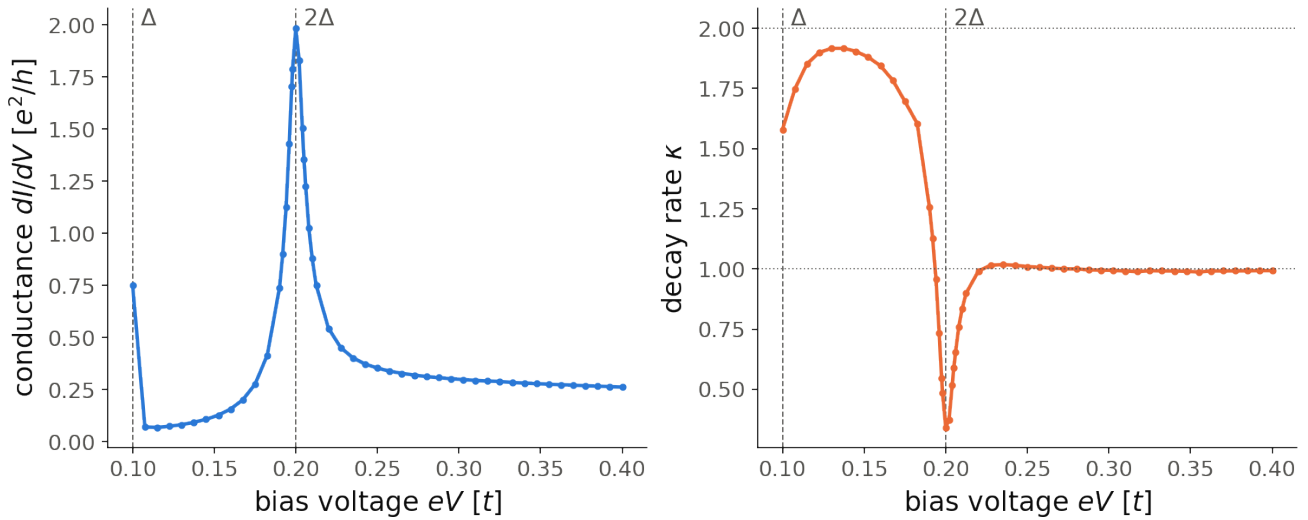


Figure 73: Conductance and decay rate of a superconducting STM tip on a superconducting sample against the bias voltage.

The figure repeats the snippet at 41 voltages from $\Delta = 0.1$ to 0.4 , in steps of 0.0075 , and at 11 more voltages between 0.19 and 0.21 , around the peak at 2Δ . It takes about 20 minutes. On the left, the conductance has its largest peak, $1.98 e^2/h$, at $eV = 0.2$, the sum of the gaps of tip and sample, where the coherence peaks of the two densities of states meet. The peak is narrow, its half width is about 0.008 , so it needs the finer grid. A second peak, $0.75 e^2/h$, is at $eV = \Delta$, the voltage $2\Delta/2$ of the first Andreev reflection. Between the two the conductance is small, 0.07 to $0.41 e^2/h$. Above 2Δ it goes down to $0.26 e^2/h$ at $eV = 0.4$, the tunneling between the two continua.

On the right, κ is between 0.99 and 1.02 above $eV = 0.22$, so a single quasiparticle carries the current there. Between Δ and 2Δ , κ grows to 1.92 at $eV \approx 0.13$, close to 2: the current comes from Andreev reflection, which transfers two electrons. Near 2Δ it drops, and at 2Δ itself $\kappa = 0.34$, below one, because the peak of the conductance there grows more slowly with T than the normal conductance.

These values need converged sidebands. With `tol=5e-4` and `nmax_max=60`, as in the snippet, the conductance and κ change by less than 0.5% with respect to `tol=2e-3` and `nmax_max=40`. A looser tolerance is not enough: with `tol=5e-2` and `nmax_max=12` the conductance at $eV = 0.25$ is 7% too small, and inside the gap κ comes out as large as 6, while the converged value is below 2. With `tol=2e-3` one point, $eV = 0.2725$, still came out at half its converged value. At a loose tolerance, `get_kappa` can also stop with an error, when the conductance changes sign between the two transparencies it needs, and the error asks for a tighter `tol` or a larger `nmax_max`.

The figure starts at Δ : below it the higher orders of multiple Andreev reflection need more sidebands, and we did not converge those points.

The code of this figure is in `documentation/figures/transport_stmtip.py`.

`get_kappa` also takes a temperature `temp`, and then every conductance is the thermal average at that temperature. Like `didv`, it takes a single `energy` or an array `energies=[...]`:

```
k = lp.get_kappa(energies=[0.1,0.25,0.4],temp=0.02,nmax=4,nmax_max=12,tol=5e-2)
```

`HT.get_kappa()` of a junction takes the same `temp` and `energies`.

24 Single defects in infinite systems

We now see how to compute the effect of a single defect, an impurity or a vacancy, in an infinite crystal, with `embedding.Embedding`. A supercell calculation cannot do this exactly: the defect is repeated in every supercell, and it interacts with its own copies. The Green's function embedding keeps the infinite pristine crystal and changes only one cell.

Let H_0 be the pristine periodic Hamiltonian, and $G_0(E)$ its local Green's function on one unit cell,

$$G_0(E) = \frac{1}{N_k} \sum_k [E + i\delta - H_0(k)]^{-1}.$$

The rest of the infinite crystal acts on this cell as a self-energy $\Sigma(E)$, defined by

$$G_0(E) = [E + i\delta - H_0^{\text{cell}} - \Sigma(E)]^{-1},$$

where H_0^{cell} is the pristine intracell Hamiltonian. The defect changes the Hamiltonian of this cell to V , and the rest of the crystal is not changed, so the Green's function of the defective cell is the Dyson equation

$$G(E) = [E + i\delta - V - \Sigma(E)]^{-1}.$$

This is exact for a defect inside one cell, in a crystal that is infinite in all directions. The local density of states is $\rho_n(E) = -\frac{1}{\pi} \text{Im} G_{nn}(E)$, and around the defect it is what STM measures near a single adatom or vacancy.

In `pyqula`, `h` is the pristine Hamiltonian and `m` is a Hamiltonian with the defect; only its intracell matrix is used as V . Here the defect is an onsite energy 1.0 on one site of a chain:

```
from pyqula import geometry
from pyqula import embedding
g = geometry.chain() # create the geometry
h = g.get_hamiltonian() # pristine, infinite Hamiltonian
hv = h.copy()
hv.add_onsite(lambda r: 1.0 if r[0]<0.01 else 0.0) # a single-site onsite defect
eb = embedding.Embedding(h,m=hv) # embed the defect in the infinite system
(x,y,d) = eb.get_ldos(energy=0.0,delta=1e-2,nsuper=200,nk=400) # LDOS around the defect
```

`eb.get_ldos()` returns the positions `x`, `y` of the sites in a window of `nsuper` unit cells around the defect, and the LDOS `d` on each of them, at the energy `energy`. The positions are half-integers, from -99.5 to 99.5 here, and the defect is the site at $x = 0.5$. In a metal, as this chain, you see Friedel oscillations: the LDOS oscillates with wavevector $2k_F$ around the defect. At $E = 0$ we have $k_F = \pi/2$, so the LDOS alternates from one site to the next, between 0.38 and 0.26 next to the defect, with a converged `k`-mesh (see below). In one dimension the oscillation at a single energy does not decay with the distance. Here it decays slowly, to 0.34 and 0.29 at 90 sites from the defect, only because of the broadening δ , over a length of about $v_F/(2\delta) = 100$ sites. In a gapped system, a strong enough defect gives a bound state inside the gap, and its LDOS is localized around the defect.

The k-mesh has to resolve the broadening. With $\delta = 0.01$, `nk=400` as in the snippet is not converged: next to the defect it gives 0.49 and 0.29 instead of 0.38 and 0.26, the values that `nk=2000` and `nk=8000` both give. So when you make `delta` smaller, make `nk` larger, and compare two values of `nk`. The call also writes `LDOS.OUT`; pass `write=False` to avoid it.

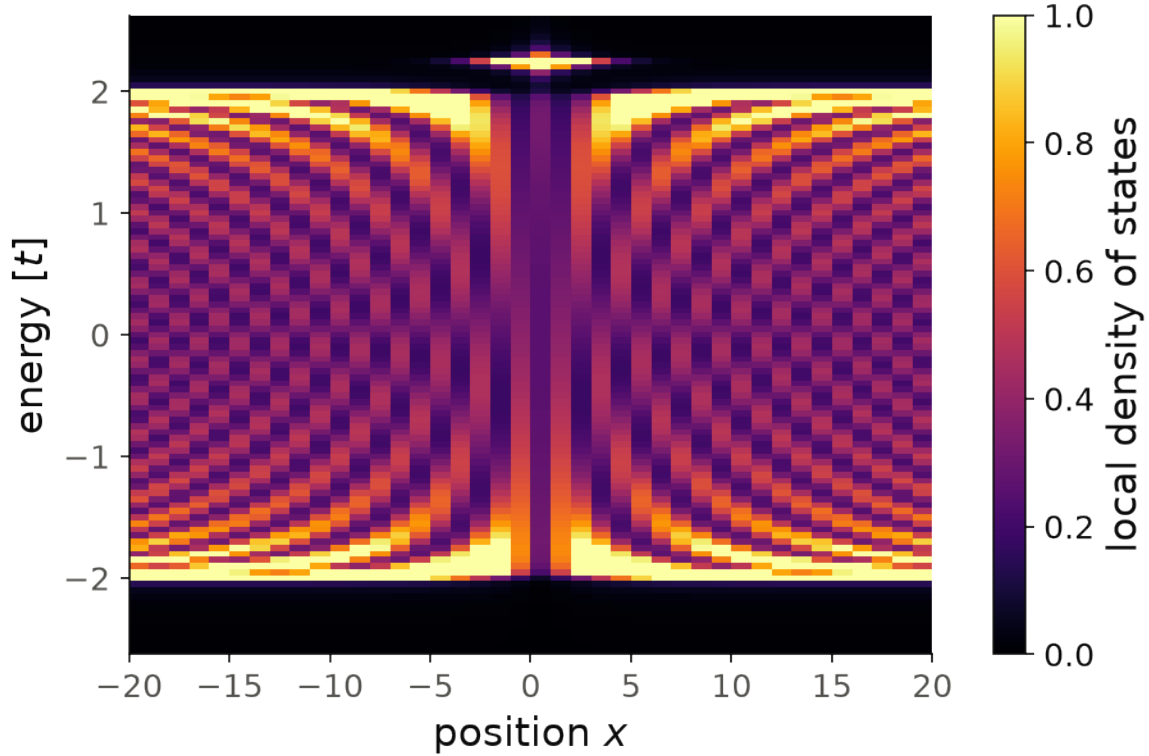


Figure 74: Local density of states around a single onsite defect in an infinite chain, as a function of position and energy.

For the figure we compute the same LDOS at 105 energies, on the 40 sites around the defect, and with the converged `nk=2000` instead of 400. The defect is at $x = 0.5$. Inside the band, between -2 and 2 , you see the Friedel oscillations as a checkerboard. At $E = 0$ the Fermi momentum is $k_F = \pi/2$, so the period is two sites, $2k_F = \pi$, and at other energies the period follows k_F . Over these 40 sites the amplitude decays only a little, as we saw above. Above the band, at $E = 2.25$, there is a bright spot on the defect: the repulsive onsite energy $V = 1$ pulls one state out of the band, a bound state at $\sqrt{4t^2 + V^2} = 2.24$.

The code of this figure is in `documentation/figures/embedding_ldos.py`.

The other methods of the embedding object:

- `eb.get_dos()`: the total DOS of the window around the defect.
- `eb.multildos()`: the LDOS at many energies, written to the folder `MULTILDOS/`.
- `eb.get_gf()`: the Green's function $G(E)$ of the defective cell.
- `eb.get_didv()`: the differential conductance of an STM tip on the defect.

The momentum-space picture of the same defect is Quasiparticle interference. See `examples/embedding/single_impurity_1D/` and `examples/embedding/honeycomb_vacancy/main.py` for runnable scripts, the other scripts in `examples/embedding/` for vacancies, boundaries and Yu-Shiba-Rusinov states, and `jupyter-notebooks/functionalities/spectral_functions/04_` for an executed notebook.

25 Kernel polynomial method

Exact diagonalization gives all the eigenstates, and its cost grows as the cube of the number of sites. For a system with 10^5 or 10^6 sites this is not possible, and also not needed: the density of states is a trace, and a trace can be computed without the eigenstates. The kernel polynomial method (KPM) does this. It expands the density of states in Chebyshev polynomials of the Hamiltonian, and only needs products of a vector with the sparse Hamiltonian, so its cost is linear in the number of sites.

25.1 The density of states of a large system

The Chebyshev polynomials $T_m(x)$ are defined on $[-1, 1]$, so we first rescale the Hamiltonian, $\tilde{H} = H/s$, with a scale s larger than the largest energy. They obey the recursion

$$T_0(x) = 1, \quad T_1(x) = x, \quad T_{m+1}(x) = 2xT_m(x) - T_{m-1}(x).$$

The density of states at energy E , with $x = E/s$, is

$$\rho(E) = \frac{1}{\pi s \sqrt{1-x^2}} \left[g_0 \mu_0 + 2 \sum_{m \geq 1} g_m \mu_m T_m(x) \right], \quad \mu_m = \text{Tr } T_m(\tilde{H}).$$

The μ_m are the moments. They are computed with the recursion above, applied to a vector: $|v_{m+1}\rangle = 2\tilde{H}|v_m\rangle - |v_{m-1}\rangle$, so each moment costs one sparse matrix-vector product. The trace is estimated with random vectors $|r\rangle$, as $\text{Tr } A \approx \frac{1}{R} \sum_{r=1}^R \langle r|A|r\rangle$, and the error goes down as $1/\sqrt{R}$. The series is cut after N moments, and this creates oscillations (Gibbs oscillations). The factors g_m of the Jackson kernel remove them,

$$g_m = \frac{(N-m+1) \cos \frac{\pi m}{N+1} + \sin \frac{\pi m}{N+1} \cot \frac{\pi}{N+1}}{N+1},$$

and the price is that every eigenvalue becomes a peak of finite width, nearly a Gaussian. In pyqula $N = 2n_{\text{pol}}$, with n_{pol} the number of polynomials, because each step of the recursion gives two moments.

The KPM density of states is `h.get_dos()` with `mode="KPM"`:

```
from pyqula import geometry
import numpy as np
g = geometry.chain()
g = g.get_supercell(100000) # a big supercell
g.dimensionality = 0
h = g.get_hamiltonian(is_sparse=True, has_spin=False)
(x,y) = h.get_dos(mode="KPM",
    energies=np.linspace(-3.0,3.0,601), # energies
    delta=1e-2, # energy resolution, half width of the peak of a level
    ntries=40 # number of random vectors for the stochastic trace
)
```

`x` is the energy grid and `y` the density of states, the same quantity that exact diagonalization would give for this chain of 10^5 sites. For a chain you see the one-dimensional van Hove singularities at $E = \pm 2$. A sparse Hamiltonian (`is_sparse=True`) is the natural choice, since the method never needs the dense matrix. This snippet takes about one minute.

The KPM density of states, divided by the number of sites (blue), follows the one of the infinite chain, $1/(\pi\sqrt{4t^2 - E^2})$ (dashed), and the two curves lie on top of each other. You see the two van Hove peaks at $E = \pm 2$, and the minimum $1/(2\pi) = 0.16$ at zero energy. Outside the band it is zero. Inside the band, the only error is the noise of the stochastic trace. Each random vector already averages over the N sites, so the noise goes down as $1/\sqrt{NR}$. For $|E| < 1.5$ it is 0.5% on average and at most 2% at a single energy. With 3000 sites

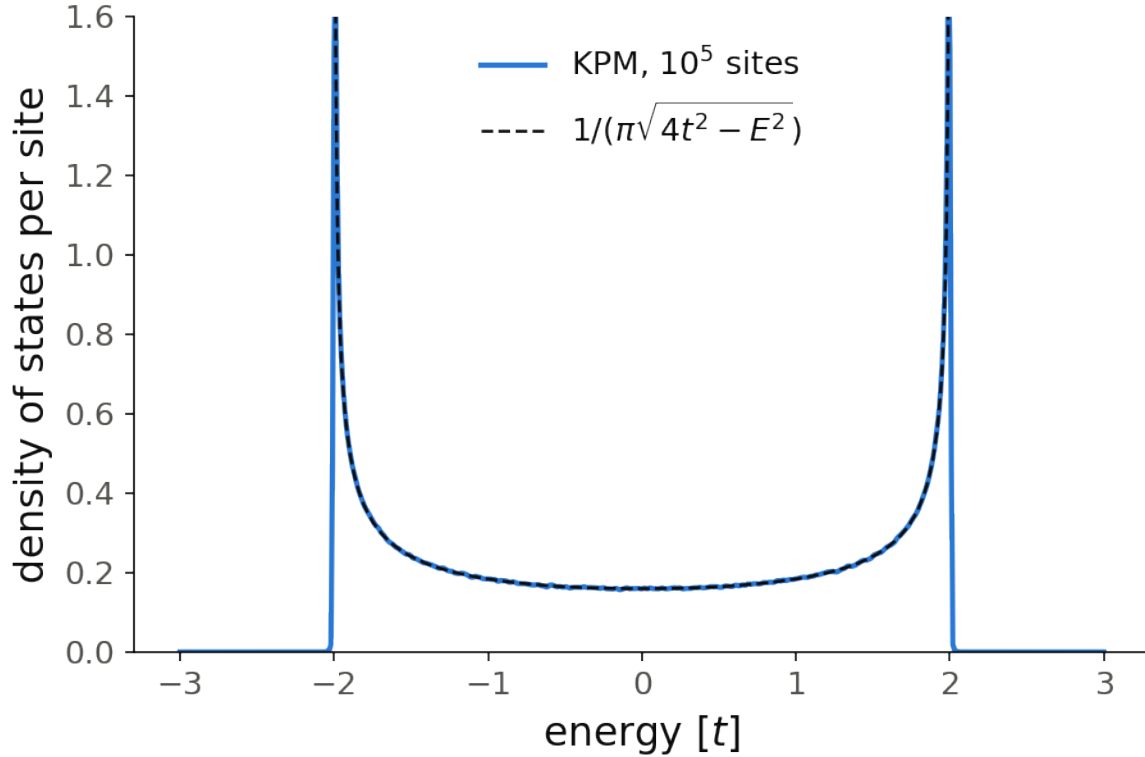


Figure 75: KPM density of states of a chain of 100000 sites, compared with the density of states of the infinite chain.

and 10 random vectors it was 5% on average and up to 18%, and with 3×10^5 sites and 80 vectors it is 0.2%, so the curve does not change any more at the scale of the figure.

At the band edges the error is of another kind. The exact density of states diverges as $1/\sqrt{2 - |E|}$, and the KPM curve cannot follow it, because every level is a peak of half width `delta`. So the divergence becomes a finite peak, 1.73 per site at $E = \pm 1.99$, and just inside the edge the curve is a little above the exact one. This does not improve with more sites or more random vectors. Only a smaller `delta` helps: with `delta=5e-3` the peak is 2.15, and it keeps growing as `delta` goes down.

The code of this figure is in `documentation/figures/kpm_dos.py`.

Two numbers control the result:

- **delta**: the energy resolution, the half width at half maximum of the peak of a single level. It is the same width that `delta` gives as a Lorentzian in `mode="ED"`, so the two curves can be compared. The method uses $n_{\text{pol}} \approx 1.85 s/\delta$ polynomials, so a sharper density of states costs more. The KPM peak is nearly a Gaussian, about 1.5 times taller than the Lorentzian and without its long tails.
- **ntries**: the number of random vectors R . The noise in y goes as $1/\sqrt{R}$.

`delta` must be larger than the distance between levels, here $2\pi t/N \approx 6 \times 10^{-5}$ in the middle of the band, otherwise `y` shows the single levels of the finite chain. The spectrum must be inside $[-s, s]$, since $T_m(x)$ grows exponentially outside $[-1, 1]$. The default is `scale=10` in units of the hopping. A Hamiltonian with a larger spectrum, for example with a large onsite energy or exchange field, needs a larger `scale`. If `scale` is too small, the code raises a `ValueError`.

25.1.1 Green's functions between two sites

The same expansion also gives the Green's function between two sites i and j of this chain, without inverting a matrix. The function `kpm.dm_ij_energy()` gives

$$\pi \langle i | \delta(E - H) | j \rangle = \pi \sum_n \psi_n(i) \psi_n^*(j) \delta(E - E_n) = \frac{i}{2} [G_{ij}^R(E) - G_{ij}^A(E)],$$

the spectral part of the Green's function:

```
from pyqula import kpm
(x,y) = kpm.dm_ij_energy(h.intra,npol=200,i=0,j=9,ne=1000)
```

For $i = j$ it is π times the local density of states of site i . For $i \neq j$ it integrates to zero over all energies, because the eigenstates form a complete basis. Its integral up to the Fermi energy is π times the density matrix element $\langle c_j^\dagger c_i \rangle$, so its sign along the energy axis tells which part of the spectrum correlates the two sites. Here `npol` is the number of polynomials n_{pol} , and each level becomes a peak of half width $1.85 s/n_{\text{pol}}$, the inverse of the relation for `delta` above.

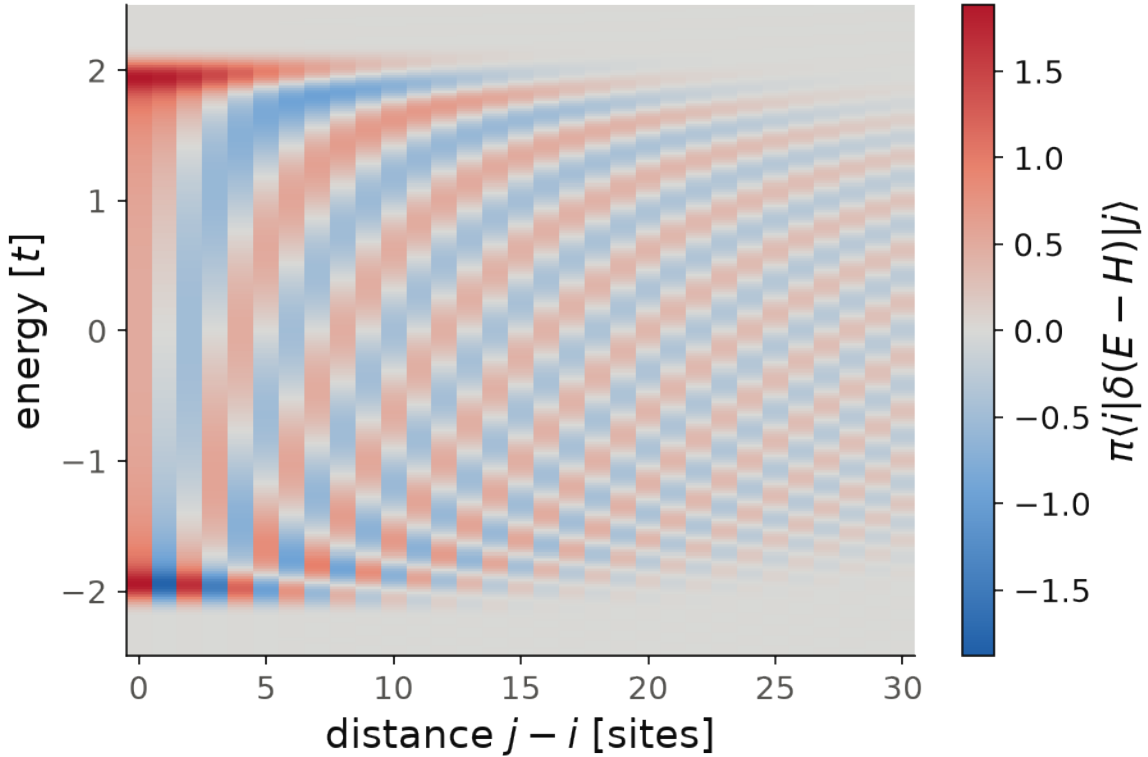


Figure 76: Spectral part of the Green's function between a site and its neighbors in a long chain, as a function of distance and energy.

For the figure we take the site $i = 1500$, in the middle of the chain and far from its ends, and compute the same quantity for $j = i$ to $i + 30$. The first column, $j = i$, is π times the LDOS: $0.50 = \pi/(2\pi)$ at zero energy, with peaks at the band edges. At zero energy the value along the chain is $0.5, 0, -0.5, 0, 0.5, \dots$, the oscillation $\cos(k_F d)$ with $k_F = \pi/2$. Further away the sign changes faster along the energy axis, and the finite resolution of 200 polynomials washes it out. The integral over all energies is π for $j = i$ and zero for the other sites. The integral up to zero energy, divided by π , is 0.5 for $j = i$, the occupation of a site at half filling, and $-0.318 = -1/\pi$ for the first neighbor.

The code of this figure is in `documentation/figures/kpm_green.py`.

The full Green's function, with its real part, comes from `kpm.correlator0d()`. It returns the energies, $\text{Re } G_{ij}^R$ and $-\text{Im } G_{ij}^R$. With `kernel="lorentz"` each pole becomes a Lorentzian of half width $1.5 s/n_{\text{pol}}$, which is the $G_{ij}^R(E + i\delta)$ that a matrix inversion gives. The cutoff of the expansion leaves an error of about 10% on a sharp pole. The keyword `lorentz_lambda`, called λ here, reduces it. The half width is then $\lambda s/(2n_{\text{pol}})$, so for the same

δ you need $n_{\text{pol}} = \lambda s / (2\delta)$. The error falls roughly as $e^{-\lambda}$. On a small chain it is 9% at the default $\lambda = 3$ and 0.3% at $\lambda = 8$.

See `examples/0d/kpm_dos/main.py` and `examples/0d/kpm_correlator/main.py` for runnable scripts, the second one with a comparison against the exact Green's function, and `jupyter-notebooks/functionalities/chebyshev_kpm_alg` for the local and non-local spectral functions of a system too large to diagonalize.

25.1.2 On a GPU

The Chebyshev moments of the same density of states can also be computed on a GPU, with the switch described in GPU. The keyword `kpm_prec` chooses single or double precision for the moments:

```
from pyqula import gpu
gpu.set_gpu(True) # Chebyshev moments on the GPU from now on
(x,y) = h.get_dos(mode="KPM",
                 energies=np.linspace(-3.0,3.0,200),
                 delta=1e-4,ntries=10,
                 kpm_prec="single") # single precision moments
```

If there is no GPU, the same code runs on the CPU with a warning.

25.2 The local density of states of a large system

We now compute the local density of states of every site of a large system, which is the map that STM measures, with `h.get_multidos()` and `h.get_ldos()` in `mode="KPM"`. The local density of states of orbital i is $D(\omega, i) = \langle i | \delta(\omega - H) | i \rangle$. It has the same expansion as the total density of states, but with the moments of one orbital,

$$D(\omega, i) = \frac{g_0 \mu_0(i) + 2 \sum_{m \geq 1} g_m \mu_m(i) T_m(\omega/s)}{\pi s \sqrt{1 - (\omega/s)^2}}, \quad \mu_m(i) = \langle i | T_m(H/s) | i \rangle,$$

so no random vectors are needed: the recursion starts from orbital i .

Doing this recursion over the whole system for every orbital costs the square of the number of sites. `pyqula` avoids this. After m steps, the vector that started on orbital i has only reached the sites within m hops of i . The $2n_{\text{pol}}$ moments only need $n_{\text{pol}} - 1$ steps, because $T_{2m} = 2T_m T_m - T_0$ and $T_{2m+1} = 2T_{m+1} T_m - T_1$ give two moments per step. So the moments computed on the sites within $n_{\text{pol}} - 1$ hops of i are exactly those of the whole system, and the full map costs a time linear in the number of sites.

```
from pyqula import geometry
import numpy as np
g = geometry.honeycomb_lattice().get_supercell(80) # 12800 sites
g.dimensionality = 0 # a finite island with zigzag edges
h = g.get_hamiltonian(is_sparse=True,has_spin=False) # keep it sparse
(x,y,es,ldos) = h.get_multidos(energies=np.linspace(-1.,1.,41), # energies
                              mode="KPM", # Chebyshev expansion of every site
                              delta=0.1, # energy resolution, half width of the peak of a level
                              write=False) # return the maps without writing them
```

`ldos` contains one map per energy, one row for each energy of `es`. All the rows come from the same moments, so 41 energies cost about the same as one. `h.get_ldos(e=0.,mode="KPM",delta=0.1)` gives only the map at zero energy. In this island the 318 sites at the zigzag edges, about 2.5% of all the sites, carry 56% of the weight at zero energy: these are the zigzag edge states. At $E = \pm 1$ the same sites carry only 1.4%. The sum of a map over the sites is the density of states at that energy, with the normalization of `h.get_dos(mode="KPM")`. The result is the same as the expansion over the whole island, up to roundoff.

Both maps show the whole island, a rhombus with four zigzag edges. At zero energy (left) the color scale is logarithmic, because the map covers three decades. The outer sites of the edges are bright: these are the zigzag edge states. Their typical LDOS is 1.1, and 2.8 at the sharp corners, so they are saturated, since the color scale is cut at 1.07. Their weight decays into the island over a few rows, and the bulk is dark, with a typical LDOS of

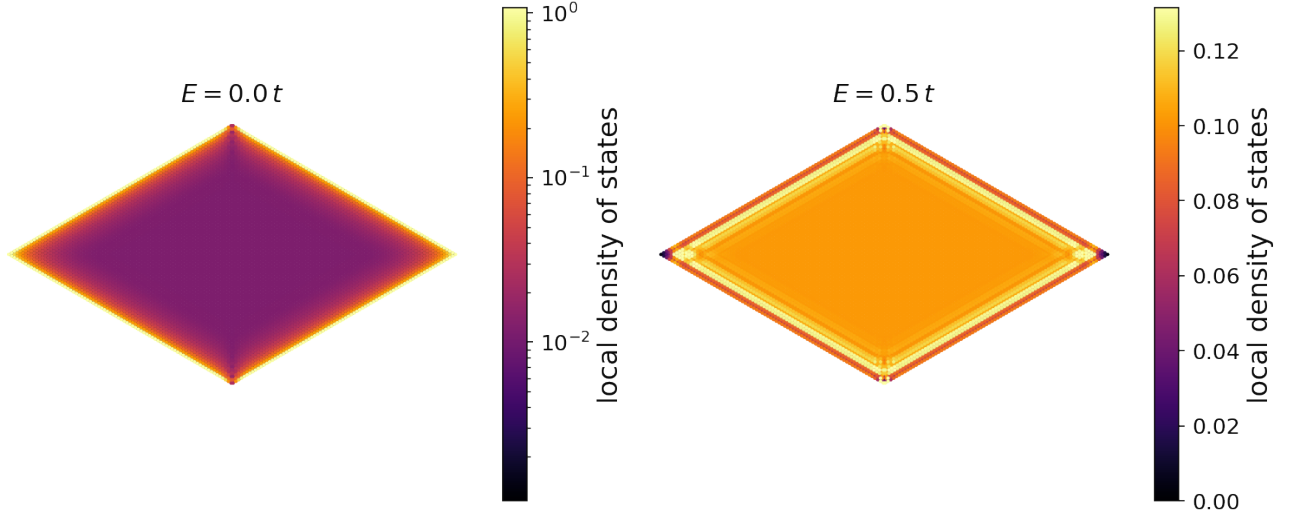


Figure 77: KPM local density of states of a whole honeycomb island of 12800 sites, at zero energy on a logarithmic color scale and at $E=0.5$.

0.012. At the two blunt corners the edge states are suppressed. At $E = 0.5$ (right, linear scale) the map is the opposite. The bulk is almost uniform, about 0.10, and the edge sites are no longer special: they carry 2.4% of the weight, about the same as their share of the sites. The brighter lines parallel to the edges are standing waves of the bulk states, reflected at the edge. The color scale is cut at 0.13, and the brightest sites of the map reach 0.22.

The code of this figure is in `documentation/figures/kpm_ldos.py`.

The cost of one site is set by the resolution. A peak of half width δ needs about $1.85 s/\delta$ polynomials, here 62, and the region of 61 hops around a site contains about 5700 sites of the honeycomb lattice. In two dimensions this region grows as n_{pol}^2 , so halving δ makes each site eight times more expensive. You can think of it in this way: with a resolution δ , a site only sees its environment up to a distance of about v/δ , where v is the velocity of the electrons, whatever the size of the sample.

The vector has almost no weight near the border of this region, so `kpm_radius` smaller than n_{pol} gives a cheaper map that is still converged in practice. On a disordered honeycomb island of 20000 sites with `npol=100`, the map with `kpm_radius=80, 70, 60` and `50` differs from the exact one by 10^{-11} , 2×10^{-7} , 5×10^{-5} and 10^{-3} of its largest value, and with 60 it takes about one third of the time. How small the radius can be depends on the velocity of the states, so you should converge it as you converge a k-mesh, by comparing two values.

The keywords are:

- **delta**: energy resolution, the half width at half maximum of the peak of a level, as in `h.get_dos(mode="KPM")`.
- **npol**: number of polynomials, instead of **delta**; 200 if neither is given.
- **kpm_radius**: radius in hops of the region around each site; **None** (default) gives the exact map.
- **operator**: an operator A independent of k , giving the local matrix element $\text{Re}(A\rho)_{ii}$, for example `operator="sz"` for the spin-resolved map.
- **scale**: half width s of the energy window; estimated from the Hamiltonian if not given.
- **kpm_prec**: precision, "double" on the CPU and "single" on the GPU by default.
- **nk**: k-points per direction for a periodic Hamiltonian; the map is averaged over the Brillouin zone.

The same recursion is used by the KPM mean field of Mean-field Hubbard models, so a self-consistent state of 10^5 sites can be read with the map that STM would measure on it. See `examples/0d/kpm_ldos_map/main.py` for a runnable script.

26 Wannierization

Often we only care about a few bands, for example the bands near the Fermi energy. Wannierization builds a smaller tight-binding model that describes only these bands. We do this with `h.get_wannier_hamiltonian()`.

26.1 Maximally localized Wannier functions

Take a group of J bands that is separated from the other bands by a gap in the whole Brillouin zone. From their Bloch states $|\psi_{m\mathbf{k}}\rangle$ we build localized orbitals, the Wannier functions,

$$|w_{n\mathbf{R}}\rangle = \frac{1}{N_k} \sum_{\mathbf{k}} e^{-i\mathbf{k}\cdot\mathbf{R}} \sum_{m=1}^J U_{mn}(\mathbf{k}) |\psi_{m\mathbf{k}}\rangle,$$

where $\mathbf{R}$ is a unit cell, N_k is the number of $\mathbf{k}$ -points of the mesh, and $U(\mathbf{k})$ is a unitary $J \times J$ matrix at each $\mathbf{k}$ -point. The Bloch states have an arbitrary phase (a gauge), and $U(\mathbf{k})$ uses this freedom. Maximally localized Wannier functions are the ones with the $U(\mathbf{k})$ that minimizes the total spread

$$\Omega = \sum_n [\langle w_{n0} | r^2 | w_{n0} \rangle - |\langle w_{n0} | \mathbf{r} | w_{n0} \rangle|^2],$$

the sum of the squared widths of the Wannier functions. The centre of function n is $\langle w_{n0} | \mathbf{r} | w_{n0} \rangle$. In the basis of the Wannier functions the Hamiltonian has hoppings $t_{nm}(\mathbf{R}) = \langle w_{n0} | H | w_{m\mathbf{R}} \rangle$, and this is the new tight-binding model. It has one orbital per selected band, and it is a normal pyqula Hamiltonian.

As an example we take the honeycomb lattice with a sublattice potential. The potential opens a gap, so the lower band (the valence band) is isolated and can be Wannierized alone:

```
from pyqula import geometry
g = geometry.honeycomb_lattice()
h = g.get_hamiltonian(has_spin=False)
h.add_onsite([0.8,-0.8]) # sublattice potential opens a gap

# Wannierize just the lowest (valence) band: bands=[0,0] selects band
# index 0 at every k-point of a 12x12 Monkhorst-Pack wannierization mesh
hwan = h.get_wannier_hamiltonian(bands=[0,0],nk=12)

print("Number of Wannier functions:",hwan.intra.shape[0])
print("Wannier centres (Cartesian):\n",hwan.wannier_centres)
print("Wannier spreads:",hwan.wannier_spreads)
print("Total spread Omega:",hwan.wannier_spread_total)
```

`bands=[a,b]` selects the bands from index `a` to index `b`, both included, counted from the lowest one. One band is selected, so `hwan` has one orbital per cell. Its centre is close to the site with the lower onsite energy, where the valence band has most of its weight. The spreads are in units of the squared length of the geometry (the first-neighbor distance is 1). A larger gap gives a more localized Wannier function and a smaller Ω . The minimization starts from the orbitals of the original model where the selected bands have most weight, so the same call always gives the same result; `trial_vectors=` sets another starting point.

The bands of `hwan` can be compared with those of the original model:

```
(k,e) = h.get_bands(write=False)
(kw,ew) = hwan.get_bands(write=False)
```

`ew` is equal to the lower band of `e` at the $\mathbf{k}$ -points of the mesh, and between them the Wannier model interpolates smoothly. So we have a model with one orbital that carries the valence band of the model with two orbitals. The quality of this interpolation depends on how localized the Wannier functions are. A band that comes close to another band somewhere in the Brillouin zone has wide Wannier functions, and needs a denser mesh `nk`.

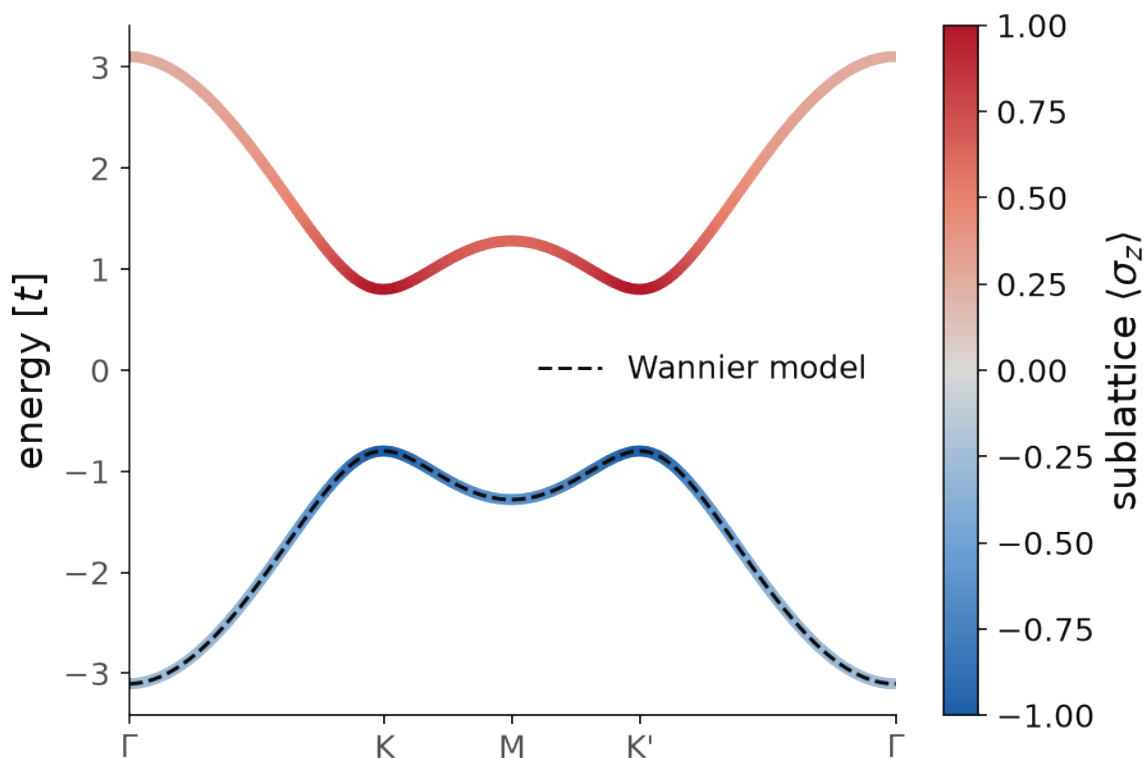


Figure 78: Bands of the gapped honeycomb lattice colored by sublattice, with the band of the one-orbital Wannier model on top.

For the figure the bands of `h` are computed with `operator="sublattice"`, so the color gives the sublattice polarization of each state. The valence band (blue) lives mostly on the sublattice with onsite energy -0.8 , and most strongly at K and K', where the gap is smallest, 2×0.8 . The band of the Wannier model (dashed) lies on top of the valence band along the whole path, from -3.10 at Γ to -0.80 at K. Its Wannier centre is at $(0.5, 0)$, exactly on the site with onsite energy -0.8 , and its spread is $\Omega = 0.32$.

The code of this figure is in `documentation/figures/wannier_valence.py`.

The mesh only fixes each hopping up to a translation by the supercell of `nk` cells. As in Wannier90, each hopping is put on the cell closest to the origin (the Wigner-Seitz cell of that supercell), so the Bloch Hamiltonian of the Wannier model is Hermitian at every k , also between mesh points.

See `examples/wannier/get_wannier_hamiltonian/main.py` for a runnable script, and `jupyter-notebooks/functionalities` and `02_exact_reproduction.ipynb` in the same folder for executed notebooks.

26.2 Symmetry-enforced Wannierization

The Wannier functions should respect the point-group symmetry of the lattice. With `symmetries="auto"`, the point group is found from the geometry and the Hamiltonian, and the code first checks that the selected bands are a union of complete multiplets of this group at every k -point. If the selection cuts through a degeneracy protected by symmetry, the code raises a `ValueError` instead of returning a model with the wrong symmetry. A list of `symmetrytk.pointgroup.SymmetryOperation` can be given instead of "auto", to enforce only a subgroup.

The flat band of the kagome lattice is a good example. It is the lowest of the three bands, at $E = -2$ in units of the hopping, and it touches the middle band at Γ . So a selection with only the flat band is not a union of complete multiplets at Γ . This touching is also the reason why the flat band alone has no exponentially localized Wannier functions:

```
from pyqula import geometry
```

```

from pyqula.symmetrytk import pointgroup

g = geometry.kagome_lattice()
h = g.get_hamiltonian(has_spin=False)

try:
    h.get_wannier_hamiltonian(bands=[0,0],nk=12,symmetries="auto") # the flat band alone
except ValueError as e:
    print("Flat band alone correctly rejected:",str(e).splitlines()[0])

# the full 3-band manifold has no such obstruction
hwan_sym = h.get_wannier_hamiltonian(bands=[0,2],nk=12,symmetries="auto")
print("Symmetries enforced:",[c.op.name for c in hwan_sym.wannier_symmetries])

```

The first call raises an error, and the message gives the symmetry and the k-point, Γ , where the selection breaks a multiplet. The second call gives a model with three orbitals, whose Wannier functions transform into each other under the operations listed in `wannier_symmetries`.

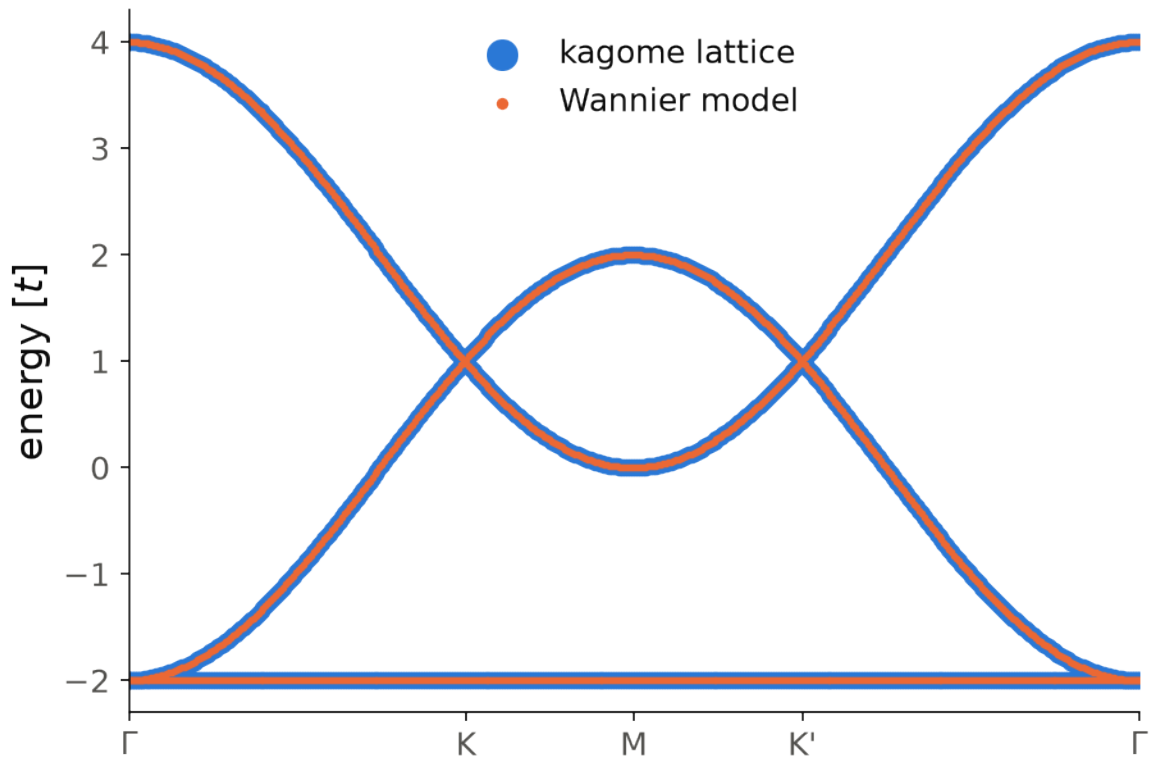


Figure 79: Bands of the kagome lattice, with the bands of the symmetric three-orbital Wannier model on top.

The three bands of the Wannier model (small orange dots) lie on the bands of the kagome lattice (blue): the flat band at $E = -2$, the Dirac crossings at K and K' at $E = 1$, and the top of the band at $E = 4$ at Γ . The largest difference between the two sets of energies along the path is 4×10^{-10} . You also see the touching at Γ between the flat band and the middle band, which is the reason why the flat band alone is rejected.

The code of this figure is in `documentation/figures/wannier_kagome.py`.

See `examples/wannier/symmetric_wannierization/main.py` for a runnable script, and `jupyter-notebooks/functionality` for the executed notebook.

26.3 Disentanglement

A selection of bands only works if these bands are separated from all the others by a gap in the whole Brillouin zone. Often they are not. The simplest case is graphene: its two p_z bands touch at K, so neither band alone is a smooth set of states. Disentanglement solves this. From a larger set of offered bands it extracts, at each k-point, the subspace of `num_wann` states that changes most smoothly with k, meaning that it minimizes the gauge-invariant part Ω_I of the spread. This is the method of Souza, Marzari and Vanderbilt. It is switched on by giving a `num_wann` smaller than the number of selected bands; then `bands=[a,b]` only says which bands are offered.

```
import numpy as np
from pyqula import geometry
g = geometry.honeycomb_lattice()
h = g.get_hamiltonian(has_spin=False)

# one Wannier function out of both pz bands, with the deep part of the
# valence band frozen (kept exactly rather than optimized)
hwan = h.get_wannier_hamiltonian(bands=[0,1],num_wann=1,nk=12,
                                dis_froz_max=-1.0,cutoff=0.0)

print("Wannier functions:",hwan.wannier_num_wann)
print("window used:",hwan.wannier_disentanglement_window)

f0,fw = h.get_hk_gen(),hwan.get_hk_gen()
k = [2./12.,1./12.,0.] # a wannierization-mesh k-point
e0 = np.sort(np.linalg.eigvalsh(f0(k))) ; ew = np.linalg.eigvalsh(fw(k))
print("frozen state reproduced to:",abs(ew[0]-e0[0]))
```

The model now reproduces exactly only the states inside the frozen window, below $E = -1$ here, at every k-point of the mesh. Outside the frozen window it does not reproduce the offered bands: it follows a smoother subspace instead. So the last line prints a number at machine precision for the frozen valence state, while the conduction band is not in the model at all. This is the price of a smooth, localized orbital, not an error. With `cutoff=0.0` all the real-space hoppings are kept; the default `cutoff=1e-6` drops the small ones and changes the last digits.

Below $E = -1$ (shaded, the frozen window) the Wannier band (orange) is on top of the valence band of graphene (blue), from -3 at Γ to -1 at M. Above $E = -1$ it leaves the valence band. Near K and K' it does not follow the kink of the Dirac cone; it goes smoothly through a maximum close to $E = 0$ instead. This smooth band is what makes the Wannier function localized. With these keywords the call also prints a warning, because the minimization of Ω_I is not converged after the default 200 iterations; the frozen window is reproduced anyway.

The code of this figure is in `documentation/figures/wannier_disentangle.py`.

The keywords are:

- `num_wann`: number of Wannier functions; smaller than the number of selected bands switches on disentanglement.
- `dis_froz_min`, `dis_froz_max`: the frozen (inner) window; states inside it are reproduced exactly. Give both, or only `dis_froz_max`.
- `dis_win_min`, `dis_win_max`: the outer window; only the states inside it are offered at each k-point, so their number can change across the Brillouin zone.
- `dis_num_iter`: maximal number of iterations of the Ω_I minimization (default 200); if it is not converged you get a warning, and the frozen window is reproduced anyway.

The returned Hamiltonian has the attributes `wannier_num_wann` and `wannier_disentanglement_window` (the four window values used, or `None` without disentanglement), together with the centres and spreads.

A window without a smaller `num_wann`, `dis_froz_min` without `dis_froz_max`, or a `num_wann` outside 1 to the number of bands raises an error, since the window would be ignored otherwise. Disentanglement does not work together with a Nambu (BdG) Hamiltonian, with `symmetries=`, or with `auto_split_clusters=True`; these

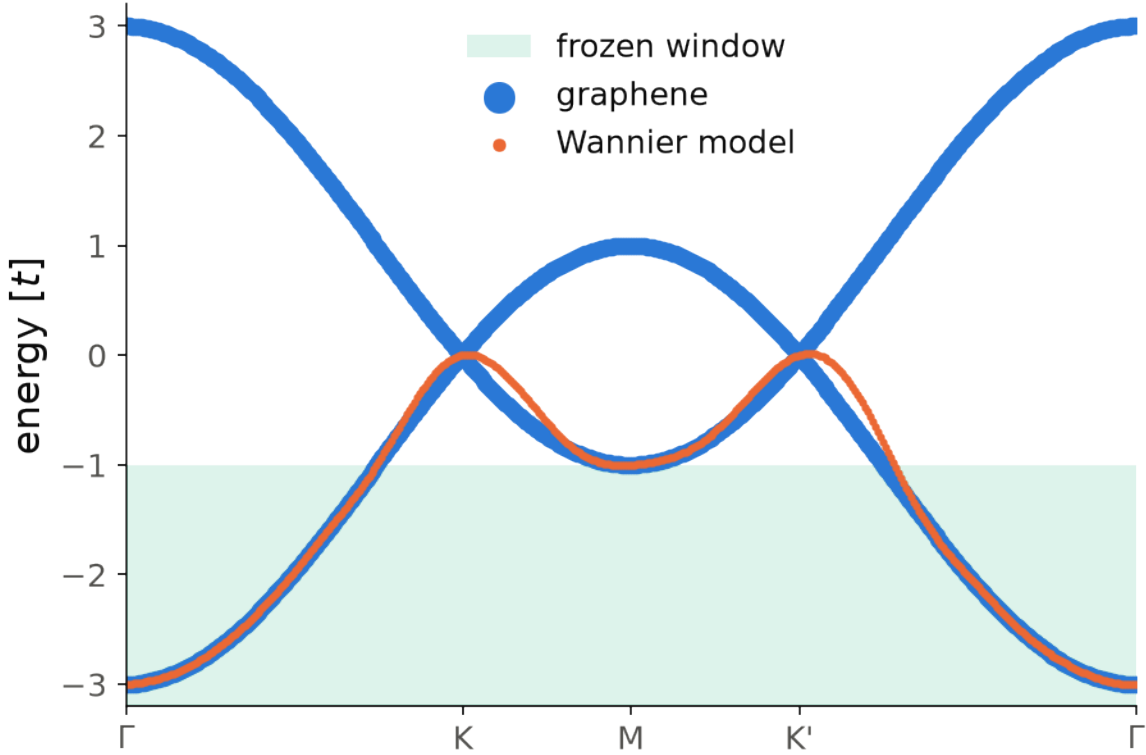


Figure 80: Bands of graphene and the band of the disentangled one-orbital Wannier model, with the frozen window shaded.

combinations raise a `NotImplementedError`. Always give the window with the `dis_*` keywords and not through `win_keywords=`, since this last one is passed on without any check.

See `jupyter-notebooks/functionalities/wannierization/04_dimensionality_and_bdg.ipynb` for Wannierization in other dimensions and in the Nambu basis, and `05_band_disentanglement.ipynb` in the same folder for disentanglement.

27 Twisted bilayer graphene structural relaxation

In a twisted graphene bilayer the local stacking changes along the moiré pattern, between AA regions, where the atoms of the two layers are on top of each other, and AB and BA (Bernal) regions. AA stacking costs more energy than AB and BA. So for small twist angles, below a few degrees, the real lattice does not stay rigid: the atoms move in the plane, the AA regions shrink, and triangular AB and BA domains grow, separated by domain walls. This relaxation changes the band structure, so it should be included before the electronic structure is computed. We do this with `GrapheneGeometry` and its method `.relax()`.

27.1 The energy that is minimized

`.relax()` finds the in-plane displacement $\mathbf{u}$ of each atom that minimizes the sum of two energies, with the parameters of Carr, Massatt, Torrisi, Cazeaux, Luskin and Kaxiras, [arXiv:1805.06972](https://arxiv.org/abs/1805.06972).

The first one is the interlayer energy, the generalized stacking fault energy. It depends on the local shift between the two layers, written as two phases (v, w) , so that it is periodic in the shift:

$$V(v, w) = c_0 + c_1 [\cos v + \cos w + \cos(v + w)] + c_2 [\cos(v + 2w) + \cos(v - w) + \cos(2v + w)] + c_3 [\cos 2v + \cos 2w + \cos(2v + 2w)],$$

per unit cell, with $c_0 = 6.832$, $c_1 = 4.064$, $c_2 = -0.374$ and $c_3 = -0.095$ meV. AA stacking is its maximum, and AB and BA are its two degenerate minima.

The second one is the elastic energy of each layer, which is the cost of deforming it,

$$E_{\text{el}} = \frac{1}{2} [G (\partial_x u_x + \partial_y u_y)^2 + K ((\partial_x u_x - \partial_y u_y)^2 + (\partial_x u_y + \partial_y u_x)^2)],$$

per unit cell, with $G = 69518$ meV and $K = 47352$ meV. The first term is the cost of a change of area, the second one the cost of a shear. The atoms only move in the plane; corrugation out of the plane is not included.

The result is a balance between the two: the interlayer energy wants large AB and BA domains, and the elastic energy wants a smooth displacement. For small angles the moiré cell is large, so the domains can grow while the deformation stays smooth, and the relaxation is strong.

27.2 The electronic structure of the relaxed lattice

`GrapheneHamiltonian` builds the tight-binding Hamiltonian from a relaxed or a rigid `GrapheneGeometry`. Its hopping between two sites depends on their distance r in three dimensions,

$$t(\mathbf{r}) = -t_{\parallel} \frac{r_{\parallel}^2}{r^2} e^{-\lambda(r-1)} e^{-\lambda_z r_z^2} - t_{\perp} \frac{r_z^2}{r^2} e^{-\lambda_{\perp}(r-d)},$$

where $r_{\parallel}$ and r_z are the in-plane and out-of-plane parts of $\mathbf{r}$. The first term is the in-plane hopping, $t = 1$, and the second one the interlayer hopping, $t_{\perp} = 0.12$, with an interlayer distance $d = 3$ in units of the carbon-carbon distance. Note that the hopping here is $-t$, the opposite sign of the $+1$ of `g.get_hamiltonian()`, so the bands of the two Hamiltonians are mirrored in energy. Because the hoppings depend on the real distances, the relaxed positions change the electronic structure without any other change.

```
from pyqula import specialgeometry
from pyqula.graphenetc.geometry import GrapheneGeometry
from pyqula.graphenetc.hamiltonian import GrapheneHamiltonian

g0 = specialgeometry.twisted_bilayer(m0=15) # ~2 degree twist
g = GrapheneGeometry(g0).relax() # AA area shrinks, AB/BA domains grow
h = GrapheneHamiltonian(g)
(k,e) = h.get_bands(num_bands=20)
```

`.relax()` returns a new geometry and does not change the rigid one, so you can compare the two. `GrapheneGeometry` works for any graphene multilayer (bilayer, twisted bilayer, twisted trilayer). The relaxation is stronger for smaller twist angles. The band structure of the relaxed lattice is the one to compare with an experiment on a real sample, for example with ARPES or with the local density of states measured by STM. The relaxation takes a few seconds, while `h.get_bands(num_bands=20)` on the 2884 atoms of this moiré cell takes several minutes, about 8 on a laptop.

The left panel shows the displacement $\mathbf{u}$ of the atoms of the top layer, the difference between the positions of $\mathbf{g}$ and of $\mathbf{g0}$, over several moiré cells. The atoms turn around each AA region (the dark centres of the rings, at the origin and at its copies by the moiré lattice vectors), and the bottom layer turns in the opposite direction. The displacement is zero at AA and at the centres of the AB and BA domains (the dark points between the rings), and it is largest, 0.025 in units of the carbon-carbon distance, on a ring around AA. So the relaxation is mostly a local rotation of the two layers around each AA region.

The right panel shows the bands of the relaxed lattice (orange) and, as a reference, of the rigid one (gray), with the default `nk=400` points on the k-path, as in the snippet. The two bands closest to zero energy are the central bands of the moiré, with their Dirac points at K and K'. The relaxation changes them by about 0.004 in units of the hopping: at M they go from -0.024 and 0.029 to -0.028 and 0.033 , and the levels closest to zero at Γ go from -0.057 and 0.068 to -0.062 and 0.072 . The next bands come closer to the central ones: at M they go from -0.083 and 0.090 to -0.076 and 0.082 .

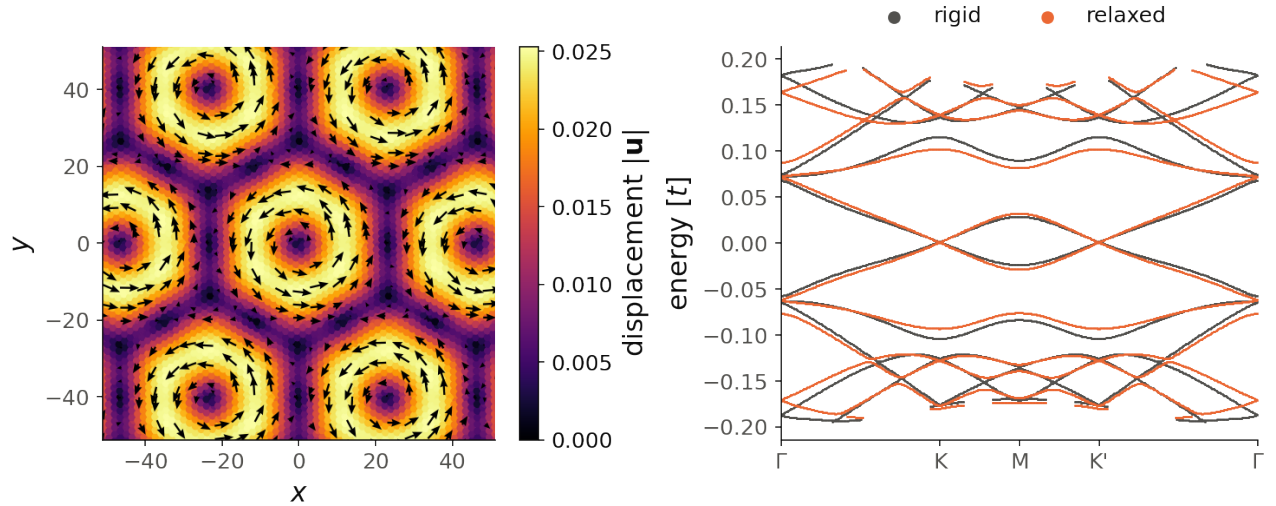


Figure 81: In-plane displacement of the atoms of the top layer of the relaxed twisted bilayer, and the bands of the rigid and of the relaxed lattice.

The code of this figure is in `documentation/figures/relaxation_twisted.py`.

See `examples/2d/graphene_relax/main.py` for a runnable script that compares the rigid and the relaxed lattices, and `jupyter-notebooks/functionalities/single_particle_hamiltonians/10_twisted_bilayer_relaxation.ipynb` for an executed notebook.

28 Classical spin models

All the other pages of this guide treat quantum models of electrons. Here we treat classical spins: unit vectors $\vec{S}_i$ on the sites of a lattice, with `classicalspin.SpinModel`. This is the right description of a magnet with large local moments at the classical level, and it gives the magnetic ground state, for example a spiral or a skyrmion texture, which can then be used as an exchange field in a quantum Hamiltonian.

28.1 The energy

Each spin is given by two angles, $\vec{S}_i = (\sin \theta_i \cos \phi_i, \sin \theta_i \sin \phi_i, \cos \theta_i)$, and the energy is

$$E = \sum_{i \neq j} \vec{S}_i \cdot J_{ij} \cdot \vec{S}_j + \sum_i \vec{b}_i \cdot \vec{S}_i,$$

where J_{ij} is a 3×3 exchange tensor and $\vec{b}_i$ a magnetic field. The sum runs over i and j separately, so every bond appears twice, once as (i, j) and once as (j, i) . A positive J is antiferromagnetic. Note the sign of the field term: the energy is lowest when the spin points opposite to $\vec{b}$.

The model uses the lattice and the neighbor shells of a `Geometry`, but it has nothing else in common with the quantum `Hamiltonian`. The ground state is found by minimizing E over the angles, with a local gradient minimization started several times from random angles. The spin configuration repeats with the unit cell you give, so a spiral or any other texture with a longer period needs a supercell large enough to contain it.

28.2 The 120-degree state

The simplest frustrated case is the triangular lattice with an antiferromagnetic first-neighbor exchange. The three bonds of a triangle cannot be all antiparallel at the same time.

```

from pyqula import geometry
from pyqula import classicalspin

g = geometry.triangular_lattice() # geometrically frustrated lattice
g = g.get_supercell(3)

sm = classicalspin.SpinModel(g) # classical spin model on this geometry
sm.add_heisenberg(Jij=[1.0]) # first-neighbor antiferromagnetic exchange
sm.minimize_energy(tries=10) # multistart local minimization

mx,my,mz = sm.get_magnetization() # per-site magnetization components

```

`mx`, `my` and `mz` are the three components of the spin on each site. The result is the 120-degree state. Two neighboring spins form an angle of 120 degrees, so $\vec{S}_i \cdot \vec{S}_j = -1/2$ on every bond, the three spins of a triangle add up to zero, and so does the total magnetization. The energy is $-3J$ per site: six neighbors, $-J/2$ per bond, and each bond counted twice. The 3×3 supercell is needed because this state has three sublattices; on the primitive cell the only possible configuration is the ferromagnetic one. `tries` is the number of random starting points. A frustrated model has many local minima, so more starting points make it more likely to find the lowest one, and the lowest energy found is kept.

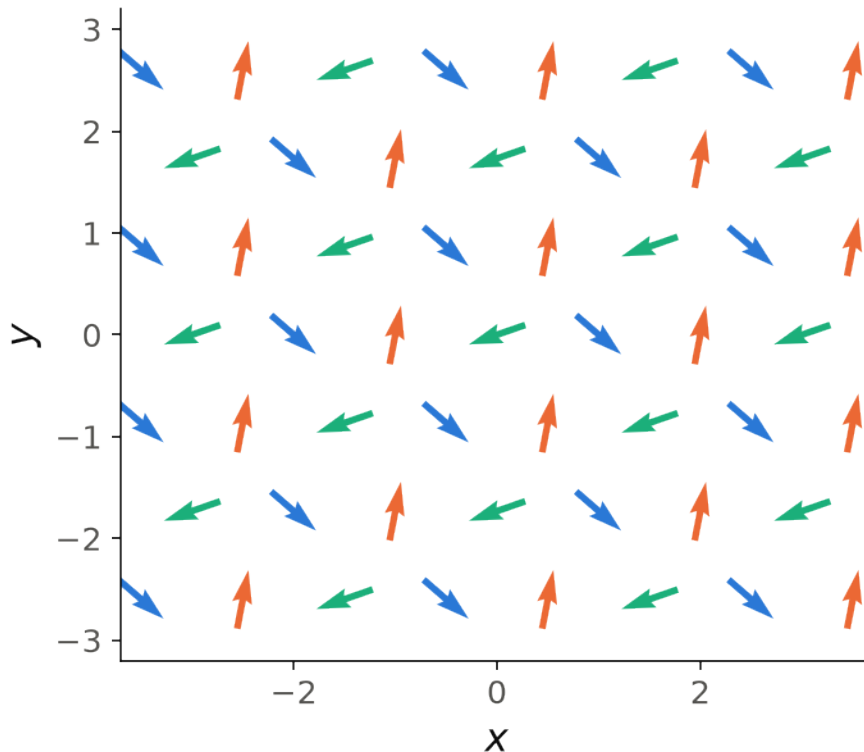


Figure 82: Spins of the 120-degree state on the triangular lattice, drawn as arrows in the plane of the spins and colored by sublattice.

The arrows are the spins, rotated with `most_perp_basis()` into the frame where they lie in the plane of the figure, and the 3×3 supercell is repeated to fill the window. The three colors are the three sublattices. The spins of each sublattice are parallel, and the three directions are at 120 degrees from each other, so every triangle has one spin of each color. The minimum energy printed by the run is -27 for the 9 sites, so $-3J$ per site, the value found above. The common direction of the spins is random, since the energy does not depend on it; the script fixes the random seed, so it is the same in every run.

The code of this figure is in `documentation/figures/classicalspins_120.py`.

28.3 Other couplings

- `add_heisenberg(Jij=[J1,J2,...])`: isotropic exchange to first, second, third neighbors, as in `g.get_hamiltonian(tij=...)`.
- `add_heisenberg(..., Jm=[Jx,Jy,Jz])`: anisotropic diagonal exchange, for example an XXZ model.
- `add_field(v)`: the same field $\vec{b}$ on every site.
- `classicalspin.generating_functions(name=...)`: ready-made couplings between two positions, "Linear" (dipolar, $1/r^3$), "RKKYTI" (RKKY on the surface of a topological insulator), "ZZ" and "XYZ" (Ising and anisotropic diagonal), "DM" (Dzyaloshinskii-Moriya).
- `add_tensor(fun)`: add the coupling `fun` between the sites of the unit cell.
- `add_tensor_2d(fun,ncells=...,vspiral=...)`: the same, summed over the periodic images, with an optional rotation of the coupling from image to image to describe a spin spiral with wavevector `vspiral`.
- `get_local_energy()`: the energy of each site, for a map of a skyrmion or a domain wall.
- `classicalspintk.align.most_perp_basis()`: rotates a texture into the frame where it is mostly in-plane, the frame to draw it with arrows.

See `examples/classicalspin/` for runnable scripts, among them the 120-degree state drawn with arrows and its local energy (`perpendicular/`), and a ladder with a modulated exchange whose texture is then used in a quantum Hamiltonian (`modulated_ladder/`). See `jupyter-notebooks/functionalities/classical_spin_models_monte_carlo/01_c` and `02_magnetization_textures.ipynb` in the same folder for executed notebooks.

29 Lattice gas models

A lattice gas describes classical particles that either occupy a site or not: adsorbates on a surface, vacancies in a crystal, or any classical order parameter with two values. We treat it with `latticegas.LatticeGas`.

29.1 The energy and the Monte Carlo sampling

The occupations are $n_i \in \{0, 1\}$, and the energy is

$$E = \sum_{i \neq j} J_{ij} n_i n_j + \sum_i \mu_i n_i,$$

where J_{ij} is the interaction between two occupied sites and μ_i is a chemical potential on each site. A positive J_{ij} is a repulsion. The sum runs over i and j separately, so each bond is counted twice. As for the classical spin models, the lattice and the neighbor shells come from a `Geometry`, and nothing else from the quantum Hamiltonian.

The configurations are sampled with Monte Carlo and the Metropolis rule. A move, the exchange of the occupations of two sites, changes the energy by ΔE , and it is accepted with probability $\min(1, e^{-\Delta E/T})$, with the temperature T in the same units as J ($k_B = 1$). These exchange moves keep the number of particles fixed, so the filling, the fraction of occupied sites, does not change.

29.2 Ordering of repulsive particles

We take a triangular lattice with one third of the sites occupied, and a repulsion to first, second and third neighbors, so a pair of particles at the distance of first, second or third neighbors costs an energy $J = 1$:

```
from pyqula import geometry
from pyqula import supercell
from pyqula import latticegas

g = geometry.triangular_lattice()
g = supercell.turn_orthorhombic(g)
g = g.get_supercell(10) # periodic, so no site sits at an edge

lg = latticegas.LatticeGas(g,filling=1./3.) # 1/3 of the sites randomly occupied
```

```
lg.add_interaction(Jij=[1.,1.,1.]) # first, second and third neighbor repulsion
es = lg.optimize_energy(temp=0.5,ntries=1e4) # Metropolis run at fixed temperature
```

`lg.den` is the occupation of each site, 0 or 1, and `es` is the energy along the Monte Carlo run. The energy goes down and then becomes flat once the particles have found their arrangement. The supercell is periodic, so every site has the same neighbors as in the bulk. With `g.dimensionality = 0` before building the model, the bonds across the border are cut instead, and we get a finite flake, where the sites at the edge have fewer neighbors and the order near the edge is different.

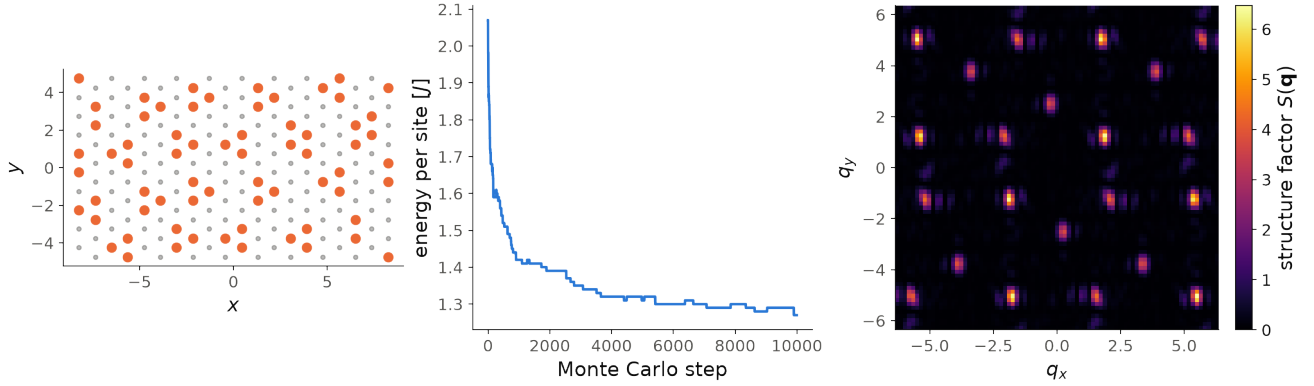


Figure 83: Occupied sites of the lattice gas after the run, the energy per site along the run, and the structure factor over a grid of wavevectors.

For the figure we also compute `lg.get_structure_factor(nq=81)`, explained in the next paragraph. The energy per site (middle) goes down from 2.07 for the random start to 1.27, and after about 4000 moves it only decreases slowly. At one third filling the particles cannot avoid all the three shells of neighbors. In the run of the figure each occupied site (left, orange) has on average 2.0 occupied first neighbors, 1.1 second and 0.66 third neighbors, so the particles form chains, and the energy per site is $(2.0 + 1.1 + 0.66)/3 \approx 1.27$. The structure factor (right) is zero at $\mathbf{q} = 0$, and it has sharp peaks at finite $\mathbf{q}$, so the occupations are ordered. The script fixes the random seed. With another seed the positions of the peaks and the final energy change a little: one run of 10^4 moves is 50 moves per site, and it can stop with domains of different orientations.

The code of this figure is in `documentation/figures/latticegas_order.py`.

To find which order the particles form, look at the density-density correlator in real space, `get_correlator()`, and in reciprocal space, `get_structure_factor()`. The structure factor has peaks at the ordering wavevector of the ordered state, for example a stripe order or a honeycomb arrangement of vacancies. This is the quantity that diffraction measures.

29.3 Keywords and other methods

- `temp`: temperature T ; higher values accept more moves that increase the energy.
- `ntries`: number of attempted moves.
- `anneal()`: runs `optimize_energy()` with a decreasing temperature and keeps the best configuration, so the system does not stay in a local minimum.
- `optimize_energy_multistart()`: keeps the best of several independent runs.
- `optimize_grand_canonical()`: flips single occupations at chemical potential `lg.mu`, so the filling changes; use it to scan the filling as a function of the chemical potential.
- `get_specific_heat()`, `get_susceptibility()`: thermodynamic quantities from the fluctuations along a run at fixed temperature.
- `get_local_energy()`, `get_local_mu()`: the energy of each site, and the cost $\mu_i + \sum_k J_{ik} n_k$ of occupying it, in the current configuration.
- `add_tensor()`: couplings that are not given by neighbor shells.
- `write()`, `read()`: save a configuration to a file and read it back.

See `examples/latticegas/` for runnable scripts of the annealing, the map of the local energy, the correlators, and the scan of the filling against the chemical potential, and `jupyter-notebooks/functionalities/classical_spin_models_mon` and `05_correlators_thermodynamics.ipynb` in the same folder for executed notebooks.

30 Ising models

The Ising model has a classical spin $s_i \in \{-1, +1\}$ on each site. We treat it with `latticeising.LatticeIsing`, which works in the same way as the lattice gas.

30.1 The energy

The energy is

$$E = - \sum_{i \neq j} J_{ij} s_i s_j - \sum_i b_i s_i,$$

where J_{ij} is the coupling between two spins and b_i is a magnetic field on each site. A positive J_{ij} is ferromagnetic, it favors parallel spins. Note that the sign convention is the opposite of the lattice gas, where the coupling has no minus sign and a positive J is a repulsion.

The sum runs over i and j separately, so each bond is counted twice, and the energy is two times the usual sum over bonds. As a result, all the energies and temperatures are two times larger than in the usual convention. For example, the ferromagnet on the square lattice orders at $T_c = 2 \times 2.269 J \approx 4.54 J$, not at $2.269 J$.

The spins are sampled with Monte Carlo and the Metropolis rule, as in the lattice gas: a move that changes the energy by ΔE is accepted with probability $\min(1, e^{-\Delta E/T})$, with $k_B = 1$.

30.2 The ferromagnetic transition

The simplest case is a ferromagnet on the square lattice. The lattice is bipartite, so nothing frustrates the order:

```
from pyqula import geometry
from pyqula import latticeising

g = geometry.square_lattice() # bipartite, so ferromagnetic order is not frustrated
g = g.get_supercell(12) # periodic, so no site sits at an edge

li = latticeising.LatticeIsing(g,m=0.0) # random +-1 spins, zero net magnetization
li.add_interaction(Jij=[1.]) # first-neighbor ferromagnetic coupling
es,ms = li.anneal(temps=[3.,1.,0.3,0.1,0.03],ntries=1e4) # simulated annealing
```

`li.s` contains the spins, ± 1 , `es` the energy along the run, and `ms` the total magnetization $\sum_i s_i$ along the run. In `ms` you see the phase transition. All the temperatures of the snippet are below $T_c \approx 4.54 J$, so already in the first stage, at $T = 3$, the magnetization per site `ms/li.nsites` grows from zero to $+1$ or -1 , all the spins parallel, and the sign is chosen at random. To see the disordered phase, where the magnetization fluctuates around zero, add a first stage above T_c , as in the figure below. As in the lattice gas, the supercell is periodic, and `g.dimensionality = 0` makes it a finite flake.

For the figure we add a first stage at $T = 6$, above T_c , and we keep the spins at its end with `checkpoint_at`. At $T = 6$ the magnetization per site (left) fluctuates around zero, between -0.6 and 0.5 on this small lattice, and the spins form domains of both signs (middle). At $T = 3$ it grows to $+1$ after about 3000 steps, and it stays there at the lower temperatures. At the end all the spins are parallel (right), and the energy is $-4J$ per site, two times the $-2J$ of the usual sum over bonds.

The code of this figure is in `documentation/figures/ising_ferro.py`.

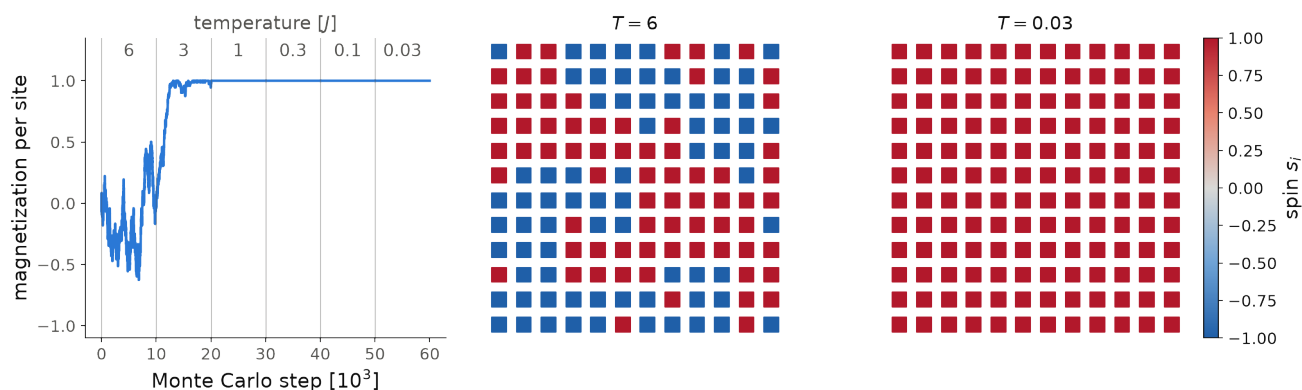


Figure 84: Magnetization per site along the annealing run, and the spins at the end of the first and of the last temperature.

30.3 Keywords and other methods

- `li.optimize_energy()`: Metropolis with single spin flips; the magnetization is not conserved. Returns the energy and the total magnetization along the run, (`es`, `ms`), and `ms` can be given to `latticegas.get_susceptibility()`.
- `li.optimize_conserved()`: Kawasaki dynamics, exchanges of two spins; the total magnetization is conserved, as the filling in the lattice gas.
- `li.anneal()`: `optimize_energy()` with a decreasing temperature, given by `temps`.
- `optimize_energy_multistart()`: keeps the best of several independent runs.
- `get_local_energy()`, `get_local_field()`: the energy and the effective field of each site in the current configuration.
- `get_correlator()`, `get_structure_factor()`: spin-spin correlator in real and reciprocal space; the peak gives the ordering wavevector, Γ for a ferromagnet and M for the checkerboard antiferromagnet.
- `add_tensor()`, `regroup()`, `get_specific_heat()`, `get_susceptibility()`: the same as in the lattice gas.

See `examples/latticeising/` for runnable scripts of the annealing, a scan in temperature (magnetization and specific heat), and maps of the local energy and field of a frustrated antiferromagnet on the triangular lattice, and `jupyter-notebooks/functionalities/classical_spin_models_monte_carlo/04_ising_monte_carlo.ipynb` and `05_correlators_thermodynamics.ipynb` in the same folder for executed notebooks.

31 Parallelism and reproducibility

Many calculations are loops over independent items: k-points for a band structure, energies for a density of states, q-points for a response function, restarts for an annealing. By default they run one after the other. `parallel.set_cores(n)` runs them on `n` processes:

```
from pyqula import parallel
parallel.set_cores(4)           # spread the sweeps over 4 processes
```

The number of cores changes only the time of the calculation, not the result. This is also true for the calculations that use random numbers, such as the stochastic trace of the kernel polynomial method or the restarts of the classical models: `cores=1` and `cores=8` give the same numbers. So if you fix the random seed of numpy in your script,

```
import numpy as np
np.random.seed(42)
```

the whole calculation gives the same result every time. This is useful when you compare two calculations that differ in one parameter only.

`parallel.set_enabled(False)` makes everything run on one core, with no parallel processes and no threads. Use it to debug a calculation, or to measure its time on a single core. `parallel.set_enabled(True)` allows

parallel runs again, but it does not restore the previous number of cores, so call `parallel.set_cores(n)` again.

32 Running on a GPU

Some of the heaviest calculations can run on a GPU: the Chebyshev moments of the kernel polynomial method, the sum over states of the spin and charge response, and the diagonalization of the Bloch Hamiltonian on a k-mesh, which is behind band structures, densities of states and Berry curvatures. One switch decides where they run, and by default everything runs on the CPU:

```
from pyqula import gpu
gpu.set_gpu(True) # every routine with a GPU path now uses the device
```

`gpu.set_gpu(False)` goes back to the CPU, and `gpu.is_gpu_available()` says if a GPU is found, so `gpu.set_gpu(gpu.is_gpu_available())` uses the GPU when there is one. A script that never calls `gpu.set_gpu()` gives the same result on every machine. If you ask for a GPU and there is none, the same code runs on the CPU with a warning, and gives the same numbers.

32.1 When the GPU helps

The gain depends on the kind of calculation:

- The response functions (the Lindhard sum) gain the most. Their cost grows as the fourth power of the number of sites in the unit cell, and the whole k-mesh is computed at once.
- The diagonalization on a k-mesh only gains for Hamiltonians with about 32 orbitals or more; for smaller ones it stays on the CPU even with the switch on.
- The kernel polynomial density of states gains little, a factor 1 to 3.
- The KPM mean field gains more: on a consumer card, an iteration for a unit cell with one to three thousand orbitals is 6 to 10 times faster than on the CPU.
- The static polarizability of the screened interaction (`get_polarizability`, `get_screened_interaction`, and `get_bse(screening=...)`) is faster on a consumer card only in single precision, from about 36 orbitals on, by about 7 times. In double precision it is slower than the CPU at every size.

32.2 Precision

The precision is chosen in each call, since it changes the numbers and not only the place where they are computed: `kpm_prec` for the Chebyshev moments, `chi_prec` for the response functions, and `eigh_prec` for the diagonalization, each "single" or "double". The default is double precision, except for the response functions and the KPM mean field on the GPU, which use single precision, because a consumer GPU can be ten times slower in double precision. Single precision keeps about six digits. This is enough for a density of states or a band structure, but not for a quantity computed as the difference of two numbers that are almost equal.

A few routines exist only in jax, for example the mean-field solvers with `use_jax=True`, the graphene relaxation and the jax Keldysh current. They follow the same switch, so they also run on the CPU by default.

See `jupyter-notebooks/functionalities/interacting_mean_field_hamiltonians/22_gpu_execution.ipynb` for an executed notebook that compares the CPU and the GPU for each case.

33 Files written to the working directory

Many methods that compute a quantity also write it to a file in the working directory: `BANDS.OUT` for the band structure, `DOS.OUT` for the density of states, `CHERN.OUT` for the Chern number, and so on. This is useful for a script that is plotted later, but in a notebook or inside a larger program the returned arrays are enough, and the files are not needed. Each of these methods has a keyword `write` with its own default, and `filewrite.set_write()` sets it for all of them at once:

```
from pyqula import geometry
from pyqula import filewrite
filewrite.set_write(False) # no routine writes its files from now on
```

```

g = geometry.honeycomb_lattice() # honeycomb lattice
h = g.get_hamiltonian() # first-neighbor tight-binding Hamiltonian
(k,e) = h.get_bands() # the band structure, and no BANDS.OUT
(k,e) = h.get_bands(write=True) # a write given in the call wins over the switch
filewrite.set_write(None) # back to the default of each routine

```

The rule is always the same: first the `write` given in the call, then the switch, and only if neither is set, the default of the method. So `filewrite.set_write(True)` makes every method write its file, also the few that do not write by default, such as `topology.chern_density()`, and `write=False` in one call keeps that call silent, whatever the switch says. The returned arrays do not depend on this, only the files do. The methods whose only job is to write a file, such as `g.write()` or `h.write_hopping()`, always write.

34 Errors and unsupported inputs

Many calculations only make sense for some Hamiltonians: a Berry curvature needs a two-dimensional Hamiltonian, a spin operator needs the spin degree of freedom, a superconducting quantity needs the Nambu one, and a keyword such as `mode=`, `solver=` or `channel=` accepts only some strings. `pyqula` checks these conditions before the calculation starts, and the error message says what is needed:

- `ValueError`: a value that cannot work, for example an unknown `mode`, or a spin operator on a spinless Hamiltonian.
- `NotImplementedError`: a combination that is not available yet.
- `TypeError`: an argument of the wrong type.

For a keyword given as a string, the message lists the accepted values, so a typo is easy to find:

```
ValueError: unknown mode ED2; the DOS accepts 'ED', 'KPM', 'adaptive', 'Green' and 'RG'
```

Two messages tell you directly what to do. A superconducting quantity on a normal Hamiltonian asks you to call `h.setup_nambu_spinor()` first, and a spin quantity on a spinless Hamiltonian asks you to call `h.turn_spinful()`:

```

from pyqula import geometry
h = geometry.honeycomb_lattice().get_hamiltonian(has_spin=False)
h.extract("mz")
# ValueError: the magnetization 'mz' needs a spinful Hamiltonian; call
# h.turn_spinful() first, or build it with g.get_hamiltonian(has_spin=True)

```

The check is only on reading a degree of freedom that the Hamiltonian does not have. The methods that add a spin term, such as `h.add_exchange()`, `h.add_zeeman()` and `h.add_kane_mele()`, make a spinless Hamiltonian spinful instead of raising an error (see Building a Hamiltonian). In the same way, quantities resolved by sublattice need a unit cell with sublattice labels, and `h.extract("CDW")` on a cell without them raises an error instead of returning `None`.

Some other checks exist because the calculation could run, but the number it gives would have no meaning:

- `h.get_average_spin_splitting()`, `h.get_spin_splitting_density()` and `h.get_spin_splitting_vs_energy()` need S_z to be conserved, and raise an error for a Hamiltonian with spin-orbit coupling or non-collinear order, giving the largest element that mixes the spins.
- `h.get_dos()` of a non-Hermitian Hamiltonian only works with `mode="ED"`, since the other modes assume real energies (see Non-Hermitian Hamiltonians).
- `h.get_total_energy(fermi=...)` does not accept a Nambu Hamiltonian, because there a Fermi energy moves the electron and hole bands in opposite directions; use `h.shift_fermi(-mu)` on the Hamiltonian instead.
- The surface density of states does not accept an operator that depends on momentum, such as "valley", since the momentum is already integrated.
- The lead Green's functions behind `HT.didv()`, `h.get_dos(mode="RG")` and `Embedding.get_gf()` raise an error for an energy exactly on a level of the lead with a broadening below about $1e-7$; use a larger `delta`.

35 Geometry functions and methods

35.1 `g.get_hamiltonian()`

Generate the tight-binding Hamiltonian of the geometry. By default it has first-neighbor hopping $t = 1$ and is spinful. See Building a Hamiltonian.

Optional arguments

- `tij=None`: the hoppings. `None` is first-neighbor hopping $t = 1$. A list gives the 1st, 2nd, 3rd... neighbor hoppings, for example `[1.0, 0.2]`. It can also be a function of the two positions that returns the hopping between them, or a `specialhopping.HoppingGenerator`
- `has_spin=True`: include the spin degree of freedom
- `is_sparse=False`: store the matrices as sparse matrices, for large unit cells
- `non_hermitian=False`: build a non-Hermitian Hamiltonian (see Non-Hermitian Hamiltonians)
- `is_multicell=False`: include hoppings to unit cells beyond the first neighbor cells
- `nc=2`: how many neighbor cells are included when `is_multicell=True`

A misspelled keyword raises `TypeError` with the list of accepted ones.

Returns the Hamiltonian

35.2 `g.remove()`

Return a copy of the geometry with some sites removed. Use it to make a vacancy or an antidot in a flake or in a supercell, before building the Hamiltonian (see Operators for an example).

Arguments

- `i=0`: the sites to remove: one index of `g.r`, a list of indices, or a function of the position, which removes every site where it returns `True`

Returns a new geometry; `g` is not changed

35.3 `g.get_supercell()`

Generate a supercell

Arguments

- `nsuper`: size of the supercell. A number or a tuple multiplies the lattice vectors. A 3x3 integer matrix M gives a general supercell, with lattice vectors $\mathbf{a}'_i = \sum_j M_{ij} \mathbf{a}_j$ (see Band unfolding)

Optional arguments

- `store_primal=False`: keep the primitive geometry inside the supercell. It is needed to unfold the bands with `operator="unfold"` (see Band unfolding)

For a 2D geometry, a non-integer `nsuper` gives a supercell whose area is `nsuper`² times the primitive one, so `np.sqrt(3)` gives the $\sqrt{3} \times \sqrt{3}$ cell.

Returns a new geometry

35.4 `g.get_unfolded_kpath()`

Return the k-path of the primitive cell, written in the reduced coordinates of the supercell. This is the path along which an unfolded band structure is computed (see Band unfolding). The same method exists as `h.get_unfolded_kpath()`. The supercell must be built with `store_primal=True`.

Arguments

- `kpath=None`: as in `g.get_kpath()`, a list of k-points or a list of high-symmetry labels, both read in the primitive cell
- `g0=None`: the primitive geometry, if the supercell does not contain it

- nk: number of k-points along the path

Returns an array of k-points, one per row, to give as `kpath=` to `h.get_bands()` or `h.get_kdos_bands()`

36 Hamiltonian functions and methods

The methods `h.add_...()` change `h` in place. The methods `h.get_...()` return a result and leave `h` unchanged. Energies are in units of the first-neighbor hopping t , and the Fermi energy is at zero energy.

36.1 `h.get_bands()`

Compute the band structure $\epsilon_n(\mathbf{k})$ along a k-path. See Band structures and DOS.

Optional arguments:

- nk=400: number of k-points along the path
- operator: an operator, or a list of operators. For each eigenstate the expectation value $\langle \Psi_n(\mathbf{k}) | A | \Psi_n(\mathbf{k}) \rangle$ is also returned
- kpath: the k-path, as a list of k-points in reduced coordinates or as a list of high-symmetry labels ("G", "M", "K", "X", "Y", and in three dimensions "Z", "R", "A", "B")
- ewindow: a function of the energy that says which bands to keep, for example `ewindow=lambda e: abs(e)<0.5`
- eigmode="complex": non-Hermitian Hamiltonians only, which part of the complex eigenvalue is returned, "complex", "real" or "imag" (see Non-Hermitian Hamiltonians)
- biorthogonal=False: non-Hermitian Hamiltonians only, weight each state by $\langle L | A | R \rangle / \langle L | R \rangle$ instead of the right eigenvector alone
- write=True: also write `BANDS.OUT` and the k-path files (see Output files)

Without `kpath`, the path is Γ -M for a square-like 2D lattice, Γ -K-M-K'- Γ for a triangular-like one, and Γ -X-M- Γ -R for a 3D one.

Returns the index of the k-point along the path and the energies, plus one array per operator if `operator` is given

36.2 `h.get_kdos_bands()`

Compute the momentum-resolved spectral function along a k-path,

$$A(\mathbf{k}, \omega) = -\frac{1}{\pi} \text{Im Tr} [A G(\mathbf{k}, \omega + i\delta)],$$

with A the identity or a projection operator. This is what ARPES measures. See Spectral functions and Band unfolding.

Optional arguments:

- kpath: the k-path (generated if not given)
- operator=None: operator that weights the spectral function, for example "unfold" to unfold the bands of a supercell
- energies, delta, nk: energies, broadening δ and number of k-points
- mode="ED": "ED" (diagonalization), "green" (Green's function) or "KPM" (Chebyshev expansion, for large unit cells)
- frand=None: generator of the random vectors of the KPM trace, only with `mode="KPM"`
- biorthogonal=False: non-Hermitian Hamiltonians only, use the Green's-function spectral function
- write=True: also write `KDOS_BANDS.OUT`

Returns the position along the k-path, the energy and the spectral weight

36.3 `h.get_dos()`

Compute the density of states

$$\rho(\omega) = \frac{1}{N_k} \sum_{n,\mathbf{k}} \delta(\omega - \epsilon_n(\mathbf{k})),$$

with the delta function broadened by a Lorentzian of width δ . Its integral over energy is the number of states per unit cell. See Band structures and DOS.

Optional arguments:

- `energies`: array of energies
- `delta=None`: broadening. By default `5/nk`, so 0.05 for `nk=100`. A broadening smaller than the level spacing of the k-mesh shows the mesh, not the physics
- `mode="ED"`: "ED", "KPM", "adaptive", "Green" or "RG" (see Band structures and DOS)
- `nk=100`: k-points per direction
- `operator=None`: operator the DOS is projected on, a name, a matrix or an `Operator`

For a non-Hermitian Hamiltonian only `mode="ED"` exists (see Non-Hermitian Hamiltonians).

Returns energies and DOS

36.4 `h.get_gap()`

Return the indirect gap, the smallest energy difference between an empty and an occupied state anywhere in the Brillouin zone. The two states can be at different k-points. The band edges are found by minimizing over $\mathbf{k}$, not on a fixed mesh, so a gap that closes at a generic k-point is found.

Optional arguments:

- `ntries=1`: number of minimizations; the smallest result is kept

Returns a number. It is zero for a metal or a Dirac semimetal

36.5 `h.get_bandwidth()`

Return the bottom and the top of the spectrum, (`emin,emax`). Note that it returns the two band edges, not their difference.

36.6 `h.get_filling()`

Return the filling, the fraction of the states below zero energy,

$$\nu = \frac{1}{NN_k} \sum_{n,\mathbf{k}} \Theta(-\epsilon_n(\mathbf{k})),$$

with N the number of states per unit cell. Half filling is 0.5. Use `h.set_filling(nu)` to fix a filling, and this method to check it (see Building a Hamiltonian).

For a Nambu Hamiltonian the spectrum is particle-hole symmetric, so half of it is always below zero. There the filling is computed from the electron part of the occupied states, so that with zero pairing it is the filling of the normal state.

Optional arguments:

- `nk`: k-points per direction

36.7 `h.get_total_energy()`

Return the total energy of the occupied states,

$$E = \frac{1}{N_k} \sum_{n,\mathbf{k}} \epsilon_n(\mathbf{k}) \Theta(\epsilon_F - \epsilon_n(\mathbf{k})),$$

per unit cell. For a mean-field Hamiltonian this is only the band energy; the interacting energy, with the double counting removed, is returned by `h.get_mean_field_hamiltonian(...,return_total_energy=True)`.

Optional arguments:

- `nk=10`: k-points per direction
- `fermi=0.0`: the Fermi energy ϵ_F . Not accepted for a Nambu Hamiltonian, because a chemical potential shifts the electron and hole parts in opposite directions; use `h.shift_fermi(-mu)` instead
- `mode="mesh"`: sum on a k-mesh; `use_kpm=True` uses a Chebyshev expansion instead, for large systems

For a Nambu Hamiltonian it returns the energy of the electrons, so with zero pairing it is the energy of the normal state.

36.8 `h.get_density_matrix()`

Return the density matrix of the occupied states, as a matrix in the basis of the Hamiltonian. See Mean-field Hubbard models.

Note the convention: $\text{dm}_{ij} = \sum_{\text{occ}} \psi_i^* \psi_j$, the transpose of the usual $\rho_{ij} = \sum_{\text{occ}} \psi_i \psi_j^*$. So an expectation value is `np.trace(dm.T@A)`, not `np.trace(dm@A)`. The two are the same for a real operator (the density, σ_x , σ_z) and differ by a sign for a purely imaginary one (σ_y , the valley operator, a current). It is simpler to use `h.get_vev(operator)`.

36.9 `h.get_ipr()`

Return the inverse participation ratio of every eigenstate,

$$\text{IPR}_n = \sum_i |\psi_n(i)|^4,$$

as `(energies,ipr)`. A state spread over N sites has $\text{IPR} \sim 1/N$, and a state on a single site has $\text{IPR} \sim 1$, so it tells localized from extended states, for example in Anderson localization or for in-gap bound states. Only for finite (0D) systems; a periodic Hamiltonian raises `NotImplementedError`, and there the IPR operator can be used instead (see Operators).

36.10 `h.get_vev()` / `h.get_single_vev()` / `h.get_several_vev()`

Ground-state expectation values of operators, summed over the occupied states.

- `h.get_vev(operator=...)` returns one real number per site, the expectation value of `operator` on each site, for example "sz" for a magnetization map. Without `operator` it returns the occupation of each site
- `h.get_single_vev(A)` returns one number, $\langle A \rangle$ summed over the whole system
- `h.get_several_vev([A,B,...])` returns $\langle A \rangle$, $\langle B \rangle$, ... with a single diagonalization

An operator that depends on $\mathbf{k}$, like "unfold", is applied inside the sum over the Brillouin zone, so `h.get_vev(operator="unfold")` gives the unfolded weight on each site:

```
from pyqula import geometry
g = geometry.honeycomb_lattice() # primitive geometry
gs = g.get_supercell(2,store_primal=True) # supercell, keeping the primitive cell
h = gs.get_hamiltonian(has_spin=False)
h.add_onsite(lambda r: 100.0 if sum((r-gs.r[0])**2)<1e-2 else 0.0) # a vacancy
d = h.get_vev(operator="unfold") # unfolded weight on each site
```

For a Nambu Hamiltonian a normal observable is counted once, so with zero pairing the result is the one of the normal Hamiltonian. A pairing operator like "spair" gives the anomalous expectation value, the local s-wave pairing amplitude, per site with `h.get_vev("spair")` and summed with `h.get_single_vev("spair")`.

Optional arguments:

- `nk=30`: k-points per direction

36.11 `h.get_magnetization()`

The magnetization of each site, as an `(nsites,3)` array. There are two quantities with this name, chosen by `mode`:

- `mode="vev"` (default): the spin expectation value of each site, $(\langle\sigma_x\rangle, \langle\sigma_y\rangle, \langle\sigma_z\rangle)$, summed over the occupied states. It needs a k-mesh, given by `nk` or the one remembered from a self-consistent calculation. In a metal at zero temperature it changes in steps of order $1/n_k$, so use a fine mesh to compare weakly polarized states
- `mode="field"`: the magnetic term written in the Hamiltonian, the coefficients of $\sigma_{x,y,z}$ on each site. After a mean-field calculation this is the exchange field, proportional to the moment

Note the sign: `h.add_zeeman()` and `h.add_exchange()` add $+\vec{B}\cdot\vec{\sigma}$, so the electrons lower their energy with the spin against $\vec{B}$, and the moment is antiparallel to the field:

```
from pyqula import geometry
h = geometry.chain().get_hamiltonian()
h.add_exchange([0.,0.,0.5])
h.get_magnetization(nk=40)           # the moment: [0,0,-0.15], against the field
h.get_magnetization(mode="field")    # [0,0,0.5], the field you put in
```

Optional arguments:

- `mode="vev"`: "vev" or "field"
- other keywords (for example `nk`) are passed to `h.get_vev()` with `mode="vev"`

36.12 `h.get_topological_invariant()`

Return the topological invariant of the occupied bands that fits the dimension and the symmetry of the Hamiltonian:

- 1D: the Z_2 invariant (± 1) of a time-reversal symmetric superconductor (class DIII), and otherwise the Berry (Zak) phase
- 2D: the Z_2 invariant if the Hamiltonian is time-reversal symmetric (also a BdG Hamiltonian), and otherwise the Chern number
- 3D: the strong and weak Z_2 indices (`nu0`, (`nu1`, `nu2`, `nu3`)), each 0 or 1, if it is time-reversal symmetric, and otherwise the Chern vector (`C1`, `C2`, `C3`)

A 0D Hamiltonian raises `ValueError`. See Topological invariants.

36.13 `h.add_soc()`

Add the Kane-Mele spin-orbit coupling, the same as `h.add_kane_mele()`.

Arguments:

- `t`: strength of the spin-orbit coupling

36.14 `h.add_zeeman()`

Add a Zeeman field $\sum_i \vec{B}_i \cdot \vec{\sigma}$ (see Building a Hamiltonian).

Arguments:

- `zeeman`: the field $\vec{B}$, as a number (meaning $(0,0,B_z)$), a 3-vector, one vector per site, or a function of the position

36.15 `h.add_rashba()`

Add the Rashba spin-orbit coupling between first neighbors,

$$H_R = i\lambda_R \sum_{\langle ij \rangle} c_i^\dagger \hat{z} \cdot (\mathbf{r}_{ij} \times \vec{\sigma}) c_j,$$

with $\mathbf{r}_{ij}$ the bond vector. It comes from a field perpendicular to the plane, which breaks the mirror symmetry $z \rightarrow -z$.

Arguments:

- `c`: the Rashba coupling λ_R , a number or a function of the position

36.16 `h.add_onsite()`

Add an onsite energy $\sum_i \mu_i c_i^\dagger c_i$ (see Building a Hamiltonian).

Arguments:

- `fermi`: the onsite energy, a number, an array with one value per site, or a function of the position

36.17 `h.add_sublattice_imbalance()`

Add a staggered onsite energy, $+m$ on one sublattice and $-m$ on the other,

$$H_m = m \sum_i \tau_i c_i^\dagger c_i, \quad \tau_i = \pm 1.$$

In the honeycomb lattice this opens a gap $2m$ at the Dirac points, as in boron nitride.

Arguments:

- `mass`: the imbalance m , a number or a function of the position

The geometry needs a sublattice with two values. A chain, a square lattice or a kagome lattice raise `ValueError`. For a bipartite lattice without sublattice you can use `g = g.get_supercell(2)` and then `g.get_sublattice()`; otherwise write the potential with `h.add_onsite(f)`.

36.18 `h.add_antiferromagnetism()`

Add a staggered exchange field: a Neel pattern $\pm m \sigma_z$ on two sublattices, and a 120-degree pattern on three or more.

Arguments:

- `mass`: the exchange field, a number or a function of the position

The geometry needs a sublattice; without one it raises `ValueError`.

36.19 `h.add_haldane()`

Add the Haldane term, an imaginary second-neighbor hopping,

$$H_H = it \sum_{\langle\langle ij \rangle\rangle} \nu_{ij} c_i^\dagger c_j, \quad \nu_{ij} = (\mathbf{d}_1 \times \mathbf{d}_2)_z,$$

where $\mathbf{d}_1$ and $\mathbf{d}_2$ are the two first-neighbor bonds that go from i to j . In the honeycomb lattice $\nu_{ij} = \pm\sqrt{3}/2$, the sign given by the direction of the turn. It breaks time-reversal symmetry and opens a gap of $9t$ at both Dirac points, and the result is a Chern insulator with $C = \pm 1$ (see Topological invariants).

Arguments

- `t`: the second-neighbor hopping t , a number or a function of the position

36.20 `h.add_modified_haldane()` / `h.add_antihaldane()`

The same hopping as `h.add_haldane()`, but with an extra sign τ_i between the two sublattices. The two valleys get opposite masses and the Chern number is zero: it is a valley Hall insulator instead of a Chern insulator. The two names are the same method.

Arguments

- `t`: the second-neighbor hopping

36.21 `h.add_kane_mele()`

Add the Kane-Mele spin-orbit coupling, the Haldane hopping with opposite sign for the two spins,

$$H_{KM} = it \sum_{\langle\langle ij \rangle\rangle} \nu_{ij} c_i^\dagger \sigma_z c_j.$$

It keeps time-reversal symmetry, and the honeycomb lattice becomes a quantum spin Hall insulator, with gap $9t$ and $Z_2 = 1$. A spinless Hamiltonian is made spinful first.

Arguments

- `t`: the spin-orbit coupling t

36.22 `h.add_anti_kane_mele()`

The Kane-Mele term with the extra sublattice sign τ_i , in the same way as `h.add_modified_haldane()` is related to `h.add_haldane()`.

Arguments

- `t`: the second-neighbor hopping

36.23 `h.add_kekule()` / `h.add_chiral_kekule()`

Add a Kekule modulation of the first-neighbor hoppings of the honeycomb lattice, with a period of three bonds. It couples the two Dirac points, which are folded to Γ , and opens a gap. `add_kekule(t)` takes the amplitude of the modulation. `add_chiral_kekule(t1=..., t2=...)` adds the chiral version, with two independent amplitudes.

36.24 `h.add_valley_exchange()`

Add a valley exchange field $\vec{v} \cdot \vec{\tau}$, with $\vec{\tau} = (\tau_x, \tau_y, \tau_z)$ the valley Pauli matrices. It is the analog of `h.add_exchange()` for the valley pseudospin (see Operators).

Arguments

- `v = (vx, vy, vz)`: the valley field

For a periodic Hamiltonian the in-plane components couple the two valleys, so the unit cell must contain both: a supercell of the honeycomb lattice with a size multiple of 3. Otherwise it raises `ValueError`. A finite flake works always.

36.25 `h.add_peierls()` / `h.add_orbital_magnetic_field()`

Add an out-of-plane magnetic field as a Peierls phase on every hopping, $t_{ij} \rightarrow t_{ij} e^{i\phi_{ij}}$ (see Building a Hamiltonian). The two names are the same method.

Arguments

- `mag_field`: the field B , as the flux through a unit area in units of the flux quantum. In the Landau gauge the phase on a bond is $\phi_{ij} = 2\pi B y \Delta x$. In a periodic system the flux through the unit cell must be an integer number of flux quanta, so build a supercell first and choose `mag_field` to match it

Optional arguments

- `gauge="Landau"`: "Landau" or "symmetric"

A Hamiltonian with pairing raises `NotImplementedError`: a Cooper pair has charge $2e$, so the pairing term picks up a different phase, and a field in a superconductor means vortices. Add the field to the normal Hamiltonian first, and then the pairing.

36.26 `h.add_inplane_bfield()`

Add an in-plane magnetic field, as a Peierls phase that depends on the out-of-plane coordinate. It matters for a multilayer, where an in-plane field gives a flux between the layers.

Optional arguments

- `b=0.0`: strength of the field
- `phi=0.0`: in-plane direction of the field, in units of π

Only for 0D, 1D and 2D systems

36.27 `h.add_strain()`

Change the hoppings according to a strain field. In graphene a non-uniform strain acts as a pseudo-magnetic field, with opposite sign in the two valleys.

Arguments

- `sr`: function of the position that gives the local strain

Optional arguments

- `mode="scalar"`: "scalar" multiplies each hopping by `sr(r)`; "directional" also uses the direction of the bond, for anisotropic strain

36.28 `h.add_crystal_field()`

Add a crystal field: an onsite energy that depends on the neighbors of each site, so that sites with fewer or farther neighbors (at an edge, next to a vacancy, or in different stackings of a bilayer) get a different energy.

Arguments

- `v`: strength of the crystal field. The average is removed, so `v` sets the spread of the onsite energies, not their offset

Optional arguments

- `rcut=6.0`: cutoff distance of the sum over neighbors

36.29 `h.generate_spin_spiral()`

Rotate the spin quantization axis from cell to cell, with a wavevector $\mathbf{q}$, to describe a spin spiral in the unit cell of the lattice (generalized Bloch theorem). It is used as a starting point for a spiral mean-field calculation, and to compute the energy of a spiral as a function of $\mathbf{q}$ (see Exchange interactions).

Optional arguments

- `vector=[0.,0.,1.]`: the axis around which the spins rotate
- `qspiral=[1.,0.,0.]`: the wavevector $\mathbf{q}$ of the spiral
- `fractional=True`: `qspiral` in reduced coordinates. With `False` only the hoppings between cells are rotated

Needs a spinful Hamiltonian

36.30 `h.add_pairing()`

Add a superconducting pairing term with a given symmetry. The Hamiltonian becomes a Bogoliubov-de Gennes (BdG) Hamiltonian if it is not already (see Superconductivity).

Optional arguments

- `delta=0.0`: pairing amplitude Δ , a number or a function of the position
- `mode="swave"`: the symmetry of the pairing, for example "swave", "extended_swave", "triplet", "pwave", "chiral_pwave", "dx2y2", "dxy", "nodal_dwave", "chiral_dwave", "nodal_fwave", "chiral_fwave", "chiral_gwave". An unknown name raises `ValueError` with the list of all accepted ones. It can also be a function of two positions that returns the 2×2 pairing matrix $D(\mathbf{r}_1, \mathbf{r}_2)$. This function must respect the Fermi antisymmetry, $D(\mathbf{r}_1, \mathbf{r}_2) = \sigma_y D(\mathbf{r}_2, \mathbf{r}_1)^T \sigma_y$, meaning that the singlet part is even and the triplet part odd under exchange of the two positions; otherwise it raises `ValueError`
- `d=[0,0,1]`: the d-vector of a triplet pairing, a 3-vector or a function of the position

A `delta` or `d` that depends on the position must have the periodicity of the lattice, otherwise it raises `ValueError`. A modulated pairing is built on a supercell that fits the modulation. In a 0D system any function works.

36.31 `h.setup_nambu_spinor()`

Write the Hamiltonian in the Nambu (electron-hole) basis, with zero pairing. This is needed before any superconducting calculation on a normal Hamiltonian, for example a mean field with a pairing channel. It is the same as `h.add_swave(0.0)`. A Nambu Hamiltonian is always spinful, so a spinless Hamiltonian is made spinful first. Changes `h` in place

36.32 `h.turn_nambu()`

Add the electron-hole degree of freedom without pairing, making a spinless Hamiltonian spinful first. Normally `h.setup_nambu_spinor()` is the one to use. Changes `h` in place

36.33 `h.turn_spinful()`

Add the spin degree of freedom to a spinless Hamiltonian; every matrix is multiplied by the identity in spin space. Changes `h` in place, and does nothing if `h` is already spinful

36.34 `h.remove_spin()`

Keep one spin block and remove the other, which gives a spinless Hamiltonian.

Optional arguments

- `channel="up"`: the block that is kept, "up" or "dn"

The part of the Hamiltonian that mixes the two spins is dropped, so this is only meaningful when s_z is conserved. With spin-orbit coupling or non-collinear magnetism the result is a different model.

36.35 `h.turn_multicell()`

Store the hoppings as a dictionary indexed by the lattice vector between the cells, which allows hoppings beyond the neighboring cells. Some methods need it, and their error messages say so. Changes `h` in place. `h.get_no_multicell()` returns a copy in the other form when possible

36.36 `h.get_hk_gen()`

Return the Bloch Hamiltonian as a function of $\mathbf{k}$,

$$H(\mathbf{k}) = \sum_{\mathbf{R}} t(\mathbf{R}) e^{2\pi i \mathbf{k} \cdot \mathbf{R}},$$

with $\mathbf{k}$ in reduced coordinates (in units of the reciprocal lattice vectors), $t(\mathbf{R})$ the hopping matrix between cells separated by $\mathbf{R}$, and no positions of the sites inside the cell in the phase. For example `h.get_hk_gen()([0.3,0.,0.])` is $H(\mathbf{k})$ at $\mathbf{k} = 0.3\mathbf{b}_1$.

36.37 `h.get_gf()`

Return the Green's function of the infinite system at one energy,

$$G(\omega) = \frac{1}{N_k} \sum_{\mathbf{k}} [\omega + i\delta - H(\mathbf{k})]^{-1},$$

as a matrix in the basis of the unit cell.

Optional arguments

- `energy=0.0`: the energy ω
- `delta=1e-5`: the imaginary part δ
- `mode="adaptive"`: how the integral over $\mathbf{k}$ is done, "adaptive" (with error control), "full" (a fixed mesh of `nk` points) or "renormalization"
- `gtype="bulk"`: "bulk" or "surface" Green's function

$-\text{Im Tr } G/\pi$ is the density of states, which is what `h.get_dos(mode="Green")` computes

36.38 `h.get_hopping_dict()`

Return the hoppings as a dictionary `{(n1,n2,n3): matrix}`, indexed by the lattice vector between the two cells, with `(0,0,0)` the hopping inside the cell. It is used to give an interaction with a range beyond one cell to the response functions (see Spin response and magnons), and to look at or change the hoppings directly. `h.get_multihopping()` returns the same as a `MultiHopping` object, which can be added and multiplied by numbers, and `h.set_multihopping()` writes it back

36.39 `h.get_rkky()`

Compute the RKKY interaction between two magnetic impurities, the effective exchange J_{ij} mediated by the electrons of `h` (see Charge response and RKKY).

Optional arguments

- `mode="pm"`: "pm" adds two local exchange fields to the Hamiltonian and takes the energy difference between parallel and antiparallel alignment, $J = -(E_{\uparrow\uparrow} - E_{\uparrow\downarrow})/(4\delta J^2)$, with δJ the strength of the two fields, at fixed filling. "LR" uses linear response, $J_{ij} \propto \chi_{ij}(\omega = 0)$, which is faster for many separations
- for `mode="pm"`: `ri`, `rj` the positions of the two impurities (required), `nk=10`, `dj=1e-1` the strength of the local exchange fields
- for `mode="LR"`: `R=[0,0,0]` the lattice vector between the two cells, `ii=0`, `jj=0` the sites in them, `nk=100`, `delta` the broadening (by default $1/\text{nk}$)

`rkky.rkky_map(h,n=...)` computes it for many separations. On a bipartite lattice at half filling it is ferromagnetic between sites of the same sublattice and antiferromagnetic between different sublattices.

36.40 `h.get_ldos()`

Compute the local density of states at one energy,

$$\rho_i(\omega) = -\frac{1}{\pi} \text{Im } G_{ii}(\omega + i\delta),$$

on every site. This is what STM measures. See Local density of states.

Optional arguments:

- `e`: the energy ω (**energy** is also accepted)
- `delta=None`: broadening δ , 0.001 by default. With `mode="KPM"` it is the energy resolution, with `npol=200` polynomials by default
- `operator=None`: operator the LDOS is projected on, a name, a matrix or an `Operator`
- `mode="arpack"`: **"arpack"** (diagonalization on a k-mesh), **"green"** (Green's function, 2D only) or **"KPM"** (Chebyshev expansion of each site, for large systems, see Kernel polynomial method)
- `npol`, `kpm_radius`, `scale`, `kpm_prec`: only for `mode="KPM"`, the number of polynomials, the radius in hops around each site (`None` for the exact result), the energy window of the spectrum, and the precision
- `projection="TB"`: **"TB"** (one value per site), **"TBRs"** (interpolated in real space) or **"atomic"** (with atomic orbitals)
- `eigmode="complex"`: non-Hermitian Hamiltonians only, whether **e** is on the real or the imaginary axis (see Non-Hermitian Hamiltonians)

Returns the x position, the y position and the LDOS

36.41 `h.get_multildos()`

Compute the LDOS at many energies. Returns `(x,y,energies,ldos)`, with `ldos` one row per energy. It also writes one file per energy in the folder `MULTILDOS/`, with the total DOS in `MULTILDOS/DOS.OUT`, unless `write=False`. The maps and the DOS have the same normalization as `h.get_ldos()` and `h.get_dos()`.

Optional arguments:

- `energies=linspace(-1,1,100)`: the energies
- `delta, nk`: broadening and k-points, as in `h.get_ldos()`; by default `delta=0.01`
- `mode="diagonalization"`: **"diagonalization"** or **"KPM"**, with the options of `h.get_ldos()` in that mode
- `operator=None`: operator the LDOS is projected on
- `projection="TB"`: **"TB"** or **"atomic"**

36.42 `h.get_chi()`

Compute a non-interacting response function between two operators A and B (density-density by default), $\chi_{AB}(\mathbf{q}, \omega)$, the linear response of $\langle A \rangle$ to a perturbation B with momentum $\mathbf{q}$ and frequency ω . It is the Lindhard sum over pairs of states $|n\mathbf{k}\rangle$ and $|m\mathbf{k} + \mathbf{q}\rangle$, each pair weighted by the difference of their Fermi occupations and by the matrix elements of A and B , with a pole at the energy difference of the pair, broadened by δ . The result is the response of one site, averaged over the sites of the unit cell. See Charge response and RKKY.

Optional arguments:

- `q=None`: the momentum transfer. With `None` the response is averaged over all $\mathbf{q}$ of the mesh, the local response; use `q=[0.,0.,0.]` for the uniform response
- `A=None, B=None`: the two operators (by default the identity, the charge response)
- `energies, delta, nk`: energies, broadening (0.1 by default) and k-points per direction (60 by default)

Returns energies and the response function

36.43 `h.get_spinchi_ladder()`

Compute the transverse spin susceptibility $\chi_{+-}(\mathbf{q}, \omega)$, dressed in the random phase approximation (RPA) with the Hubbard U of a mean-field Hamiltonian,

$$\chi^{RPA} = \chi_0 (1 - U\chi_0)^{-1}.$$

Its poles are the magnons, the spin waves that inelastic neutron scattering and spin-polarized STM measure. See Spin response and magnons.

Optional arguments:

- `q=None, energies, delta, nk`: as in `h.get_chi()`. With `q=None` each $\mathbf{q}$ is dressed and then averaged, which gives the local response

- `RPA=True`: dress the response with the RPA; `False` gives the bare one. An interaction between different sites is treated in the pair basis of `h.get_transverse_spinchi()`
- `T=None`: temperature of the occupations, by default equal to `delta`
- `chi_prec=None`: precision of the sum, "single" or "double". By default single on the GPU and double on the CPU (see GPU)

36.44 `h.get_rpa_kernel_poles()`

Compute the poles of the RPA kernel $1 - V(\mathbf{q})\chi(\mathbf{q}, \omega)$, the frequencies of the collective modes or of the instabilities of the interacting system.

Optional arguments:

- `V=None` (required): the interaction. A matrix for an onsite interaction, or a dictionary `{(n1,n2,n3): matrix}` (or a `MultiHopping`) for an interaction between cells, which is Fourier transformed to $V(\mathbf{q})$
- `A=None`, `B=None`, `q=None`, energies, `delta`, `nk`: as in `h.get_chi()`. A pole belongs to one $\mathbf{q}$, so with `q=None` the kernel is built at $\mathbf{q} = 0$ from the averaged response

Returns an `(npoles, 2)` array, one row per collective mode, sorted by frequency: the frequency of the pole and its imaginary part. A small imaginary part means a sharp mode.

36.45 `h.get_magnon_bands()`

Compute the magnon bands of a magnetic mean-field state along a $\mathbf{q}$ -path. See Spin response and magnons.

Optional arguments:

- `method="rpa"`: "rpa", "pair" or "tdhf". "rpa" takes the poles of the RPA spin response; it works for metals and insulators and needs a grid of frequencies. "pair" keeps the pair index of the interaction, for density-density or exchange interactions between different sites. "tdhf" solves the time-dependent Hartree-Fock (electron-hole pair) eigenvalue problem; it needs no frequency grid, has an exact Goldstone mode, and needs a gapped state converged on the same `nk` (or `metal=True`)
- `qpath=None`: the $\mathbf{q}$ -path, the default path of the geometry, a list of high-symmetry labels, or a list of vectors in reduced coordinates
- `nq=None`: with a list of vectors, the number of points of the path through them, so `qpath=[[0,0,0], [0.5,0,0]]`, `nq=26` is 26 points from 0 to 1/2; without it the vectors are the $\mathbf{q}$ -points themselves. For the default or a labeled path, the density of points, 20 when not given
- energies, `delta`, `nk`: as above, for `method="rpa"`
- `nk`, `n`, channel, `V`: for `method="tdhf"`, the k -mesh (the same as in the mean-field calculation), the number of branches per $\mathbf{q}$ -point, the block ("auto", "spinflip", "all"), and an interaction instead of `h.V`

Returns `(qs, ws, gammas)` for `method="rpa"`: the index of the $\mathbf{q}$ -point along the path, the frequency of the pole and its imaginary part. With `method="tdhf"` it returns `(qs, es)`, with `es` the magnon energy; an imaginary part means that the mean-field state is unstable.

36.46 `h.get_transverse_spinchi()`

Compute the RPA spin response in the basis of the pairs of sites of the interaction, instead of single sites. This allows a density-density interaction between neighbors and an exchange interaction, which the site-basis RPA cannot include. It works for metals and for non-collinear states, and gives the full spin- and site-resolved $\chi(\omega)$.

Optional arguments:

- `W=None`: the interaction, by default the one of the mean-field calculation
- `q=None`, energies, `delta`, `nk`: as in `h.get_chi()`
- `component=None`: a pair of spin indices (`a, b`) to return only that block
- `chi_prec=None`: precision of the sum, as in `h.get_spinchi_ladder()`; on the CPU only "double"

Returns `(energies, chi)`, with `chi` a $3N \times 3N$ matrix per frequency, ordered as $(S_x, S_y, S_z) \times \text{site}$.

36.47 `h.get_magnon_energies()`

Return the magnon energies at one momentum $\mathbf{Q}$, from the spin-flip channel of the time-dependent Hartree-Fock (Bethe-Salpeter) equation. Same arguments as `h.get_magnon_bands(method="tdhf")`, with $\mathbf{Q}=[\mathbf{qx}, \mathbf{qy}, \mathbf{qz}]$ instead of the $\mathbf{q}$ -path. A large imaginary part means that the mean-field state is unstable against that excitation.

36.48 `h.get_goldstone_residual()`

Return how far a magnetic mean-field state is from having a zero-energy magnon at $Q = 0$, as required by the Goldstone theorem without spin-orbit coupling. It is $\|Mv\|/\|v\|$, with M the time-dependent Hartree-Fock matrix and v a uniform rotation of all spins. It is proportional to the convergence error of the mean field, so check it before trusting a magnon dispersion. It also detects a mean field converged on a different $\mathbf{nk}$.

Optional arguments:

- `nk=10`, `V=None`, `channel="auto"`: as in `h.get_magnon_energies()`
- `relative=True`: divide by the largest transition energy, to compare models with different bandwidths

36.49 `h.get_densitychi_RPA()`

Compute the RPA charge response

$$\chi^{RPA}(\mathbf{q}, \omega) = \chi_0 [1 - V(\mathbf{q})\chi_0]^{-1}$$

for a density-density interaction with onsite U , first, second and third neighbor V_1, V_2, V_3 and a general $V(r)$. The interaction is given here, not taken from a mean-field calculation. Its poles are plasmons or charge-density-wave instabilities.

Optional arguments:

- `V1=0.0`, `V2=0.0`, `V3=0.0`, `U=0.0`, `Vr=None`: the interaction, as in `h.get_mean_field_hamiltonian()`
- `rcut=None`: range of $\mathbf{Vr}$, every pair of sites closer than `rcut`, a site with itself included; `None` means 5 in a periodic system and all pairs in a finite one
- `q=None`, `energies`, `delta`, `nk`: as in `h.get_chi()`

36.50 `h.get_plasmon_bands()`

Compute the plasmon bands, the poles of the RPA charge response along a $\mathbf{q}$ -path. The charge analog of `h.get_magnon_bands()`.

Optional arguments:

- `V1=0.0`, `V2=0.0`, `V3=0.0`, `U=0.0`, `Vr=None`: as in `h.get_densitychi_RPA()`
- `qpath=None`, `nq=None`, `energies`, `delta`, `nk`: as in `h.get_magnon_bands()`

Returns `(qs, ws, gammas)`, as `h.get_magnon_bands()`.

36.51 `h.get_bse()`

Solve the Bethe-Salpeter equation for excitons, the bound electron-hole pairs, on top of a mean-field Hamiltonian, and return a BSE object. In the basis of transitions $v\mathbf{k} \rightarrow c\mathbf{k} + \mathbf{Q}$,

$$(\epsilon_{c\mathbf{k}+\mathbf{Q}} - \epsilon_{v\mathbf{k}})A_{v\mathbf{k}} + \sum_{v'c'\mathbf{k}'} K_{v\mathbf{k}, v'c'\mathbf{k}'} A_{v'c'\mathbf{k}'} = E_X A_{v\mathbf{k}},$$

with K the direct (attractive) minus the exchange (repulsive) kernel. The excitons below the band gap are the peaks of optical absorption. See Excitons.

Optional arguments:

- `V=None`: the electron-hole interaction. `None` uses `h.V`, the one of the mean-field calculation. Otherwise a dictionary `{(n1,n2,n3): matrix}`, or a matrix for an onsite interaction; `bsetk.interaction.density_interaction` builds one from `U, V1, V2, V3, Vr`
- `Q=None`: total momentum **Q** of the exciton, by default zero
- `nk=10`: k-points per direction
- `nv=None, nc=None`: only the `nv` highest valence and `nc` lowest conduction bands; `None` takes all
- `kernel="full"`: `"full"`, `"direct"` (only the attraction), `"exchange"` (only the exchange, which is the RPA) or `"none"` (free transitions)
- `tda=False`: Tamm-Dancoff approximation, keeping only the resonant block
- `max_memory=2.0`: stop, with an error, if the matrix needs more than this many GB
- `screening=None`: the interaction in the direct term. `None` uses the bare one, which is time-dependent Hartree-Fock. `"rpa"` uses the static RPA screened interaction $W = \varepsilon^{-1}v$ from the bands of `h`. `"crpa"` does the same without the transitions inside the `nv/nc` window. A `ScreenedInteraction` uses one computed before. The exchange term always uses the bare interaction. A fitted Hubbard U is already screened and should not be screened again
- `nkW=None`: k-mesh for the screening, by default `nk`; it must be a multiple of `nk`
- `channel="charge"`: `"charge"` builds the dielectric matrix on the sites and keeps spin rotation symmetry; `"orbital"` uses the full spin-orbital matrix. They are the same for a spinless Hamiltonian
- `solver="dense"`: `"dense"` diagonalizes the full matrix, and is the only one without the Tamm-Dancoff approximation. `"iterative"` finds the lowest excitons without building the matrix, for large k-meshes. `"qtt"` uses a tensor-network compression, with a cost that grows as $\log N_k$. The last two need `tda=True` (see Excitons)
- `neig=None`: number of excitons for `solver="iterative"` (4 by default); `"qtt"` only gives one
- `gauge="auto"`: fixes the arbitrary phase of the Bloch states, needed by `solver="qtt"`. It does not change any energy

`solver="qtt"` also takes `tolerance` (default `1e-6`), `maxbonddim`, `maxdim`, `nsweep`, `cutoff`, `coarse_nk` and `unfolding`.

The BSE object has `energies`, `amplitudes` (the resonant amplitudes $A_{vc}(\mathbf{k})$), `amplitudesY` (the antiresonant ones, zero with `tda`), `pairs` (the label `(ik,iv,ic)` of each transition), and the methods `get_energies(n)`, `get_binding_energies(n)` and `get_lowest_transition()`. The last one is the lowest free transition, from which binding energies are measured.

36.52 `h.get_exciton_energies()`

Return the exciton energies E_X , sorted. Same arguments as `h.get_bse()`, and `n=None` to keep only the `n` lowest.

36.53 `h.get_exciton_binding_energies()`

Return the binding energies of the excitons, the lowest free transition minus E_X . A positive value means a bound exciton. Same arguments as `h.get_exciton_energies()`.

36.54 `h.get_exciton_states()`

Return `(energies,amplitudes)` of the excitons, with the amplitudes $A_{vc}(\mathbf{k})$ indexed by the transition, whose label `(ik,iv,ic)` is in `pairs.labels` of the BSE object. Same arguments as `h.get_exciton_energies()`.

36.55 `h.get_exciton_bands()`

Return the exciton band structure $E_X(\mathbf{Q})$, one Bethe-Salpeter calculation per q-point along a path.

Optional arguments:

- `qpath=None, nq=None`: the q-path and its number of points, as in `h.get_magnon_bands()`
- `n=None`: keep only the `n` lowest excitons at each q-point
- `V, nk, nv, nc, kernel, tda, max_memory, screening, nkW`: as in `h.get_bse()`

Returns (**qs**,**es**): the index of the q-point along the path and the exciton energy. The energy is complex at a q-point where the mean-field state is unstable. Note that **nv** and **nc** must not cut a degenerate group of bands (a spinful Hamiltonian without spin-orbit coupling or magnetism needs even **nv** and **nc**); otherwise a warning is printed.

36.56 **h.get_screened_interaction()**

Return the static RPA screened interaction $W(\mathbf{q}) = \varepsilon^{-1}(\mathbf{q})v(\mathbf{q})$, with $\varepsilon = 1 - v\chi_0$ built from the bands of **h**, as a **ScreenedInteraction** object.

Optional arguments:

- **V=None**: the bare interaction v , as **V** in **h.get_bse()**. **None** uses **h.V**; note that a fitted Hubbard U is already screened
- **nk=10**: k-mesh for the sum over **k** and for the grid of **q**
- **screening="rpa"**: **"rpa"** (all transitions screen) or **"crpa"** (the transitions in **exclude** do not)
- **exclude=None**: (**vbands**,**cbands**) that do not screen, needed by **"crpa"**
- **channel="charge"**: **"charge"** or **"orbital"**, as in **h.get_bse()**
- **chi_prec=None**: precision of χ_0 , **"single"** or **"double"**; by default single on the GPU and double on the CPU

The object has **qs**, **Wq**, **chi0**, **bare**, **epsmin** (the smallest eigenvalue of ε on the mesh), **.at(q)** for the value at a q-point and **.get_dict()** for the interaction in real space. If an eigenvalue of $\varepsilon(\mathbf{q})$ reaches zero it raises an error: the mean-field state has a charge or spin instability at that wavevector.

36.57 **h.get_polarizability()**

Return (**qs**,**chi0**), the static polarizability $\chi_0(\mathbf{q})$ on the k-mesh, in the spin-orbital basis, with shape (**nq**,**norb**,**norb**). Takes **nk**, **exclude** and **chi_prec** as **h.get_screened_interaction()**.

36.58 **h.get_fermi_surface()**

Compute the spectral weight $A(\mathbf{k},\omega)$ on a 2D k-mesh at one energy. At $\omega = 0$ it is the Fermi surface, as seen by ARPES. See Spectral functions.

Optional arguments:

- **e=0.0**: the energy ω
- **nk=50**: k-points per direction
- **delta**: broadening
- **operator=None**: operator that weights the spectral function, for example **"sz"**, **"valley"** or **"unfold"**
- **primal_mesh=False**: use the Brillouin zone of the primitive cell, for a supercell built with **store_primal=True**; this is the mesh of an unfolded Fermi surface

Returns **kx**, **ky** and the spectral weight

36.59 **h.get_multi_fermi_surface()**

Compute the Fermi surface at many energies, and write one file per energy in the folder **MULTIFERMISURFACE/**.

Optional arguments:

- **energies=[0.0]**: the energies
- **nk**, **delta**, **operator**, **primal_mesh**: as in **h.get_fermi_surface()**

36.60 **h.get_surface_kdos()**

Compute the spectral function at the surface and in the bulk of a semi-infinite system, from the surface Green's function. The surface states of a topological insulator appear here as bands inside the bulk gap. See Spectral functions.

Optional arguments:

- kpath: the k-path (generated if not given)
- energies, delta: energies and broadening

Returns k, energy, surface spectral weight and bulk spectral weight; also writes `KDOS.OUT`

36.61 `h.get_qpi()`

Compute the quasiparticle interference pattern, the Fourier transform of the LDOS modulation around a defect, as measured by Fourier-transform STM (2D systems only). See Quasiparticle interference. Returns `(q,energies,qpi)`, with `qpi` one row per energy and one column per wavevector, and also writes the maps to `MULTIQPI/` and the DOS to `DOS.OUT`.

Optional arguments:

- energies, nk, delta: as above
- mode="response": `"pm"` computes the autocorrelation of the momentum-resolved spectral function, the QPI of a real scatterer; `"response"` uses a joint density of states of the clean bands, which is faster; `"tmatrix"` computes a single impurity in the infinite system with the T-matrix, and `"born"` the same impurity to first order
- impurity=None: the impurity of `"tmatrix"` and `"born"`, a number (onsite potential on the first site of the cell), one onsite potential per site, or a matrix with the dimension of the cell
- nunfold=1: unfold the QPI of a supercell to the Brillouin zone of the primitive cell, given as the linear size of the supercell (`np.sqrt(3)` for the $\sqrt{3} \times \sqrt{3}$ one)
- write=True: with `False` no file is written

36.62 `h.get_qpi_impurity()`

Compute the quasiparticle interference with real impurities: they are put in a supercell, the LDOS is computed in real space, and it is Fourier transformed (2D systems only). Returns `(r,ldos_r,q,qpi_q)`.

Optional arguments:

- nsuper=10: size of the supercell (a number or `(n1,n2)`)
- impurities=[]: list of dictionaries, each `{"position"|"index": ..., "onsite": v}` or `{"position"|"index": ..., "vacancy": True}`
- energies=0.0, delta, nk: as above
- num_waves=20: initial number of eigenstates near each energy; it is increased automatically when needed, so a small value only costs time
- write=True, output_folder="QPI_IMPURITY": also write the maps to files

36.63 `h.get_spin_splitting_density()`

Compute the spin splitting of a collinear magnet as a function of energy. Each pair of spin-up and spin-down bands contributes its squared splitting $(\epsilon_{n\uparrow} - \epsilon_{n\downarrow})^2$, broadened by a Lorentzian at the mean energy of the pair, so the integral over energy is the squared splitting summed over bands and averaged over the Brillouin zone. It is the quantity that shows an altermagnet, spin-split bands without net magnetization. Returns `(energies,values)`. See Altermagnets.

Optional arguments:

- nk=20: k-points per direction
- energies: the energies (by default 400 points from -3 to 3)
- delta=1e-2: broadening
- tol=1e-7: largest spin-mixing matrix element accepted; above it the method raises an error, since the spin splitting is not defined with spin-orbit coupling or non-collinear order

36.64 `h.get_average_spin_splitting()`

Return one number: the spin splitting of a typical band, averaged over the bands and over the Brillouin zone. It does not depend on the size of the unit cell used to describe the crystal.

Optional arguments:

- `nk=20`: k-points per direction
- `tol=1e-7`: as above

Use `h.get_spin_splitting_vs_energy()` to get the largest splitting instead of the typical one.

36.65 `h.get_spin_splitting_vs_energy()`

Compute the largest spin splitting $|\epsilon_{n\uparrow} - \epsilon_{n\downarrow}|$ over the Brillouin zone, as a function of the mean energy of the pair. Returns `(energies, values)`, in the same form as `h.get_spin_splitting_density()`.

Optional arguments:

- `nk=100`: k-points per direction
- `energies=None`: centers of the energy bins; otherwise `nbins` points from `emin` to `emax`
- `nbins=400`, `emin=None`, `emax=None`: used without `energies`; by default the range of the energies found
- `tol=1e-7`: as above

Empty bins are 0.0.

36.66 `h.get_spin_chern()`

Return the spin Chern number $(C_+ - C_-)/2$ of a two-dimensional spinful insulator. The occupied states are split by the sign of the projected spin Ps_zP , so the result stays quantized when s_z is not conserved, for example with Rashba coupling. See Topological invariants.

Optional arguments: - `operator="sz"`: the spin component, a name, a matrix or an `Operator` - `nk=40`: k-points per direction - `write=True`: with `False` nothing is written to `CHERN.OUT` and `BERRY_CURVATURE.OUT`

It raises `ValueError` if the projected spin has no gap or the system is a metal.

36.67 `h.get_mirror_chern()`

Return the mirror Chern number $(C_{+i} - C_{-i})/2$ of a two-dimensional insulator with the mirror symmetry $z \rightarrow -z$, with $C_{\pm i}$ the Chern numbers of the two mirror sectors. The mirror is found from the geometry and the Hamiltonian; without it the method raises `ValueError`.

Optional arguments: - `nk=40`: k-points per direction

36.68 `h.get_winding_number()`

Return the winding number of a one-dimensional Hamiltonian with chiral symmetry,

$$W = \frac{1}{2\pi i} \int dk \partial_k \log \det q(k),$$

with $q(k)$ the off-diagonal block of $H(k)$ in the basis where the chiral operator is diagonal. It is the number of zero modes at each end of an open chain. See Topological invariants.

Optional arguments: - `chiral=None`: the chiral operator, a name, a matrix or an `Operator`; by default the sublattice operator of a normal Hamiltonian and $\sigma_y \tau_y$ on each site of a Nambu one. It raises `ValueError` if it does not anticommute with the Hamiltonian - `nk=200`: k-points in the Brillouin zone

36.69 `h.get_quadrupole_moment()`

Return the quadrupole moment q_{xy} , 0 or 1/2, of a two-dimensional insulator with the mirrors M_x and M_y , from the nested Wilson loop. A value 1/2 means corner charges $\pm e/2$, a higher-order topological insulator. See Topological invariants.

Optional arguments: - `nk=40`: k-points per direction - `nocc=None`: number of occupied bands, counted from the lowest; by default the bands below zero energy - `tol=1e-4`: smallest distance of the Wannier bands from 0 and 1/2; below it the Wannier gap is closed and the method raises `ValueError`

36.70 `h.get_wannier_sector_polarization()`

Return the polarization of one sector of the Wannier bands, along the direction perpendicular to the first Wilson loop, from the nested Wilson loop. It is given modulo 1, in $-1/4, 3/4$, so that 0 and 1/2 are not at the edge of the interval.

Optional arguments: - `loop=0`: direction of the first Wilson loop - `sector="+"`: the Wannier bands with $0 < \nu < 1/2$, or `"-"` for $-1/2 < \nu < 0$ - `gauge="lattice"`: all orbitals at the origin of the cell, or `"atomic"`, each orbital at its position - `nk=40`, `nocc=None`, `tol=1e-4`: as in `h.get_quadrupole_moment()`

36.71 `h.get_chern()`

Return the Chern number of the occupied bands,

$$C = \frac{1}{2\pi} \int_{BZ} \Omega(\mathbf{k}) d^2k,$$

with Ω the Berry curvature. It is the quantized Hall conductivity $\sigma_{xy} = C e^2/h$, and the number of chiral edge states. See [Topological invariants].

Optional arguments: - `nk`: k-points per direction, 10 with `integration="grid"` and 20 with `integration="qtci"`; with `integration="wannier"`, k-points along each Wilson loop, 30 by default - `integration="grid"`: how the integral is done. `"grid"` sums the Berry curvature on an `nk x nk` mesh, and the result is always an integer. `"qtci"` uses tensor cross interpolation, which samples adaptively; it is accurate for a smooth curvature (a large gap) but not for a sharp peak. `"wannier"` counts the winding of the hybrid Wannier centers - `nt=100`: for `integration="wannier"`, the number of momenta in the other direction; a small gap needs more - `nocc=None`: for `integration="wannier"`, the number of occupied bands; by default the bands below zero energy - `operator=None`: an operator that restricts the Berry curvature, as in `h.get_berry_curvature()` - `write=True`: also write the Chern number to `CHERN.OUT` and the Berry curvature to `BERRY_CURVATURE.OUT`

36.72 `h.get_berry_curvature()`

Return the Berry curvature of the occupied bands,

$$\Omega(\mathbf{k}) = i \sum_{n \in \text{occ}} \left(\langle \partial_{k_x} u_n | \partial_{k_y} u_n \rangle - \langle \partial_{k_y} u_n | \partial_{k_x} u_n \rangle \right),$$

as a map over the Brillouin zone, `(kx,ky,berry)`, three flat arrays. This is the curvature that `h.get_chern()` integrates.

In two dimensions the map covers a whole number of Brillouin zones, and the curvature is per unit area of the reduced zone, whose area is one. So the Chern number is `np.mean(berry)/(2*np.pi)`, the same as `h.get_chern()` when `nk` resolves the curvature.

Optional arguments:

- `nk=100`: k-points per direction (the map has `nk*nk` points, and `berry.reshape(nk,nk)` has `kx` along the first index)

- `reciprocal=True`: `kx,ky` in Cartesian coordinates, to plot a hexagonal Brillouin zone without distortion. The map covers a rectangle centered at Γ made of whole zones: one zone for a square lattice, a square of side $|\mathbf{b}_1|$; two zones for a hexagonal lattice, of sides $|\mathbf{b}_1|$ and $\sqrt{3}|\mathbf{b}_1|$. A lattice without such a rectangle (an oblique one) gets the parallelogram of its two reciprocal lattice vectors, one zone, whose points are not on a rectangular grid. With `reciprocal=False`, `kx,ky` are reduced coordinates from -1 to 1 , four zones
- `mode="Wilson"`: "Wilson" uses the Fukui-Hatsugai-Suzuki plaquettes; "Green" uses the Green's-function Kubo formula, and is used automatically with `operator`
- `operator=None`: restrict the curvature to a subspace, for example "valley" for the valley-resolved curvature (see Operators)
- `nsuper=1`: number of copies of the window in each direction
- `kpath`: compute along a k-path instead of on a 2D mesh; for a curvature to plot next to the bands, `h.get_berry_curvature_path()` is simpler
- `delta=0.001`: broadening for `mode="Green"`
- `write=True`: also write the map to `BERRY_MAP.OUT`

36.73 `h.get_berry_curvature_path()`

Return the Berry curvature of the occupied bands along a k-path, the same $\Omega(\mathbf{k})$ as `h.get_berry_curvature()`, as `(index,berry)`: the index of each k-point of the path and the curvature there. It is the curvature to plot under a band structure, where its peaks show the k-points of small gaps.

Optional arguments:

- `kpath=None`: the k-path, a list of k-points or of high-symmetry labels; by default the k-path of `h.get_bands()`
- `nk=600`: number of k-points of the path
- `reciprocal=False`: with `True`, the k-points of `kpath` are Cartesian, in the coordinates of `h.get_berry_curvature()`, instead of reduced
- `dk=0.01`: side of the plaquette of the Wilson loop, in reduced coordinates
- `mode="Wilson"`, `operator=None`, `delta=0.001`: as in `h.get_berry_curvature()`
- `write=True`: also write the k-points and the curvature to `BERRY_CURVATURE.OUT`

36.74 `h.get_quantum_geometric_tensor()`

Return the quantum geometric tensor of a set of bands S at one k-point,

$$Q_{\mu\nu}(\mathbf{k}) = \sum_{n \in S} \langle \partial_\mu u_n | (1 - P_S) | \partial_\nu u_n \rangle,$$

with P_S the projector on the bands. Its real part is the quantum metric and its imaginary part is $-\Omega/2$, with Ω the Berry curvature. See Quantum geometry.

Optional arguments: - `k=[0.,0.,0.]`: the k-point - `occ_idxs=None`: the band indices of S ; by default the bands with $E < 0$, as in `h.get_chern()` - `non_abelian=False`: with `True`, return the full tensor in the orbital basis, $\sum_{m,n \in S} |u_m\rangle Q^{mn} \langle u_n|$, a matrix for each pair of directions, which does not depend on the choice of basis inside a degenerate subspace - `degeneracy_tol=1e-8`: if a band of S is degenerate with a band outside it within this tolerance, the method raises `ValueError` - `gauge="atomic"`: "atomic" uses the positions of the orbitals in the Bloch phase, the physical quantum geometry; "lattice" puts every orbital at the origin of its cell - `coordinates="reduced"`: derivatives along the reduced momenta, or "cartesian" along k_x, k_y, k_z

36.75 `h.get_quantum_metric()`

Return the quantum metric $g_{\mu\nu} = \text{Re } Q_{\mu\nu}$, with the same arguments as `h.get_quantum_geometric_tensor()`.

36.76 `h.get_dvector_non_unitarity()`

Return the non-unitarity vector of a triplet superconductor, $\vec{q} = i(\vec{d} \times \vec{d}^*)$, on each site, for a BdG Hamiltonian (see Superconductivity). It is real, zero for a unitary state, and otherwise is the spin of the Cooper pairs: parallel to the magnetization in a ferromagnetic triplet superconductor, and along $+z$ for pure $\Delta_{\uparrow\uparrow}$ pairing.

Optional arguments

- `nk = 10`: k-points per direction

Returns an array of shape (number of sites, 3).

36.77 `h.write_non_unitarity()`

Write $\vec{q}$ as a map in real space to `NON_UNITARITY_MAP.OUT`, with columns (x, y, z, q_x, q_y, q_z) .

Optional arguments

- `nrep = 2`: number of copies of the unit cell in each direction
- `nk = 10`: k-points per direction

36.78 `h.get_average_dvector()`

Return the squared components $(|d_x|^2, |d_y|^2, |d_z|^2)$ of the d-vector of a BdG Hamiltonian, averaged over $\mathbf{k}$.

Optional arguments

- `nk = 10`: k-points per direction
- `spatial_sum = True`: average over the sites, returning one 3-vector
- `non_unitarity = False`: average the squared components of $\vec{q}$ instead. This only says whether the state is non-unitary, not the direction; use `h.get_dvector_non_unitarity()` for the vector.

36.79 `h.get_superfluid_weight()`

Return the superfluid weight tensor D_s^{ab} of a BdG Hamiltonian in one, two or three dimensions, the response of the supercurrent to a phase gradient,

$$D_s^{ab} = \left. \frac{\partial^2 \Omega(\mathbf{A})}{\partial A_a \partial A_b} \right|_{\mathbf{A}=0},$$

with Ω the grand potential and $\mathbf{A}$ a vector potential. It sets the London penetration depth, $\lambda^{-2} \propto D_s$. See Superconductivity.

Optional arguments:

- `nk=20`: k-points per direction
- `T=0.0`: temperature
- `mode="kubo"`: "`kubo`" uses the multiband Kubo formula, and "`finite_difference`" the numerical second derivative of the grand potential
- `gauge="atomic"`: "`atomic`" uses the full bond vector in the phase (the physical Peierls substitution), "`lattice`" only the lattice vector
- `decompose=False`: with `True`, return a dictionary with "`total`", "`conventional`", "`geometric`", "`delta`" and "`gauge`"; the conventional part comes from the band dispersion and the geometric part from the quantum metric

Returns a `(dim,dim)` array, or a dictionary with `decompose=True`. `decompose=True` raises `ValueError` when the decomposition is not defined, and with `mode="finite_difference"`.

36.80 `h.get_bkt_temperature()`

Return the Berezinskii-Kosterlitz-Thouless temperature of a 2D BdG Hamiltonian, from the Nelson-Kosterlitz condition $T_{\text{BKT}} = (\pi/8)D_s(T_{\text{BKT}})$, with $|\Delta|$ kept fixed. Arguments `nk=20`, `tmax=None`, `tol=1e-6`, `maxite=60`, `maxexpand=20`, and `gauge`. `tmax` is the upper end of the search, $(\pi/8)D_s(0)$ by default; it is doubled up to `maxexpand` times if needed, and if no crossing is found the method raises `ValueError`. Returns a number.

36.81 `h.get_nonlinear_drude_conductivity()`

Compute the l -th order nonlinear Drude conductivity $\sigma^{x^{l_1}y^{l_2};b}$ of a collinear magnet, the current along b of order E^l in the electric field, for spin s ,

$$\sigma_s^{x^{l_1}y^{l_2};b} = \frac{(-e/\hbar)^{l+1}}{(i\omega + 1/\tau)^l} \int \frac{d^D k}{(2\pi)^D} f_s^{(0)} \frac{\partial^{l+1} \epsilon_s}{\partial k_x^{l_1} \partial k_y^{l_2} \partial k_b},$$

with $l = l_1 + l_2$, $f_s^{(0)}$ the Fermi function and ϵ_s the bands of spin s . The lowest order that is not zero gives the wave index of an altermagnet (d : 1, f : 2, g : 3, i : 5; a p -wave magnet has none, and $l = 0$ is always zero). Spin must be conserved. See Altermagnets.

Optional arguments

- `field = "x"`: one character per power of the electric field, for example "yyyyy" for the fifth-order response to E_y
- `current = "x"`: direction of the current
- `channel = "spin"`: "spin", "charge", "up" or "dn"
- `nk = 100`: k-points per direction
- `T = 0.01`: temperature of the occupations
- `mu = 0.0`: chemical potential
- `tau = 1.0`: relaxation time τ ; the l -th order scales as τ^l
- `omega = 0.0`: frequency, which enters as $1/(i\omega + 1/\tau)^l$
- `degeneracy_tol = 1e-8`: relative band spacing below which bands are taken as degenerate

Returns a complex number

36.82 `h.get_nonlinear_drude_components()`

All components of the l -th order nonlinear Drude conductivity, as a dictionary `{"x^l1 y^l2;b": value}`. It uses one k-mesh for all components, so it is faster than separate calls.

36.83 `h.get_nonlinear_drude_orders()`

For each order $l = 0$ to `lmax`, the largest $|\sigma^{x^{l_1}y^{l_2};b}|$ over the components of that order. The lowest l with a nonzero value is the wave index.

Optional arguments

- `lmax = 6`: highest order
- every optional argument of `h.get_nonlinear_drude_conductivity()`

Returns a list of numbers

36.84 `h.get_optical_conductivity()`

Compute the optical conductivity tensor $\sigma_{ab}(\omega)$ with the Kubo-Greenwood formula, with an intraband (Drude) and an interband part. $\text{Re } \sigma_{aa}$ is the optical absorption, and the DC limit of σ_{xy} is the Hall conductivity. See Optical conductivity.

Optional arguments:

- `energies=None`: the frequencies ω

- `nk=20`: k-points per direction
- `T=None`: temperature (by default equal to `delta`)
- `delta=0.1`: broadening η
- `intraband=True, interband=True`: include each of the two parts
- `component=None`: for example "xy", to return only that component
- `degeneracy_tol=1e-6`: relative tolerance to treat two bands as degenerate

Returns `(energies,sigma)`, with `sigma` complex of shape `(nw,3,3)`, or `(nw,)` with `component`. The unit is $e^2/\hbar$, so e^2/h is $1/(2\pi)$. 3D and Nambu Hamiltonians raise `NotImplementedError`.

36.85 `h.get_drude_weight()`

Return the Drude weight tensor, the weight of the intraband $\delta(\omega)$ peak of $\text{Re } \sigma(\omega)$. Arguments `nk=20`, `T=0.05`, `degeneracy_tol=1e-6`. Returns a real `(3,3)` array.

36.86 `h.get_sum_rule_weight()`

Return the diamagnetic weight W of the optical f-sum rule, $\int \text{Re } \sigma_{aa}(\omega) d\omega = \pi W_{aa}$. Arguments `nk=20`, `T=0.05`. Returns a real symmetric `(3,3)` array.

36.87 `h.get_entanglement_entropy()`

Return the entanglement entropy of a region A of the ground state,

$$S = - \sum_n [\zeta_n \log \zeta_n + (1 - \zeta_n) \log(1 - \zeta_n)],$$

with ζ_n the eigenvalues of the correlation matrix $C_{ij} = \langle c_i^\dagger c_j \rangle$ restricted to A . See Entanglement.

Optional arguments:

- `region=None`: the region, a list of site indices, a boolean mask, a function of the position, or a fraction of the cells; `None` takes half of the system
- `nsuper=10`: number of unit cells in the ring that is cut (periodic Hamiltonians)
- `direction=None`: lattice direction perpendicular to the cut, by default the last periodic one
- `kpar=None`: momentum parallel to the cut; with `None` a 2D Hamiltonian is averaged over `nk` momenta
- `nk=20`: number of momenta of that average
- `fermi=0.0`: the states with $E < \text{fermi}$ are occupied (must be 0 for BdG)

Returns a number. A state exactly at the Fermi energy raises an error, since the ground state is then not unique.

36.88 `h.get_entanglement_spectrum()`

Return the eigenvalues of the entanglement Hamiltonian, $\xi_n = \ln[(1 - \zeta_n)/\zeta_n]$, of a region. Same arguments as `h.get_entanglement_entropy()` (with `nk=41` by default).

Returns a sorted array of ξ_n for a 0D or 1D Hamiltonian or a single `kpar`. For a 2D Hamiltonian with `kpar=None` it returns `(ks,xis)`, with `xis` of shape `(nk,nA)`, the Li-Haldane entanglement spectrum as a function of the parallel momentum. In a topological phase it shows the same spectral flow as the edge states.

36.89 `h.get_wannier_hamiltonian()`

Build maximally localized Wannier functions for a set of bands, and return the tight-binding Hamiltonian in this basis. See Wannierization.

Arguments:

- `bands = [a,b]`: first and last band (counted from 0, both included)

Optional arguments:

- `nk=12`: k-points per direction
- `cutoff=1e-6`: hoppings smaller than this are dropped (the hopping inside the cell is always kept). With 0.0 the bands are reproduced exactly
- `symmetries=None`: "auto" to find and impose the point group, or a list of `symmetrytk.pointgroup.SymmetryOperation`
- `num_wann=None`: number of Wannier functions. `None` uses the whole set of bands, which is then reproduced exactly. A smaller number uses the Souza-Marzari-Vanderbilt disentanglement, and then only the frozen window is reproduced exactly. A value outside `1..len(bands)` raises `ValueError`
- `dis_win_min, dis_win_max=None`: outer energy window, the states that can enter the Wannier functions at each k-point
- `dis_froz_min, dis_froz_max=None`: frozen window, whose states are reproduced exactly. `dis_froz_max` alone is enough; `dis_froz_min` alone raises `ValueError`
- `dis_num_iter=200`: maximum number of iterations of the disentanglement; if it does not converge, a warning is printed

A `dis_*` argument without a smaller `num_wann` raises `ValueError`. Disentanglement with a BdG Hamiltonian, with `symmetries=`, or with `auto_split_clusters=True` raises `NotImplementedError`.

Returns a new, smaller Hamiltonian. `.wannier_centres`, `.wannier_spreads` and `.wannier_spread_total` give the centers and spreads of the Wannier functions, and `.wannier_num_wann` and `.wannier_disentanglement_window` what was used (the window is `None` without disentanglement)

36.90 `h.get_szsz_mean_field_hamiltonian()`

Self-consistent Hartree-Fock mean field of an Ising exchange interaction $\sum_{ij} J_{ij} S_i^z S_j^z$ (see Exchange interactions). $J > 0$ is antiferromagnetic and $J < 0$ ferromagnetic.

Optional arguments:

- `J1, J2, J3 = 0.`: first, second and third neighbor couplings
- `Jr=None`: a coupling as a function of the distance, as `Vr` in `h.get_mean_field_hamiltonian()`; `Jr(0)` is an onsite term $J_r(0) S_i^z S_i^z / 2$, so a coupling between bonds must be zero at zero distance
- `filling, mf, nk, maxerror, mix, constrains`: as in `h.get_mean_field_hamiltonian()`
- `return_total_energy=False`: also return the total energy

It also works for BdG Hamiltonians, with both the normal and the pairing channels.

Returns the self-consistent Hamiltonian (it raises an error if the loop does not converge)

36.91 `h.get_sxsx_mean_field_hamiltonian()` / `h.get_sysy_mean_field_hamiltonian()`

The same as `h.get_szsz_mean_field_hamiltonian()`, for an $S_i^x S_j^x$ or $S_i^y S_j^y$ interaction. Also works for BdG Hamiltonians.

36.92 `h.get_exchange_mean_field_hamiltonian()`

Self-consistent mean field of an anisotropic exchange $\sum_{ij} (J_{ij}^x S_i^x S_j^x + J_{ij}^y S_i^y S_j^y + J_{ij}^z S_i^z S_j^z)$.

Optional arguments:

- `Jx1, Jx2, Jx3, Jy1, Jy2, Jy3, Jz1, Jz2, Jz3 = 0.`: first, second and third neighbor couplings for each spin component
- `Jxr, Jyr, Jzr=None`: couplings as a function of the distance, one per component
- `mf, filling, nk, maxerror, mix, constrains`: as above

It also works for BdG Hamiltonians, with the normal and the pairing channels. An exchange interaction can give superconductivity: for example an antiferromagnetic J alone, started from a random guess, can converge to a superconducting state.

Returns the self-consistent Hamiltonian (it raises an error if the loop does not converge)

36.93 h.get_combined_mean_field_hamiltonian()

Self-consistent mean field of density-density interactions (onsite U , neighbor V_1, V_2, V_3 , and V_r) together with a Heisenberg exchange $\sum_{ij} J_{ij} \vec{S}_i \cdot \vec{S}_j$. See Mean-field Hubbard models and Exchange interactions.

Optional arguments:

- U, V_1, V_2, V_3, V_r : as in `h.get_mean_field_hamiltonian()`
- $J_1, J_2, J_3 = 0.$: Heisenberg exchange for the first, second and third neighbors
- $J_r=None$: an exchange $J_r(\mathbf{r}_1, \mathbf{r}_2)$ as a function of the positions, as V_r . A site with itself is included, so $J_r(0)$ is an onsite term $J_r(0) \vec{S}_i \cdot \vec{S}_i/2$, which at the mean-field level is a Hubbard $U = -3J_r(0)/4$; an exchange between bonds must be zero at zero distance
- $rcut=None$: range of V_r and J_r , every pair of sites closer than $rcut$, a site with itself included ($V_r(0)$ is an onsite $U = V_r(0)$); `None` means 5 in a periodic system and all pairs in a finite one
- $J_{1x}, J_{1y}, J_{1z} = 0.$: anisotropic correction to J_1 , only for first neighbors (the first-neighbor J_z is $J_1 + J_{1z}$)
- $mf, filling, nk, maxerror, mix, constrains$: as above
- $integration="ed"$: the density matrix of each iteration is computed by exact diagonalization. `"kpm"` uses a Chebyshev (KPM) expansion instead, for systems too large to diagonalize. With `is_sparse=True` in `g.get_hamiltonian()` the memory grows linearly with the number of sites. A Nambu Hamiltonian also works, sparse or dense, and a dense one comes back dense. With `"kpm"`: `scale` is the energy window (found automatically if not given; if given, it must contain the whole spectrum, or it raises `ValueError`), `npol` the number of Chebyshev moments, `ne` the number of energies in the occupied window, `cores` the number of parallel workers, and `kpm_radius` the number of hops around each site that are kept (`None` for the whole system). A finite `kpm_radius` makes the time linear in the number of sites, and it is accurate for a gapped state or at finite temperature (see Kernel polynomial method)
- `use_jax=True, solver=...`: find the self-consistent solution with a nonlinear solver instead of simple mixing. `"newton"` is the default and `"error_gradient"` the most robust. They can stop before `maxerror`, and then the method raises an error as for any calculation that does not converge. Only for a normal (non-BdG) Hamiltonian, with exact diagonalization and without `constrains`:

```
from pyqula import geometry
h = geometry.chain().get_hamiltonian()
hmf = h.get_combined_mean_field_hamiltonian(U=4.0, J1=-0.5, filling=0.5,
                                             use_jax=True, solver="newton") # JAX-derivative-based SCF solver
```

For a BdG Hamiltonian all the channels include the normal and the pairing terms, so an exchange alone can give superconductivity and not only magnetism (see Exchange interactions).

Returns the self-consistent Hamiltonian (it raises an error if the loop does not converge)

`filling` can also be an array with one value per site, with the same meaning as the number (the fraction of the two states of the site that is occupied). Then $\langle n_i \rangle = \text{filling}[i]$ is imposed on each site, with one Lagrange multiplier (a local chemical potential) per site found in the same self-consistent loop. The converged potentials are in the `fermi` attribute of the result. If the mean field or the constraint does not converge, the method raises an error. Only for a normal (non-BdG) Hamiltonian with `mu=None`, with `integration="ed"` or `"kpm"`. `SpinonHamiltonian` uses this to impose exactly one fermion per site.

36.94 SpinonHamiltonian(g)

Abrikosov pseudofermion mean-field Hamiltonian of a spin-1/2 Heisenberg model on the geometry `g`. Each spin is written as $\vec{S}_i = \frac{1}{2} f_i^\dagger \vec{\sigma} f_i$ with one fermion per site, and the exchange is decoupled in the hopping (RVB) and magnetic channels. See Pseudofermion mean field. It is imported with `from pyqula.spinon import SpinonHamiltonian`; it has no bare hopping, and the couplings are given to `h.get_mean_field_hamiltonian()` as $J_1, J_2, J_3, J_r, J_{1x}, J_{1y}, J_{1z}$.

- `h.get_mean_field_hamiltonian(J1=..., nk=..., ...)`: the same arguments as `h.get_combined_mean_field_hamiltonian()` except `filling`, since there is always exactly one fermion per site. Returns the self-consistent Hamiltonian (it raises an error if the loop does not converge), with two more attributes:
 - `h2.local_occupation`: $\langle n_i \rangle$ on each site (between 0 and 2, it should be 1.0)

- `h2.constraint_lambda`: the Lagrange multiplier of each site (a local chemical potential)

36.95 KondoLatticeHamiltonian(hc)

Abrikosov pseudofermion (Read-Newns) mean-field Hamiltonian of a Kondo lattice, built from the conduction-electron Hamiltonian `hc`. A localized f site, with no bare hopping, is added next to each conduction site, and the Kondo coupling $J \vec{s}_i \cdot \vec{S}_i$ is decoupled in a hybridization V_i between c and f electrons. A finite V_i is the Kondo (heavy Fermi liquid) state. See Pseudofermion mean field. It is imported with `from pyqula.kondolattice import KondoLatticeHamiltonian`, and J is given to `h.get_mean_field_hamiltonian()` as J .

- `h.get_mean_field_hamiltonian(J=...,filling=...,mf=(V,lam),nk=...,...)`: finds the hybridization and the Lagrange multiplier of the constraint self-consistently. Returns the self-consistent Hamiltonian (it raises an error if the loop does not converge), with three more attributes:
 - `h2.local_occupation`: $\langle n_f \rangle$ on each localized site (it should be 1.0)
 - `h2.hybridization`: V_j on each localized site
 - `h2.constraint_lambda`: the Lagrange multiplier of each site

36.96 GrapheneGeometry(g)

A `Geometry` for graphene multilayers `g` (bilayer, twisted bilayer, twisted trilayer...), with a method `.relax()` (see Structural relaxation). It is imported with `from pyqula.graphenetc.geometry import GrapheneGeometry`. `g` must have `has_sublattice=True`, otherwise it raises `ValueError`.

- `g.relax(nrep=1,maxiter=500,verbose=False,layer_pairs=None,gsfe_coeffs=...,elastic_coeffs=...)`: minimizes the stacking energy between layers (the generalized stacking fault energy, GSFE) plus the elastic energy inside each layer, with respect to in-plane displacements, and returns a new relaxed `GrapheneGeometry`. `layer_pairs` gives the pairs of layers (ordered by z) with a stacking energy, by default all neighboring pairs; `gsfe_coeffs` and `elastic_coeffs` replace the graphene parameters.

36.97 GrapheneHamiltonian(geometry)

A Hamiltonian for a graphene multilayer `geometry` (usually a `GrapheneGeometry`, relaxed or not; see Structural relaxation). It is imported with `from pyqula.graphenetc.hamiltonian import GrapheneHamiltonian`. The hoppings decay with the distance, as in `specialhopping.twisted_matrix(ti=0.12,lambi=8.0,lamb=12.0,dl=3.0)`, the same as `specialhamiltonian.twisted_bilayer_graphene`, so the relaxed positions change the electronic structure. `mgenerator=...` gives another hopping function.

36.98 h.get_central_heterostructure()

Build a two-terminal `Heterostructure` with `h`, a finite (0D) Hamiltonian, as the central region, connected to two semi-infinite chains at sites i and j (see Quantum transport).

Optional arguments:

- `i=0, j=None`: the sites where the leads are attached (counted from 0); by default j is the last site
- `left=None, right=None`: the Hamiltonians of the leads; by default a spinless `geometry.chain()`. Add pairing (`add_swave`) to one of them, or to `h`, for a normal-superconductor junction. Only one of `h`, `left` and `right` can have pairing

Returns a `Heterostructure`, so `didv`, `get_dos`, `get_kappa` and the other methods work. `landauer` raises `NotImplementedError` when there is pairing; use `didv` then. A central region that is not 0D raises `NotImplementedError`.

37 Other classes

37.1 Heterostructure functions and methods

37.1.1 `HT.get_dc_current()`

Compute the time-averaged (DC) current through a two-terminal junction at a bias voltage V , with the Floquet-Keldysh formalism. The leads can be normal or superconducting, built with `heterostructures.build(h1,h2)`, with or without a central region. With superconducting leads this includes multiple Andreev reflection. See Quantum transport.

Arguments:

- voltage: bias voltage eV

Optional arguments:

- nmax=6, nmax_max=40: initial and maximal number of Floquet sidebands. The number grows until the current converges to `tol`; a warning is printed if `nmax_max` is reached first
- tol=1e-3: relative tolerance on the current
- temperature=0.: temperature of the leads
- delta=None: broadening, by default `HT.delta`. It must be much smaller than the smallest superconducting gap

Returns the DC current

37.1.2 `HT.get_iv_curve()`

`HT.get_dc_current()` for an array of voltages.

Arguments:

- voltages: array of bias voltages

Returns an array of DC currents

37.1.3 `HT.didv_curve()` / `lp.didv_curve()`

The differential conductance dI/dV for an array of energies, the same as `[ht.didv(energy=e) for e in es]`. It is also available as `didv(energies=...)`.

Arguments:

- energies: array of bias energies
- any keyword argument of `didv` (`method`, `delta`, `temp`, ...)

Returns an array of dI/dV values

Note that `T` means different things on the two classes. On a `Heterostructure`, `T` is the temperature. On a `LocalProbe`, `T` is the transparency of the probe, the coupling between probe and sample, and the temperature is `temp`. A `T=` or a `delta=` given to one call applies only to that call. `HT.with_delta(delta)`, `lp.with_delta(delta)` and `lp.with_coupling(T)` return a copy with a new broadening or coupling, to use in several calls.

37.1.4 `HT.get_kappa()`

Compute the exponent ratio κ , which tells single-particle tunneling from Andreev reflection (also `LocalProbe.get_kappa()`, see Quantum transport).

Optional arguments:

- energy=0.0: energy at which κ is computed (returns a number)
- energies: array of energies instead (returns an array); do not give both
- temp=0.: temperature. A finite value averages the conductances over the Fermi window

- T=1e-2: coupling around which the exponent is computed

Returns κ , a number or an array like **energies**

κ is the exponent $d \log G / d \log T$ of the conductance with the coupling T , divided by the same exponent with the pairing removed. It is 1 for single-particle tunneling, since $G \propto T$, and 2 for Andreev reflection, since $G \propto T^2$. Inside a superconducting gap the broadening **delta** of a **LocalProbe** acts as a finite quasi-particle lifetime, and it gives a normal conductance $G_N \delta / \Delta$. The Andreev conductance is $G_N^2 / 4$ in the tunneling limit, so the plateau at 2 needs **delta** well below $\Delta G_N / 4$ at the smallest coupling, for example **delta=1e-10** for a gap of 0.1 down to **T=1e-3**. See **examples/transport/localprobe_kappa_1D** and **jupyter-notebooks/functionalities/quantum_transport/07_differential_decay_rate.ipynb**.

37.2 SpinModel functions and methods

A **SpinModel** is a classical spin model with energy

$$E = \sum_i \vec{b}_i \cdot \vec{S}_i + \sum_{ij} \vec{S}_i \cdot J_{ij} \cdot \vec{S}_j,$$

where $\vec{S}_i$ is a unit vector with angles (θ_i, ϕ_i) , $\vec{b}_i$ a magnetic field, and J_{ij} a 3×3 exchange tensor. See Classical spin models.

37.2.1 sm.add_heisenberg()

Add exchange couplings between neighbor shells. Several calls add up.

Optional arguments:

- Jij=None: list of couplings of each shell, for example [J1,J2,J3] for first, second and third neighbors. It accepts everything that **tij** of **g.get_hamiltonian()** accepts; None is first neighbors with $J = 1$
- Jm=[1.,1.,1.]: diagonal weights (J_x, J_y, J_z) multiplying Jij; use different values for XXZ or Ising anisotropy

Modifies **sm** in place, no return value

37.2.2 sm.add_field()

Add a uniform magnetic field $\vec{b}$ to every site. Several calls add up.

Arguments:

- v: the field, a 3-vector, for example [0.,0.,1.]

37.2.3 sm.add_tensor() / sm.add_tensor_2d()

Add a general coupling tensor J_{ij} given by a function **fun(r1,r2) -> 3x3 matrix** (see **generating_functions** below). **add_tensor** only couples sites inside the unit cell. **add_tensor_2d** also sums the periodic images $i_a \mathbf{a}_1 + j_a \mathbf{a}_2$ with i_a, j_a in **[-ncells,ncells]**, and can rotate the tensor of each image around z by the angle $\pi(v_0 i_a + v_1 j_a)$, with (v_0, v_1) given by **vspiral**. This puts a spin-spiral wavevector into the exchange.

Arguments:

- fun: coupling function, for example one from **generating_functions**

Optional arguments (**add_tensor_2d** only):

- ncells=1: number of periodic images on each side, along each lattice vector
- vspiral=[0.,0.]: rotation angles per image, in units of π

Modifies **sm** in place, no return value

37.2.4 `classicalspin.generating_functions()`

Return a coupling function `fun(r1,r2) -> 3x3 matrix` of a common type, to use with `add_tensor` and `add_tensor_2d`.

Optional arguments:

- `name="Heisenberg"`: the type of coupling. "Heisenberg" is isotropic exchange, cut by `fc(distance)`. "Linear" is the tensor $J\hat{r}\hat{r}/r^3$, with $\hat{r}$ the bond direction. "RKKYTI" is the RKKY coupling on the surface of a topological insulator, $J(\frac{3}{2}\hat{r}\hat{r} - 1)/r^3$. "ZZ" is Ising, only $S_i^z S_j^z$. "XYZ" is diagonal with weights `v`. "DM" is a first-neighbor Dzyaloshinskii-Moriya coupling $\vec{D}_{ij} \cdot (\vec{S}_i \times \vec{S}_j)$, with $\vec{D}_{ij}$ along $\mathbf{r}_{ij} \times \vec{v}$, so `v` plays the role of the direction of the intermediate ion (or a function of the bond vector)
- `J=1.0`: strength of the coupling
- `v=[0.,0.,1.]`: vector used by the type, or a function of the bond vector
- `fc=None`: function of the distance that cuts or modulates the coupling; by default first neighbors ($0.9 < d < 1.1$)
- `fdiff=lambda x,y: x-y`: how the bond vector is computed from `(r1,r2)`
- `fr=None`: extra rotation matrix `fr(r1,r2)` applied to the coupling (by default the identity)

Returns the coupling function

37.2.5 `sm.minimize_energy()`

Find the classical ground state, by minimizing E over the angles (θ_i, ϕ_i) of all spins from several random starting points. Only states with the periodicity of the unit cell are found, so a spiral needs a supercell that fits it.

Optional arguments:

- `theta0=None, phi0=None`: starting angles. With `None` each try starts from random angles, which is what makes `tries` useful. With explicit arrays every try starts at the same point and gives the same result
- `tries=10`: number of minimizations; the one with the lowest energy is kept
- `calle=None`: a function `calle(sm) -> float` added to the energy, for example a penalty that favors a texture
- `silent=False`: if `True`, the minimum energy is not printed

Sets `sm.theta`, `sm.phi` and `sm.magnetization` to the best result, and returns `(theta,phi)`

37.2.6 `sm.get_energy()` / `sm.energy()`

The energy E above for the current angles. Returns a number.

37.2.7 `sm.get_local_energy()`

The energy of each site: its field term plus half of each exchange term it belongs to. The values add up to `sm.get_energy()`.

Returns an array over sites

37.2.8 `sm.get_magnetization()`

The Cartesian components of the spins, $\vec{S}_i = (\sin \theta_i \cos \phi_i, \sin \theta_i \sin \phi_i, \cos \theta_i)$.

Returns `(mx,my,mz)`, arrays over sites

37.2.9 `align.most_perpendicular_vector()` / `align.most_perp_basis()`

(`classicalspintk.align`) Find the direction most perpendicular to a set of vectors, for example a magnetization, and rotate the vectors so that this direction is the new z axis. The vectors are then mostly in the new xy plane, which helps to plot a skyrmion or a spiral as arrows in the plane. See [examples/classicalspin/perpendicular/main.py](#).

Returns the perpendicular unit vector, or (for `most_perp_basis`) the vectors in the rotated basis

37.3 LatticeGas functions and methods

A `LatticeGas` is a classical lattice gas with occupations $n_i = 0, 1$ and energy

$$E = \sum_i \mu_i n_i + \sum_{i \neq j} J_{ij} n_i n_j,$$

where the sum runs over ordered pairs, so each bond counts twice. See Lattice gas models.

37.3.1 `lg.add_interaction()`

Add a density-density coupling J_{ij} between neighbor shells. Several calls add up.

Optional arguments:

- `Jij`: list of couplings of each shell, for example `[J1,J2,J3]`. A positive value is repulsive and a negative one attractive. It accepts everything that `tij` of `g.get_hamiltonian()` accepts

Modifies `lg` in place, no return value

37.3.2 `lg.set_filling()`

Replace the occupations `lg.den` by a random configuration with a given filling, for example to start a new anneal.

Arguments:

- `filling`: fraction of occupied sites (rounded to an integer number of sites)

37.3.3 `lg.get_energy()`

The energy E above for the current occupations `lg.den`. Returns a number.

37.3.4 `lg.optimize_energy()`

Lower the energy with Metropolis moves at fixed filling. In each step one to three random pairs of an occupied and an empty site are swapped. The move is accepted if the energy does not increase, and with probability $e^{-\Delta E/T}$ otherwise.

Optional arguments:

- `temp=0.1`: Metropolis temperature T . `temp=0` only accepts moves that do not increase the energy
- `ntries=1e5`: number of attempted swaps
- `resync_every=1000`: every how many attempts the energy is recomputed from scratch, to avoid accumulated rounding errors
- `patience=None`: if given, stop after this many attempts without a new lowest energy
- `checkpoint_at=None`: an integer or a list of integers; a copy of `lg.den` after that many attempts is stored in `lg.checkpoints`, a dictionary `step -> den`

Sets `lg.den` to the final configuration, and returns the energy after each attempt

37.3.5 `lg.anneal()`

Simulated annealing: `lg.optimize_energy()` at each temperature of a decreasing list. The lowest-energy configuration of the whole schedule is kept.

Optional arguments:

- `temps=None`: temperatures, from high to low; by default 10 temperatures on a geometric scale from 2.0 to 0.05
- `ntries=1e4`: attempts per temperature
- `checkpoint_at=None`: as in `lg.optimize_energy()`, but counted over the whole schedule
- any other keyword of `lg.optimize_energy()`, used at every temperature

Sets `lg.den` to the best configuration, and returns the energies of the whole schedule

37.3.6 `lg.optimize_energy_multistart()`

Run `nstart` independent optimizations from random configurations at the current filling, and keep the lowest energy. This lowers the chance of ending in a metastable state. The runs are parallel after `parallel.set_cores()` (see Parallelism).

Optional arguments:

- `nstart=10`: number of runs
- any other keyword of `lg.optimize_energy()` (`temp`, `ntries`, `patience`...), the same for every run

Sets `lg.den` to the best configuration, and returns its energy

37.3.7 `lg.optimize_grand_canonical()`

Metropolis sampling in the grand-canonical ensemble. Single sites are switched between occupied and empty, so the filling is not fixed and is set by the chemical potential `lg.mu`. Use it to scan a phase diagram as a function of the chemical potential, or to sample at a fixed temperature for `latticegas.get_specific_heat()` and `get_susceptibility()`.

Optional arguments: the same as `lg.optimize_energy()` (`temp`, `ntries`, `resync_every`, without `patience`), but the default is `temp=1.0`

Sets `lg.den` to the final configuration, and returns (`es`, `ns`): the energy and the number of occupied sites after each attempt

37.3.8 `latticegas.get_specific_heat()` / `latticegas.get_susceptibility()`

Thermodynamic quantities from a trajectory sampled at a fixed temperature T , by `lg.optimize_energy()` or `lg.optimize_grand_canonical()`. `get_specific_heat(es, temp, burn=0.2)` gives the specific heat $C = \text{Var}(E)/T^2$ from the energies. `get_susceptibility(ns, temp, burn=0.2)` gives the compressibility $dN/d\mu = \text{Var}(N)/T$ from the number of particles; it only makes sense in the grand-canonical case, where N fluctuates. `burn` is the fraction at the start of the trajectory that is thrown away as thermalization.

37.3.9 `lg.add_tensor()`

Add a coupling $J_{ij} = \text{fun}(\mathbf{r}_i, \mathbf{r}_j)$ between all pairs of sites, for interactions beyond neighbor shells, for example a screened or dipolar $1/r^n$ interaction. Pairs with zero coupling are not stored.

Modifies `lg` in place, no return value

37.3.10 `lg.regroup()`

Merge repeated pairs from several `add_interaction()` or `add_tensor()` calls, summing their couplings. The energy does not change, only the calculation gets faster.

Modifies `lg` in place, no return value

37.3.11 `lg.write()` / `lg.read()`

Write the occupations `lg.den` to a text file, and read them back. `read()` raises `ValueError` if the number of sites in the file is not `lg.nsites`.

Optional arguments:

- `name="DENSITY.OUT"`: file name

37.3.12 `lg.get_local_energy()` / `lg.get_local_mu()`

`get_local_energy()` gives the energy of each site, $\mu_i n_i + \sum_j J_{ij} n_i n_j$; the values add up to `lg.get_energy()`. `get_local_mu()` gives the same expression with site i occupied, meaning the energy cost of putting a particle on site i with its neighbors as they are.

Optional arguments:

- `normalize=False`: divide the value of each site by the sum of its couplings

Returns an array over sites

37.3.13 `lg.get_correlator()`

The density-density correlator of the current occupations, as a function of the neighbor shell. It shows whether the ground state is ordered. `n` is the number of shells, `normalized` divides out the mean occupation.

Returns (`distances`, `correlators`)

37.3.14 `lg.get_structure_factor()`

The structure factor of the current occupations,

$$S(\mathbf{q}) = \frac{1}{N} \left| \sum_i (n_i - \bar{n}) e^{-i\mathbf{q} \cdot \mathbf{r}_i} \right|^2,$$

computed on the real positions of the sites. The correlator gives the length scale of the order, and $S(\mathbf{q})$ gives its wavevector. Since the mean occupation $\bar{n}$ is subtracted, $S(\mathbf{q} = 0) = 0$, so a peak at another $\mathbf{q}$ signals order. This is the quantity measured in a diffraction experiment.

Optional arguments:

- `qpath=None`: array of $\mathbf{q}$ vectors. By default a square grid of `nq`×`nq` points between $-q_{max}$ and q_{max}
- `nq=60`: points per direction of the default grid
- `qmax=None`: size of the default grid; by default $2\pi/d$, with d the first-neighbor distance

Returns (`qpath`, `sq`), the $\mathbf{q}$ vectors and $S(\mathbf{q})$

37.4 LatticeIsing functions and methods

A `LatticeIsing` is a classical Ising model with spins $s_i = \pm 1$ and energy

$$E = - \sum_i b_i s_i - \sum_{i \neq j} J_{ij} s_i s_j,$$

where the sum runs over ordered pairs, so each bond counts twice, which is two times the usual convention. See Ising models.

37.4.1 `li.add_interaction()`

Add an exchange J_{ij} between neighbor shells. Several calls add up.

Optional arguments:

- `Jij`: list of couplings of each shell, for example `[J1,J2,J3]`. A positive value is ferromagnetic (note the opposite sign to `LatticeGas.add_interaction()`). It accepts everything that `tij` of `g.get_hamiltonian()` accepts

Modifies `li` in place, no return value

37.4.2 `li.add_field()`

Add a magnetic field b_i .

Arguments:

- `h`: a number (the same on every site) or an array with one value per site

Modifies `li` in place, no return value

37.4.3 `li.set_magnetization()`

Replace the spins `li.s` by a random configuration with a given magnetization.

Arguments:

- `m=0.0`: average magnetization, between -1 and 1 (rounded to an integer number of up spins)

37.4.4 `li.get_energy()`

The energy E above for the current spins `li.s`. Returns a number.

37.4.5 `li.get_magnetization()`

The magnetization per site, $m = \frac{1}{N} \sum_i s_i$, between -1 and 1.

37.4.6 `li.optimize_energy()`

Metropolis dynamics with single spin flips. In each step one random spin is flipped. The flip is accepted if the energy does not increase, and with probability $e^{-\Delta E/T}$ otherwise. The magnetization is not fixed.

Optional arguments:

- `temp=1.0`: Metropolis temperature T ; `temp=0` only accepts flips that do not increase the energy
- `ntries=1e5`: number of attempted flips
- `resync_every=1000`: every how many attempts the energy is recomputed from scratch
- `checkpoint_at=None`: an integer or a list of integers; a copy of `li.s` after that many attempts is stored in `li.checkpoints`

There is no `patience` option, because the trajectory is meant for `latticegas.get_specific_heat()` and `get_susceptibility()`, and stopping early would bias them.

Sets `li.s` to the final configuration, and returns `(es, ms)`: the energy and the total magnetization $\sum_i s_i$ after each attempt

37.4.7 `li.optimize_conserved()`

Kawasaki dynamics: in each step a random up spin and a random down spin are exchanged, so the magnetization is fixed. Raises `ValueError` if `li.s` does not contain both up and down spins.

Optional arguments: the same as `li.optimize_energy()`, but with default `temp=0.1`, and also:

- `patience=None`: if given, stop after this many attempts without a new lowest energy

Sets `li.s` to the final configuration, and returns the energy after each attempt

37.4.8 `li.anneal()`

Simulated annealing: `li.optimize_energy()` at each temperature of a decreasing list, keeping the lowest-energy configuration of the whole schedule.

Optional arguments:

- `temps=None`: temperatures from high to low; by default 10 temperatures on a geometric scale from 2.0 to 0.05

- `ntries=1e4`: attempts per temperature
- `checkpoint_at=None`: as in `li.optimize_energy()`, counted over the whole schedule
- any other keyword of `li.optimize_energy()`, used at every temperature

Sets `li.s` to the best configuration, and returns `(es, ms)` for the whole schedule

37.4.9 `li.optimize_energy_multistart()`

Run `nstart` independent optimizations from random spins with the current magnetization, and keep the lowest energy. Parallel as in `lg.optimize_energy_multistart()`.

Optional arguments:

- `nstart=10`: number of runs
- any other keyword of `li.optimize_energy()` (`temp`, `ntries`), the same for every run

Sets `li.s` to the best configuration, and returns its energy

37.4.10 `li.get_local_energy()` / `li.get_local_field()`

`get_local_energy()` gives the energy of each site; the values add up to `li.get_energy()`. `get_local_field()` gives the effective field on each site, $h_i^{\text{eff}} = b_i + 2 \sum_k J_{ik} s_k$, so that flipping s_i costs exactly $2s_i h_i^{\text{eff}}$.

Returns an array over sites

37.4.11 `li.get_correlator()` / `li.get_structure_factor()`

The spin-spin correlator and the structure factor of the current spins `li.s`. The arguments are the same as in `lg.get_correlator()` and `lg.get_structure_factor()`.

37.4.12 `li.add_tensor()` / `li.regroup()`

The same as `lg.add_tensor()` and `lg.regroup()`.

Modifies `li` in place, no return value

37.4.13 `li.write()` / `li.read()`

Write the spins `li.s` to a text file, and read them back. `read()` takes the sign of each value, so the spins are exactly ± 1 , and raises `ValueError` if the number of sites in the file is not `li.nsites`.

Optional arguments:

- `name="SPIN.OUT"`: file name