pyqcm Documentation

Release v.1.0.7(git hash 20ce301)

David Sénéchal

Sep 22, 2022

CONTENTS:

CHAPTER

ONE

INTRODUCTION

1.1 What is pyqcm?

pyqcm is a python module that interfaces with a library written in C++: **qcm**. This library provide a collection of functions that help implement quantum cluster methods. Specifically, it provides an exact diagonalization solver for small clusters on which a Hubbard-like model is defined and provides functions to define infinite-lattice models and to embed the clusters into the lattice via *Cluster Pertrubation Theory* (CPT). Methods like the *Variational Cluster Approximation* (VCA) and *Cluster Dynamical Mean Field Theory* (CDMFT) are then implemented from qcm by the pyqcm module, which is written in Python only.

This document is not a review of the above methods; the reader is referred to the appropriate review articles for that. It is strictly a user's manual for the pyqcm module. Some degree of understanding of the above methods is however necessary to proceed. This document also provides insights as to the inner working of the code, and therefore consitutes also a embryonic developer's manual.

1.2 Installation

Pyqcm is written in Python and thus requires no compilation. However, it is based on a C++ library (**qcm**) provided with the distribution, and depends on **CUBA** for numerical integration and **BLAS-LAPACK** for elementary (non sparse) linear algebra.

The source code can be cloned with the following command:

```
git clone https://dsenech@bitbucket.org/dsenechQCM/qcm_wed.git
```

or, from github:

```
git clone https://github.com/dsenech/qcm_wed.git
git checkout master
```

Compiling can be done with pip:

```
cd <path_to_qcm_wed_folder>
pip install .
```

CHAPTER

TWO

MODELS AND CLUSTERS

Pyqcm can be applied to Hubbard and extended Hubbard models with one or more bands defined in 0 to 3 dimensions of space. These models are defined on a lattice, which may not be a Bravais lattice, but which contains a Bravais lattice as a subset. In other words, the *sites* of the lattice constitute the Bravais lattice itself and its *basis* within each unit cell.

2.1 Lattices and superlattices

In quantum cluster methods, the lattice is divided into repeated units that are generally (but not necessarily) larger than the unit cell. The repeated unit is called the the *super unit cell* (SUC) and is made of one or more *clusters*. The repetition defines a superlattice. For instance, the figure below shows how the honeycomb lattice is tiled with an 8-site super unit cell (shaded in blue), itself made of two 4-site clusters. The original Bravais basis vectors are $\mathbf{e}_1$ and $\mathbf{e}_2$. The superlattice vectors are $\mathbf{E}_1$ and $\mathbf{E}_2$. The inter-cluster links are indicated by dashed lines and the intra-cluster links by full lines.

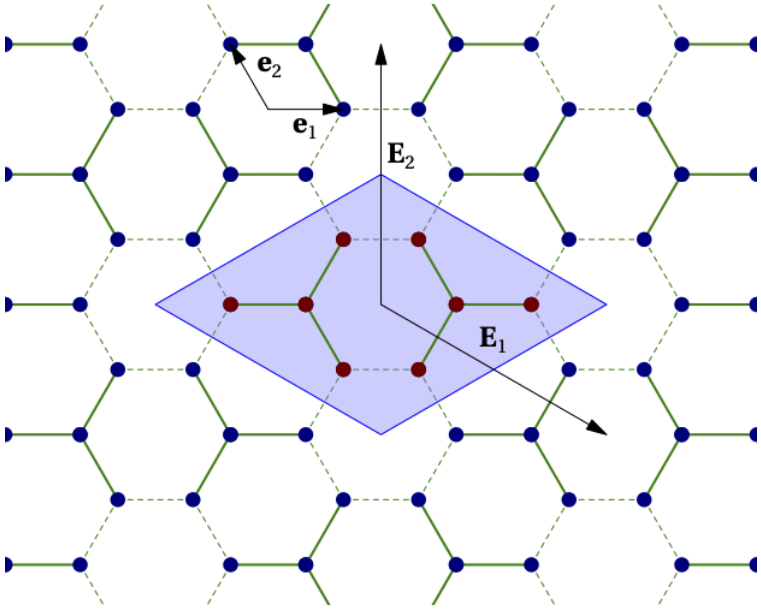

Fig. 1: Figure 1

Another example is the triangular lattice, where the supercell can be made of a two contiguous and inverted 15-site triangles, each of them being a cluster with the point group of the lattice, but not a repeatable pattern on the lattice. Only by adjoining two such clusters does one recover a repeatable unit:

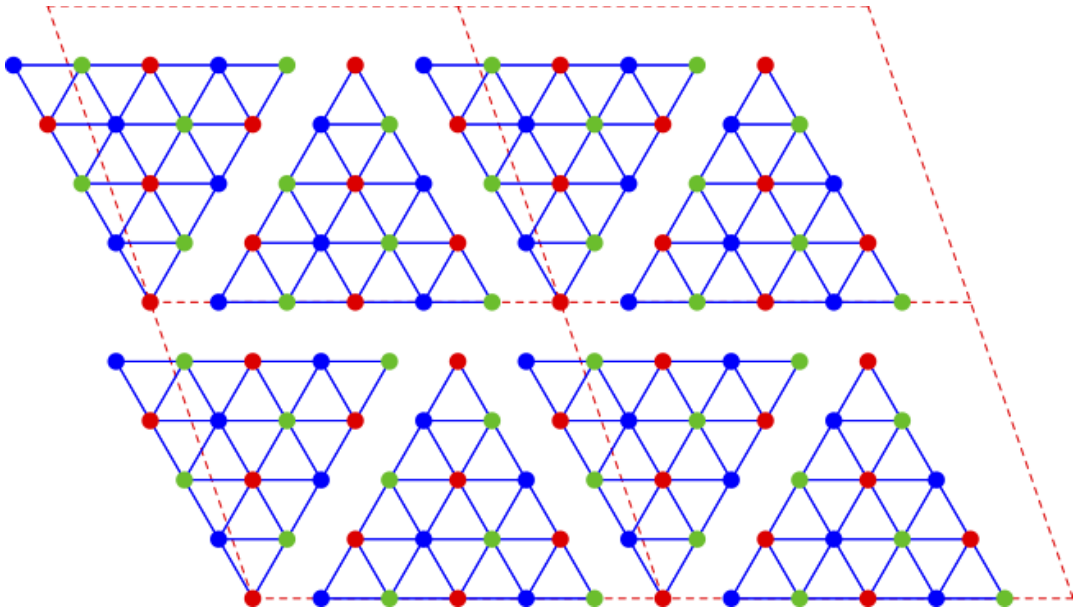

Fig. 2: Figure 2

Note that the triangular lattice is often thought of as made of three intertwined sublattices (A, B and C) whose sites are represented in blue, red and green on the figure. Note that the 15-site cluster treats these three sublattices symmetrically, which is useful when studying spiral magnetic order based on these three sublattices.

Let us summarize:

- A model is defined on a lattice of dimension D
- Cluster methods assume that this lattice is an assembly of small clusters and that a small number (possibly one) of these clusters form a repeated unit forming the basis of a superlattice of dimension D

2.2 Lattice models

In quantum cluster methods, one deals with a Hubbard-like model defined on the infinite lattice, the so-called *lattice model*, which we are attempting to solve approximately. The road to this approximate solution goes through defining closely related models on small clusters (the so-called *impurity models*), but more on this below. Let us start by explaining the range of lattice models that can be studied and defined in **pyqcm**.

Lattice models are defined by a Hamiltonian in dimension $D = 0, 1, 2$ or 3. D is the dimension of the Brillouin zone, i.e. the number of directions in which the model is infinite and a basic unit periodically repeated. For instance, one could consider a two-dimensional Hubbard model (a frequent case), or maybe a two-dimensional model with a finite number of layers in the third dimension, for instance to study heterostructures or a slab geometry, which still constitutes a two-dimensional model because it is infinite in two directions only. The model is made of one-body terms and interaction terms and has the general form

$$H = \sum_a h_a H_a$$

where each index a represent either a one-body operator or an interaction term, and h_a is the (real) value of the coefficient of each term. The operator H_a is Hermitian.

The parameters h_a of the lattice model are commonly referred to as 'lattice parameters' in this document and in the code, to distinguish them from the parameters of the various cluster Hamiltonians that make up the lattice.

In the code, the definition of an operator H_a, the value of its coefficient h_a in the Hamiltonian and of its expectation value $\langle H_a \rangle$ are encapsulated in the same data structure, so that the same symbol is often used to refer to all three concepts. This has to be kept in mind in this manual. When we say *the value of operator* H_a, we mean the value of its coefficient h_a, and we sometimes use the words *operator* and *parameter* interchangeably.

One distinguishes the following types of terms:

2.2.1 One-body operators

$$H_a = \sum_{\alpha\beta} t^{(a)}_{\alpha\beta} c^\dagger_\alpha c_\beta$$

where each of the indices α and β stands for a given site and spin (composite index (i, σ)). The hopping amplitude matrix $t^{(a)}_{\alpha\beta}$ is Hermitian. For a given pair of sites, labelled 1 and 2, the general form of the one-body term is

$$\sum_{\alpha\beta} \sum_{i,j} c^\dagger_{i\alpha} \tau^{(a)}_{ij} \sigma^{(b)}_{\alpha\beta} c_{j\beta}$$

The use of Pauli matrices $\sigma^b_{\alpha\beta}$ and τ^a_{ij} makes sure that the operator is Hermitian ($a = 0$, $b = 0$ correspond to the identity matrix). Case $a = 1$ and $b = 0$ corresponds to an ordinary hopping term, without spin flip. In specifying the precise shape of the one-body operator, it will suffice to specify the matrix types a and b for the spatial and spin parts respectively.

2.2.2 Anomalous operators

Anomalous (or pairing) operators are useful in describing superconductivity. They have the general form

$$H_a = \sum_{\alpha\beta} \left(\Delta^{(a)}_{\alpha\beta} c_\alpha c_\beta + \text{H.c.} \right)$$

where again the indices α and β are composite indices. The pairing amplitude $\Delta^{(a)}_{\alpha\beta}$ is antisymmetric. It may be symmetric in site indices and antisymmetric in spin indices (singlet superconductivity), or vice-versa (triplet superconductivity). A common way of representing these possibilities is the so-called *d-vector* :

$$H_a = \sum_{i,j,\sigma,\sigma'} \left(\Delta^{(a)}_{ij,b} c_{is} (i\sigma_b \sigma_2)_{ss'} c_{js'} + \text{H.c.} \right)$$

where the index b can take the values 0 to 3. The case $b = 0$ corresponds to singlet superconductivity (in which case $(\Delta^{(a)}_{ij,0} = \Delta^{(a)}_{ji,0})$ and the cases $b = 1, 2, 3$ corresponds to triplet superconductivity (in which case $(\Delta^{(a)}_{ij,b} = -\Delta^{(a)}_{ji,b})$.

pyqcm provides functions to define pairing operators by specifying the vectors $i - j$ and the values of b.

2.2.3 Density waves

Density wave operators are defined with a spatial modulation characterized by a wave vector $\mathbf{Q}$. They can be based on sites or on bonds. If the operator is a site density wave, its expression is

$$x \sum_{\mathbf{r}} A_{\mathbf{r}} \cos(\mathbf{Q} \cdot \mathbf{r} + \phi)$$

where

$$A_{\mathbf{r}} = n_{\mathbf{r}}, S^z_{\mathbf{r}}, S^x_{\mathbf{r}}$$

If it is a bond density wave, its expression is

$$\sum_{\mathbf{r}} \left[x c^\dagger_{\mathbf{r}} c_{\mathbf{r}+\mathbf{e}} e^{i(\mathbf{Q}\cdot\mathbf{r}+\phi)} + \text{H.c} \right]$$

where $\mathbf{e}$ is the link vector.

If it is a pair density wave, its expression is

$$\sum_{\mathbf{r}} \left[x c_{\mathbf{r}} c_{\mathbf{r}+\mathbf{e}} e^{i(\mathbf{Q}\cdot\mathbf{r}+\phi)} + \text{H.c} \right]$$

where $\mathbf{e}$ is the link vector and $\mathbf{r}$ a site of the lattice.

In **pyqcm** the different types of density waves are associated with keywords:

- 'N' : normal, i.e., a charge density wave
- 'Z' or 'spin' : a spin density wave, for S_z
- 'X' : spin density wave, for S_x
- 'singlet' : a pair-density wave, with singlet pairing. In this case and the following, the first creation operator in the above equation is replaced by an annihilation operator.
- 'dx', 'dy' 'dz' : a pair density wave, with triplet pairing, with d-vector in the directions x, y or z.

2.2.4 On-site interaction terms

Interaction terms can of the Hubbard or extended Hubbard type. The Hubbard on-site interaction is expressed as

$$H_a = \sum_i U_i^{(a)} n_{i\uparrow} n_{i\downarrow}$$

where $n_{is} = c^\dagger_{is} c_{is}$ is the number operator at site i and spin projection $s = \uparrow, \downarrow$.

2.2.5 Extended interaction terms

The extended (Coulomb) interaction is expressed as

$$H_a = \sum_{ij} V_{ij}^{(a)} n_i n_j \qquad (n_i = n_{i\uparrow} + n_{i\downarrow})$$

2.2.6 Hund's coupling terms

It is also possible to add a Hund's coupling term:

$$\hat{O} = \sum_{i,j} J_{ij} H_{ij}$$

where i and j stand for site indices and where

$$H_{ij} = -n_{i\uparrow} n_{j\uparrow} - n_{i\downarrow} n_{j\downarrow} + c^\dagger_{i\uparrow} c_{j\uparrow} c^\dagger_{j\downarrow} c_{i\downarrow} + c^\dagger_{j\uparrow} c_{i\uparrow} c^\dagger_{i\downarrow} c_{j\downarrow} + c^\dagger_{i\uparrow} c_{j\uparrow} c^\dagger_{i\downarrow} c_{j\downarrow} + c^\dagger_{j\uparrow} c_{i\uparrow} c^\dagger_{j\downarrow} c_{i\downarrow}$$

See, e.g., A. Liebsch and T. A. Costi, The European Physical Journal B, Vol. 51, p. 523 (2006). This can also be written as

$$H_{ij} = -n_{i\uparrow} n_{j\uparrow} - n_{i\downarrow} n_{j\downarrow} + (c^\dagger_{i\uparrow} c_{j\uparrow} + \text{H.c.})(c^\dagger_{i\downarrow} c_{j\downarrow} + \text{H.c.})$$

or as

$$H_{ij} = -c^\dagger_{i\uparrow} c^\dagger_{j\uparrow} c_{j\uparrow} c_{i\uparrow} - c^\dagger_{i\downarrow} c^\dagger_{j\downarrow} c_{j\downarrow} c_{i\downarrow} + c^\dagger_{i\uparrow} c^\dagger_{i\downarrow} c_{j\downarrow} c_{j\uparrow} + c^\dagger_{j\uparrow} c^\dagger_{j\downarrow} c_{i\downarrow} c_{i\uparrow} - c^\dagger_{j\uparrow} c^\dagger_{i\downarrow} c_{i\uparrow} c_{j\downarrow} - c^\dagger_{i\uparrow} c^\dagger_{j\downarrow} c_{j\uparrow} c_{i\downarrow}$$

2.2.7 Heisenberg coupling terms

Finally, it is possible to add a Heisenberg coupling term:

$$\hat{O} = \sum_{i,j} J_{ij} \mathbf{S}_i \cdot \mathbf{S}_j$$

2.3 Clusters

A cluster is a unit of the system (or *impurity*, in the DMFT jargon) that is solved exactly by the *impurity solver*, in our case by exact diagonalization. There may be more than one cluster in the repeated unit (or super unit cell). The spatial correlations are exactly taken care of only within the cluster. The size of the cluster is limited by the capacity to perform exact diagonalizations. Clusters may also be attached to bath sites, which are not part of the lattice model *per se* but serve to simulate each cluster's environment in cluster (or cellular) dynamical mean field theory (CDMFT). The cluster Hamiltonian H', or *reference Hamiltonian*, has the same form as the lattice Hamiltonian (except for the possible presence of a bath), but the values of its one-body terms, noted h'_a, may differ. The interaction terms are the same on the cluster and on the lattice. The case of extended interactions requires a special treatment because of the bonds broken across cluster boundaries, which must be treated within the Hartree approximation.

2.4 Multiband models

Multiband models are treated in **pyqcm** in a seemingly restrictive fashion, which in fact poses no restriction at all. It is assumed that each geometric site on the lattice correspond to a single orbital (with two spins). Models with more than one band must necessarily be accounted for by assigning differents sites to each band. The perfect example of this is the Hubbard model on the honeycomb (aka graphene) lattice. The lattice is not a Bravais lattice, since it contains one vacancy for every two occupied sites on an underlying triangular lattice. But there is no obligation in **qcm** for the lattice to be a Bravais lattice, i.e., for every site of the lattice to be occupied by an orbital (empty sites are allowed). The reason for doing things this way is that sometimes the two bands are equivalent, like in graphene. For instance, one can then define a 6-site cluster centered on a vacancy (the vertices of a hexagon). See Fig. 3 below. This cluster, interesting to use because of its symmetry, is a repeatable unit of the honeycomb lattice, but does not contain three identical unit cells of graphene, and could not be used if bands were treated only on a unit-cell basis. The concept of band in fact is only relevant to the lattice itself, not to the clusters, which ignore it.

Note that we will use the term *band* in a rather loose fashion, since most of the time we work in the *orbital* basis, not the *band* (or *Bloch*) basis. Since the word *orbital* is commonly used here to refer to the different sites of the impurity model, we use the word *band* to refer to the different *orbitals* of the unit cell of the lattice model, which is certainly a misnomer.

2.5 Bath sites

Each cluster can be associated with a bath of uncorrelated sites. This is illustrated on Fig. 4 for the four-site cluster of Fig. 1. From the point of view of the ED solver, the distinction between bath sites and physical sites is irrelevant. The only difference is that only physical sites have interactions. All the sites associated with a cluster are labelled consecutively, starting with the physical sites. The bath sites do not really have a position, even though they are pictured on Fig. 4 as if they occupied neighboring sites on the lattice.

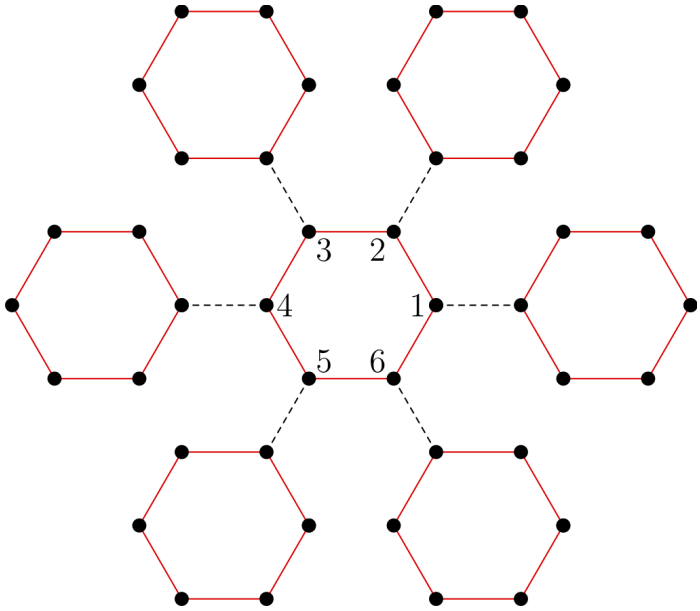

Fig. 3: Figure 3

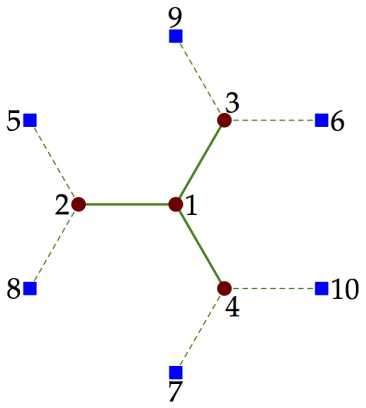

Fig. 4: Figure 4

2.6 Site and orbital labels

By convention, orbitals (or degrees of freedom) within a cluster are numbered and labelled consecutively as follows, where N_s is the number of physical sites, N_b the number of bath sites, and $N_o = N_s + N_b$:

- From 0 to $N_s - 1$, the spin up orbitals of the cluster proper.
- From N_s to $N_o - 1$, the spin up orbitals of the bath.
- From N_o to $N_o + N_s - 1$, the spin down orbitals of the cluster proper.
- From $N_o + N_s$ to $2N_o - 1$, the spin down orbitals of the bath.

The above numbering is relevant to the ED solver but not to the lattice model. Instead, orbitals within the super unit cell are labelled in the order of consecutive clusters and keep the same ordering as they already have within clusters, except that bath sites are excluded. This means in particular that the spin down orbitals of the super unit cell are no longer grouped together; rather, it is the cluster index that is the most external index. Explicitly, the ordering of labels in the super unit cell is as follows:

- From 0 to $N_{s1} - 1$, the spin up orbitals of cluster # 1 containing N_{s1} physical sites.
- From N_{s1} to $2N_{s1} - 1$, the spin down orbitals of cluster # 1.
- From $2N_{s1}$ to $2N_{s1} + N_{s2} - 1$, the spin up orbitals of cluster # 2 containing N_{s2} physical sites.
- From $2N_{s1} + N_{s2}$ to $2N_{s1} + 2N_{s2} - 1$, the spin down orbitals of cluster # 2.
- etc.

2.7 Green function indices and mixing states

The above scheme describes the indices labeling the degrees of freedom. A slightly different scheme labels the indices of the Green function, depending on the *mixing state*:

Normal mixing. If there are no anomalous terms nor spin-flip terms in the model, then the Green function (cluster or lattice) does not mix up and down spins and the Gorkov function vanishes. The cluster Green function for cluster i is a $N_{si} \times N_{si}$ matrix, associated with the destructions operators forming an array

$$(c_{j\uparrow}) \qquad j = 0, \dots, N_{si} - 1$$

The CPT Green function for the super unit cell (SUC) is then a $N_s \times N_s$ matrix, where N_s is the total number of physical sites in the repeated unit:

$$N_s = \sum_i N_{si}$$

This mixing state is called *normal mixing*.

Spin asymmetric mixing. If the model is not spin symmetric, i.e., if the up and down spins are not equivalent, then the down part of the Green function is different, but is still a $N_s \times N_s$ matrix. This case is called *spin asymmetric mixing*. It entails separate computations for the up and down spin Green functions.

Spin-flip mixing. If there are spin-flip terms, but sill no anomalous terms, the cluster Green function is a $2N_{si} \times 2N_{si}$ matrix, associated with the destructions operators forming an array

$$(c_{i\uparrow}) \oplus (c_{i\downarrow}) \qquad i = 0, \dots, N_{si} - 1$$

The CPT Green function for the SUC is then a $2N_s \times 2N_s$ matrix, and the cluster index is the outermost index. This is called *spin-flip mixing*.

Simple Nambu mixing. If there are anomalous terms, but no spin-flip terms, the cluster Green function is a $2N_{si} \times 2N_{si}$ matrix, associated with the destruction and creation operators forming an array

$$(c_{i\uparrow}) \oplus (c^\dagger_{i\downarrow}) \qquad i = 0, \dots, N_{si} - 1$$

The CPT Green function for the SUC is then a $2N_s \times 2N_s$ matrix, and the cluster index is still the outermost index. This is called *simple Nambu mixing*.

Full Nambu mixing. If there are both anomalous and spin-flip terms, the cluster Green function is a $4N_{si} \times 4N_{si}$ matrix, associated with the destruction and creation operators forming an array

$$(c_{i\uparrow}) \oplus (c_{i\downarrow}) \oplus (c^\dagger_{i\uparrow}) \oplus (c^\dagger_{i\downarrow}) \qquad i = 0, \dots, N_{si} - 1$$

The CPT Green function for the SUC is then a $4N_s \times 4N_s$ matrix, and the cluster index is still the outermost index. This is called *full Nambu mixing*. This also applies if there are no spin-flip terms, but triplet anomalous terms of the types *dx`and `dy*.

Different clusters may have different mixings, for instance if one of them describes a normal layer and another one a superconducting layer. However, the lattice model will have the more general mixing of the two and the Green function of each cluster will be *upgraded* to the lattice mixing as needed, for instance by doubling it by adding a Nambu transformed part.

CHAPTER

THREE

DEFINING MODELS

This section explains how to define models through python calls.

3.1 Defining cluster models

Clusters are the building blocks of lattice models. One needs to define them first. This is done through calls to *new_cluster_model()*. For instance:

```
from pyqcm import *
new_cluster_model('2x2_C2v', 4, 0, [[3, 4, 1, 2], [2, 1, 4, 3]])
```

The function `new_cluster_model(name, Ns, Nb, perm, bath_irrep)` takes the following arguments:

1. A name given to the cluster model
2. The number N_s of physical sites
3. The number N_b of bath sites
4. (optional) A list of permutations of the $N_o = N_s + N_b$ orbitals that define generators of the symmetries of the cluster.
5. (optional) A boolean flag that, if true, signals that bath orbitals belong to irreducible representations of the symmetry group of the cluster, instead of being part of permutations of the different orbitals of the cluster-bath system.

In the above example, a four-site cluster is defined, without any bath sites. The positions of the sites are not relevant to the impurity solver, and so are not defined at this stage. However, a plaquette geometry is implicit here, with the following site labels:

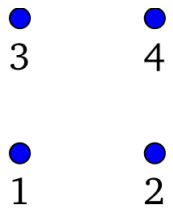

Fig. 1: Figure 1

The cluster symmetries (permutations) passed to the function thus correspond to reflexions with respect to the horizontal and vertical axes, respectively. The cluster symmetries will be used by the ED solver to lighten the exact diagonalization task. Large clusters will take less memory, convergence of the Lanczos method will be faster, and computing the Green function at a given frequency will be more efficient. The different permutations that constitute the generators must

commute with each other. They generate an Abelian group with g elements. Such a group has an equal number g of irreducible representations, numbered from 0 to $g-1$.

3.1.1 Defining operators on the cluster

Most operators in the model are best defined on the lattice and their restriction to the cluster is defined automatically, so there is no need to define them explicitly on each cluster of the super unit cell. This is not the case if one wants to use the ED solver as a standalone program without reference to a lattice model. Bath operators, on the other hand, need to be defined explicitly within the cluster model since they do not exist on the lattice model.

The following code defines the cluster and bath operators for the cluster illustrated in the last section, which we reproduce here:

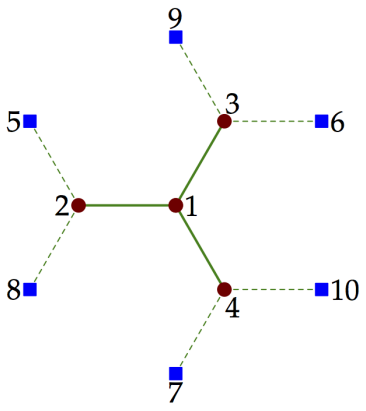

Fig. 2: Figure 2

Content of the cluster definition file:

```
from pyqcm import *

no = 10
ns = 4
new_cluster_model('clus', ns, no-ns, [[1, 3, 4, 2, 6, 7, 5, 9, 10, 8]])


new_cluster_operator('clus', 'bt1', 'one-body', [
    (2, 5, 1.0),
    (3, 6, 1.0),
    (4, 7, 1.0),
    (2+no, 5+no, 1.0),
    (3+no, 6+no, 1.0),
    (4+no, 7+no, 1.0)
])

new_cluster_operator('clus', 'bt2', 'one-body', [
    (2, 8, 1.0),
    (3, 9, 1.0),
    (4, 10, 1.0),
    (2+no, 8+no, 1.0),
    (3+no, 9+no, 1.0),
    (4+no, 10+no, 1.0)
```

(continues on next page)

(continued from previous page)

```
])

new_cluster_operator('clus', 'be1', 'one-body', [
    (5, 5, 1.0),
    (6, 6, 1.0),
    (7, 7, 1.0),
    (5+no, 5+no, 1.0),
    (6+no, 6+no, 1.0),
    (7+no, 7+no, 1.0)
])

new_cluster_operator('clus', 'be2', 'one-body', [
    (8, 8, 1.0),
    (9, 9, 1.0),
    (10, 10, 1.0),
    (8+no, 8+no, 1.0),
    (9+no, 9+no, 1.0),
    (10+no, 10+no, 1.0)
])
```

Note that the symmetry defined here is a rotation by 120 degrees. This generates the group C_3, which has complex representations. **pyqcm** can only deal with Abelian groups (the correct treatment of non-Abelian symmetries is too complex when computing Green functions for the benefits it would provide). In the above example, a better strategy when no complex operators are present would be to define only a C_2 symmetry based on one of three possible reflections. This would only provide 2 symmetry operations instead of 3, but the representations would be real instead of complex, thus saving more time and memory.

The function `new_cluster_operator(model, name, type, elements)` takes the following arguments:

1. The name of the model (the same given to `new_model()`)
2. The name of the operator
3. The type of operator; one of 'one-body', 'anomalous', 'interaction', 'Hund', 'Heisenberg'
4. An array of real matrix elements. Each element of the array is a 3-tuple giving the labels of the orbitals involved and the value of the matrix element itself. Note that spin-up and spin-down orbital labels are separated by the total number of orbitals on the custer, here *no=10*.

If a complex-valued operator is needed, then the function `new_cluster_operator_complex()` must be used, the only difference being that the actual matrix elements are complex numbers.

3.2 Defining lattice models

3.2.1 A simple example

The following simple example illustrates how to define a Hubbard model on the square lattice in two dimensions:

```
from pyqcm import *

new_cluster_model(clus', 4, 0, [[4, 3, 2, 1]])
add_cluster('clus', [0, 0, 0], [[0, 0, 0], [1, 0, 0], [0, 1, 0], [1, 1, 0]])
lattice_model('2x2_C2', [[2, 0, 0], [0, 2, 0]])
```

(continues on next page)

(continued from previous page)

```
interaction_operator('U')
hopping_operator('t', [1, 0, 0], -1)
hopping_operator('t', [0, 1, 0], -1)
hopping_operator('t2', [1, 1, 0], -1)
hopping_operator('t2', [-1, 1, 0], -1)
hopping_operator('t3', [2, 0, 0], -1)
hopping_operator('t3', [0, 2, 0], -1)
anomalous_operator('D', [1, 0, 0], 1)
anomalous_operator('D', [0, 1, 0], -1)
density_wave('M', 'Z', [1, 1, 0])
```

There is a call to `new_cluster_model()` to define a 2×2 plaquette with a C_2 symmetry (rotation by 180 degrees). Then this cluster is added to the lattice model via the function `add_cluster()` which takes 3 arguments:

1. The name of the cluster model (here `clus`)
2. The base position of the cluster within the super unit cell (here `[0,0,0]`)
3. An array of integer positions of the different cluster sites. This is where the geometry of the cluster appears.

The funcion `add_cluster()` is used to add individual clusters to the super unit cell. Only at the end of this process can the lattice model itself be defined properly by the function `lattice_model()`, which is the signal that no more clusters are needed. In the above example, the super unit cell contains a single cluster. Therefore `add_cluster()` is called only once and the lattice model can then be wrapped up by a call to `lattice_model()`, which takes three arguments (the last one optional):

1. The name of the model, for reporting purposes
2. The d vectors defining the superlattice vectors. d is the dimension of the model.
3. The d vectors defining the lattice vectors of the model. In the above example this is omitted because the default value is used: `[[1,0,0], [0,1,0]]`.

Note that all vectors used in these specifications have three components (even for lower dimensional models) and have **integer** components. They are in fact coefficients of basis vectors generating a working Bravais lattice and are therefore integers by definition (not all vectors of this working Bravais lattice need belong to the actual Bravais lattice of the model; the case of the graphene lattice is illustrated below).

Once the geometry of the model is defined, operators can be added to the model via various functions. The Hubbard on-site interaction is added with a call to `interaction_operator('U')`, whose sole argument in this case is the name we choose for that operator (more arguments are needed for other types of interactions, multi-band models, etc; see the detailed documents in the reference section on functions).

Nearest-neighbor hopping is defined with a call to `hopping_operator()`, which has three mandatory arguments:

1. The name of the operator
2. The link on which the operator is defined
3. The amplitude of the operator on that link.

Optional keyword arguments are needed for multi-band models, spin-flip operators, etc. In the above example this function is called twice with the same name but different links (along [1,0,0] and [0,1,0] respectively). The matrix elements generated by the two calls are simply added to the list associated with this operator. Similar calls are performed for the second-neighbor hopping `t2` and the third-neighbor hopping `t3`.

A d-wave pairing operator is defined via a call to `anomalous_operator()` , which takes the same arguments as `hopping_operator()`, i.e., name, link and amplitude. Note that two calls are made with the same name `D`, one in the x direction, the other one in the y direction, with amplitudes 1 and -1, in accordance with the d-wave character of the operator.

Finally, a density wave corresponding to (π, π) antiferromagnetism is added, with a call to `density_wave()`, which has 3 mandatory arguments:

1. The name of the operator (here 'M')
2. **The type of density-wave (here 'Z' for a spin density wave in the z component of the spin). Other possibilities are:**
    - 'N' : charge density wave
    - 'X' : spin density wave in the x component of the spin
    - 'spin' : same as Z
    - 'singlet' : singlet pairing
    - 'dx' : triplet pairing in the x direction using the $\mathbf{d}$ vector formalism.
    - 'dy' : same, in the y direction
    - 'dz' : same, in the z direction (this one does not require Nambu doubling)
3. The wavevector $\mathbf{Q}$ of the density wave. It has real components (in multiples of π) and it must be commensurate with the super unit cell within some small numerical tolerance.

Additional keyword arguments to `density_wave()` include the link on which the density wave is defined (for bond-density waves), bands involved (for multi-band models), additional phases and amplitudes, etc. Again, see the reference section for details.

3.2.2 A more complex example

The following example defines a model on the graphene lattice using two cluster within the super unit cell and the graphene lattice, as illustrated below:

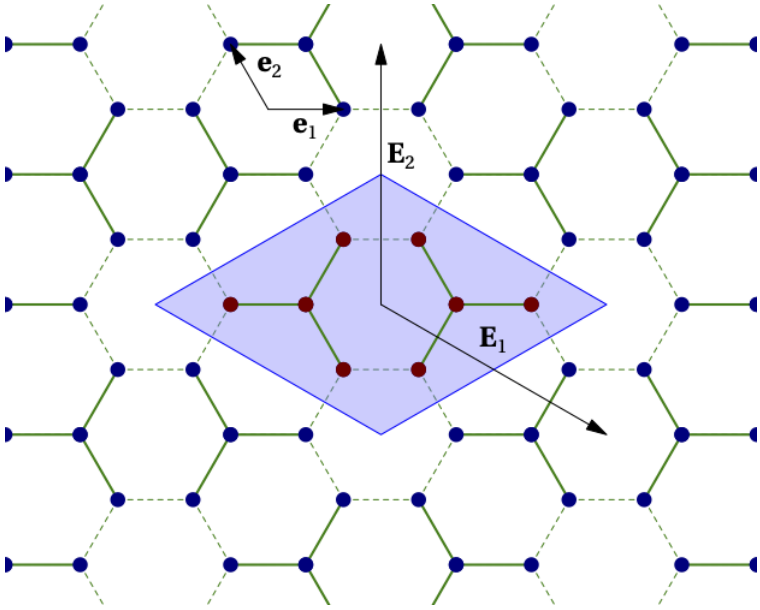

Fig. 3: Figure 3

Of possible set of function calls to define the Hubbard model on this system is:

```
import cluster_h4_6b_C3
from pyqcm import *

add_cluster('clus', [-1, 0, 0], [[0, 0, 0], [-1, 0, 0], [1, 1, 0], [0, -1, 0]])
add_cluster('clus', [1, 0, 0], [[0, 0, 0], [1, 0, 0], [-1, -1, 0], [0, 1, 0]])
lattice_model('h4_6b_C3', [[2, -2, 0],[2, 4, 0]], [[1, -1, 0], [2, 1, 0]])
set_basis([[1, 0, 0], [-0.5, 0.866025403784438, 0], [0, 0, 1]])

interaction_operator('U')
hopping_operator('t', [1,0,0], -1, band1=2, band2=1)
hopping_operator('t', [0,1,0], -1, band1=2, band2=1)
hopping_operator('t', [-1,-1,0], -1, band1=2, band2=1)
```

The first statement, `import cluster_h4_6b_C3`, imports a cluster definition file, for instance the one associated with Fig. 2 above. Using the name `clus` given to the cluster in that file, two copies of the clusters are added to the super unit cell. The positions associated with the two copies are different, but the cluster model is the same, which means that only one copy of the Hilbert space operators and bases necessary for the exact diagonalization will be constructed. The origin has been placed exactly between the two clusters. The positions in each cluster are defined relative to the base position of each cluster.

The integer positions are defined in terms of the basis defined by the call to `set_basis()`. The argument of that function is a set of real-valued vectors defining the basis vectors of the working Bravais lattice. On Fig. 3, the first two of these vectors are $\mathbf{e}_1$ and $\mathbf{e}_2$.

The call to `lattice_model()` defines both the superlattice vectors $\mathbf{E}_1$ and $\mathbf{E}_2$ (second argument) and the basis vectors of the model's physical Bravais lattice (third argument). The lattice basis vectors only serve to attribute band labels to the different sites. In *qcm*, each degree of freedom of a given spin (i.e. each orbital) must have its own site on the working Bravais lattice. The basis vectors of the physical Bravais lattice then define band labels (from 1 to $N_{\rm band}$) attributed to each site. The order in which bands are labelled depends on the order in which the sites appear. Given the above definitions and Fig. 3, the A sublattice of graphene corresponds to band 2 and the B sublattice to band 1.

Given that the model has two bands, the definition of the hopping operator `t` must contain band information: the keywords `band1` and `band2` are used to specify the band numbers associated with the two sites separated by the bond vector (link) given in argument. Internally, a loop is done over all sites of the super unit cell; the bond vector is used to identify a second site; if that site exists and if the bands associated with the two sites agree with `band1` and `band2`, then a matrix element is added to the operator. In the above example, three calls are needed because of the three directions (bonds).

The greatest risk in such calls is to mislabel the bands. In order to check that operators were defined properly, a call to `print_model()` is warranted. This function takes one mandatory argument: the name of a text file in which a detailed enumeration of model properties will be printed. Additional keyword arguments control the production of asymptote files (.asy) that graphically illustrate each operator. This is a powerful way to check the validity of the model definition.

3.2.3 Other examples

The distribution contains a folder (*notebooks*) that contains many examples of models and codes. New users are encouraged to study a few of these models and to consult the reference section for more detailed information about model building.

CHAPTER

FOUR

PARAMETER SETS, HILBERT SPACE SECTORS AND MODEL INSTANCES

4.1 Parameter sets

Each term in the lattice Hamiltonian is associated with a parameter h_a, and each cluster of the super unit cell has an equal number of parameters, for the corresponding terms of its Hamiltonian. If a bath is added to a cluster, then even more parameters are necessary to specify the cluster Hamiltonian. By default, the values of the cluster parameters are inherited from those of the lattice parameters. But their values may be different if desired. In addition, even within the lattice or within a cluster, one may want to impose automatic constraints between parameters, such as between the Coulomb repulsion U and the chemical potential μ, or other constraints coming from accidental symmetries. The large number of parameters and this inheritance of values needs to be managed. This is done via a *parameter set*.

On a cluster, the name of the operator will received a suffix `_i`, i being the cluster label (from 1 to the number of clusters $N_{\rm clus}$). Thus, from a lattice operator `t`, an operator `t_1` will be constructed on cluster #1, and an operator `t_2` on cluster #2, and so on. The values of the coefficients of these operators will be designated by the same symbols. This suffix scheme will apply to all operators defined in the model and on the clusters.

The qcm library contains a single instance of a structure called *parameter_set* that can define equivalences between parameters beyond those imposed by default, or release the default constraints. This structure is defined in pyqcm through the function `set_parameters(str)`. This may be the most important function of the library. It takes a single, long string argument, for instance:

```
set_parameters("""
    t = 1
    U = 8
    mu = 0.5*U
    M = 0
    M_1 = 0.1
""")
```

The values of the parameters are set by the equal sign. Dependent parameters are specified by a multiple of another parameter and the multiplication (*) sign. Note that a prefactor is always required, even when it is unity. Thus, you should write `M_2 = 1*M_1` and not `M_2=M_1`. Notice that setting a parameter to `0` causes qcm to *not create the operator* in the model.

4.2 Hilbert space sectors

For each cluster in the super unit cell, one must specify in which Hilbert space sector to look for the ground state. Each sector is specified by a Python string indicating which sectors of the Hilbert space must be considered in the exact diagonalization procedure. For instance, if the model conserves the number of particles and the z-component of the spin, the string `R0:N4:S0` means that the ground state of the cluster must be looked for in the Hilbert space sector with N=4 electrons and S=0 spin projection. R0 means that the ground state should belong to the trivial representation (labeled 0) of the point group. The spin projection is expressed by an integer $S = 2S_z$. Thus, `S1` means $S_z = \frac{1}{2}$ and `S-2` means $S_z = -1$.

The label of the irreducible representation is taken from the character table of the point group. For instance, in the important case of a C_{2v} symmetry, each irreducible representation is either even or odd with respect to horizontal and vertical reflections. The binary digits 0 and 1 are associated with even and odd symmetries, respectively. Thus, the binary number 10 = 2 labels a representation that is odd in y and even in x, 01 = 1 labels a representation that is even in y and odd in x, 11 = 3 is odd in both directions and 00=0 is even in both directions. These possibilities are thus represented by the strings R2, R1, R3 and R0, respectively.

Hilbert space sectors are a crucial element of the use of the library and may be the source of physical errors. Performance issues dictate that not all Hilbert space sectors should be checked for the true ground state for every calculation. Some judgement must be applied as to which sector or subset of sectors contains the true ground state. For a given cluster, a subset of sectors may be provided instead of a single one, by separating the sector keywords by slashes (/). For instance, the string indicating that the ground state should be search in the sectors of the trivial representation, with N=4 electrons and spin projection -1 or 1 is `R0:N4:S-1/R0:N4:S1`. Such a set of possible sectors is then specified with this call:

```
set_target_sectors(['R0:N4:S-1/R0:N4:S1'])
```

Note that the argument to that function is a list of strings, which in the above example contains a single element because the super unit cell contains a single cluster.

If spin is not conserved because of the presence of spin-flip terms, then the spin label must be omitted. For instance, the string "R0:N4" denotes the sector containing 4 electrons, in the trivial point group representation. An error message will be issued if the user specifies a spin sector in such cases, or inversely if the spin sector is not specified when spin is conserved.

The same is true in cases where particle number is not conserved (superconductivity): the number label must be omitted. For instance, the string `R0:S0` denotes the sector with zero spin, in the trivial point-group representation and an undetermined number of electrons.

When the target Hilbert space sector (or subset of sectors) specified in `set_target_sectors(...)` does not contain the true ground state, then the Green function computed thereafter will likely have the wrong normalization and analytic properties, because "excited states" obtained from the pseudo ground state by applying creation or annihilation operators may have a lower energy than the latter.

The ED solver has a global real-valued parameter called `temperature` which allows a certain mixtures of Hilbert space sectors in the density matrix of the system. This temperature must be low, as the ED solver is not a finite-temperature impurity solver. Only a few low-lying states may be computed. Within a Hilbert space sector, only the lowest-energy state will be found, unless the Davidson method is used and the global parameter `Davidson_states` is set to an integer value greater than one. A finite number of low-energy states will then be computed, and will contribute to the density matrix, provided their Boltzmann weight is larger than some minimum defined by the global parameter `minimum_weight`. The same applies to states coming from different sectors. This feature allows for a smoother transition between sectors when looping over chemical potential.

4.3 Model instances

Once a set of parameters has been recorded in the library's parameter set, one may define an instance of the lattice model by a call to `new_model_instance(lab)` , where `lab` is a non-negative integer that defines a label for that instance. All computations from the qcm library are performed on a given model instance, associated with particular values of the lattice and cluster parameters. In most cases a single model instance is needed at a given time, so that the label `lab` is 0 by default. A call to `new_model_instance(lab)` will erase any previous instance of the model with the same label.

Computations on a model instance are **lazy**. For instance, computing the ground state of the cluster is done only when needed, for instance when asking for the ground state averages or the Green function; computing the cluster Green function representation (either Lehmann or through continued fractions) is done only when a Green-function related quantity is needed, etc.

CHAPTER

FIVE

REFERENCE: LIST OF FUNCTIONS

pyqcm.**CPT_Green_function**(*z*, *k*, *spin_down=False*, *label=0*)

computes the CPT Green function at a given frequency

Parameters

- **z** – complex frequency
- **k** – single wavevector (ndarray(3)) or array of wavevectors (ndarray(N,3)) in units of π
- **spin_down** (*boolean*) – True is the spin down sector is to be computed (applies if mixing = 4)
- **label** (*int*) – label of the model instance

Returns

a single or an array of complex-valued matrices

pyqcm.**CPT_Green_function_inverse**(*z*, *k*, *spin_down=False*, *label=0*)

computes the inverse CPT Green function at a given frequency

Parameters

- **z** – complex frequency
- **k** – array of wavevectors (ndarray(N,3)) in units of π
- **spin_down** (*boolean*) – True is the spin down sector is to be computed (applies if mixing = 4)
- **label** (*int*) – label of the model instance

Returns

a single or an array of complex-valued matrices

pyqcm.**Green_function_average**(*cluster=0*, *spin_down=False*)

Computes the cluster Green function average (integral over frequencies)

Parameters

- **cluster** (*int*) – label of the cluster (0 to the number of clusters-1)
- **spin_down** (*boolean*) – true is the spin down sector is to be computed (applies if mixing = 4)

Returns

a complex-valued matrix

pyqcm.**Green_function_dimension**()

returns the dimension of the CPT Green function matrix :return: int

pyqcm.**Green_function_solve**(*label=0*)

Usually, the Green function representation is computed only when needed, in a just-in-time fashion (i.e. in a lazy way). This forces the computation of the Green function representation for the current instance (i.e. non lazy).

Parameters
label (*int*) – label of the model instance

Returns
None

pyqcm.**Lehmann_Green_function**(*k*, *band=1*, *spin_down=False*, *label=0*)

computes the Lehmann representation of the periodized Green function for a set of wavevectors

Parameters

- **k** – single wavevector (ndarray(3)) or array of wavevectors (ndarray(N,3)) in units of π
- **band** (*int*) – band index (starts at 1)
- **spin_down** (*boolean*) – True is the spin down sector is to be computed (applies if mixing = 4)
- **label** (*int*) – label of the model instance

Returns
a list of pairs {poles, residues}, each of poles and residues being itself a list.

exception pyqcm.**MinimizationError**

exception pyqcm.**MissingArgError**

exception pyqcm.**OutOfBoundsError**(*variable*)

exception pyqcm.**ParseError**

pyqcm.**Potthoff_functional**(*hartree=None*, *file='sef.tsv'*, *label=0*, *symmetrized_operator=None*)

computes the Potthoff functional for a given instance

Parameters

- **label** (*int*) – label of the model instance
- **file** (*str*) – name of the file to append with the result
- **hartree** (*(class hartree)*) – Hartree approximation couplings (see pyqcm/hartree.py)
- **symmetrized_operator** (*str*) – name of an operator wrt which the functional must be symmetrized

Returns
the value of the self-energy functional

pyqcm.**QP_weight**(*k*, *eta=0.01*, *band=1*, *spin_down=False*, *label=0*)

computes the k-dependent quasi-particle weight from the self-energy derived from the periodized Green function

Parameters

- **k** – single wavevector (ndarray(3)) or array of wavevectors (ndarray(N,3)) in units of π
- **eta** (*float*) – increment in the imaginary axis direction used to computed the derivative of the self-energy
- **band** (*int*) – band index (starts at 1)

- **spin_down** (*boolean*) – True is the spin down sector is to be computed (applies if mixing = 4)
- **label** (*int*) – label of the model instance

Returns
a single float or an array of floats, depending on the shape of k

exception pyqcm.**SolverError**

exception pyqcm.**TooManyIterationsError**

pyqcm.**V_matrix**(*z*, *k*, *spin_down=False*, *label=0*)

Computes the matrix $V = G_0^{-1} - G_0^{c-1}$ at a given frequency and wavevectors, where G_0 is the noninteracting Green function on the infinite lattice and G_0^c is the noninteracting Green function on the cluster.

Parameters

- **z** (*complex*) – frequency
- **k** (*wavevector*) – wavevector (ndarray(3)) in units of π
- **spin_down** (*boolean*) – True is the spin down sector is to be computed (applies if mixing = 4)
- **label** (*int*) – label of the model instance

Returns
a single (d,d) or an array (N,d,d) of complex-valued matrices. d is the reduced GF dimension.

exception pyqcm.**VarParamMismatchError**

exception pyqcm.**WrongArgumentError**

pyqcm.**add_cluster**(*name*, *pos*, *sites*, *ref=0*)

Adds a cluster to the repeated unit

Parameters

- **name** (*str*) – name of the cluster model
- **pos** (*[int]*) – base position of cluster (array of ints)
- **sites** (*[[int]]*) – list of positions of sites (2D array of ints)
- **ref** (*[int]*) – label of a previous cluster (starts at 1) to which this one is entirely equivalent (0 = no equivalence)

Returns
None

pyqcm.**anomalous_operator**(*name*, *link*, *amplitude*, *band1=None*, *band2=None*, ***kwargs*)

Defines an anomalous operator

Parameters

- **name** (*str*) – name of operator
- **link** (*[int]*) – bond vector (3-component integer array)
- **amplitude** (*complex*) – pairing multiplier
- **band1** (*int*) – number of the first band (None by default)
- **band2** (*int*) – number of the second band (None by default)

Keyword Arguments

- type (*str*) – one of 'singlet' (default), 'dz', 'dy', 'dx'

Returns
None

pyqcm.**averages**(*ops=[]*, *label=0*, *file='averages.tsv'*)

Computes the lattice averages of the operators present in the model

Parameters

- **label** (*int*) – label of the model instance
- **file** (*str*) – name of the file in which the information is appended

Return {str,float}
a dict giving the values of the averages for each parameter

pyqcm.**band_Green_function**(*z*, *k*, *spin_down=False*, *label=0*)

computes the periodized Green function at a given frequency and wavevectors, in the band basis (defined in the noninteracting model). It only differs from the periodized Green function in multi-band models.

Parameters

- **z** (*complex*) – frequency
- **k** – single wavevector (ndarray(3)) or array of wavevectors (ndarray(N,3)) in units of π
- **spin_down** (*boolean*) – true is the spin down sector is to be computed (applies if mixing = 4)
- **label** (*int*) – label of the model instance

Returns
a single (d,d) or an array (N,d,d) of complex-valued matrices. d is the reduced GF dimension.

pyqcm.**cluster_Green_function**(*cluster*, *z*, *spin_down=False*, *label=0*, *blocks=False*)

Computes the cluster Green function

Parameters

- **cluster** (*int*) – label of the cluster (0 to the number of clusters-1)
- **z** (*complex*) – frequency
- **spin_down** (*boolean*) – true is the spin down sector is to be computed (applies if mixing = 4)
- **label** (*int*) – label of the model instance (default 0)
- **blocks** (*boolean*) – true if returned in the basis of irreducible representations

Returns
a complex-valued matrix

pyqcm.**cluster_Green_function_dimension**(*clus=0*)

returns the dimension of the cluster Green function matrix :param int clus: label of the cluster :return: int

pyqcm.**cluster_QP_weight**(*cluster=0*, *eta=0.01*, *band=1*, *spin_down=False*, *label=0*)

computes the cluster quasi-particle weight from the cluster self-energy

Parameters

- **cluster** (*int*) – cluster label (starts at 0)

- **eta** (*float*) – increment in the imaginary axis direction used to computed the derivative of the self-energy
- **band** (*int*) – band index (starts at 1)
- **spin_down** (*boolean*) – True is the spin down sector is to be computed (applies if mixing = 4)
- **label** (*int*) – label of the model instance

Returns
a float

pyqcm.**cluster_averages**(*label=0*)

Computes the average and variance of all operators of the cluster model in the cluster ground state.

Parameters
label (*int*) – label of the cluster model instance

Returns
a dict str : (float, float) with the averages and variances as a function of operator name

pyqcm.**cluster_hopping_matrix**(*clus=0*, *spin_down=False*, *label=0*, *full=0*)

returns the one-body matrix of cluster no i for instance 'label'

Parameters

- **cluster** – label of the cluster (0 to the number of clusters - 1)
- **spin_down** (*boolean*) – True is the spin down sector is to be computed (applies if mixing = 4)
- **label** (*int*) – label of the model instance
- **full** (*boolean*) – if True, returns the full hopping matrix, including bath

Returns
a complex-valued matrix

pyqcm.**cluster_info**()

Return:A list of 4-tuples
(str, int, int, int, int): name of the cluster model, number of physical sites, number of bath sites, dimension of the Green function, number of point-group symmetry operations

pyqcm.**cluster_parameters**(*label=0*)

returns the values of the cluster parameters in a given instance, as well as the cluster model name

Parameters
label (*int*) – label of the cluster model instance

Returns
a tuple: dict{string,float}, str

pyqcm.**cluster_self_energy**(*cluster*, *z*, *spin_down=False*, *label=0*)

Computes the cluster self-energy

Parameters

- **cluster** (*int*) – label of the cluster (0 to the number of clusters -1)
- **z** (*complex*) – frequency
- **spin_down** (*boolean*) – true is the spin down sector is to be computed (applies if mixing = 4)

- **label** (*int*) – label of the model instance (default 0)

Returns
a complex-valued matrix

pyqcm.**complex_HS**(*label=0*)

returns True if the model needs a complex Hilbert space. False otherwise.

pyqcm.**density_matrix**(*sites*, *label=0*)

Computes the density matrix of subsystem A, defined by the array of site indices "sites" :param [int] sites: list of sites defining subsystem A :return [complex], [int32], [int32]: the density matrix, the left and right bases (spins up and down)

pyqcm.**density_wave**(*name*, *t*, *Q*, ***kwargs*)

Defines a density wave

Parameters

- **name** (*str*) – name of operator
- **t** (*str*) – type of density-wave – one of 'Z', 'X', 'Y', 'N'='cdw', 'singlet', 'dz', 'dy', 'dx'
- **Q** (*wavevector*) – wavevector of the density wave (in multiple of pi)

Keyword Arguments

- link (*[int]*) – bond vector, for bond density waves
- amplitude (*complex*) – amplitude multiplier. **Caution**: A factor of 2 must be used in some situations (see *Density waves*)
- band (*int*) – Band label (0 by default = all bands)
- phase (*float*) – real phase (as a multiple of pi)

Returns
None

pyqcm.**dispersion**(*k*, *spin_down=False*, *label=0*)

computes the dispersion relation for a single or an array of wavevectors

Parameters

- **k** (*wavevector*) – single wavevector (ndarray(3)) or array of wavevectors (ndarray(N,3)) in units of π
- **spin_down** (*boolean*) – True is the spin down sector is to be computed (applies if mixing = 4)
- **label** (*int*) – label of the model instance

Returns
a single (ndarray(d)) or an array (ndarray(N,d)) of real values (energies). d is the reduced GF dimension.

pyqcm.**dos**(*z*, *label=0*)

computes the density of states at a given frequency.

Parameters

- **z** (*complex*) – frequency
- **label** (*int*) – label of the model instance

Returns
ndarray(d) of real values, d being the reduced GF dimension

pyqcm.**epsilon**(*y*, *pr=False*)

Performs the epsilon algorithm for accelerated convergence

Parameters

- **y** (*[float]*) – sequence to be extrapolated
- **pr** (*boolean*) – if True, prints the resulting extrapolation

Return [float]
the extrapolated values

pyqcm.**explicit_operator**(*name*, *elem*, ***kwargs*)

Defines an explicit operator

Parameters

- **name** (*str*) – name of operator
- **elem** (*[(list, list, complex)]*) – List of tuples. Each tuple contains three elements (in order): a list representing position, a list representing link and a complex amplitude.

Keyword Arguments

- tau (*int*) – specifies the tau Pauli matrix (0,1,2,3)
- sigma (*int*) – specifies the sigma Pauli matrix (0,1,2,3)
- type (*str*) – one of 'one-body' [default], 'singlet', 'dz', 'dy', 'dx', 'Hubbard', 'Hund', 'Heisenberg', 'X', 'Y', 'Z'

Returns
None

pyqcm.**get_global_parameter**(*name*, *value=None*)

gets the value of a global parameter.

Parameters
name (*str*) – name of the global option

Returns
the value, according to type

pyqcm.**ground_state**(*file=None*)

Returns
a list of pairs (float, str) of the ground state energy and sector string, for each cluster of the system

pyqcm.**hopping_operator**(*name*, *link*, *amplitude*, *band1=None*, *band2=None*, ***kwargs*)

Defines a hopping term or, more generally, a one-body operator

Parameters

- **name** (*str*) – name of operator
- **link** (*[int]*) – bond vector (3-component integer array)
- **amplitude** (*float*) – hopping amplitude multiplier
- **band1** (*int*) – number of the first band (None by default)
- **band2** (*int*) – number of the second band (None by default)

Keyword Arguments

- tau (*int*) – specifies the tau Pauli matrix (0,1,2,3)
- sigma (*int*) – specifies the sigma Pauli matrix (0,1,2,3)

Returns
None

pyqcm.**hybridization_function**(*clus*, *z*, *spin_down=False*, *label=0*)

returns the hybridization function for cluster 'cluster' and instance 'label'

Parameters

- **clus** (*int*) – label of the cluster (0 to the number of clusters-1)
- **z** (*complex*) – frequency
- **spin_down** (*boolean*) – True is the spin down sector is to be computed (applies if mixing = 4)
- **label** (*int*) – label of the model instance

Returns
a complex-valued matrix

pyqcm.**interaction_operator**(*name*, *link=None*, *band1=None*, *band2=None*, ***kwargs*)

Defines an interaction operator of type Hubbard, Hund, Heisenberg or X, Y, Z

Parameters

- **name** (*str*) – name of the operator
- **band1** (*int*) – number of the first band (None by default)
- **band2** (*int*) – number of the second band (None by default)
- **link** (*list*) – link of the operator (None by default)

Keyword Arguments

- amplitude (*float*): amplitude multiplier
- type (*str*): one of 'Hubbard', 'Heisenberg', 'Hund', 'X', 'Y', 'Z'

Returns
None

pyqcm.**interactions**(*clus=0*)

returns the one-body matrix of cluster no i for instance 'label'

Parameters
cluster – label of the cluster (0 to the number of clusters - 1)

Returns
a real-valued matrix

pyqcm.**lattice_model**(*name*, *superlattice*, *lattice=None*)

initiates the lattice model.

Parameters

- **name** (*str*) – the name of the model
- **superlattice** (*[[int]]*) – array of integers of shape (d,3), d being the dimension

- **lattice** (*[[int]]*) – array of integers of shape (d,3), d being the dimension. If None, will be replaced by the unit lattice.

Returns
None

pyqcm.**matrix_elements**(*model*, *op*)

returns the type and matrix elements defining a Hermitian operator

Parameters

- **model** (*str*) – name of the cluster model
- **op** (*str*) – name of the operator

Returns
a tuple (typ, elem)

pyqcm.**mixing**()

returns the mixing state of the system:

- 0 – normal. GF matrix is n x n, n being the number of sites
- 1 – anomalous. GF matrix is 2n x 2n
- 2 – spin-flip. GF matrix is 2n x 2n
- 3 – anomalous and spin-flip (full Nambu doubling). GF matrix is 4n x 4n
- 4 – up and down spins different. GF matrix is n x n, but computed twice, with spin_down = false and true

Returns
int

pyqcm.**model_is_closed**()

returns True if the model can no longer be modified. False otherwise.

pyqcm.**model_size**()

Returns
a 5-tuple:

1. the size of the supercell
2. the number of bands
3. a tuple containing the sizes of each cluster
4. a tuple containing the sizes of each cluster's bath
5. a tuple containing the references of each cluster (label of reference cluster, from 0 to the nunber of clusters-1)

pyqcm.**momentum_profile**(*name*, *k*, *label=0*)

computes the momentum-resolved average of an operator

Parameters

- **name** (*str*) – name of the lattice operator
- **k** – array of wavevectors (ndarray(N,3)) in units of π
- **label** (*int*) – label of the model instance

Returns
an array of values

pyqcm.**new_cluster_model**(*name*, *n_sites*, *n_bath=0*, *generators=None*, *bath_irrep=False*)

Initiates a new model (no operators yet)

Parameters

- **name** (*str*) – name to be given to the model
- **n_sites** (*int*) – number of cluster sites
- **n_bath** (*int*) – number of bath sites
- **generators** (*[[int]]*) – symmetry generators (2D array of ints)
- **bath_irrep** (*[boolean]*) – True if bath orbitals belong to irreducible representations of the symmetry group

Returns
None

pyqcm.**new_cluster_model_instance**(*name*, *values*, *sec*, *label=0*)

Initiates a new instance of the cluster model

Parameters

- **name** (*str*) – name of the cluster model
- **values** (*{str,float}*) – values of the operators
- **sec** (*str*) – target Hilbert space sectors
- **label** (*int*) – label of model_instance

Returns
None

pyqcm.**new_cluster_operator**(*name*, *op_name*, *op_type*, *elem*)

creates a new operator from its matrix elements

Parameters

- **name** (*str*) – name of the cluster model to which the operator will belong
- **op_name** (*str*) – name of the operator
- **op_type** (*str*) – type of operator ('one-body', 'anomalous', 'interaction', 'Hund', 'Heisenberg', 'X', 'Y', 'Z')
- **elem** (*[(int,int,float)]*) – array of matrix elements (list of tuples)

Returns
None

pyqcm.**new_cluster_operator_complex**(*name*, *op_name*, *op_type*, *elem*)

creates a new operator from its complex-valued matrix elements

Parameters

- **name** (*str*) – name of the cluster model to which the operator will belong
- **op_name** (*str*) – name of the operator
- **op_type** (*str*) – type of operator ('one-body', 'anomalous', 'interaction', 'Hund', 'Heisenberg', 'X', 'Y', 'Z')

- **elem** (*[(int, int, complex)]*) – array of matrix elements (list of tuples)

Returns
None

pyqcm.**new_model_instance**(*label=0*, *record=False*)

Creates a new instance of the lattice model, with values associated to terms of the Hamiltonian.

Parameters

- **label** (*int*) – label of the model instance
- **record** (*boolean*) – if True, keeps a str-valued record of the instance in memory, containing all the data necessary to read the instance back without solving.

Return (class model_instance)
an instance of the class *model_instance*

pyqcm.**parameter_set**(*opt='all'*)

returns the content of the parameter set

Parameters
opt (*str*) – governs the action of the function

Returns
depends on opt

if opt = 'all', all parameters as a dictionary {str,(float, str, float)}. The three components are

(1) the value of the parameter,

(2) the name of its overlord (or None),

(3) the multiplier by which its value is obtained from that of the overlord.

if opt = 'independent', returns only the independent parameters, as a dictionary {str,float} if opt = 'report', returns a string with parameter values and dependencies.

pyqcm.**parameter_string**(*lattice=True*, *CR=False*)

Returns a string with the model parameters. :param boolean lattice : if True, only indicates the independent lattice parameters. :param boolean CR : if True, put each parameter on a line.

pyqcm.**parameters**(*label=0*)

returns the values of the parameters in a given instance

Parameters
label (*int*) – label of the model instance

Returns
a dict {string,float}

pyqcm.**params_from_file**(*out_file*, *n=0*)

reads an output file for parameters

Parameters

- **out_file** (*str*) – name of output file from which parameters are read
- **n** (*int*) – line number of data in output file (excluding titles).

Returns
a dict of (parameter, value)

pyqcm.**periodized_Green_function**(*z*, *k*, *spin_down=False*, *label=0*)

computes the periodized Green function at a given frequency and wavevectors

Parameters

- **z** (*complex*) – frequency
- **k** – single wavevector (ndarray(3)) or array of wavevectors (ndarray(N,3)) in units of π
- **spin_down** (*boolean*) – true is the spin down sector is to be computed (applies if mixing = 4)
- **label** (*int*) – label of the model instance

Returns

a single (d,d) or an array (N,d,d) of complex-valued matrices. d is the reduced GF dimension.

pyqcm.**periodized_Green_function_element**(*r*, *c*, *z*, *k*, *spin_down=False*, *label=0*)

computes the element (r,c) of the periodized Green function at a given frequency and wavevectors (starts at 0)

Parameters

- **r** (*int*) – a row index (starts at 0)
- **c** (*int*) – a column index (starts at 0)
- **z** (*complex*) – frequency
- **k** – array of wavevectors (ndarray(N,3)) in units of π
- **spin_down** (*boolean*) – true is the spin down sector is to be computed (applies if mixing = 4)
- **label** (*int*) – label of the model instance

Returns

a vector of complex numbers

pyqcm.**potential_energy**(*label=0*)

computes the potential energy for a given instance, as the functional trace of Sigma x G

Parameters

label (*int*) – label of the model instance

Returns

the value of the potential energy

pyqcm.**print_averages**(*ave*)

Prints the averages nicely on the screen

Parameters

ave (*dict*) – the dictionary produced by qcm.averages()

pyqcm.**print_cluster_averages**(*ave*)

Prints the averages nicely on the screen

Parameters

ave (*dict*) – the dictionary produced by qcm.averages()

pyqcm.**print_graph**(*name*, *sites*)

prints a graphiz (dot) program for the cluster

Parameters

- **name** (*str*) – name of the cluster model

- **sites** (*[[int]]*) – list of positions of sites (2D array of ints)

Returns
None

pyqcm.**print_model**(*filename*)

Prints a description of the model into a file

Parameters
filename (*str*) – name of the file

Returns
None

pyqcm.**print_options**(*opt=0*)

Prints the list of global options and parameters on the screen

Parameters
opt (*int*) – 0 -> prints to screen. 1 -> prints to latex. 2 -> prints to RST

pyqcm.**print_parameter_set**()

prints the content of the parameter set

pyqcm.**print_parameters**(*P*)

Prints the parameters nicely on the screen

Parameters
P (*dict*) – the dictionary of parameters

pyqcm.**print_wavefunction**(*label=0*, *pr=True*)

prints the ground state wavefunction(s) on the screen

Parameters

- **label** (*int*) – label of the model instance
- **pr** (*bool*) – prints wavefucntion to screen if pr=True

Returns
the wavefunction

pyqcm.**projected_Green_function**(*z*, *spin_down=False*, *label=0*)

computes the projected Green function at a given frequency, as used in CDMFT.

Parameters

- **z** (*complex*) – frequency
- **spin_down** (*boolean*) – true is the spin down sector is to be computed (applies if mixing = 4)
- **label** (*int*) – label of the model instance

Returns
the projected Green function matrix (d x d), d being the dimension of the CPT Green function.

pyqcm.**properties**(*label=0*)

Returns two strings of properties of a model instance

Parameters
label (*int*) – label of the model instance

Returns
a pair of strings (the description line and the data line).

pyqcm.**qmatrix**(*label=0*)

Returns the Lehmann representation of the Green function

Parameters
label (*int*) – label of the cluster model instance

Returns
2-tuple made of 1. the array of M real eigenvalues, M being the number of poles in the representation 2. a rectangular (L x M) matrix (real of complex), L being the dimension of the Green function

pyqcm.**read_cluster_model_instance**(*S*, *label=0*)

reads the solution from a string

Parameters

- **S** (*str*) – long string containing the solution
- **label** (*int*) – label of model_instance

Returns
None

pyqcm.**read_from_file**(*out_file*, *n=0*)

reads an output file for parameters

Parameters

- **out_file** (*str*) – name of output file from which parameters are read
- **n** (*int*) – line number of data in output file (excluding titles)

Returns
string to be added to an eventual input file

pyqcm.**read_from_file_legacy**(*filename*)

reads model parameters from a text file, for legacy results :param str filename: name of the input text file

pyqcm.**read_model**(*filename*)

Reads the definition of the model from a file

Parameters
file – name of the file, the same as that of the model, i.e., without the '.model' suffix.

Returns
None

pyqcm.**reduced_Green_function_dimension**()

returns the dimension of the reduced Green function, i.e. a simple multiple of the number of bands n, depending on the mixing state: n, 2n or 4n, and the number of bands

pyqcm.**sectors**(*R=None*, *N=None*, *S=None*)

Alternative method of setting target sectors (see *set_target_sectors(...)*). An example of usage is, R = [[1,2], [3,4]] –> ["R1.../R2...", "R3.../R4..."].

Parameters

- **R** (*int or [int] or [[int]]*) – Symmetry sector.
- **N** (*int or [int] or [[int]]*) – Particle number.
- **S** (*int or [int] or [[int]]*) – Total spin.

Returns
None

pyqcm.**self_energy**(*z*, *k*, *spin_down=False*, *label=0*)

computes the self-energy associated with the periodized Green function at a given frequency and wavevectors

Parameters

- **z** (*complex*) – frequency
- **k** – single wavevector (ndarray(3)) or array of wavevectors (ndarray(N,3)) in units of π
- **spin_down** (*boolean*) – True is the spin down sector is to be computed (applies if mixing = 4)
- **label** (*int*) – label of the model instance

Returns
a single (d,d) or an array (N,d,d) of complex-valued matrices. d is the reduced GF dimension.

pyqcm.**set_basis**(*B*)

Parameters
B – the basis (a (D x 3) real matrix)

Returns
None

pyqcm.**set_global_parameter**(*name*, *value=None*)

sets the value of a global parameter. If the global parameter is Boolean, then value is True by default and should not be specified.

Parameters

- **name** (*str*) – name of the global option
- **value** – value of that option (None, int, float or str)

Returns
None

pyqcm.**set_parameter**(*name*, *value*, *pr=False*)

sets the value of a parameter within a parameter_set

Parameters

- **name** (*str*) – name of the parameter
- **value** (*float*) – its value

Returns
None

pyqcm.**set_parameters**(*params*, *dump=True*)

Defines a new set of parameters, including dependencies

param tuple/str params
the values/dependence of the parameters (array of 2- or 3-tuples), or string containing syntax

param boolean dump
if True, sets the global str parameter_set_str tothe value

Return [tuple]
list of tuples of the form (str, float) or (str, float, str). The first form gives the parameter name and its value. The second gives the parameter name, a multiplier and the name of the reference parameter. See the documentation on the hierarchy of parameters.

pyqcm.**set_params_from_file**(*out_file*, *n=0*)

reads an output file for parameters

Parameters

- **out_file** (*str*) – name of output file from which parameters are read
- **n** (*int*) – line number of data in output file (excluding titles)

Returns
nothing

pyqcm.**set_target_sectors**(*sec*)

Define the Hilbert space sectors in which to look for the ground state

Parameters
sec (*[str]*) – the target sectors

Returns
None

pyqcm.**site_and_bond_profile**()

Computes the site and bond profiles in all clusters of the repeated unit

Returns
A pair of ndarrays

site profile – the components are x y z n Sx Sy Sz psi.real psi.imag

bond profile – the components are x1 y1 z1 x2 y2 z2 b0 bx by bz d0.real dx.real dy.real dz.real d0.imag dx.imag dy.imag dz.imag

pyqcm.**spatial_dimension**()

returns the spatial dimension (0, 1, 2, 3) of the model

pyqcm.**spectral_average**(*name*, *z*, *label=0*)

returns the contribution of a frequency to the average of an operator

Parameters

- **name** – name of the operator
- **z** – complex frequency
- **label** (*int*) – label of the model instance

Returns
float

pyqcm.**spin_spectral_function**(*freq*, *k*, *band=1*, *label=0*)

computes the k-dependent spin-resolved spectral function

Parameters

- **freq** – complex freqency
- **k** – single wavevector (ndarray(3)) or array of wavevectors (ndarray(N,3)) in units of π
- **band** (*int*) – band index (starts at 1)

- **label** (*int*) – label of the model instance

Returns
depending on the shape of k, a nd.array(3) of nd.array(N,3)

pyqcm.**susceptibility**(*op_name*, *freqs*, *label=0*)

computes the dynamic susceptibility of an operator

Parameters

- **op_name** (*str*) – name of the operator
- **freqs** (*[complex]*) – array of complex frequencies
- **label** (*int*) – label of cluster model instance

Returns
array of complex susceptibilities

pyqcm.**susceptibility_poles**(*op_name*, *label=0*)

computes the dynamic susceptibility of an operator

Parameters

- **name** (*str*) – name of the operator
- **label** (*int*) – label of cluster model instance

Returns [(float,float)]
array of 2-tuple (pole, residue)

pyqcm.**switch_cluster_model**(*name*)

switches cluster model to 'name'. Hack used in DCA.

pyqcm.**tk**(*k*, *spin_down=False*, *label=0*)

computes the k-dependent one-body matrix of the lattice model

Parameters

- **k** – single wavevector (ndarray(3)) or array of wavevectors (ndarray(N,3)) in units of π
- **spin_down** (*boolean*) – True is the spin down sector is to be computed (applies if mixing = 4)
- **label** (*int*) – label of the model instance

Returns
a single or an array of complex-valued matrices

pyqcm.**update_bath**(*label=0*)

updates the model parameters without creating a new instance or reseting the instance specified

Parameters
label (*int*) – label of the model instance

Returns
None

pyqcm.**variational_parameters**()

Returns
a list of the names of the variational parameters

pyqcm.**wavevector_grid**(*n=100*, *orig=[-1.0, -1.0]*, *side=2*, *k_perp=0*, *plane='z'*)

Produces a set of wavevectors for a MDC

Parameters

- **n** (*int*) – number of wavevectors on the side
- **orig** (*[float]*) – origin (in multiples of pi)
- **side** (*float*) – length of the side (in multiples of pi)
- **k_perp** (*float*) – momentum component in the third direction (in multiples of pi)
- **plane** (*str*) – momentum plane, 'xy'='z', 'yz'='x'='zy' or 'xz'='zx'='y'

Returns

ndarray of wavevectors (n*n x 3)

pyqcm.**wavevector_path**(*n=32*, *shape='triangle'*)

Builds a wavevector path and associated tick marks

Parameters

- **n** (*int*) – number of wavevectors per segment
- **shape** (*str*) – the geometry of the path, one of: line, halfline, triangle, graphene, graphene2, diagonal, cubic, cubic2, tetragonal, tetragonal2 OR a tuple with two wavevectors for a straight path between the two OR a filename ending with ".tsv". In the latter case, the file contains a tab-separated list of wavevectors (in units of pi) and tick marks: the first three columns are the x,y,z components of the wavevectors, and the last columns the strings (possibly latex) for the tick marks (write - in that column if you do not want a tick mark for a specific wavevector).

Returns tuple

1) a ndarray of wavevectors 2) a list of tick positions 3) a list of tick strings

pyqcm.**write_cluster_instance_to_file**(*filename*, *clus=0*)

Writes the solved cluster model instance to a text file

Parameters

- **filename** (*str*) – name of the file
- **clus** (*int*) – label of the cluster model instance

Returns

None

5.1 List of classes

class pyqcm.**model**

__init__()

class pyqcm.**model_instance**(*model*, *label*)

__init__(*model*, *label*)

CHAPTER

SIX

SPECTRAL PROPERTIES

This submodule provides functions that plot spectral properties (mostly the spectral function) as a function of frequency and/or wavevector.

6.1 List of functions

pyqcm.spectral.**DoS**(*w*, *eta=0.1*, *label=0*, *sum=False*, *progress=True*, *labels=None*, *colors=None*, *file=None*, *plt_ax=None*, ***kwargs*)

Plots the density of states (DoS) as a function of frequency

Parameters

- **wmax** (*float*) – the frequency range is from -wmax to wmax if w is a float. If wmax is a tuple then the range is (wmax[0], wmax[1]). wmax can also be an explicit list of real frequencies
- **eta** (*float*) – Lorentzian broadening, if w is real
- **label** (*int*) – label of the model instance
- **sum** (*boolean*) – if True, the sum of the DoS of all bands is plotted in addition to each band individually
- **progress** (*boolean*) – if True, prints computation progress
- **labels** (*[str]*) – labels of the different curves
- **colors** (*[str]*) – colors of the different curves
- **file** (*str*) – if not None, saves the plot in a file with that name
- **plt_ax** – optional matplotlib axis set, to be passed when one wants to collect a subplot of a larger set
- **kwargs** – keyword arguments passed to the matplotlib 'plot' function

Returns

None

pyqcm.spectral.**Fermi_surface**(*nk=64*, *label=0*, *band=None*, *quadrant=False*, *plane='xy'*, *k_perp=0.0*, *file=None*, *plt_ax=None*, ***kwargs*)

Plots the Fermi surface of the non-interacting model (2D)

Parameters

- **nk** (*int*) – number of wavevectors on each side of the grid
- **label** (*int*) – label of the model instance

- **band** (*int*) – if None, plots all the bands. Otherwise just plots the FS for that band (starts at 1)
- **quadrant** (*boolean*) – if True, plots the first quadrant of a square Brillouin zone only
- **plane** (*str*) – momentum plane, 'xy'='z', 'yz'='x'='zy' or 'xz'='zx'='y'
- **k_perp** (*float*) – momentum component in the third direction (in multiple of π)
- **file** (*str*) – if not None, saves the plot in a file with that name
- **plt_ax** – optional matplotlib axis set, to be passed when one wants to collect a subplot of a larger set
- **kwargs** – keyword arguments passed to the matplotlib 'plot' function

Returns
None

pyqcm.spectral.**G_dispersion**(*nk=64*, *label=0*, *band=None*, *period='G'*, *contour=False*, *inv=False*, *quadrant=False*, *datafile=None*, *max=None*, *k_perp=0.0*, *plane='xy'*, *file=None*, *plt_ax=None*, ***kwargs*)

Plots the eigenvalues of the inverse Green function at zero frequency

Parameters

- **nk** (*int*) – number of wavevectors on each side of the grid
- **label** (*int*) – label of the model instance
- **band** (*int*) – if 0, plots all the bands. Otherwise just shows the plot for that band (starts at 1)
- **period** (*str*) – periodization scheme ('G', 'M')
- **contour** (*boolean*) – True for a contour plot; otherwise a 3D plot.
- **inv** (*boolean*) – True if the inverse eigenvalues (inverse energies) are plotted instead
- **quadrant** (*boolean*) – if True, plots the first quadrant of a square Brillouin zone only
- **datafile** (*str*) – if different from None, just writes the data in a file and does not plot
- **max** (*float*) – energy range (from -max to max)
- **k_perp** (*float*) – momentum component in the third direction (in multiple of pi)
- **plane** (*str*) – momentum plane, 'xy'='z', 'yz'='x'='zy' or 'xz'='zx'='y'
- **file** (*str*) – if not None, saves the plot in a file with that name
- **plt_ax** – optional matplotlib axis set, to be passed when one wants to collect a subplot of a larger set
- **kwargs** – keyword arguments passed to the matplotlib 'plot' function

Returns
None

pyqcm.spectral.**Luttinger_surface**(*nk=200*, *label=0*, *band=1*, *quadrant=False*, *k_perp=0*, *plane='xy'*, *file=None*, *plt_ax=None*, ***kwargs*)

Plots the Luttinger surface (zeros of the Green function) in the Brillouin zone (2D)

Parameters

- **nk** (*int*) – number of wavevectors on each side of the grid

- **label** (*int*) – label of the model instance
- **band** (*int*) – band number (starts at 1)
- **quadrant** (*boolean*) – if True, plots the first quadrant of a square Brillouin zone only
- **k_perp** (*float*) – for 3D models, value of the component of k perpendicular to the plane
- **plane** (*str*) – for 3D models, plane of the plot ('z'='xy', 'y'='xz', 'x='yz')
- **file** (*str*) – if not None, saves the plot in a file with that name
- **plt_ax** – optional matplotlib axis set, to be passed when one wants to collect a subplot of a larger set
- **kwargs** – keyword arguments passed to the matplotlib 'plot' function

Returns
None

pyqcm.spectral.**cluster_spectral_function**(*wmax=6*, *eta=0.05*, *matsubara=False*, *clus=0*, *label=0*, *offset=2*, *full=False*, *opt=None*, *spin_down=False*, *blocks=False*, *file=None*, *plt_ax=None*, *realpart=False*, *color='b'*, ***kwargs*)

Plots the spectral function of the cluster in the site basis

Parameters

- **wmax** (*float*) – the frequency range is from -wmax to wmax if w is a float. If wmax is a tuple then the range is (wmax[0], wmax[1]). wmax can also be an explicit list of real frequencies
- **eta** (*float*) – Lorentzian broadening
- **matsubara** (*boolean*) – If True, the frequency range is along the imaginary frequency axis
- **clus** (*int*) – label of the cluster within the super unit cell (starts at 0)
- **label** (*int*) – label of the model instance
- **offset** (*float*) – vertical offset in the plot between the curves associated to successive wavevectors
- **full** (*boolean*) – if True, plots off-diagonal components as well
- **self** (*boolean*) – if True, plots the self-energy instead of the spectral function
- **opt** (*str*) – if None, G is computed. Other options are 'self' (self-energy) and 'hyb' (hybridization function)
- **spin_down** (*boolean*) – if True, plots the spin down part, if different
- **blocks** (*boolean*) – if True, gives the GF in the symmetry basis (block diagonal)
- **file** (*str*) – if not None, saves the plot in a file with that name
- **plt_ax** – optional matplotlib axis set, to be passed when one wants to collect a subplot of a larger set
- **color** – matplotlib color of the curves
- **kwargs** – keyword arguments passed to the matplotlib 'plot' function

Returns
the array of frequencies, the spectral weight

pyqcm.spectral.**gap**(*k*, *band=1*, *threshold=0.001*)

Computes the spectral gap for a series of wavevectors

:param k : set of wavevectors :param int band : band number (starts at 1) :param float threshold : weight below which a Lehmann contribution is deemed zero returns: an array of gap values

pyqcm.spectral.**hybridization_function**(*wmax=6*, *eta=0.01*, *matsubara=False*, *clus=0*, *realpart=False*, *label=0*, *file=None*, *plt_ax=None*, ***kwargs*)

This function plots the imaginary part of the hybridization function Gamma as a function of frequency. Only the diagonal elements are plotted, but for all clusters if there is more than one. The arguments have the same meaning as in *plot_spectrum*, except 'realpart' which, if True, plots the real part instead of the imaginary part.

Parameters

- **wmax** (*float*) – the frequency range is from -wmax to wmax if w is a float. If wmax is a tuple then the range is (wmax[0], wmax[1]). wmax can also be an explicit list of real frequencies
- **eta** (*float*) – Lorentzian broadening
- **matsubara** (*boolean*) – If True, the frequency range is along the imaginary frequency axis
- **clus** (*int*) – cluster index (starts at 0)
- **realpart** (*boolean*) – if True, the real part of the Green function is shown, not the imaginary part
- **label** (*int*) – label of the model instance
- **file** (*str*) – if not None, saves the plot in a file with that name
- **plt_ax** – optional matplotlib axis set, to be passed when one wants to collect a subplot of a larger set
- **kwargs** – keyword arguments passed to the matplotlib 'plot' function

Returns

None

pyqcm.spectral.**mdc**(*nk=200*, *eta=0.1*, *label=0*, *band=None*, *spin_down=False*, *quadrant=False*, *opt='GF'*, *k_perp=0*, *freq=0.0*, *max=None*, *plane='xy'*, *size=1.0*, *band_basis=False*, *sym=None*, *file=None*, *plt_ax=None*, ***kwargs*)

Plots the spectral weight at zero frequency in the Brillouin zone (2D)

Parameters

- **nk** (*int*) – number of wavevectors on each side of the grid
- **eta** (*float*) – Lorentzian broadening
- **label** (*int*) – label of the model instance
- **band** (*int*) – if None, sums all the bands. Otherwise just shows the weight for that band (starts at 1)
- **spin_down** (*boolean*) – true is the spin down sector is to be computed (applies if mixing = 4)
- **quadrant** (*boolean*) – if True, plots the first quadrant of a square Brillouin zone only
- **opt** (*str*) – The quantity to plot. 'GF' = Green function, 'self' = self-energy, 'Z' = quasiparticle weight
- **k_perp** (*float*) – momentum component in the third direction (in multiple of pi)
- **freq** (*float*) – frequency at which the spectral function is computed (0 by default)

- **max** (*float*) – maximum value of the plotting range (if None, maximum of the data)
- **plane** (*str*) – momentum plane, 'xy'='z', 'yz'='x'='zy' or 'xz'='zx'='y'
- **band_basis** – uses the band basis instead of the orbital basis (for multiband models)
- **sym** (*str*) – symmetrization option for the mdc
- **size** (*float*) – size of the plot, in multiple of the default (2 pi on the side)
- **file** (*str*) – if not None, saves the plot in a file with that name
- **plt_ax** – optional matplotlib axis set, to be passed when one wants to collect a subplot of a larger set
- **kwargs** – keyword arguments passed to the matplotlib 'plot' function

Returns
the contour plot object

pyqcm.spectral.**mdc_anomalous**(*nk=200*, *w=0.1j*, *label=0*, *bands=(1, 1)*, *self=False*, *im_part=False*, *quadrant=False*, *k_perp=0.0*, *plane='xy'*, *file=None*, *plt_ax=None*, ***kwargs*)

Plots the anomalous Green function or self-energy (2D)

Parameters

- **nk** (*int*) – number of wavevectors on each side of the grid
- **w** (*complex*) – complex frequency at which the Green function is computed
- **label** (*int*) – label of the model instance
- **bands** (*int*) – shows the weight for (b1,b2) (starts at 1), or numpy array of spin-Nambu projection
- **self** (*boolean*) – if True, plots the anomalous self-energy instead of the spectral function
- **im_part** (*boolean*) – if True, plots the imaginary part instead of the real part
- **quadrant** (*boolean*) – if True, plots the first quadrant of a square Brillouin zone only
- **k_perp** (*float*) – for 3D models, value of the component of k perpendicular to the plane
- **plane** (*str*) – for 3D models, plane of the plot ('z'='xy', 'y'='xz', 'x='yz')
- **file** (*str*) – if not None, saves the plot in a file with that name
- **plt_ax** – optional matplotlib axis set, to be passed when one wants to collect a subplot of a larger set
- **kwargs** – keyword arguments passed to the matplotlib 'plot' function

Returns
None

pyqcm.spectral.**momentum_profile**(*op*, *nk=50*, *label=0*, *quadrant=False*, *k_perp=0.0*, *plane='xy'*, *file=None*, *plt_ax=None*, ***kwargs*)

Plots the momentum-resolved average of operator op in the Brillouin zone (2D)

Parameters

- **op** (*str*) – name of the lattice operator
- **nk** (*int*) – number of wavevectors on each side of the grid
- **label** (*int*) – label of the model instance
- **quadrant** (*boolean*) – if True, plots the first quadrant of a square Brillouin zone only

- **k_perp** (*float*) – momentum component in the third direction (in multiple of pi)
- **plane** (*str*) – momentum plane, 'xy'='z', 'yz'='x'='zy' or 'xz'='zx'='y'
- **file** (*str*) – if not None, saves the plot in a file with that name
- **plt_ax** – optional matplotlib axis set, to be passed when one wants to collect a subplot of a larger set
- **kwargs** – keyword arguments passed to the matplotlib 'plot' function

Returns
None

pyqcm.spectral.**plot_dispersion**(*nk=64*, *label=0*, *spin_down=False*, *band=None*, *contour=False*, *datafile=None*, *quadrant=False*, *k_perp=0*, *plane='xy'*, *file=None*, *plt_ax=None*, *view_angle=None*, ***kwargs*)

Plots the dispersion relation in the Brillouin zone (2D)

Parameters

- **nk** (*int*) – number of wavevectors on each side of the grid
- **label** (*int*) – label of the model instance
- **spin_down** (*boolean*) – True is the spin down sector is to be computed (applies if mixing = 4)
- **band** (*int*) – if None, sums all the bands. Otherwise just shows the weight for that band (starts at 1)
- **contour** (*boolean*) – True if a contour plot is produced instead of a 3D plot.
- **datafile** (*str*) – if given, name of the data file (no extension please) in which the data is printed, for plotting with an external program. Does not plot. Will produce one file per band, with the .tsv extension.
- **quadrant** (*boolean*) – if True, plots the first quadrant of a square Brillouin zone only
- **k_perp** (*float*) – momentum component in the third direction (in multiple of pi)
- **plane** (*str*) – momentum plane, 'xy'='z', 'yz'='x'='zy' or 'xz'='zx'='y'
- **file** (*str*) – if not None, saves the plot in a file with that name
- **plt_ax** – optional matplotlib axis set, to be passed when one wants to collect a subplot of a larger set
- **view_angle** (*tuple*) – optional projection angle to pass to view_init() in the format of (elevation, azimuth) in degrees
- **kwargs** – keyword arguments passed to the matplotlib 'plot_surface' function

Returns
None

pyqcm.spectral.**segment_dispersion**(*path='triangle'*, *nk=64*, *label=0*, *file=None*, *plt_ax=None*, ***kwargs*)

Plots the dispersion relation in the Brillouin zone along a wavevector path

Parameters

- **path** (*str*) – wavevector path, as used by the function wavevector_path()
- **nk** (*int*) – number of wavevectors on each side of the grid
- **label** (*int*) – label of the model instance

- **file** (*str*) – if not None, saves the plot in a file with that name
- **plt_ax** – optional matplotlib axis set, to be passed when one wants to collect a subplot of a larger set
- **kwargs** – keyword arguments passed to the matplotlib 'plot' function

Returns
None

pyqcm.spectral.**spectral_function**(*wmax=6.0*, *eta=0.05*, *path='triangle'*, *nk=32*, *label=0*, *band=None*, *offset=2*, *opt='A'*, *Nambu_redress=True*, *inverse_path=False*, *title=None*, *file=None*, *plt_ax=None*, ***kwargs*)

Plots the spectral function $A(\mathbf{k}, \omega)$ along a wavevector path in the Brillouin zone. This version plots the spin-down part with the correct sign of the frequency in the Nambu formalism.

Parameters

- **wmax** (*float*) – the frequency range is from -wmax to wmax if w is a float. If wmax is a tuple then the range is (wmax[0], wmax[1]). wmax can also be an explicit list of real frequencies
- **eta** (*float*) – Lorentzian broadening
- **path** (*str*) – a keyword that is passed to pyqcm.wavevector_path() to produce a set of wavevectors along a path, or a tuple
- **nk** (*int*) – the number of wavevectors along each segment of the path (passed to pyqcm.wavevector_grid())
- **label** (*int*) – label of the instance of the model
- **band** (*int*) – if not None, only plots the spectral function associated with this orbital number (starts at 1). If None, sums over all bands.
- **offset** (*float*) – vertical offset in the plot between the curves associated to successive wavevectors
- **opt** (*str*) – 'A' : spectral function, 'self' : self-energy, 'Sx' : spin (x component), 'Sy' : spin (y component)
- **Nambu_redress** (*boolean*) – if True, evaluates the Nambu component at the opposite frequency
- **inverse_path** (*boolean*) – if True, inverts the path (k –> -k)
- **title** (*str*) – optional title for the plot. If None, a string with the model parameters will be used.
- **file** (*str*) – if not None, saves the plot in a file with that name
- **plt_ax** – optional matplotlib axis set, to be passed when one wants to collect a subplot of a larger set
- **kwargs** – keyword arguments passed to the matplotlib 'plot' function

Returns
None

pyqcm.spectral.**spectral_function_Lehmann**(*path='triangle'*, *nk=32*, *label=0*, *band=1*, *offset=0.1*, *lims=None*, *file=None*, *plt_ax=None*, ***kwargs*)

Plots a Lehmann representation of the spectral function along a wavevector path in the Brillouin zone. Singularities are plotted as impulses with heights proportionnal to the residue.

Parameters

- **path** – if a string, keyword passed to *pyqcm.wavevector_path()* to produce a set of wavevectors; else, explicit list of wavevectors (N x 3 numpy array).
- **nk** (*int*) – the number of wavevectors along each segment of the path (passed to pyqcm.wavevector_path())
- **label** (*int*) – label of the instance of the model
- **band** (*int*) – only plots the spectral function associated with this orbital number (starts at 1)
- **offset** (*float*) – vertical offset in the plot between the curves associated to successive wavevectors
- **lims** ((*float*,*float*)) – limits of the plot in frequency (2-tuple)
- **file** (*str*) – if not None, saves the plot in a file with that name
- **plt_ax** – optional matplotlib axis set, to be passed when one wants to collect a subplot of a larger set
- **kwargs** – keyword arguments passed to the matplotlib 'plot' function

Returns
None

pyqcm.spectral.**spin_mdc**(*nk=200*, *eta=0.1*, *label=0*, *band=None*, *quadrant=False*, *opt='spin'*, *freq=0.0*, *max=None*, *k_perp=0*, *plane='xy'*, *band_basis=False*, *file=None*, *plt_ax=None*, ***kwargs*)

Plots the spin spectral weight at zero frequency in the Brillouin zone (2D)

Parameters

- **nk** (*int*) – number of wavevectors on each side of the grid
- **eta** (*float*) – Lorentzian broadening
- **label** (*int*) – label of the model instance
- **band** (*int*) – if None, sums all the bands. Otherwise just shows the weight for that band (starts at 1)
- **quadrant** (*boolean*) – if True, plots the first quadrant of a square Brillouin zone only
- **opt** (*str*) – The quantity to plot. 'spin' = spin texture, 'spins' = spin texture (saturated), 'sz' = z-component, 'spinp' = modulus of xy-component
- **freq** (*float*) – frequency at which the spectral function is computed (0 by default)
- **max** (*float*) – maximum value of the plotting range (if None, maximum of the data)
- **k_perp** (*float*) – momentum component in the third direction (in multiple of pi)
- **plane** (*str*) – momentum plane, 'xy'='z', 'yz'='x'='zy' or 'xz'='zx'='y'
- **band_basis** – uses the band basis instead of the orbital basis (for multiband models)
- **file** (*str*) – if not None, saves the plot in a file with that name
- **plt_ax** – optional matplotlib axis set, to be passed when one wants to collect a subplot of a larger set
- **kwargs** – keyword arguments passed to the matplotlib 'plot' function

Returns
The contour plot object

CHAPTER

SEVEN

TOPOLOGY : BERRY CURVATURE COMPUTATIONS

This submodule provides functions that compute topological quantities (Berry curvature, Chern number, etc.)

pyqcm.berry.**Berry_curvature**(*nk=200*, *eta=0.0*, *period='G'*, *range=None*, *label=0*, *band=None*, *subdivide=False*, *plane='xy'*, *k_perp=0.0*, *file=None*, *plt_ax=None*, ***kwargs*)

Draws a 2D density plot of the Berry curvature as a function of wavevector, on a square grid going from -pi to pi in each direction.

Parameters

- **nk** (*int*) – number of wavevectors on the side of the grid
- **eta** (*float*) – imaginary part of the frequency at zero, i.e., w = eta*1j
- **period** (*str*) – type of periodization used (e.g. 'G', 'M', 'None')
- **range** (*list*) – range of plot [originX, originY, side], in multiples of pi
- **label** (*int*) – label of the model instance (0 by default)
- **band** (*int*) – the band to use in the computation (1 to number of bands). None (default) means a sum over all bands.
- **subdivide** (*int*) – True if plaquette subdivision is used.
- **k_perp** (*float*) – momentum component in the third direction (x pi)
- **plane** (*str*) – momentum plane, 'xy'='z', 'yz'='x'='zy' or 'xz'='zx'='y'
- **file** (*str*) – Name of the file to save the plot. If None, shows the plot on screen.
- **plt_ax** – optional matplotlib axis set, to be passed when one wants to collect a subplot of a larger set
- **kwargs** – keyword arguments passed to the matplotlib 'plot' function

Returns

the contourplot object of matplotlib

pyqcm.berry.**Berry_field_map**(*nk=40*, *nsides=4*, *plane='z'*, *k_perp=0.0*, *label=0*, *band=None*, *file=None*, *plt_ax=None*, ***kwargs*)

Creates a plot of the Berry flux as a function of wavevector

Parameters

- **nk** (*int*) – number of wavevector grid points on each side
- **nsides** (*int*) – number of sides of the polygon used to compute the circulation of the Berry field.
- **plane** (*str*) – momentum plane, 'xy'='z', 'yz'='x'='zy' or 'xz'='zx'='y'

- **k_perp** (*str*) – offset in wavevector in the direction perpendicular to the plane (x pi)
- **label** (*int*) – label of the model instance (0 by default)
- **band** (*int*) – the band to use in the computation (1 to number of bands). None (default) means a sum over all bands.
- **file** (*str*) – Name of the file to save the plot. If None, shows the plot on screen.
- **plt_ax** – optional matplotlib axis set, to be passed when one wants to collect a subplot of a larger set
- **kwargs** – keyword arguments passed to the matplotlib 'plot' function

Returns
the contourplot object of matplotlib, the quiver object of matplotlib

pyqcm.berry.**Berry_flux**(*k0*, *R*, *nk=40*, *plane='xy'*, *label=0*, *band=None*)

Computes the integral of the Berry connexion along a closed circle

Parameters

- **k0** (*int*) – center of the circle
- **R** (*float*) – radius of the circle
- **nk** (*int*) – number of wavevectors on the circle
- **plane** (*str*) – momentum plane, 'xy'='z', 'yz'='x'='zy' or 'xz'='zx'='y'
- **label** (*int*) – label of the model instance (0 by default)
- **band** (*int*) – the band to use in the computation (1 to number of bands). None (default) means a sum over all bands.
- **label** – label of the model instance (0 by default)

Returns float
the flux

pyqcm.berry.**Berry_flux_map**(*nk=40*, *plane='z'*, *dir='z'*, *k_perp=0.0*, *label=0*, *band=None*, *npoints=4*, *radius=None*, *file=None*, *plt_ax=None*, ***kwargs*)

Creates a plot of the Berry flux as a function of wavevector

Parameters

- **nk** (*int*) – number of wavevector grid points on each side
- **plane** (*str*) – momentum plane, 'xy'='z', 'yz'='x'='zy' or 'xz'='zx'='y'
- **dir** (*str*) – direction of flux, 'xy'='z', 'yz'='x'='zy' or 'xz'='zx'='y'
- **k_perp** (*str*) – offset in wavevector in the direction perpendicular to the plane (x pi)
- **label** (*int*) – label of the model instance (0 by default)
- **band** (*int*) – the band to use in the computation (1 to number of bands). None (default) means a sum over all bands.
- **npoints** (*int*) – nombre de points sur chaque boucle
- **file** (*str*) – Name of the file to save the plot. If None, shows the plot on screen.
- **plt_ax** – optional matplotlib axis set, to be passed when one wants to collect a subplot of a larger set
- **kwargs** – keyword arguments passed to the matplotlib 'plot' function

Returns
the contourplot object of matplotlib

pyqcm.berry.**Chern_number**(*nk=100*, *eta=0.0*, *period='G'*, *offset=[0.0, 0.0, 0.0]*, *label=0*, *band=None*, *subdivide=False*)

Computes the Chern number by summing the Berry curvature over wavevectors on a square grid going from (0,0) to (pi,pi)

Parameters

- **nk** (*int*) – number of wavevectors on the side of the grid
- **eta** (*float*) – imaginary part of the frequency at zero, i.e., w = eta*1j
- **period** (*str*) – type of periodization used (e.g. 'G', 'M', 'None')
- **offset** (*wavevector*) – wavevector offset of the computation grid
- **label** (*int*) – label of the model instance (0 by default)
- **band** (*int*) – the band to use in the computation (1 to number of bands). None (default) means a sum over all occupied bands.
- **subdivide** (*boolean*) – recursivity flag (wavevector grid subdivision)

Returns float
The Chern number

pyqcm.berry.**monopole**(*k*, *a=0.01*, *nk=20*, *label=0*, *band=None*, *subdivide=False*)

computes the topological charge of a node in a Weyl semi-metal

Parameters

- **k** (*[double]*) – wavevector, position of the node
- **a** (*float*) – half-side of the cube surrounding the node
- **nk** (*int*) – number of divisions along the side of the cube
- **label** (*int*) – label of the model instance (0 by default)
- **band** (*int*) – band to compute the charge of (if None, sums over all bands)
- **subdivide** (*booleean*) – True if subdivision is allowed (False by default)
- **label** – label of the model instance (0 by default)

Return float
the monopole charge

pyqcm.berry.**monopole_map**(*nk=40*, *label=0*, *band=None*, *plane='z'*, *k_perp=0.0*, *file=None*, *plt_ax=None*, ***kwargs*)

Creates a plot of the monopole density (divergence of B) as a function of wavevector

Parameters

- **nk** (*int*) – number of wavevector grid points on each side
- **plane** (*str*) – momentum plane, 'xy'='z', 'yz'='x'='zy' or 'xz'='zx'='y'
- **k_perp** (*str*) – offset in wavevector in the direction perpendicular to the plane (x pi)
- **label** (*int*) – label of the model instance (0 by default)
- **band** (*int*) – the band to use in the computation (1 to number of bands). None (default) means a sum over all bands.

- **file** (*str*) – Name of the file to save the plot. If None, shows the plot on screen.
- **plt_ax** – optional matplotlib axis set, to be passed when one wants to collect a subplot of a larger set
- **kwargs** – keyword arguments passed to the matplotlib 'plot' function

Returns

the contourplot object of matplotlib

CHAPTER

EIGHT

CLUSTER DYNAMICAL MEAN FIELD THEORY

This submodule provides functions that perform the CDMFT algorithm.

8.1 List of functions

pyqcm.cdmft.**cdmft**(*varia=None*, *beta=50*, *wc=2.0*, *maxiter=32*, *accur=0.001*, *accur_hybrid=0.0001*, *accur_dist=1e-10*, *alpha=0.0*, *displaymin=False*, *method='Nelder-Mead'*, *file='cdmft.tsv'*, *skip_averages=False*, *eps_algo=0*, *initial_step=0.1*, *hartree=None*, *check_sectors=None*, *grid_type='sharp'*, *counterterms=None*, *SEF=False*, *observables=None*, *verb=False*, *max_function_eval=5000000*)

Performs the CDMFT procedure

Parameters

- **varia** (*[str]*) – list of variational parameters OR tuple of two lists : bath energies and bath hybridizations OR function that returns dicts of bath energies and bath hybridizations given numeric arrays
- **beta** (*float*) – inverse fictitious temperature (for the frequency grid)
- **wc** (*float*) – cutoff frequency (for the frequency grid)
- **maxiter** (*int*) – maximum number of CDMFT iterations
- **accur** (*float*) – the procedure converges if parameters do not change by more than accur
- **accur_hybrid** (*float*) – the procedure converges on the hybridization function with this accuracy
- **accur_dist** (*float*) – convergence criterion when minimizing the distance function.
- **alpha** (*float*) – damping parameter (fraction of the previous iteration in the new one) OR (float,int) with number of iterations where damping is used (at the beginning if positive, at the end if negative)
- **displaymin** (*boolean*) – displays the minimum distance function when minimized
- **method** (*str*) – method to use, as used in scipy.optimize.minimize()
- **file** (*str*) – name of the file where the solution is written
- **skip_averages** (*boolean*) – if True, does NOT compute the lattice averages of the converged solution
- **eps_algo** (*int*) – number of elements in the epsilon algorithm convergence accelerator = 2*eps_algo + 1 (0 = no acceleration)

- **initial_step** (*float*) – initial step in the minimization routine
- **hartree** (*[class hartree]*) – mean-field hartree couplings to incorportate in the convergence procedure
- **check_sectors** (*boolean*) – the ground state is checked against the ground states of the sectors contained in target_sectors
- **grid_type** (*str*) – type of frequency grid along the imaginary axis : 'sharp', 'ifreq', 'self'
- **counterterms** (*[str]*) – list of counterterms names (cluster operators that should strive to have zero average)
- **SEF** (*boolean*) – if True, computes the Potthoff functional at the end
- **observable]** (*[class*) – list of observables used to assess convergence
- **verb** (*boolean*) – If True, prints debugging information
- **max_function_eval** (*int*) – maximum number of distance function evaluations when minimizing distance

Returns
None

pyqcm.cdmft.**cdmft_distance_debug**(*varia=None*, *vset=None*, *beta=50*, *wc=2.0*, *grid_type='sharp'*, *counterterms=None*)

Debugs the CDMFT distance function

Parameters

- **varia** (*[str]*) – list of variational parameters
- **vset** (*[[float]]*) – sets of bath parameters
- **beta** (*float*) – inverse fictitious temperature (for the frequency grid)
- **wc** (*float*) – cutoff frequency (for the frequency grid)
- **grid_type** (*str*) – type of frequency grid along the imaginary axis : 'sharp', 'ifreq', 'self'
- **counterterms** (*[str]*) – list of counterterms names (cluster operators that should strive to have zero average)
- **SEF** (*boolean*) – if True, computes the Potthoff functional at the end
- **observable** (*[class]*) – list of observables used to assess convergence

Returns
None

pyqcm.cdmft.**cdmft_forcing**(*field_name*, *seq*, *beta_seq=None*, ***kwargs*)

performs a sequence of CDMFT runs with the external field 'field_name' takes the successive values in 'seq'

Parameters

- **field_name** (*str*) – name of the forcing field
- **seq** (*[float]*) – sequence of values to be taken by the forcing field
- **beta_seq** (*[float]*) – an optional sequence of fictitious inverse temperatures (same length as seq)
- **kwargs** – named parameters passed to the CDMFT function

Returns
None

pyqcm.cdmft.**cdmft_variational_sequence**(*basic_params*, *varia_seq*, ***kwargs*)

performs a sequence of CDMFT runs with an increasing variational set.

Parameters

- **basic_params** (*str*) – specifies non variational parameters, in the format used by set_parameters()
- **varia_seq** (*[str]*) – a sequence of strings specifying additional variational parameters and their initial values
- **kwargs** – named parameters passed to the CDMFT function

Returns
None

pyqcm.cdmft.**forcing_sequence**(*f1*, *f2*, *beta1*, *beta2*, *n=6*)

generates logarithmic sequences of fields and temperatures, for use with cdmft_forcing

Parameters

- **f1** (*float*) – high value of the field
- **f2** (*float*) – low value of the field
- **beta1** (*float*) – low value of the inverse temperature
- **beta2** (*float*) – high value of the inverse temperature
- **n** (*int*) – number of values in the sequence

Return ([float],[float])
lists of field and beta values

pyqcm.cdmft.**moving_std**(*x*, *min*)

finds the subsequence with the smallest standard deviation, with minimum size min

Parameters

- **x** (*[float]*) – sequence
- **min** (*int*) – minimum length of subsequence

8.2 List of classes

class pyqcm.cdmft.**general_bath**(*name*, *ns*, *nb*, *spin_dependent=False*, *spin_flip=False*, *singlet=False*, *triplet=False*, *complex=False*, *sites=None*)

__init__(*name*, *ns*, *nb*, *spin_dependent=False*, *spin_flip=False*, *singlet=False*, *triplet=False*, *complex=False*, *sites=None*)

Defines a general bath (constructor)

Parameters

- **name** (*str*) – name of the cluster-bath model to be defined
- **ns** (*int*) – number of sites in the cluster
- **nb** (*int*) – number of bath orbitals in the cluster
- **spin_dependent** (*boolean*) – if True, the parameters are spin dependent
- **spin_flip** (*boolean*) – if True, spin-flip hybridizations are present

- **singlet** (*boolean*) – if True, defines anomalous singlet hybridizations
- **triplet** (*boolean*) – if True, defines anomalous triplet hybridizations
- **complex** (*boolean*) – if True, defines imaginary parts as well, when appropriate
- **sites** (*[[int]]*) – 2-level list of sites to couple to the bath orbitals (labels from 1 to ns). Format resembles [[site labels to bind to orbital 1], ...] .

starting_values(*c=1*, *e=(0.5, 1.5)*, *hyb=(0.5, 0.2)*, *shyb=(0.1, 0.05)*, *pr=False*)

returns an initialization string for the bath parameters

Parameters

- **c** (*int*) – cluster label (starts at 1)
- **e** (*(float,float)*) – bounds of the values for the bath energies (absolute value)
- **hyb** (*(float,float)*) – average and deviation of the normal hybridization parameters
- **shyb** (*(float,float)*) – average and deviation of the anomalous hybridization parameters
- **pr** (*boolean*) – prints the starting values if True

Return str
initialization string

starting_values_PH(*c=1*, *e=(1, 0.5)*, *hyb=(0.5, 0.2)*, *phi=None*, *pr=False*)

returns an initialization string for the bath parameters, in the particle-hole symmetric case.

Parameters

- **c** (*int*) – cluster label
- **e** (*(float)*) – range of bath energies
- **hyb** (*(float,float)*) – average and deviation of the normal hybridization parameters
- **phi** (*[int]*) – PH phases of the cluster sites proper
- **pr** (*boolean*) – if True, prints info

Return str
initialization string

varia(*H=None*, *E=None*, *c=1*, *spin_down=False*)

creates a dict of variational parameters to values taken from the hybridization matrix H and the energies E, for cluster c

Parameters

- **H** (*ndarray*) – matrix of hybridization values
- **E** (*ndarray*) – array of energy values
- **spin_down** (*boolean*) – True for the spin-down values

Return {str,float}
dict of variational parameters to values

varia_E(*c=1*)

returns a list of parameter names from the bath energies with the suffix appropriate for cluster c

Parameters
c (*int*) – label of the cluster (starts at 1)

Return [str]
list of parameter names from the bath energies with the suffix appropriate for cluster c

`varia_H(c=1)`

returns a list of parameter names from the bath hybridization with the suffix appropriate for cluster c

Parameters
c (*int*) – label of the cluster (starts at 1)

Return [str]
list of parameter names from the bath hybridization with the suffix appropriate for cluster c

CHAPTER

NINE

THE VARIATIONAL CLUSTER APPROXIMATION

This submodule provides functions that implements the Variational Cluster Approximation (VCA)

9.1 List of functions

pyqcm.vca.**plot_GS_energy**(*param*, *prm*, *clus=0*, *file=None*, *plt_ax=None*, ***kwargs*)

Draws a plot of the ground state energy as a function of a parameter param taken from the list *prm*. The results are going to be appended to 'GS.tsv'

Parameters

- **param** (*str*) – name of the parameter (independent variable)
- **prm** (*[float]*) – list of values of the parameter
- **clus** (*int*) – label of the cluster (starts at 0)
- **file** (*str*) – if not None, saves the plot in a file with that name
- **plt_ax** – optional matplotlib axis set, to be passed when one wants to collect a subplot of a larger set
- **kwargs** – keyword arguments passed to the matplotlib 'plot' function

Returns

None

pyqcm.vca.**plot_sef**(*param*, *prm*, *file='sef.tsv'*, *accur_SEF=0.0001*, *hartree=None*, *show=True*, *symmetrized_operator=None*)

Draws a plot of the Potthoff functional as a function of a parameter param taken from the list prm. The results are going to be appended to 'sef.tsv'

Parameters

- **param** (*str*) – name of the parameter (independent variable)
- **prm** (*[float]*) – list of values of the parameter
- **accur_SEF** (*float*) – precision of the computation of the self-energy functional
- **hartree** (*(class hartree)*) – Hartree approximation couplings (see pyqcm/hartree.py)
- **show** (*boolean*) – if True, the plot is shown on the screen.

Returns

None

pyqcm.vca.**transition_line**(*varia*, *P1*, *P1_range*, *P2*, *P2_range*, *delta*, *verb=False*)

Builds the second-order transition line as a function of a control parameter P1. The results are written in the file *transition.tsv*

Parameters

- **varia** (*str*) – variational parameter
- **P1** (*str*) – control parameter
- **P1_range** (*[float]*) – an array of values of P1
- **P2** (*str*) – dependent parameter
- **P2_range** (*(float)*) – 2-uple of values of P2 that bracket the transition
- **verb** (*boolean*) – If True, prints progress
- **delta** (*float*) – at each step, the new bracket for P2 will be P2c +/- delta, P2c being the previous critical value

Returns

None

pyqcm.vca.**vca**(*var2sef=None*, *names=None*, *start=None*, *steps=None*, *accur=None*, *max=None*, *file='vca.tsv'*, *accur_grad=1e-06*, *max_iter=30*, *max_iter_diff=None*, *method='NR'*, *hartree=None*, *hartree_self_consistent=False*, *symmetrized_operator=None*, *var_max_start=None*)

Performs a VCA with the QN or NR method

Parameters

- **var2sef** – function that converts variational parameters to model parameters
- **names** (*str or [str]*) – names of the variational parameters
- **start** (*float or [float]*) – starting values
- **steps** (*float or [float]*) – initial steps
- **accur** (*float or [float]*) – accuracy of parameters (also step for 2nd derivatives)
- **max** (*float or [float]*) – maximum values that are tolerated
- **accur_grad** (*float*) – max value of gradient for convergence
- **max_iter** (*int*) – maximum number of iterations in the procedure
- **max_iter_diff** (*float*) – optional maximum value of the maximum step in the quasi-Newton method
- **method** (*str*) – method used to optimize ('SYMR1', 'NR', 'BFGS', 'altNR', 'Nelder-Mead', 'COBYLA', 'Powell', 'CG', 'minimax')
- **hartree** (*(class hartree)*) – Hartree approximation couplings (see pyqcm/hartree.py)
- **hartree_self_consistent** (*boolean*) – True if the Hartree approximation is treated in the self-consistent, rather than variational, way.
- **symmetrized_operator** (*str*) – name of an operator wrt which the functional must be symmetrized
- **var_max_start** (*int*) – label of the first variable for which the function is a maximum (minimal vars first, maximal vars last)

Returns

None

CHAPTER

TEN

HARTREE APPROXIMATION AND COUNTERTERMS

10.1 The Hartree approximation

Extended interactions require an extension of quantum cluster methods: inter-cluster interactions have to be cut off and replaced by a mean field, in the Hartree approximation. An extended interaction on the cluster is then reduced to

$$\sum_{(ij)} V_{ij}^c n_i n_j + \sum_{(ij)} V_{ij}^m \left(n_i \langle n_j \rangle + n_j \langle n_i \rangle - \langle n_i \rangle \langle n_j \rangle \right)$$

where V_m stands for the terms of V_{ij} which need to be treated in the Hartree approximation, and V^c those that do not need to be. (ij) stands for a pair of sites ($i \neq j$).

Since V_m is real symmetric, the mean-field Hamiltonian may be recast into

$$H_m = \sum_{ij} V_{ij}^m \left(n_i \langle n_j \rangle - \frac{1}{2} \langle n_i \rangle \langle n_j \rangle \right)$$

where now the sum is over independent values of i and j. V^m can be diagonalized by an orthogonal transformation L: $V^m = L \Lambda \tilde{L}$. One then defines eigenoperators $O_a = L_{aj} n_j$ such that

$$H_m = \sum_a \lambda_a \left\{ O_a \langle O_a \rangle - \frac{1}{2} \langle O_a \rangle^2 \right\}$$

Now, let us consider the mean values $\langle O_a \rangle$ as adjustable variational (or mean-field) parameters. Then, in terms of the coefficients $h_a = \lambda_a \langle O_a \rangle$ of the operator O_a, the mean-field Hamiltonian takes the following form:

$$H_m = \sum_a \left\{ h_a O_a - \frac{h_a^2}{2\lambda_a} \right\}$$

The variation of parameter h_a the yields

$$\langle O_a \rangle = \frac{h_a}{\lambda_a} \rightarrow h_a = \lambda_a \langle O_a \rangle$$

The Hartree procedure consists in starting with trial values of $_a$ and iteratively performing the above assignation until convergence.

10.2 List of functions

class pyqcm.hartree.**hartree**(*Vm*, *V*, *eig*, *accur=0.0001*, *lattice=False*)

This class contains the elements needed to perform the Hartree approximation for the inter-cluster components of an extended interaction. The basic self-consistency relation is

$$v_m = ve\langle V_m \rangle$$

where *v* is the coefficient of the operator *V* and v_m that of the operator V_m, and *e* is an eigenvalue specific to the cluster shape and the interaction. $\langle V_m \rangle$ is the average of the operator V_m, taken as a lattice of as a cluster average.

attributes:

- Vm (str) : mean-field operator
- V (str) : extended interaction
- eig (float) : eigenvalue *e* of the mean-field operator in the self-consistency relation
- lattice (boolean) : True if lattice averages are used
- diff : difference between successive values of v_m‘
- ave : averasge of the operator V_m
- accur : desired accuracy

__init__(*Vm*, *V*, *eig*, *accur=0.0001*, *lattice=False*)

Parameters

- **Vm** (*str*) – name of the mean-field operator
- **V** (*str*) – name of the interaction operator
- **eig** (*float*) – eigenvalue
- **accur** (*float*) – required accuracy of the self-consistent procedure
- **lattice** (*boolean*) – if True, the lattice average is used, otherwise the cluster average

converged()

Tests whether the mean-field procedure has converged

Return boolean
True if the mean-field procedure has converged

omega()

returns the constant contribution, added to the Potthoff functional

omega_var()

returns the constant contribution, added to the Potthoff functional

update(*pr=False*)

Updates the value of the mean-field operator based on its average

Parameters
pr (*boolean*) – if True, progress is printed on the screen

10.3 Counterterms

class pyqcm.hartree.**counterterm**(*name*, *clus*, *S*, *accur=0.0001*)

That class contains information about operators that are added to the cluster Hamiltonian in order to make their average vanish. The coefficient *v* of the operator *V* is adjusted so that $\langle V \rangle = 0$. The coefficient is updated using the relation

$$\langle O_1 \rangle - \langle O_2 \rangle = S(v_1 - v_2)$$

where $v_{1,2}$ are two values of the coefficient of the operator used in succession, $\langle v_{1,2} \rangle$ are the corresponding averages and *S* is the (varying) slope.

attributes:

- name (str) : name of the countererm operator
- fullname (str) : name, including the cluster label
- clus (int) : label of the cluster
- S (float) : slope
- accur (float) : accuracy to which the average must vanish.
- v (float) : value of the coefficient of the operator
- v0 (float) : previous value of the coefficient of the operator (previous iteration)

__init__(*name*, *clus*, *S*, *accur=0.0001*)

Constructor

Parameters

- **name** (*str*) – name of the counterterm (previously defined in the model)
- **clus** (*int*) – cluster label to which it is applied
- **S** (*float*) – initial value of the slope
- **accur** (*float*) – accuracy

converged()

Performs a convergence test

Return boolean
True if converged

update(*first=False*)

updates the value of the operator given its averaged

Parameters
first (*boolean*) – True if this is the first time it is called for a given iterative sequence

CHAPTER

ELEVEN

LOOPING UTILITIES

pyqcm.loop.**Hartree**(*F*, *couplings*, *maxiter=10*, *eps_algo=0*, *file='hartree.tsv'*, *SEF=False*)

Performs the Hartree approximation

Parameters

- **F** – task to perform wihtin the loop
- **couplings** (*[class hartree]*) – list of Hartree couplings (or single coupling)
- **maxiter** (*int*) – maximum number of Hartree iterations
- **eps_algo** (*int*) – number of elements in the epsilon algorithm convergence accelerator = 2*eps_algo + 1 (0 = no acceleration)

pyqcm.loop.**controlled_loop**(*func=None*, *varia=None*, *loop_param=None*, *loop_range=None*, *control_func=None*, *adjust=False*, *predict=True*)

Performs a controlled loop for VCA or CDMFT with a predictor

Parameters

- **func** – a function called at each step of the loop
- **varia** (*[str]*) – names of the variational parameters
- **loop_param** (*str*) – name of the parameter looped over
- **loop_range** ((*float, float, float*)) – range of the loop (start, end, step)
- **control_func** – (optional) name of the function that controls the loop (returns boolean)
- **adjust** (*boolean*) – if True, adjusts the steps depending on control or status of convergence
- **predict** (*boolean*) – if True, uses a linear or quadratic predictor

pyqcm.loop.**fade**(*F*, *p1*, *p2*, *n*)

fades the model between two sets of parameters, in n steps

Parameters

- **F** – task to perform wihtin the loop
- **p1** (*dict*) – first set of parameters
- **p2** (*dict*) – second set of parameters
- **n** – number of steps

pyqcm.loop.**fixed_density_loop**(*mu*, *target_n*, *kappa=1.0*, *maxdmu=0.2*, *func=None*, *loop_param=None*, *loop_values=None*, *var_param=None*, *dens_tol=0.001*, *dir=''*, *measure=None*, *cluster_density=False*)

Performs a loop while trying to keep a fixed density

Parameters

- **mu** (*float*) – initial value of mu
- **target_n** (*float*) – desired value of the density
- **kappa** – initial guess of the compressibility
- **maxdmu** – maximum change in mu at each step (absolute value)
- **func** – function called at each step of the loop. No required argument. returns None.
- **loop_param** (*str*) – name of the parameter looped over
- **loop_values** (*float*) – an array of values of loop_param
- **var_param** (*str*) – array of variational parameters (names) to be predicted
- **dens_tol** – tolerance on the value of the density
- **dir** – directory of calling script
- **measure** – name of function to be called when the desired density is reached
- **cluster_density** – if True, uses the cluster density instead of the lattice density

pyqcm.loop.**linear_loop**(*N*, *func=None*, *varia=None*, *params=None*, *predict=True*)

Performs a loop for VCA or CDMFT with a predictor

Parameters

- **N** – number of intervals within the loop
- **func** – function called at each step of the loop
- **varia** (*[str]*) – names of the variational parameters
- **P** (*{str:(float,float)}*) – dict of parameters to vary with initial and final values
- **predict** (*boolean*) – if True, uses a linear or quadratic predictor

pyqcm.loop.**loop_from_file**(*func*, *file*)

Performs a task over a set of instances defined in a file

Parameters

- **func** – function (task) to perform
- **file** (*str*) – name of the data file specifying the solutions, in the same tab-separated format usually used to write solutions

CHAPTER

TWELVE

OTHER UTILITIES

12.1 Profile of expectation values

pyqcm.profile.**plot_profile**(*n_scale=1*, *bond_scale=1*, *current_scale=1*, *spin_scale=1*, *spin_angle=0*, *bond_spin_scale=1*, *singlet_scale=1*, *triplet_scale=1*, *file=None*, *layer=0*)

Produces a figure of various local quantities on the repeated unit, from averages computed in the ground state wavefunction

Parameters

- **n_scale** (*float*) – scale factor applied to the density
- **bond_scale** (*float*) – scale factor applied to the bond charge density
- **current_scale** (*float*) – scale factor applied to the currents on the bonds
- **spin_scale** (*float*) – scale factor applied to the spins on the sites
- **spin_angle** (*float*) – angle at which to draw the spins, from their nominal direction
- **bond_spin_scale** (*float*) – scale factor applied to the spins on the bonds
- **singlet_scale** (*float*) – scale factor applied to the singlet pairing amplitudes
- **triplet_scale** (*float*) – scale factor applied to the triplet pairing amplitudes
- **file** (*str*) – name of the output file, if not None
- **layer** (*float*) – layer number (z coordinate)

12.2 Defining models for slabs

class pyqcm.slab.**slab**(*name*, *nlayer*, *cluster*, *sites*, *superlattice*, *lattice*, *thickness=None*)

Helper class to define multi-layer systems, typically layers of planes. The fundamental unit is a 2D model that is repeated in the z direction

__init__(*name*, *nlayer*, *cluster*, *sites*, *superlattice*, *lattice*, *thickness=None*)

Constructor

Parameters

- **name** (*str*) – name of the 2D model
- **nlayer** (*int*) – total number of layers in the slab
- **cluster** (*str*) – name of the cluster model assembled (limited to one cluster)

- **sites** (*[[int, int, int]]*) – list of sites in each layer
- **superlattice** (*[[int, int, int]]*) – superlattice vectors in 2D
- **lattice** (*[[int, int, int]]*) – lattice vectors in 2D
- **thickness** (*int*) – number of inequivalent layers (<= nlayer/2). If None, all layers are different

hopping_operator(*name*, *link*, *amplitude*, ***kwargs*)

Defines a hopping operator Same arguments as the pyqcm.hopping_operator() function

interaction_operator(*name*, ***kwargs*)

Defines an interaction operator Same arguments as the pyqcm.interaction_operator() function

CHAPTER

THIRTEEN

SUMMARY OF OPTIONS

The behavior of *qcm* can be controlled by many global parameters that can be set via a call to `qcm.set_global_parameter()`. There are four kinds of global parameter, depending on the type of variable involved:

1. Boolean parameter. They are all false by default. Calling `qcm.set_global_parameter(name)` with the name of the parameter sets that parameter to True.
2. integer-valued parameter. Calling `qcm.set_global_parameter(name, val)` with the name of the parameter sets that parameter's value to `val`.
3. real-valued parameter. Same calling prototype as above.
4. char-valued parameter. Same calling prototype as above.

The following tables list the different global parameters, their default value and a short description.

13.1 Boolean options

name	default	description
CSR_sym_store	false	stores CSR matrices fully for openMP application
check_lanczos_residual	false	checks the Lanczos residual at the end of the eigenvector computation
continued_fraction	false	Uses the continued fraction solver for the Green function instead of the band Lanczos method
dual_basis	false	uses the dual basis for wavevector computations
modified_Lanczos	false	Uses the modified Lanczos method for the ground state instead of the usual Lanczos method
no_degenerate_BL	false	forbids band lanczos to proceed when the eigenstates have degenerate energies
nosym	false	does not take cluster symmetries into account
one_body_solution	false	Only solves the one-body part of the problem, for the Green function
periodic	false	considers the cluster(s) as periodic
periodized_averages	false	computes lattice averages using the periodized Green function
print_Hamiltonian	false	Prints the Hamiltonian on the screen, if small enough
print_all	false	prints dependent parameters as well
strip_anomalous_self	false	sets to zero the anomalous part of the self-energy
verb_ED	false	prints ED information and progress
verb_Hilbert	false	prints progress information on bases and operators in the Hilbert space
verb_integrals	false	prints information and progress about integrals
verb_warning	false	prints warnings
zero_dim	false	sets the spatial dimension to zero, on any model

13.2 Integer-valued options

name	default	description
Davidson_states	1	Number of states requested in the Davidson-Liu algorithm
GK_min_regions	8	minimum number of regions in the Gauss-Kronrod method
cuba2D_mineval	1024	minimum number of integrand evaluations in CUBA (2D)
cuba3D_mineval	16000	minimum number of integrand evaluations in CUBA (3D)
dim_max_print	64	Maximum dimension for printing vectors and matrices
kgrid_side	32	number of wavevectors on the side in a fixed wavevector grid
max_dim_full	256	Maximum dimension for using full diagonalization
max_iter_BL	600	Maximum number of iterations in the band Lanczos procedure
max_iter_CF	400	Maximum number of iterations in the continuous fraction Lanczos procedure
max_iter_QN	60	maximum number of iterations in the quasi-Newton method
max_iter_lanczos	600	Maximum number of iterations in the Lanczos procedure
print_precision	8	precision of printed output
seed	0	seed of the random number generator

13.3 Real-valued options

name	default	description
Qmatrix_vtol	1e-10	minimum value of a Qmatrix contribution
Qmatrix_wtol	1e-05	maximum difference in frequencies in Q-matrix
accur_Davidson	1e-05	maximum norm of residuals in the Davidson-Liu algorithm
accur_OP	0.0001	accuracy of lattice averages
accur_Q_matrix	1e-05	tolerance in the normalization of the Q matrix
accur_SEF	5e-08	Accuracy of the Potthoff functional
accur_band_lanczos	1e-12	energy difference tolerance for stopping the BL process
accur_continued_fraction	0.01	value of beta below which the simple Lanczod process stops
accur_deflation	1e-07	norm below which a vector is deflated in the Band Lanczos method
accur_lanczos	1e-12	tolerance of the Ritz residual estimate in the Lanczos method
band_lanczos_minimum_gap	1e-05	gap between the lowest two states in BL below which the method fails
cutoff_scale	1e+12	high-frequency cutoff in integrals
eta	0.0001	value of the imaginary part of the frequency in Chern number/Berry phase computations
large_scale	20	high-frequency region for imaginary frequency axis integrals
minimum_weight	0.01	minimum weight in the density matrix
small_scale	0.5	low-frequency region for imaginary frequency axis integrals
temperature	0	Temperature of the system.

13.4 Char-valued options

name	default	description
Hamiltonian_format	S	Desired Hamiltonian format: S (CSR matrix), O (individual operators), F (factorized), N (none = on the fly)
periodization	G	periodization scheme: G, S, M, C or N (None)

CHAPTER

FOURTEEN

PARALLEL PROCESSING

14.1 Generality

pyqcm could benefit from various methods to perform parallel processing in shared memory system. Three parallelisations method could be used in theory:

- Those implemented in BLAS/LAPACK,
- Those implemented in the CUBA numerical integration library
- And those implemented in QCM library (with OpenMP).

By default, pyqcm (and QCM) disable parallelization using BLAS/LAPACK and CUBA, to use its own implementation using OpenMP with all the cores available on the machine. Users can still specify fewer number of core to use by exporting the environment variable OMP_NUM_THREADS, e.g.:

```
OMP_NUM_THREADs=X #use X cores
```

User can also choose to rely on the CUBA library parallelization by setting the environment variable CUBACORES. QCM OpenMP parallelization must also be unset:

```
CUBACORES=X #use X cubacores, X must be >=2
OMP_NUM_THREADs=1 #unset OpenMP
```

However, using this method, only the integration section would be treated in parallel.

14.2 Performance

The following Figure 1 compares parallelization performance on a simulation which spend most of its time (i.e. more than 99% using a single core) in the numerical integration. Tests were performed on a 2 x Intel Xeon Gold 6238 Cascade Lake @ 2.10GHz with 22 cores each and HyperThreading disabled.

User should note that while OpenMP used the same object in memory to perform calculation in parallel, CUBA rather duplicate the object in the memory, resulting in a significant memory consumption.

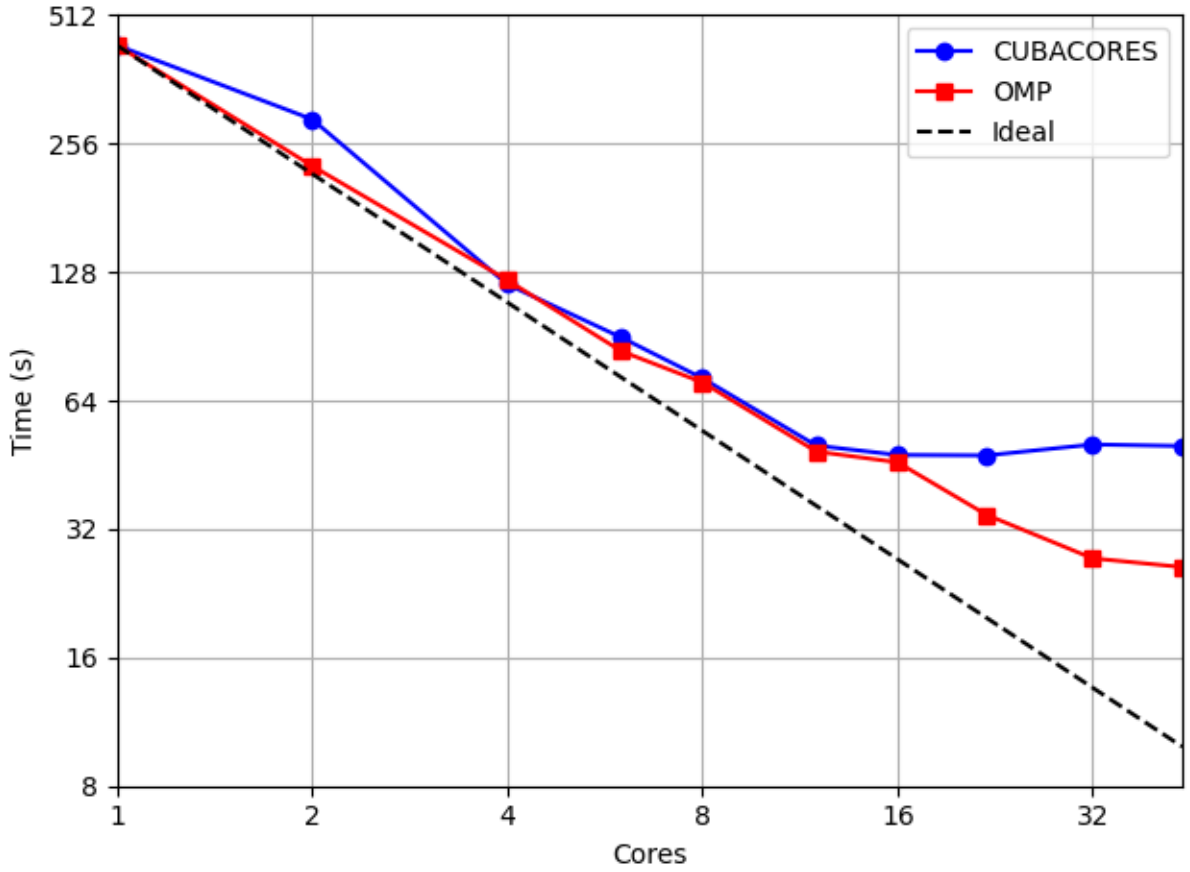

Fig. 1: Figure 1: Parallel performance using QCM OpenMP (in red) and Cuba parallelization (in blue).

CHAPTER

FIFTEEN

INDICES AND TABLES

- genindex
- modindex
- search

PYTHON MODULE INDEX

p

INDEX

Symbols

A

B

C

D

E

F

G

H

I

L

M

N

O

P

Q

R

S

T

U

V

W