Python lib qmodel

GitHub: https://github.com/magmage/qmodel [https://github.com/magmage/qmodel]

Basic information

A lightweight Python library for discrete quantum models centered on Hilbert space bases and operators. The main classes are `Basis`, `Operator` and `Vector` correspondingly, with derived classes for specialized bases and a class for vectors of operators. Here, “basis” should always be read as “Hilbert space with a fixed basis”. An operator is then always defined with respect to such a fixed basis and carries a reference to the `Basis` object it belongs to. The basis elements are indexed by quantum numbers, “qn” for short, like (n, ℓ, m) for atomic orbitals or (i, σ) for a Hubbard lattice. Those are realized as dictionaries. A basis can be tensorized to represent multiple particles, the corresponding basis elements are then realized as tuples of the constituent basis elements. Next to the full tensor product (for distinguishable particles, qubits) also symmetric and anti-symmetric tensor products are supported.

First import the necessary classes.

```
from qmodel import *
```

Getting started

A `Basis` object is initialized with a qn key (string) and a range of quantum numbers (float-valued iterable). The example initializes a 3-dimensional space with a basis that is numbered with a qn called “i” and that has the values 1,2,3. Some information about the basis is then printed.

```
b = Basis('i', (1,2,3))
print(b)
```

One of the most basic operators in this basis is the mapping $e_i \mapsto ie_i$ when e_i are the respective basis vectors. It thus always has a diagonal matrix in this basis. The corresponding method is `diag()` which yields an `Operator` object. Printing an `Operator` gives the matrix which just has $(1, 2, 3)$ in the diagonal in this case.

```
A = b.diag()
print(A)
```

Operators can be arithmetically manipulated which always creates new instances.

```
B = 3*A - A/2 + 1
print(A) # stays the same
print(B)
```

Tensor products

Tensor products between different spaces are created by calling `b1.tensor(b2)`. The two `Basis` objects have to have disjoint qn keys. Consequently, `b2.tensor(b1)` gives a different order in the basis but is effectively the same, since anyway the quantum numbers keep track of the basis elements. Let's create the space for a single particle on a 3-site lattice labelled by qn “i” with 2 internal degrees of freedom valued $(-1, 0, +1)$ and labelled by qn “f”. If the basis is printed, all basis elements are listed with their respective qns collected in a dictionary.

```
bi = Basis("i", (1,2,3))
bf = Basis("f", (-1,0,1))
b1 = bi.tensor(bf)
b2 = bf.tensor(bi)
print(b1)
print(b2)
```

The operator $\hat{A}$ from before must then be extended to act on the full tensor space. This corresponds to the operator $\hat{A} \otimes \hat{I}$ on `b1` and $\hat{I} \otimes \hat{A}$ on `b2`. The method `extend(bx)` with the respective extended basis `bx` then returns a new extension of an operator acting on the larger basis.

```
A = bi.diag()
print(A.extend(b1))
print(A.extend(b2))
```

We can also directly tensorize operators. For this define $\hat{A}$ on `bi` and $\hat{B}$ on `bf` and get $\hat{A} \otimes \hat{B}$ with `A.tensor(B)`. The corresponding tensor basis is then automatically formed but can also be provided as an optional argument.

```
A = bi.diag()
B = bf.diag()
T = A.tensor(B)
print(T)
print(T.basis)
```

Note that this kind of tensor products of bases are called “one-particle basis” even though they might represent multiple, different and distinguishable particles, since they can equally well be understood as a single particle with multiple degrees-of-freedom (qns).

Distinguishable many-particle (qubit) systems

There are three different possibilities to form an N-fold tensor product of a given space: A full tensor product of all basis elements, a symmetric, or an antisymmetric tensor product. For the state space of distinguishable particles of the same kind the full tensor product is needed. If every such particle has just one internal degree of freedom with values ± 1 (e.g. the spin in one fixed direction, always for Spin-$\frac{1}{2}$ particles and in the z-direction) then we talk about a qubit system. We first define the state space for a single system for which the basis class `SpinBasis` is provided and then form the N-fold tensor product with `ntensor(N)`.

```
N = 2
b_single = SpinBasis()
b_many = b_single.ntensor(N)
print(b_many)
```

When printing this basis, we notice that the basis elements look differently now. They are given as a tuple where the first element always signifies the type of tensor product, where `x` stands for the full tensor product, `s` for symmetric, and `a` for antisymmetric. This is followed by a dictionary for the qn of each particle. If the N-particle basis is tensorized again, say 2-fold, this just yields a $2N$ particle space.

```
b_many_many = b_many.ntensor(2)
print(b_many_many)
```

The method `sigma_z()` applied to the single-particle `SpinBasis` gives the $\hat{\sigma}_z$ Pauli operator that measures the spin in z-directions with eigenvalues ± 1. We can extend this operator to the N-particle basis to form $\hat{\sigma}_z \otimes \hat{I} \otimes \hat{I} \otimes \ldots + \hat{I} \otimes \hat{\sigma}_z \otimes \hat{I} \otimes \ldots + \ldots$ which measures the total z-spin of all particles. The same operator is given by calling `diag('s')` on the many-particle basis, which sums all values of the given qn for each basis element and returns a diagonal operator.

```
Sz = b_single.sigma_z().extend(b_many)
print(Sz)
print(b_many.diag('s'))
```

Note that in this way it is impossible to address the individual qubits because they share the same qn label. To keep them addressable separately we have to create one `SpinBasis` for each particle and give them different labels, say `s1` and `s2`, then form the tensor product. This way we can get `sigma_z()` for just one particle (e.g. the first) and subsequently extend this operator to the 2-particle basis. The same result is achieved by calling `diag('s1')` on the 2-particle basis.

```
b_spin1 = SpinBasis('s1')
b_spin2 = SpinBasis('s2')
b_many = b_spin1.tensor(b_spin2)
```

```
print(b_many)
print(b_spin1.sigma_z().extend(b_many))
print(b_many.diag('s1'))
```

Fermionic many-particle systems

If the particles are indistinguishable and described by an antisymmetric wavefunction (fermions) then the many-particle space must be formed with `ntensor(N, 'a')` or, as an alias, `wedge(N)`. Again, our particles have just an internal spin degree-of-freedom and we construct the N-particle basis and an operator to measure total spin.

```
N = 2
b_single = SpinBasis()
b_many = b_single.wedge(N)
print(b_many)
print(b_many.diag('s'))
```

The result for $N = 2$ is just a one-dimensional Hilbert space, since the only possible state is $\chi_+ \otimes \chi_- - \chi_- \otimes \chi_+$. For $N = 3$ the state space is even empty, since we just have two different internal states for each particle. We thus add a further degree-of-freedom to the individual particles, say a location labelled by `x` that has valued in $\{1, 2, 3\}$. Now, the antisymmetric wedge product results in an 15-dimensional 2-particle space.

```
b_single = Basis('x', (1,2,3)).tensor(SpinBasis())
print(b_single)
b_many = b_single.wedge(2)
print(b_many)
```

Notice, that like this we already created a spin-lattice system, yet without using the `LatticeBasis` class discussed below.

Bosonic many-particle systems

If a many-particle system is described by a symmetric wavefunction (bosons), the many-particle basis must be created with `ntensor(N, 's')`. With the `SpinBasis` from before (although a boson should not have Spin-$\frac{1}{2}$), a 2-particle system then has three basis states, $\chi_+ \otimes \chi_+$, $\chi_- \otimes \chi_-$, and $\chi_+ \otimes \chi_- + \chi_- \otimes \chi_+$. We also print the operator that measures the total spin of the system.

```
N = 2
b_single = SpinBasis()
b_many = b_single.ntensor(N, 's')
print(b_many)
print(b_many.diag('s'))
```

We can also combine different symmetries in a single, larger space. Say we want to describe the interaction between two fermions with qn `f` that can take values $\{1, 2, 3\}$ and two bosons with qn `b` and values $\{1, 2\}$. Then this is how to construct the space for the whole system.

```
b_fermion = Basis('f', (1,2,3)).wedge(2)
b_boson = Basis('b', (1,2)).ntensor(2, 's')
print(b_fermion.tensor(b_boson))
```

We see that the basis elements now get nested, with the outer combination of fermions and bosons marked with `x` (full tensor product), and then the basis elements for the antisymmetric and symmetric component systems.

Since usually the boson number is not fixed within a system, we should instead use a different description that models a Fock space. The qn `b` with values $\{1, 2\}$ is replaced by two modes, i.e. bosons of different kinds, and each mode has the number of particles $\{0, 1, 2, \ldots\}$ as qn. This is implemented as `NumberBasis` and they are constructed by (optionally) setting the maximal number (cutoff, the maximal number is `max_num-1`) and the qn label. The bases for two different modes (with different labels) can then be combined into one space with the usual tensor product.

```
b1_boson = NumberBasis(max_num=5, qn_key='n1')
print(b1_boson)
b2_boson = NumberBasis(max_num=5, qn_key='n2')
```

```
b_two_modes = b1_boson.tensor(b2_boson)
print(b_two_modes)
```

The `NumberBasis` also implements creation and annihilation operators $\hat{a}^\dagger$, $\hat{a}$ and the operator for the displacement coordinate and its derivative, $\hat{x} = \frac{1}{\sqrt{2}}(\hat{a}^\dagger + \hat{a})$, $\hat{\partial}_x = \frac{1}{\sqrt{2}}(-\hat{a}^\dagger + \hat{a})$. Finally, we test if $\hat{a}^\dagger\hat{a}$ gives the number operator, that we also get from `diag()`, as expected.

```
print(b1_boson.creator())
print(b1_boson.annihilator())
print(b1_boson.x_operator())
print(b1_boson.dx_operator())
print(b1_boson.creator()*b1_boson.annihilator())
print(b1_boson.diag())
```

Spin-lattice example

We create a basis for an M-site lattice and a separate one for the spin degree-of-freedom $s \in \pm 1$. Although this is easy to achieve with the discussed `Basis` class, we can use the specialized `LatticeBasis` and `SpinBasis` for this that offer additional functionality.

```
M = 3
b_lattice = LatticeBasis(M)
b_spin = SpinBasis()
b_onepart = b_lattice.tensor(b_spin) # one-particle space
print(b_onepart)
```

We then put $N = 2$ fermionic particles on this lattice which means the space is spanned by all 2-fold antisymmetric combinations (combinatorial product without doubles) of the 1-particle space basis vectors. The method `wedge(n)` (antisymmetric tensor product, wedge product) provides this functionality and gives a space of dimension $\binom{2M}{N} = 15$.

```
N = 2
b = b_onepart.wedge(N)
print(b)
```

We now want to create an operator for the Hubbard Hamiltonian

$$\hat{H} = -t \sum_{i,s} \left(\hat{a}^\dagger_{i+1,s} \hat{a}_{i,s} + \text{h.c.} \right) + U \sum_i \hat{n}_{i\uparrow} \hat{n}_{i\downarrow}, \tag{1}$$

where we assume a periodic chain, i.e., $i = M + 1$ corresponds to $i = 1$ again. Summation over all lattice sites is achieved with `b_lattice.sum()` that takes a callable as argument that itself has the qn as argument. For the sum over all lattice sites and spin states, `b_onepart.sum()` can be used instead.

```
b_lattice.sum(lambda qn: print(qn))
b_onepart.sum(lambda qn: print(qn))
```

The library allows us to combine the parts of the operator just like in mathematical notation, it is just important to create the operators on the full many-particle basis `b`. The operator $\hat{a}^\dagger_{i1,s1}\hat{a}_{i2,s2}$ is given by the method `hop(qn1, qn2)` (fermion one-body operator) where `qnx = {'i': ix, 's': sx}`. For $\hat{n}_{i,s} = \hat{a}^\dagger_{i,s}\hat{a}_{i,s}$ a single argument is enough, `hop(qn)` (fermion number operator). We create a little helper function that always gets the qn for the next site $i + 1$ by changing just the index "i" in a dictionary. The method `add_adj()` adds the adjoint (hermitian conjugate) to an operator and returns the result.

```
t = 1
U = 1
next_site = lambda qn: dict(qn, i=qn['i']%M+1)
H = -t * b_onepart.sum(lambda qn: b.hop(next_site(qn), qn).add_adj()) \
    +U * b_lattice.sum(lambda qni: b.hop(dict(qni, s=1)) * b.hop(dict(qni, s=-1)))
print(H)
```

The correct Fermi statistics is automatically obeyed when the operators are created on an antisymmetric tensor space.

Next, the method `H.eig()` gives the full spectrum and eigensystem of the Hamiltonian. Yet, here it is important to add the information that the operator is self-adjoint (hermitian) with `H.eig(hermitian = True)`, so that only real eigenvalues can be expected and spectrum and eigensystem get sorted (lowest first). The information is stored in a dictionary from which we retrieve the ground-state energy and the ground-state vector as well as degeneracy information. The special method `trace(b_lattice)` can then be used to sum the modulus square of the vector coefficients over all quantum numbers *except* the ones of basis `b_lattice` (which can also be a single one). Consequently, `trace(b_lattice)` gives the lattice density of the ground state when applied to the ground-state vector.

```
E = H.eig(hermitian = True)
gs_energy = E['eigenvalues'][0]
gs_vector = E['eigenvectors'][0]
gs_degeneracy = E['degeneracy'][0]
gs_density = gs_vector.trace(b_lattice)
print(gs_energy)
print(gs_density)
print(gs_degeneracy)
```

The ground state for $N = 2$ is non-degenerate, so for a Hamiltonian that is symmetric under lattice-site permutations the density is uniform. Next, add a non-uniform potential to get some variation. The potential is given as a list (NB: index starting at 0) and multiplied pointwise with the operator `b.hop(qni)` where now `qni = {'i': *}` is a dictionary just including the lattice index. This means the spin index gets automatically summed over all possibilities, thus `b.hop({'i': 1})` corresponds to $\hat{n}_{1\uparrow} + \hat{n}_{1\downarrow}$.

```
pot = [0, -1, 0]
H = H + b_lattice.sum(lambda qni: pot[qni['i']-1] * b.hop(qni))
E = H.eig(hermitian = True)
gs_energy = E['eigenvalues'][0]
gs_vector = E['eigenvectors'][0]
gs_density = gs_vector.trace(b_lattice)
print(gs_energy)
print(gs_density)
```

The ground-state energy gets lowered, as expected, and the probability of finding a particle at the favorable lattice position $i = 2$ is increased, while the other ones still have equal probability.

Operators can also be organized and manipulated as vectors. The `SpinBasis` offers a special method to create the Pauli-operator vector $\hat{S} = (\hat{\sigma}_x, \hat{\sigma}_y, \hat{\sigma}_z)$ where $\hat{\sigma}_*$ are the usual Pauli operators. We can use this to evaluate the spin expectation value of the ground state from the above example which correctly yields (approximately) zero in all three components, because no spin direction is favored by $\hat{H}$.

```
S = b_spin.sigma_vec().extend(b)
print(S.expval(gs_vector))
```

Dicke model example

In this Dicke model example, two qubits (distinguishable Spin-$\frac{1}{2}$ particles) are coupled to a single photon mode. The Hamiltonian is

$$\hat{H} = \hat{a}^\dagger \hat{a} + \frac{1}{2} - t(\hat{\sigma}_x^1 + \hat{\sigma}_x^2) + \lambda \hat{x} \otimes (\hat{\sigma}_z^1 + \hat{\sigma}_z^2). \tag{2}$$

In the coupling term the displacement coordinate of the photon mode, measured by $\hat{x}$, couples to the total spin in z-direction. The coupling strength is controlled by λ and we give a plot of the 10 lowest eigenvalues of $\hat{H}$ for different values of λ.

```
import numpy as np
import matplotlib.pyplot as plt

max_photon_num = 10
```

```
b_mode = NumberBasis(max_num=max_photon_num)
# define qubits separately so operators can act separately on them
b1_spin = SpinBasis('s1')
b2_spin = SpinBasis('s2')
b = b_mode.tensor(b1_spin).tensor(b2_spin)

num_op = b_mode.diag().extend(b) # number operator for photons
x_op = b_mode.x_operator().extend(b)
sigma_z1 = b1_spin.sigma_z().extend(b)
sigma_z2 = b2_spin.sigma_z().extend(b)
sigma_x1 = b1_spin.sigma_x().extend(b)
sigma_x2 = b2_spin.sigma_x().extend(b)

t = 1
H0 = num_op + 1/2 - t*(sigma_x1 + sigma_x2)
coupling_op = x_op*(sigma_z1 + sigma_z2)

spec = []
lam_span = np.linspace(0,5,50)
for lam in lam_span:
    H = H0 + lam*coupling_op
    res = H.eig(hermitian = True)
    spec.append(res['eigenvalues'][0:10])
plt.plot(lam_span, spec, 'b')
plt.show()
```

API reference

Basis class

Properties

`Basis.dim` dimension, number of basis elements
`Basis.qn_keys` the labels of the used qns
`Basis.part_num` number of particles when tuples are used (only with `symmetry`)
`Basis.part_basis` reference to the particle basis, or `self`
`Basis.symmetry` symmetry marker (`None` | `'x'` | `'s'` | `'a'`)
`Basis.el` list of all elements (as dicts or tuples)

Methods

`Basis([qn_key: str], [qn_range: Iterable]) → Basis`

The constructor returns a basis object with a single qn. If `qn_key` or `qn_range` is not provided, an empty basis is returned. The `qn_range` must be an `Iterable` of type `int` or `float` that lists the possible values for the qn.

`Basis.tensor(basis: Basis) → Basis`

Return the tensor product of the basis object with another basis. This method cannot be used for a tensor product with the same basis or any basis with the same qn labels. For a tensor product with the same basis, `Basis.ntensor()` must be used.

`Basis.ntensor(n: int, [symmetry]) → Basis`

Return the n-fold tensor product of the basis object as a new basis. The optional argument `symmetry` allows to define a special symmetry for the new basis:

- `symmetry = 'x'` (default): no special symmetry (distinguishable particles, qubits)
- `symmetry = 's'` : symmetric space (bosons)
- `symmetry = 'a'` : anti-symmetric space (fermions)

`Basis.wedge(n: int) → Basis`

Alias for `Basis.ntensor(n, symmetry = 'a')`.

`Basis.sum(func: Callable) → float`

Iterates all basis elements, calls `func` (a `Callable` that returns a summable value) on each basis element (dictionary valued) and returns the sum of the results.

`Basis.id() → Operator`

Returns the identity operator in this basis.

`Basis.diag([qn_key: str]) → Operator`

Returns a diagonal operator $x \mapsto x$ for a given qn.

`qn_key` is a qn of the basis. If the basis has only a single qn then this argument can be omitted.

`Basis.hop(qn1: dict, [qn2: dict]) → Operator`

The method creates an one-body (hopping) operator with matrix M_{ij} on the given basis $\{b_i\}$ after rules below.

If `qn2` is not given then `qn2 = qn1`. `qn1` and `qn1` must contain the same qn keys.

- *One-particle basis:* If `qn1` is found in the qn of b_i and `qn2` is found in the qn of b_j and the *other* qns all agree between b_i and b_j then $M_{ij} = 1$, else 0.
- *Symmetric many-particle basis:* `qn1` can be found in multiple particle sectors of b_i and likewise `qn2` in b_j, all combinations of those are iterated, the one-particle basis elements are compared like in the one-particle basis above, and the results added.
- *Antisymmetric many-particle basis:* Same as in the symmetric case, but the sign is set $(-1)^{p_1+p_2}$, where p_1 and p_2 are the index of the respective particle sectors where `qn1` and `qn2` are found.
- *Non-symmetric many-particle basis:* Same as in the symmetric case, but additionally those positions have to agree, $p_1 = p_2$.

This means the qns can, but most not, fully qualify a one-particle basis element. If they do not then this method implies a summation over the remaining part of the qn. In the (anti-)symmetric case the hopping operator corresponds to $\hat{a}_i^\dagger \hat{a}_j$ if `qn1` represents b_i and `qn2` represents b_j.

LatticeBasis class

Derived from `Basis`.

`LatticeBasis(vertex_num: int, [qn_key: str])`

The constructor returns a lattice basis object with `vertex_num` vertices numbered with a qn labelled `qn_key` from 1 onward. If `qn_key` is omitted, the qn label is "`i`".

`LatticeBasis.graph_laplacian(graph_edges: set, [qn_key: str], [hopping: float], [include_degree: bool]) → Operator`

Return an operator for the graph Laplacian on this lattice basis. The graph is specified by a set of pairs (tuples) that define the edges on the graph. If the vertices are not labelled by "`i`", then the `qn_key` must be provided.

`hopping` default is 1. This value is inserted at the position of the matrix where an edge connects two vertices.

`include_degree` default is `True`. If not `False` then the diagonal includes the negative degree of the vertex, i.e., the number of other vertices connecting to it.

Example: `b.graph_laplacian({(1,2), (2,3), (1,3)})`

Note that in graph theory usually an opposite sign convention is employed. We used this definition such that the negative graph Laplacian fulfills the same semi-positivity condition as the continuum Laplacian.

`LatticeBasis.graph_pot(pot: dict|list|ndarray, [qn_key: str]) → Operator`

Return a diagonal operator for a potential on a lattice (graph).

`pot` can be of type dict, list, or NumPy type `ndarray`. A dict must be of form `{1: p1, 2: p2, …}` where `pi` is the potential acting on vertex `i`. Alternatively, it can be a list/one-dimensional array `[p1, p2, …]`.

`qn_key` default is "`i`" and contains the corresponding qn label for the vertices.

VacuumBasis class

Derived from `Basis`.

`VacuumBasis([qn_key: str])`

The constructor returns a basis for the vaccum which is just the one-dimension Hilbert space $\mathbb{C}$. If `qn_key` is omitted, the qn label is "`vac`".

NumberBasis class

Derived from `Basis`.

`NumberBasis(max_num: int, [qn_key: str])`

The constructor returns a basis for the single-mode Fock space with number basis $|0\rangle, |1\rangle, |2\rangle, \ldots$ up to `max_num-1`. If `qn_key` is omitted, the qn label is "`n`".

`NumberBasis.creator() → Operator`

Return the creation operator $\hat{a}^\dagger$ for the number basis.

`NumberBasis.annihilator() → Operator`

Return the annihilation operator $\hat{a}$ for the number basis.

`NumberBasis.x_operator() → Operator`

Return the operator for the displacement coordinate, $\hat{x} = \frac{1}{\sqrt{2}}(\hat{a}^\dagger + \hat{a})$.

`NumberBasis.dx_operator() → Operator`

Return the operator for the derivative w.r.t. the displacement coordinate, $\hat{\partial}_x = \frac{1}{\sqrt{2}}(-\hat{a}^\dagger + \hat{a})$.

SpinBasis class

Derived from `Basis`.

`SpinBasis([qn_key: str])`

The constructor returns a basis for a Spin-$\frac{1}{2}$ particle with qn values $\{+1, -1\}$ (in that order; spin measured in z-direction). If `qn_key` is omitted, the qn label is "`s`".

SpinBasis.sigma_x() → Operator

Return the $\hat{\sigma}_x = \begin{pmatrix} 0 & 1 \\ 1 & 0 \end{pmatrix}$ Pauli operator.

SpinBasis.sigma_y() → Operator

Return the $\hat{\sigma}_y = \begin{pmatrix} 0 & -\mathrm{i} \\ \mathrm{i} & 0 \end{pmatrix}$ Pauli operator.

SpinBasis.sigma_z() → Operator

Return the $\hat{\sigma}_z = \begin{pmatrix} 1 & 0 \\ 0 & -1 \end{pmatrix}$ Pauli operator.

SpinBasis.sigma_vec() → OperatorList

Return an `OperatorList` object containing the vector $(\hat{\sigma}_x, \hat{\sigma}_y, \hat{\sigma}_z)$.

SpinBasis.sigma_plus() → Operator

Return the operator $\hat{\sigma}_+ = \begin{pmatrix} 0 & 1 \\ 0 & 0 \end{pmatrix}$.

SpinBasis.sigma_minus() → Operator

Return the operator $\hat{\sigma}_- = \begin{pmatrix} 0 & 0 \\ 1 & 0 \end{pmatrix}$.

SpinBasis.proj_plus() → Operator

Return the projection operator on the $+1$ eigenspace, $\hat{P}_+ = \begin{pmatrix} 1 & 0 \\ 0 & 0 \end{pmatrix}$.

SpinBasis.proj_minus() → Operator

Return the projection operator on the -1 eigenspace, $\hat{P}_- = \begin{pmatrix} 0 & 0 \\ 0 & 1 \end{pmatrix}$.

Vector class

Operators

The following operators are overloaded to work with `Vector` objects:
`*` with scalars from left/right
(See also overloading in the `Operator` class.)

Properties

`Vector.basis` reference to the basis of the vector
`Vector.col` NumPy `ndarray` that stores the numerical values of the vector

Methods

Vector(basis: Basis, [col: list|ndarray])

The constructor returns a vector object belonging to `basis` that can be optionally initialized with `col` of type `list` or

NumPy type `ndarray` (one-dimensional).

`Vector.copy() → Vector`

Return a copy of the object.

`Vector.inner(X: Vector) → complex`

Return the inner product of the vector with another vector. Note that the two vectors need to belong to the same basis and that the entries from the object the method is called on get complex conjugated.

`Vector.prob() → ndarray`

Return the components $|x_i|^2$ (probabilities) of the vector as an array.

`Vector.norm() → float`

Return the 2-norm of the vector.

`Vector.trace(subbasis: Basis) → ndarray`

Trace out all qn except those contained in `subbasis`, which has to be a "primitive" basis without any special symmetry (elements are only dicts), and return a density w.r.t. `subbasis`.

Example: Let `lb` be a `LatticeBasis` and `X` a vector on a many-particle basis that includes particles on the lattice (possibly with additional degrees-of-freedom). Then `X.trace(lb)` gives the density on the lattice.

Operator class

Operators

The following operators are overloaded to work with `Operator` objects:
`-` for sign change
`+/-` addition and subtraction between operators and between operators and scalars (times identity) from left/right
`*` multiplication between operators (non-commutative)
`*` multiplication between operators and scalars from left/right
`*` multiplication between operator and vector (from left)
`*` multiplication between operators and lists of scalars (from left/right) $(a_i)_i$ give an `OperatorList` object $(a_i \hat{A})_i$
`/` division of an operator by a scalar
[double asterisk] integer power of an operator
(Note that operations only work between operators (and vectors) defined on the same basis, this means the *same instance* of an `Basis` object.)

Properties

`Operator.basis` reference to the basis of the operator
`Operator.matrix` NumPy `ndarray` that stores the numerical matrix values of the operator

Methods

`Operator(basis: Basis)`

The constructor returns an operator object belonging to `basis` with a zero matrix.

`Operator.copy() → Operator`

Return a copy of the object.

`Operator.conj() → Operator`

Return an operator that is the complex conjugate.

`Operator.T() → Operator`

Return an operator that is the transpose.

`Operator.adj() → Operator`

Return an operator that is the Hermitian adjoint (complex conjugate and transpose).

`Operator.add_adj() → Operator`

Return an operator where the Hermitian adjoint is added, $\hat{A} + \hat{A}^\dagger$.

`Operator.comm(A: Operator) → Operator`

Return the commutator of the operator with another operator (first comes the operator where this method is called upon).

`Operator.acomm(A: Operator) → Operator`

Return the anticommutator of the operator with another operator.

`Operator.expval(vec: Vector, [check_real: bool], [transform_real: bool]) → float|complex`

Return the expectation value of the operator w.r.t. the given vector.

`check_real`: If `True` returns an error if the result has an imaginary part. Default is `False`.

`transform_real`: If `True` the result just returns the real part. This can be useful to also discard a `+0*1j` or very small imaginary part. Default is `False`.

`Operator.norm() → float`

Return the 2-norm (largest singular value) of an operator.

`Operator.eig([hermitian: bool]) → dict`

Get the eigenvalues and the eigenvectors of the operator. Returns a dictionary with keys `'eigenvalues'` and `'eigenvectors'` where the first is a NumPy `ndarray` and the second a list of `Vector` objects.

If `hermitian` is set to `True` (default is `False`) then a self-adjoint operator is assumed and the eigenvalues are returned in ascending order. Also, the returned dictionary includes a further key `'degeneracy'` which is a NumPy `ndarray` with the dimension of the eigenspaces ordered after the eigenvalues (e.g. `[2,1,3]` means a 2-fold degenerate groundstate, a non-degenerate 1st excited state and a 3-fold degenerate 2nd excited state). Note that even if `hermitian` is set to `True` the operator is not checked for self-adjointness (see method `test_hermiticity()`).

`Operator.test_hermiticity()`

Raises an error if the operator is not self-adjoint (hermitian). No return value.

`Operator.diag() → dict`

The method first checks for self-adjointness, then returns the diagonal of the diagonalized operator $\hat{D}$ and the corresponding transformation matrix $\hat{U}$ as NumPy `ndarray` in a dictionary with keys `'diag'` and `'transform'`. The relation to the operator is $\hat{D} = \hat{U}\hat{A}\hat{U}^\dagger$.

`Operator.extend(ext_basis: Basis) → Operator`

Extend the operator to a larger basis that includes the previous qn. If $\hat{A}$ was defined on V and is extended to $V \otimes W$, the returned operator is $\hat{A} \otimes \hat{I}$.

`Operator.tensor(A: Operator|OperatorList, [tensor_basis: Basis]) → Operator|OperatorList`

Tensor product of the operator with another one. If the argument is an `OperatorList` then the tensor product is taken with every operator in the list and an `OperatorList` is returned.

`tensor_basis` is an optional argument to fix a basis for the returned operator. If it is not given, a new basis is constructed as a tensor product itself. It might be necessary to specify `tensor_basis` in many cases, since the new operator should be defined on a particular basis object.

OperatorList class

Operators

The following operators are overloaded to work with `OperatorList` objects:
`-` for sign change of all operators in the list (vector)
`+/-` vector addition and subtraction between operator lists and between operator lists and scalars (in all components) from left/right
`*` inner product between two operator lists (non-commutative) or between an operator list and a list of numbers or `ndarray`
`*` multiplication between operator list and scalars (in all components) from left/right
`/` division of the operator list by a scalar (in all components)
`len()` gives the number of operators in the list
`[i]` returns the *i*th operator in the list

Properties

`OperatorList.basis` reference to the basis of the operator list (all operators in the list must have the same basis)
`OperatorList.operators` the list of `Operator` objects

Methods

`OperatorList(basis: Basis, [operators: list])`

The constructor returns an operator list belonging to `basis` that can already be initialized with a given list of `Operator` objects.

`OperatorList.append(operators: Operator|list)`

Append an operator or all operators in a list to the operator list.

`OperatorList.copy() → OperatorList`

Return a copy of the object.

`OperatorList.conj() → OperatorList`

Return an operator list with each operator complex conjugated.

`OperatorList.T() → OperatorList`

Return an operator list with each operator transposed.

`OperatorList.adj() → OperatorList`

Return an operator list with each operator Hermitian adjoined (complex conjugated and transposed).

`OperatorList.expval(vec: Vector, [check_real: bool], [transform_real: bool]) → float|complex`

Return the expectation values of each operator in the list w.r.t. the given vector as a NumPy `ndarray`. Flags are like in `Operator.expval()`.

`OperatorList.norm() → float`

Return the Euclidean norm of the list of operators, $\sqrt{\sum_i \|\hat{A}_i\|^2}$.

`OperatorList.sum() → Operator`

Return the sum of all operators in the list, $\sum_i \hat{A}_i$.

`OperatorList.diag([trials: int]) → dict`

Simultaneous diagonalization of the operators in the list. Since this method uses randomization, a numer of trials can be specified. If the method is not successful in any trial, an error is raised. Returns the diagonals of the diagonalized operators as a list and the corresponding transformation matrix as NumPy `ndarray` in a dictionary with keys `'diag'` and `'transform'` . The relation to the operators is $\hat{D}_i = \hat{U}\hat{A}_i\hat{U}^\dagger$.

`OperatorList.extend(ext_basis: Basis) → OperatorList`

Extend each operator in the list to a larger basis that includes the previous qn.

`Operator.tensor(A: Operator, [tensor_basis: Basis]) → OperatorList`

Tensor product of each operator in the list with another operator that returns an `OperatorList`. The optional argument `tensor_basis` is like for `Operator.tensor()`.