

Short overview over the general operator module for kwant

(Dated: November 30, 2018)

These few pages are meant to help to understand the working flow of the general operator module. I know that lots of aspects are still superficial and not very precise and that many details are not correct in this manuscript, but I hope that it still helps to understand the bigger picture working flow. At the moment, we want to calculate objects of the form

$$\psi_a^* M_a O_{ab} M_b O_{bc} \dots O_{xy} M_y \phi_y, \quad (1)$$

where $[O_{ab}, O_{bc}, \dots, O_{xy}]$ and $[M_a, M_b, \dots, M_y]$ are the user-defined lists for the hopping operator and the onsite matrices, respectively.

I. KWANT.OPERATOR.LOCALOPERATOR

To begin with, let me summarize the important steps in the `kwant.operator.LocalOperator` such that it is easier to see the differences and extensions for the general operator.

`--init--()`:

- store class variables
- normalize where to a unified format

`--call--()`:

- create `result_array` (numpy array)
- call `_operate`

`_operate (at the example of Current, but similar in other cases):`

- preparation: create 1-2 (one for the hopping=hamiltonian and maybe one for the onsite if it is not unique) instances of `BlockSparseMatrix` (BSM), `H_ab.blocks` and `M_a.blocks`, which contains all the information needed for the calculation:
 - `BSM.get(w)`: returns the operator matrix which belongs to the element 'w' of the where-list
 - `BSM.block_shapes(w,0/1)`: returns number of orbitals of the corresponding site 'where[w,0/1]'
 - `BSM.block_offsets(w,0/1)`: returns the first index which belongs to the corresponding site 'where[w,0/1]' in the wave function
- loop over all 'where'-elements and calculate the expectation value. For the chosen 'where'-element 'w'...
- ... get the number of orbitals, `a_norbs` and `b_norbs`, via `H_ab.blocks.block_shapes(w,0/1)`
- ... get the wave function start_index, `a_s` and `b_s`, via `H_ab.blocks.block_offsets(w,0/1)`
- ... get the explicit hopping matrix `H_ab` via `H_ab.blocks.get(w)`
- ... get the explicit onsite matrix `M_a` via `M_a.blocks.get(w)`
- ... sum over all orbitals:

$$\sum_{i,j=1}^{a_{norbs}} \sum_{k=1}^{b_{norbs}} \psi^*(a_s + i) M_a(i, j) H_{ab}(j, k) \phi(b_s + k) \quad (2)$$

for the given wave functions ψ and ϕ . i, j and k are orbital indexes.

II. GENERAL OPERATOR

A. The calculation of the matrix element in `_operate`

There is one major differences for the general operator compared to the `kwant.operator`: there are more (arbitrary number of) onsite and hopping operators, which is why we need lists of most objects, see the following tabular.

<code>kwant.operator(.Current)._operate()</code>	<code>general.Operator._operate()</code>
· create 2 <code>BlockSparseMats</code>	· create list of <code>BlockSparseMats</code> (for each operator)
· get <code>a_norbs</code> , <code>b_norbs</code>	· get list <code>x_norbs</code> (for each hopping site)
· get the explicit matrix M_a	· get a list of onsite mats
· get the explicit matrix H_{ab}	· get a list of hopping mats
· get wave function start indexes <code>a_s</code> and <code>b_s</code>	· get wave function start indexes <code>bra_start</code> and <code>ket_start</code>

Since we do not know how many operators there are, the looping over all needed sites and filling the lists for one given site combinations is handled by recursive functions.

$$\sum_{o1_a, o2_a, o1_b, o2_b, \dots} \underbrace{\psi^*(bra_start + o1_a) M[0](o1_a, o2_a) \times O[0](o2_a, o1_b) M[1](o1_b, o2_b) \times \dots \times}_{\text{pre rec funcs}} \underbrace{\times O[N-1](o2_x, o1_y) M[N](o1_y, o2_y) \times \phi(ket_start + o2_y)}_{\substack{\text{rec func step} \\ \text{rec func end}}} \quad (3)$$

In the code so far, there are actually 2 recursive functions, one for looping over all sites and filling the lists for a given site combinations (`sitesum_refunc`) and one for doing the actual sum over the orbitals (`orbsums_refunc`). It would be possible to combine them in one function, but for me it seems to be more clear having separated the two steps.

B. The problem of the wherelist

The only conceptual problem is how to tell the different operator for which sites/hoppings they are nonzero (or where they are supposed to be calculated). In the current state of implementation, a list of list of hoppings is given:

$$\text{wherelist} = \left[\left[\begin{pmatrix} a_1 \\ b_1 \end{pmatrix}, \begin{pmatrix} \cdot \\ \cdot \end{pmatrix}, \dots \text{a-b-hops} \right], \left[\begin{pmatrix} \cdot \\ \cdot \end{pmatrix}, \dots \text{b-c-hops} \right], \dots \right]. \quad (4)$$

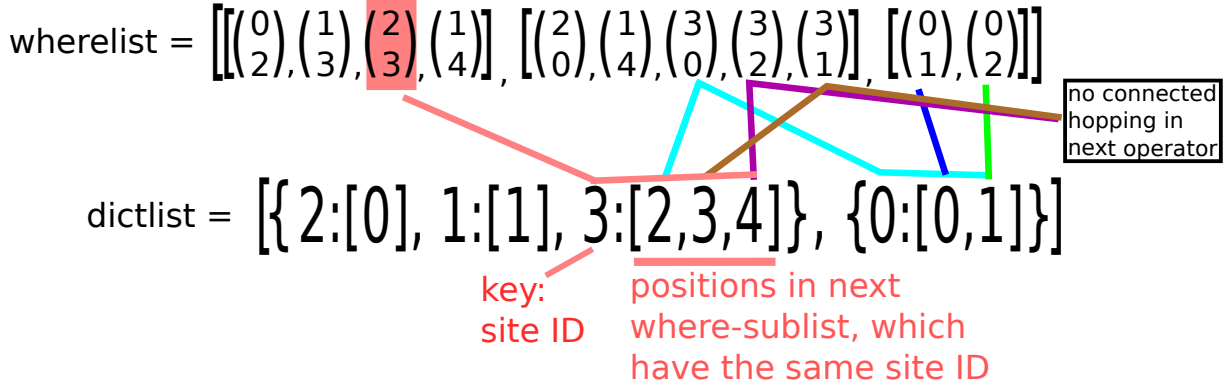
We only want to consider “connected” hoppings in the sense that for a given ‘a-b-hopping’, e.g. (a_{10}, b_4) , only ‘b-c-hoppings’ should be considered which have the same b-site, e.g. (b_4, c_7) or (b_4, c_{105}) . The same is true for all the other hoppings. Note that the wherelist above contains all the information needed, not only for the hopping operator but also for the onsite matrices.

To manage to calculate only the appropriate terms, we create in `genOp._init_()` a list of dicts which have the site-IDs as keys and returns a list of the positions of all hoppings in the next hopping-level whose first element is this particular site:

$$\text{dictlist} = \left[\{b_1 : [5, 3, 42, \dots \text{(positions in wherelist[1] whose b-site is } b_1)], b_2 : [\dots], \dots\}, \right. \\
 \left. \{c_1 : [7, 13, \dots \text{(positions in wherelist[2] whose c-site is } c_1)], c_2 : [\dots], \dots\}, \right. \\
 \left. \{\dots\}, \dots \right] \quad (5)$$

Since I know which Site is under consideration at the moment, I can find out all the related hoppings of the next operator (in the next where-sublist) with the help of `dict.list` and `wherelist` — that’s all I need. Here is an example of how the dictlist looks like for a given wherelist and how to find the related hoppings in the next where-sublist:

Ex: chosen first hopping



Note that for efficiency reasons, instead of the dicts, we use two auxiliary 1d-arrays, (wherepos_neigh and auxpos_list). Moreover, the wherelist is also flattened to 1d-arrays of hoppings together with one auxiliary list (auxwhere_list) which tells the starting positions of the prior sublists in the flattened where. At the moment I will not discuss the creation and the details of the auxiliary lists here, since these details are not necessary to understand the working mechanism of the general operator, however, they are very briefly introduced in Appendix A.

To finish, let me summarize the steps which happen in the general.operator.__init__():

- store class variables
- normalize where-sublists and flatten 'where' to 1d-array of hoppings
- create dictlist (and create two auxiliary lists to be used instead of dictlist for efficiency reasons)
- create list of all site combinations. This is used to be able to find out which output data belongs to which site-combination
- normalize hopping operators and onsite operators
- (initialize BlockSparseMat-lists)

general.operator.__call__():

- create result_array (numpy array)
- call _operate

As a last comment, I want to state that the result array is not directly obvious as opposed to the kwant.operators, in which the result array is directly related to 'where'. Since in the general operator case, all the site paths first have to be precomputed from the given wherelist, not even the result length is predetermined. For that reason, in genOp.__init__(), a list of all site-paths is created (self.sitechains) which has the same structure as the result array. This list directly tells which result in the output-array belongs to which site path.

Appendix A: Auxiliary lists

For efficiency reasons, we create 4 auxiliary lists, which contain the same information as the 'wherelist' and 'dictlist' discussed in Subsec. II B:

$$\text{wherelist} = \left[\left[\begin{pmatrix} a_1 \\ b_1 \end{pmatrix}, \begin{pmatrix} \cdot \\ \cdot \end{pmatrix}, \dots, \text{a-b-hops} \right], \left[\begin{pmatrix} \cdot \\ \cdot \end{pmatrix}, \dots, \text{b-c-hops} \right], \dots \right] \text{ becomes:}$$

- $\text{where_flat} = \left[\begin{pmatrix} a_1 \\ b_1 \end{pmatrix}, \begin{pmatrix} \cdot \\ \cdot \end{pmatrix}, \dots, \text{a-b-hops}, \begin{pmatrix} \cdot \\ \cdot \end{pmatrix}, \dots, \text{b-c-hops} \right]$
- $\text{auxwhere_list} = [\text{start index of 0th op in where_flat}, \text{start index of 1st op in where_flat}, \dots]$

and dictlist (remember that key: siteID of next where-sublist, value: list of positions of this site in the next where-sublist):

- $\text{where_neigh} = [\text{the values of 'dictlist' ordered according to where_flat}]$
- $\text{auxpos_list} = [\text{start index in where_neigh which belongs to where_flat}[0], \text{start index in where_neigh, which belongs to where_flat}[1], \dots]$

In words:

- 'where.flat' flattened list of hoppings in the sense that we have now a 1d list of hoppings (2-tupels), instead of a list of lists of hoppings
- 'auxwhere.list' has the same highest level structure as the original 'wherelist' and tells the absolute starting positions in 'where.flat'.
- 'where.neigh' contains all the relative positions of connected sites of the next operator. It is ordered as 'where.flat' in the sense that first, all the positions to connected sites relative to the hopping 'where.flat[0]' are given, then to 'where.flat[1]' and so on.
- 'auxpos.list' has the same structure as 'where.flat' and tells the absolute starting position in 'where.neigh' of the data which is related to the corresponding hopping in 'where.flat'