Short overview over the general operator module for kwant

(Dated: December 21, 2018)

Some parts in this manuscript go quite into detail. For understanding the work flow only, we recommend skipping the Subsec. II D and the final remarks in Sec. III.

In kwant, we want to use the operator classes to calculate objects of the form

$$\psi_a^\dagger O_{ab}^{(0)} O_{bc}^{(1)} O_{cd}^{(2)} \ldots O_{xy}^{(N)} \phi_y, \tag{1}$$

where the indexes a, b, ..., y denote sites and $O(0)$, ... $O^{(N)}$ are arbitrary operators, *i.e.* hopping ($i \neq j$) or onsite ($i = j$) operators. For instance, O can be the Hamiltonian or any other operator either given as a matrix or a function which returns a matrix when 1-2 sites are given.

So far, kwant can already calculate 3 different operators with the parent class kwant.operator._LocalOperator, which are:

- Density: $\psi_a^\dagger M_{aa} \phi_a$, with M being an arbitrary onsite operator
- Current: $\psi_a^\dagger M_{aa} H_{ab} \phi_b - c.c.$, with H being the Hamiltonian
- Source: $\psi_a^\dagger M_{aa} H_{aa} \phi_a - c.c.$

We decided to create the general operator module to be able to simply add new operators of the form of Eq. 1, because for the energy current in a time-dependent system, the following operators are also needed additionally:

- EnergyCurrent: $\psi_a^\dagger H_{ab} H_{bc} \phi_c - c.c.$,
- Arbitrary Hopping Operator: $\psi_a^\dagger M_{aa} O_{ab} \phi_b + c.c.$

Note that these new operators are only needed in the time-dependent case when the energy of incoming electrons in a scattering region is not conserved anymore. For the static case, the energy current can be easily calculated by having an additional E in the integral over the energy E in the Landauer-Büttiker formula.

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:

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

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 generalOperator.Operator._operate

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

kwant.operator(.Current)._operate()	general.Operator._operate()
· create 2 BlockSparseMats	· create list of BlockSparseMats (1 BSM for each operator)
· get a_norbs, b_norbs	· get list x_norbs (for each hopping site)
· get the explicit matrices M_a and H_{ab}	· get a list of all matrices
· get wave function start indexes a_s and b_s	· get wave function start indexes bra_start and ket_start

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.

$$\sum_{o_a, o_b, \ldots} \underbrace{\psi^*(bra\_start + o_a)}_{\text{pre rec funcs}} \underbrace{\times O_{ab}^{(0)}(o_a, o_b)}_{\text{rec func step}} \times \ldots \underbrace{\times O_{xy}^{(N)}(o_x, o_y) \times \phi(ket\_start + o_y)}_{\text{rec func end}}, \tag{3}$$

where o_i are denoting all degrees of freedom on a site, *e.g.* atomic orbitals, spin, etc.

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 (`sitePath_recfunc`) and one for doing the actual sum over the orbitals (`orbsums_recfunc`). 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. How to initialize the desired operator

According to Joseph's suggestions, having an arbitrary amount of different operators as in Eq. (1) is solved by first initializing each single operator and then initialize the product of those operators. In all cases, the sites/hoppings where an operator is to be calculated has to be directly given in the initialization of that particular operator.

1. *generalOperator.Operator*

The main class is the `generalOperator.Operator` class, which contains the routines to actually calculate the matrix elements of an operator as in Eq. (1). A single operator can be initialized by this class with the parameters:

`generalOperator.Operator.__init__(..)` – parameters:

- `syst`: the kwant system

- `opfunc`: the operator function, e.g. `syst.hamiltonian,`which takes two sites as arguments (and the parameters of the Hamiltonian(?)) and returns a matrix
- `in_where=None`: list of hoppings (tuples of sites) where to calculate this operator. Default value `None` uses all hoppings in the system.
- `withRevTerm=0`: its value should be ± 1 or 0. It tells whether the hermitian conjugate term is to be added, subtracted or ignored.
- `const_fac=1`: a constant factor with which the result is multiplied in the end
- `check_hermiticity=False`: If hermiticity is supposed to be checked or not
- `sum=False:`If the sum over all allowed hopping-combinations is to be returned or an array with the result of each combination

The initialization does effectively the same as for the `kwant.operator._LocalOperator`: store class variables and normalize `where`.

2. *generalOperator.Onsite*

An onsite operator can either be intialized as `generalOperator.Operator` with `where` being fake hoppings (*i.e.* both sites in the hopping are the same) or by using the `generalOperator.Onsite` class which inherits from `generalOperator.Operator`. The differences between these classes are

`generalOperator.Onsite` – (differences to `generalOperator.Operator`) :

- this operator is flagged as onsite operator, which simplifies some operations later
- `where` is now given by the user as a list of sites instead of being list of hoppings. (Note that internally, the list of sites is transformed into a list of fake hoppings to be able to have a unified structure for any operator.)
- `opfunc`: now takes only 1 site as argument (and the parameters of the Hamiltonian(?)), or is a matrix itself
- additional kw argument `willNotBeCalled=False`: if the onsite operator is initialized only to be used in a product but not to directly calculate its matrix elements (*i.e.* not being called), the path finding (connection of hoppings of different operators) can be simplified. However, this is only possible if there are no `where` restrictions in this onsite operator, *i.e.* `where==None`. See Subsec. II C for more details about the path finding and Subsec. II D for `willNotBeCalled`.

The `__init__`-method of its parent class is called.

3. *generalOperator.Op_Product*

This class inherits from the main class `generalOperator.Operator` to be able to calculate matrix elements, bind operators, ... However it is initialized differently to be able to calculate the matrix elements of a product of operators:

`generalOperator.Op_Prod.__init__(..)` – parameters:

- `*ops`: arbitrary number of operators which are to be multiplied
- `withRevTerm=0`: its value should be ± 1 or 0. It tells whether the hermitian conjugate term is to be added, subtracted or ignored.
- `const_fac=1`: a constant factor with which the result is multiplied in the end
- `check_hermiticity=False`: If hermiticity is supposed to be checked or not
- `sum=False:`If the sum over all allowed hopping-combinations is to be returned or an array with the result of each combination

- `willNotBeCalled=False`: only relevant if all operators in `ops` have been flagged as `willNotBeCalled`. It can be set to `True` in case that this product will be only used in another product, but not called directly for calculating its matrix elements, to enhance efficiency (mostly avoid path finding).

More about the `__init__`-method of `generalOperator.Op_Prod` can be found in subsection II D.

Example:
The Current operator, which is supposed to calculate $-i(\psi_a^\dagger M_{aa} H_{ab}\phi_b - c.c.)$, can be created with the help of the general Operator as

```
hamil = Operator(syst, syst.hamiltonian, in_where=??)
onsiteOp = Onsite(syst, onsite, where=None, willNotBeCalled=True)
current = Op_Product(onsiteOp, hamil, withRevTerm=-1, const_fac=-1j)
```

where we omit here some options like `sum` for simplicity.
Note that the 5 operator examples from above are already implemented in `generalOperator.pyx` and can be directly used by the user.

C. The 'Pathfinding' – how to tell the code which hopping combinations are to be considered

The only new conceptual problem compared to the existing `kwant.operator._LocalOpereator`'s is how to tell the product which hoppings in `where` of one operator are to be connected with which hoppings in `where` of the next operator. I will show here the procedure for a product of 2 operators, but it is straight forward to extend it to an arbitrary amount of operators.

Example:

- op1: `where` $= \left[\begin{pmatrix}41\\12\end{pmatrix}, \begin{pmatrix}14\\15\end{pmatrix}\right]$
- op2: `where` $= \left[\begin{pmatrix}15\\23\end{pmatrix}, \begin{pmatrix}12\\4\end{pmatrix}, \begin{pmatrix}15\\2\end{pmatrix}\right]$

Probably, I want to connect the 1st hopping of op1 with the 2nd hopping of op2 (since it has the same site ID '12') to calculate $\psi_{41}^\dagger O^{(1)}_{41,12} O^{(2)}_{12,4}\phi_4$, and to connect the second hopping of op1 with the 1st and 3rd hopping of op2 to calculate both, $\psi_{14}^\dagger O^{(1)}_{14,15} O^{(2)}_{15,23}\phi_{23}$ and $\psi_{14}^\dagger O^{(1)}_{14,15} O^{(2)}_{15,2}\phi_2$.

However, maybe for some reason, I only want to connect the hopping $\begin{pmatrix}14\\15\end{pmatrix}$ of op1 with $\begin{pmatrix}15\\2\end{pmatrix}$ of op2 but not with $\begin{pmatrix}15\\23\end{pmatrix}$. Therefore, we need to have a means to tell the product which hoppings of the 1st operator are to be connected with the hoppings of the 2nd operator, and so on. To that end, we use a new list (at the moment called `wherepos_neigh`), which has the same length as the `where` of the first operator. Each element is again a list of the *positions* of the related hoppings in the `where` of the 2nd operator:

$$\texttt{wherepos\_neigh} = \Big[\big[\text{positions of related hoppings to } \texttt{op1.where[0]}\big], \big[\text{positions of related hoppings to } \texttt{op1.where[1]}\big], \ldots\Big]. \tag{4}$$

Thus, for our example above, in the case that we want to relate all hoppings with the same sites, we get:

$$\texttt{wherepos\_neigh} = \Big[[1], [0, 2]\Big] \tag{5}$$

Note that also the other case mentioned in the example above (connect the 2nd hopping of op1 only with the third hoppping of op2), is possible with `wherepos_neigh` $= \Big[[1], [2]\Big]$. As far as we can see, any possible combination of hoppings can be achieved using `wherepos_neigh`. Note that in the code, `wherepos_neigh`, which is so far a list of lists, is flattened and an auxiliary list called `auxpos_list` is created for bookkeeping, in which the starting positions of each previous sublist in the now flattened `wherepos_neigh`-list are stored.

One possibility is that the user has to tell us this `wherepos_neigh`-list, *i.e.* which "paths" he wants to have. However, for most physical cases (at least for the energy current and all other example operators above), we just want to go through all possible *connected* hoppings of the operators. (According to our definition, a hopping (a, b) of operator1 is connected to a hopping (c, d) of operator2, iff $b == c$.) This case of looking for all connected hoppings in the next operator is the default case and for that one, the user does not have to give us his desired `wherepos_neigh`-list. Instead we create the `wherepos_neigh`-list by searching for all possible combinations, *i.e.* all possible paths which are allowed by the given `where`'s of the operators.

This path finding algorithm is implemented with the help of a dictionary. For each `where` (*i.e.* for each operator) a dictionary is created whose keys are the site-Ids of the first site in the hoppings and the values are lists, at which position the corresponding hoppings are to be found. Creating this dictionary should be of order of the number of hoppings in `where`.

To create the `wherepos_neigh` from this dictionary, we only have to copy the list which is returned by the dictionary when called with the 2nd site of the considered hopping. We illustrate that principle with the example from above.

Example:

- op1: `where` $= \left[\begin{pmatrix} 41 \\ 12 \end{pmatrix}, \begin{pmatrix} 14 \\ 15 \end{pmatrix} \right]$
- op2: `where` $= \left[\begin{pmatrix} 15 \\ 23 \end{pmatrix}, \begin{pmatrix} 12 \\ 4 \end{pmatrix}, \begin{pmatrix} 15 \\ 2 \end{pmatrix} \right]$

The dictionary which is related to op2 is:
`_new_dict` $= \{15 : [0, 2], 12 : [1]\}$. (Remember: keys are the Site-IDs, the values are lists of the position where to find them.)

To create the corresponding `wherepos_neigh`, we take the first hopping of `op1.where`, use its 2nd site-ID which is '12' and ask the dictionary `_new_dict[12]` to yield the corresponding list of positions, which is [1]. As a formula, we have

$$\texttt{wherepos\_neigh[i]} = \texttt{\_new\_dict}\big[\texttt{op1.where[i][1]}\big] \tag{6}$$

and we are done. Accessing a dictionary seems to be $\mathcal{O}(1)$, which makes this process in total $\mathcal{O}(N_{\text{sites}})$, assuming that the number of elements in the `where`-lists is $\mathcal{O}(N_{\text{sites}})$.

Note that we did not have to create a dictionary for `op1.where`, because it is the starting object (we do not have to connect hoppings from the left).

In the actual calculation, `wherepos_neigh` is used in the recursive function `sitePath_recfunc` to go along a path and store the matrices belonging to the operators for the given hoppings in dummy lists. For that we start with the first hopping from the left most-operator and call the recursive function `sitePath_recfunc`. In that recursive function, we loop over all connected sites which are given by `wherepos_neigh[0]`, store the according operator-matrix and number of orbitals at those sites and call the recursive function `sitePath_recfunc` again. This process is repeated until we reach the right-most operator (recursion end), where we initialize the sum over all orbitals, which is again handled by a recursive function.

D. Overview of Op_Prod.__init__(..)

For the most part, the class variables for the operator-product can be directly copied/deduced from the individual operators. The only exceptions are the class variables which are related to the path finding (= creating `wherepos_neigh`).

To avoid the tedious path finding process if possible, the boolean variable `willNotBeCalled` of onsite operators can be `True`, because in that case the path finding can be circumvented as shown below. In total, there are 4 possibilities:

1. for all operators `willNotBeCalled==True` AND the operator product is not allowed to be called
2. for all operators `willNotBeCalled==True` AND the operator product is allowed to be called
3. some operators have `willNotBeCalled==True` other `willNotBeCalled==False` ⇒ operator product is callable
4. for all operators `willNotBeCalled==False` ⇒ operator product is callable

In the first case, `wherepos_neigh` does not have to be created since it will not be called anyway.

In the second case, `wherepos_neigh` has to be created but is trivial since we are dealing with only onsite operators (only onsite can have `willNotBeCalled==True`).

The forth case is discussed in detail in Subsec. II C, *i.e.* `wherepos_neigh` is either given by the user or created with the help of dictionaries.

The third case needs a little more discussion and is the reason why `willNotBeCalled` was introduced in the first place. One reason to have the `willNotBeCalled`-Option is to avoid unnecessary path finding. Another reason is that the ordering of the result array might change as compared to the ordering in `where`.

This change of ordering could happen for instance for the Spin Current. Let's assume `where` $= \left[\begin{pmatrix}41\\12\end{pmatrix}, \begin{pmatrix}14\\15\end{pmatrix}, \begin{pmatrix}41\\26\end{pmatrix}\right]$. To calculate the z-spin current for these hoppings, we would need something like:

```
hamil = Operator(syst, syst.hamiltonian, in_where=where)
sz = Onsite(syst, sigma_z)
sz_current = Op_Product(sz, hamil, withRevTerm=-1, const_fac=-1j)
```

The relevant part of `where` for the sz-onsite operator would be something like `sz.where` $= \left[\begin{pmatrix}41\\41\end{pmatrix}, \begin{pmatrix}14\\14\end{pmatrix}\right]$. For this product, the path finding algorithm would lead a `wherepos_neigh` $= \left[[0,2],[1]\right]$. Since the recursive function first treats all paths that start at the same site before considering other starting sites, the result array would have the form

$$\texttt{result} = \left[\text{SpinCurrent at } \begin{pmatrix}41\\12\end{pmatrix}, \text{SpinCurrent at } \begin{pmatrix}41\\26\end{pmatrix}, \text{SpinCurrent at } \begin{pmatrix}14\\15\end{pmatrix}\right]$$

instead of

$$\texttt{result} = \left[\text{SpinCurrent at } \begin{pmatrix}41\\12\end{pmatrix}, \text{SpinCurrent at } \begin{pmatrix}14\\15\end{pmatrix}, \text{SpinCurrent at } \begin{pmatrix}41\\26\end{pmatrix}\right]$$

as it would be the case for the `kwant.operator.Current` because it corresponds to the ordering of the initial `where`.

Long story short, to avoid this reordering and to avoid the path finding at all, we used the option `willNotBeCalled`. When a `'willNotBeCalled==True'`-Operator is multiplied with a `'willNotBeCalled==False'`-Operator, the `where` of the callable operator is copied for the uncallable operator as fake hoppings, since the uncallable operator is an onsite operator. (Depending on the position of the uncallable operator either the 1st sites in the hoppings are used or the 2nd sites.)

The important `wherepos_neigh` becomes trivially: `wherepos_neigh` $= \left[[0],[1],[2],\ldots,[\texttt{Nhops\_tot-1}]\right]$ and we have everything that is needed.

III. FINAL REMARKS

Note that most of the discussion in this manuscript is for two operators, but the implementation works for a product of an arbitrary amount of operators. Most importantly, the product of operators, which were themselves initialized as products of operators, is possible.

Caveat: At the moment an arbitrary amount of operators can be multiplied at once only if all of the operators have the same `willNotBeCalled` (either all `False` or all `True`), which are the cases 1., 2., and 4. in the list in Subsec. II D. In the case of a mixture (case 3.), only the multiplication of exactly 2 instances of the `generalOperator.Operator`-class (which could themselves be products of operators) is implemented at the moment. The reason for the restriction to 2 operators is the following. Imagine you want to do a product of 3 operators, the first 2 are uncallable, the 3rd one is callable. The 'products' are executed from left to right which means that first, the uncallable operators are multiplied. However neither of them has yet a `where`, so it is not yet clear which one they should obtain. When the 'product' between 2nd and 3rd operator is executed, everything is fine since the 3rd operator has a `where` which is then copied for the 2nd operator. However, we either would need to copy this `where` also to the first operator retrospectively, or start from the product of 2nd and 3rd operator and then move to the left (and to the right if there would be more operators). Since this implementation might be a little more tedious but the restriction of two operators does not limit at all the generality of the operator product – we could just define a product of only the 1st and 2nd operator with `willNotBeCalled=True` and then multiply this resulting product operator with the 3rd operator – we decided to keep it like that at the moment. In the future, an elegant way to circumvent this 2 operator restriction would be to automatically check for the total number of operators to be multiplied. If it is larger than 2, just multiply the first

2 operators, then multiply their product with the third one, then multiply this product with the 4th operator, and so on. Like this, the user does not have to worry about the number of operators multiplied at all.