Defining the hadron operators

In our code the quarks are always defined as follows:

$$u = 1\,, \quad \bar{u} = 1.2\,, \quad \bar{d} = 2\,, \quad \bar{d} = 2.2\,, \quad \bar{s} = 3\,, \quad \bar{s} = 3.2\,. \tag{1}$$

The operators are expressed using the list notation in Python by their flavor constituents. Here we have to distinguish between two cases:

1. Without superposition: A hadron operator, composed of m hadrons combined with a total of n quarks, with a Dirac structure $d^{\alpha\beta\cdots\gamma}$, a color structure $E^{ab\cdots c}$, and a constant factor C, takes the form

$$\mathcal{O} = C\, d^{\alpha\beta\cdots\gamma} E^{ab\cdots c} (f_1)^{\alpha\, a}\, (f_2)^{\beta\, b} \cdots (f_n)^{\gamma\, c}\;\;, \tag{2}$$

and is defined as

```
HadronO = [C, [f1, f2, ..., fn], ['T1', 'T2', ..., 'Tm'] ]
```

where the coefficients T_i represents the types of the m hadrons. Here are some examples:

- $\Delta^{++} = d^{\Delta k}_{\alpha\beta\gamma}\, \Phi^{uuu}_{\alpha\beta\gamma}$ is represented by:

```
 Deltapp = [1, [1, 1, 1], "d2"]
```

- $\pi^{-} = -d^{\pi k}_{\alpha\beta} \Phi^{ud}_{\alpha\beta}$ is represented by:

```
Pim = [-1, [1.2, 2], "M"]
```

- $\pi^{+} N^{+} = d^{Nk}_{\alpha\beta\gamma} d^{\pi k}_{\kappa\lambda}\, \Phi^{du}_{\kappa\lambda}\, \Phi^{uud}_{\alpha\beta\gamma}$ is represented by:

```
#two_hadron_operator([1/2, 1], 3/2, 3/2, 'Nucleon', 'Pi')=
[-1, [1, 1, 2, 2.2, 1], ["n", "M"] ]
```

- $\pi^{+}\pi^{+}\pi^{+} = d^{\pi k}_{\alpha\beta}\, d^{\pi k}_{\gamma\kappa}\, d^{\pi k}_{\lambda\omega}\, \Phi^{du}_{\alpha\beta}\, \Phi^{du}_{\gamma\kappa}\, \Phi^{du}_{\lambda\omega}$ is represented by:

```
#three_hadron_operator([1, 1, 1], 3, 3, 3, 'Pi', 'Pi', 'Pi')=
[1, [2.2, 1, 2.2, 1, 2.2, 1], ["M", "M", "M"] ]
```

2. Superposition of hadrons operators: In general, a super position of hadron operators can be written as:

$$\mathcal{O}_S = d^{\alpha\beta\cdots\gamma}_1 E^{ab\cdots c}_1 \Big(C_1\, (f_1)^{\alpha\, a}\, (f_2)^{\beta\, b} \cdots (f_n)^{\gamma\, c} + C_2\, (g_1)^{\alpha\, a}\, (g_2)^{\beta\, b} \cdots (g_3)^{\gamma\, c} + \cdots \Big)\;, \tag{3}$$

and is defined as

```
HadronO = [  [C1, [f1, f2, ..., fn], ['T1', 'T2', ..., 'Tm']],
             [C2, [g1, g2, ..., gn], ['t1', 't2', ..., 'tm']], ...   ]
```

Here are some examples on operators with superpositions:

- $\pi^{0} = \frac{1}{\sqrt{2}} d^{\pi k}_{\alpha\beta} \left(\Phi^{dd}_{\alpha\beta} - \Phi^{uu}_{\alpha\beta} \right)$ is represented by:

```
Pi0 = [[1/sp.sqrt(2), [2.2, 2],'M'], [-1/sp.sqrt(2), [1.2, 1],'M']]
```

- $N^{+}N^{0} - N^{0}N^{+} = d^{Nk}_{\alpha\beta\gamma} d^{Nk}_{\kappa\lambda\omega} \left(\Phi^{uud}_{\alpha\beta\gamma} \Phi^{ddu}_{\kappa\lambda\omega} - \Phi^{ddu}_{\alpha\beta\gamma}\, \Phi^{uud}_{\kappa\lambda\omega} \right)$ is represented by:

```
#two_hadron_operator([1/2, 1/2], 0, 0, 'Nucleon', 'Nucleon') =
[[-1, [2, 2, 1, 1, 1, 2], ['n', 'n']], [1, [1, 1, 2, 2, 2, 1], ['n', 'n']]]
```

- $\frac{1}{\sqrt{5}} N^{0}\pi^{+}\pi^{+} + \sqrt{\frac{2}{5}} N^{+}\pi^{0}\pi^{+} + \sqrt{\frac{2}{5}} N^{+}\pi^{+}\pi^{0}$ is represented by:

```
#three_hadron_operator([1, 1, 1/2], 5/2, 3/2, 2, 'Pi', 'Pi', 'Nucleon')=
[[-sp.sqrt(5)/5, [2, 2, 1, 2.2, 1, 2.2, 1], ['n', 'M', 'M']],
 [sp.sqrt(5)/5, [1, 1, 2, 2.2, 2, 2.2, 1], ['n', 'M', 'M']],
 [-sp.sqrt(5)/5, [1, 1, 2, 1.2, 1, 2.2, 1], ['n', 'M', 'M']],
 [sp.sqrt(5)/5, [1, 1, 2, 2.2, 1, 2.2, 2], ['n', 'M', 'M']],
 [-sp.sqrt(5)/5, [1, 1, 2, 2.2, 1, 1.2, 1], ['n', 'M', 'M']]]
```

Performing the quark contractions

We want now to obtain all Wick contractions of a correlator of the form

$$\mathcal{C} = \left\langle \mathcal{O}_1\, \mathcal{O}_2 \cdots \mathcal{O}_n \bar{\mathcal{O}}_m\, \bar{\mathcal{O}}_{m-1} \cdots \right\rangle \,, \tag{4}$$

where n and m are the number of operators on the sink and on the source, respectively. For that we build a code, whose structure is as follows:

1. We start with two input-lists, one at the sink, and one at the source. Each list contains operators of single-hadron operators. The input-lists take the form

```
Sink = [O1, O2, ...]
Source = [OB1, OB2, ...]
```

2. As next, we obtain the overall factor ($z0$) for the correlator as follows

```
z0 = 1
for i in range(len(Sink)):
    z0 *= Sink[i][0]
for j in range(len(Source)):
    z0 *= Source[j][0]
```

3. After determining the overall factor, we multiply the flavor of the quarks on the sink by -1. We also assign a number to each quark on the sink and source according to its position, from 0 to $len(Sink) - 1$ and 0 to $len(Source) - 1$, respectively. After that we combine all quarks and the corresponding numbers into one list called f_t_o:

```
total_operator = []
counter = 0
for i in range(len(Sink)):
    sub_list = []
    for quark in Sink[i][1]:
        sub_list.append([-1 * quark, counter])
        counter += 1
    total_operator.append(sub_list)
counter = 0
for j in range(len(Source)):
    sub_list = []
    for quark in Source[j][1]:
        sub_list.append([quark, counter])
        counter += 1
    total_operator.append(sub_list)
    f_t_o = [quark for hadron in total_operator for quark in hadron]
```

4. After that we take the list f_t_o and generate from it all possible Wick contractions by using the function *combinations* from the package *itertools* by doing the following steps:

   - We form all possible pair combinations from the elements in the list f_t_o and keep only those with a flavor difference of 0.2, i.e. those of the same flavor. Afterward, we generate all $(len(f\_t\_o)/2)$-combinations from the contracted pairs and retain only the valid ones, i.e., those that are unique, ensuring each element appears exactly once to avoid duplication:

```
def is_valid_contraction(candidate, Start_List):
    all_elements = [elem1 for pairK in candidate for elem1 in pairK]
    return len(set(map(tuple, all_elements))) == len(Start_List)

def Generate_Contraction(Start_List):
    pairs = [list(pair) for pair in combinations(Start_List, 2)]
    possible_pairs = []
    for paar_contraction in pairs:
        if np.abs(paar_contraction[0][0]) == (np.abs(paar_contraction[1][0]) +
            0.2) or (np.abs(paar_contraction[0][0]) + 0.2) == np.abs(
            paar_contraction[1][0]):
            possible_pairs.append(paar_contraction)
    pairs2 = [list(pairX) for pairX in combinations(possible_pairs, len(
        Start_List) // 2)]
    final_solution = []
    for candidate in pairs2:
        if is_valid_contraction(candidate, Start_List):
            final_solution.append(candidate)
    return final_solution
```

- All (unsimplified) topologies for the Wick contractions are now obtained from

```
first_Step = Generate_Contraction(f_t_o)
```

The reason why we call it *first_Step* is because we still have to determine the sign coming from the Grassmann algebra and multiply each topology with the overall factor $z0$. But before doing that, we must put all operators, which are in bar, on the right hand side, so it coincides with the standard notation for the propagator:

```
second_Step = []
for diagram in first_Step:
    new_diagram = []
    for paar_contraction in diagram:
        if (np.abs(paar_contraction[0][0]) + 0.2) == np.abs(paar_contraction
            [1][0]):
            new_diagram.append(paar_contraction)
        else:
            new_diagram.append([paar_contraction[1], paar_contraction[0]])
    second_Step.append(new_diagram)
```

5. The sign of each topology can be determined as follows: Initially, the quarks in *first_Step* can be numbered as follows, where m_c represents the number of quarks on the sink, and m_{cc} on the source:

$$[0, 1, \cdots, m_c - 1, m_c, m_c + 1, \cdots, m_{cc} - 1]$$

Hence, for any topology, if the numbering places a larger number before a smaller number, this gives one negative sign. The total overall sign coming from the Grassmann algebra can thus be obtained as follows:

```
def permutation_sign(permuted_elements):
    inversions = 0
    nF = len(permuted_elements)
    for i in range(nF):
        for j in range(i + 1, nF):
            if permuted_elements[i] > permuted_elements[j]:
                inversions += 1
    sign = (-1) ** inversions
    return sign
sign_of_diagram = [0 for _ in second_Step]
diagram_N = 0
for diagram in second_Step:
    numbers = []
    for paar_contraction in diagram:
        if paar_contraction[0][0] < 0:
            t1 = paar_contraction[0][1]
        else:
            t1 = paar_contraction[0][1] + m_c
        numbers.append(t1)
        if paar_contraction[1][0] < 0:
            t2 = paar_contraction[1][1]
        else:
            t2 = paar_contraction[1][1] + m_c
        numbers.append(t2)
    sign_of_diagram[diagram_N] = permutation_sign(numbers)
    diagram_N += 1

last_step = [0 for _ in second_Step]
for i, diagram in enumerate(second_Step):
    Diagram_Propagator_Form = [0 for _ in diagram]
    for j, pair_contraction in enumerate(diagram):
        Diagram_Propagator_Form[j] = pair_contraction
    last_step[i] = [sign_of_diagram[i] * z0, Diagram_Propagator_Form]
```

The list *last_step* contains all possible Wick contraction topologies, where each topology contains the corresponding sign and overall factor. Before moving to the next step about simplifying the diagrams and making the output readable for the user, we discuss input-lists, which contain superpositions of hadrons. Sink and/or source with operators with superpositions can be dealt as follows: We start by creating an empty list, *elements*, to store the processed operators. Then, we iterate through each operator in the input list of hadrons:

- If an operator is a superposition (i.e., its first element is a list), we append it directly to *elements*.
- If it is not a superposition (i.e., its first element is not a list), we wrap it in a list and then append it to *elements*.

After processing all operators, we compute the Cartesian product of the lists in *elements* using the *product* function. This generates all possible combinations of operators. Finally, we convert each combination (which is a tuple) into a list, so that each element of the output is a list representing a combination of sink/source operators. All of this can be done using the following function:

```
def hadron_multiplication(hadrons):
    elements = []
    for item in hadrons:
        if isinstance(item[0], list):
            elements.append(item)
        else:
            elements.append([item])
    result0 = list(product(*elements))
    result = [list(tup) for tup in result0]
    return result
```

That is, in general, after receiving input from the user, we can always pass it to the multiplication function and perform all Wick contractions for each combination separately. Afterward, we save the results in a single list:

```
Sink = hadron_multiplication(Sink)
Source = hadron_multiplication(Source)
number_of_correlators = len(Sink) * len(Source)
Total_Wick_Contractions = [0 for i in range(number_of_correlators)]
zeiger = 0
for i in range(len(Sink)):
    for j in range(len(Source)):
        sink1 = Sink[i]
        source1 = Source[j]
        Wick_Contractions = Determine_Contractions(sink1, source1)#This function
            determines the Wick contractions for input without superpositions and
            works as explained in the previous steps.
        Total_Wick_Contractions[zeiger] = Wick_Contractions
        zeiger += 1
Total_Wick_Contractions = [Final_element for sublist_1 in Total_Wick_Contractions for
     Final_element in sublist_1]
```

6. Having the Wick contractions, we can now perform the simplifications. The simplification is, in general, due to the relations between the Dirac structures of some hadrons:

$$
\begin{aligned}
&d^{\Sigma\,k}_{\alpha\beta\gamma} = d^{\Sigma\,k}_{\beta\alpha\gamma}\,, \quad d^{\Xi\,k}_{\alpha\beta\gamma} = d^{\Xi\,k}_{\beta\alpha\gamma}\,, \quad d^{\Lambda\,k}_{\alpha\beta\gamma} = -d^{\Lambda\,k}_{\beta\alpha\gamma}\,, \quad d^{N\,k}_{\alpha\beta\gamma} = d^{N\,k}_{\beta\alpha\gamma}\,,\\
&d^{\Sigma\,k}_{\alpha\beta\gamma} = d^{\Sigma\,k}_{\beta\alpha\gamma} = d^{\Sigma\,k}_{\alpha\gamma\beta} = d^{\Sigma\,k}_{\beta\gamma\alpha} = d^{\Sigma\,k}_{\gamma\beta\alpha} = d^{\Sigma\,k}_{\gamma\alpha\beta}\,, \quad d^{\Delta\,k}_{\alpha\beta\gamma} = d^{\Delta\,k}_{\beta\alpha\gamma} = d^{\Delta\,k}_{\alpha\gamma\beta} = d^{\Delta\,k}_{\beta\gamma\alpha} = d^{\Delta\,k}_{\gamma\beta\alpha} = d^{\Delta\,k}_{\gamma\alpha\beta}\,.
\end{aligned}
\tag{5}
$$

In principle, the simplification can be performed on two levels. The first level, without isospin symmetry, can be achieved using the following steps:

- We take each topology from *Total_Wick_Contractions* and check if it reduces to any other existing topology by exchanging the positions of the quarks according to Eg. (5).
- If it reduces to another topology, we remove this element and add its numerical factor to that of the element it was compared with, together with the corresponding relative sign.

To perform the simplification on this level we define a function, whose arguments are:

- The list of all topologies, together with their numerical factors, that we want to simplify *result_to_simplify*.
- The list contains two elements, representing the positions of the quarks we want to simplify *simplification_pair*.
- The relative sign between the two topologies that reduce to each other is determined by Eq. (5). For example, the relative sign due to the exchange of the first two quarks in the nucleon is -1.
- The location of the pairs that we want to exchange, i.e. *Sink* or *Source*

```python
def to_tuple(item):
    return tuple(map(to_tuple, item)) if isinstance(item, list) else item#This function converts all nested lists into tuples within a given structure. It is needed when the order of elements in sets does not play a role. In our case, those elements are the contracted quark pairs.

def pair_simplifying(location, simplification_pair, relative_sign, result_to_simplify):
    Simplified_Result0 = copy.deepcopy(result_to_simplify)
    lenR = len(result_to_simplify)
    x = simplification_pair[0]
    y = simplification_pair[1]
    if location == 'sink':
        for jX in range(lenR):
            for jY in range(lenR):
                if jX < jY:
                    re_construct = []#We reconstruct the topology by exchanging x and y
                    for jZ in result_to_simplify[jY][1]:#Hier result_to_simplify[jY][1] is the topology, and result_to_simplify[jY][0] the corresponding numerical factor
                        q_f = jZ[0][0]
                        q_n = jZ[0][1]
                        Source_q_n = jZ[1]#The condition q_f < 0 ensures that it is on the sink.
                        if q_f < 0 and q_n == x: #If the quark position is equal to x, we exchange it with y.
                            re_construct.append([[q_f, y], Source_q_n])
                        elif q_f < 0 and q_n == y:#If the quark position is equal to y, we exchange it with x.
                            re_construct.append([[q_f, x], Source_q_n])
                        else:
                            re_construct.append(jZ)#Now we check if the reconstructed topology is equal to any other existing topology
                    if Simplified_Result0[jY] != [0, 0] and Simplified_Result0[jX] != [0, 0] and set(to_tuple(re_construct)) == set(to_tuple(Simplified_Result0[jX][1])):#If this is the case, we remove one of the topologies being compared and add the numerical factor, with the relative sign, to the remaining one
                        Simplified_Result0[jX][1] = copy.deepcopy(Simplified_Result0[jY][1])
                        Simplified_Result0[jX][0] = Simplified_Result0[jY][0] + relative_sign * Simplified_Result0[jX][0]
                        Simplified_Result0[jY][0] = 0
                        Simplified_Result0[jY][1] = 0
        Simplified_Result = [item for item in Simplified_Result0 if item != [0, 0]]
        return Simplified_Result

    if location == 'source':#Here, we repeat the same procedure, but this time on the source
        for jX in range(lenR):
            for jY in range(lenR):
                if jX < jY:
                    re_construct = []
                    for jZ in result_to_simplify[jY][1]:
                        q_f = jZ[1][0]# Remember we exchange the pairs at the source q_f >0
                        q_n = jZ[1][1]#We exchange quarks at source -> we look at jZ[1][1] always!
                        Sink_q_n = jZ[0]
                        if q_f > 0 and q_n == x:
                            re_construct.append([Sink_q_n, [q_f, y]])
                        elif q_f > 0 and q_n == y:
                            re_construct.append([Sink_q_n, [q_f, x]])
                        else:
                            re_construct.append(jZ)
                    if Simplified_Result0[jY] != [0, 0] and Simplified_Result0[jX] != [0, 0] and set(to_tuple(re_construct)) == set(to_tuple(Simplified_Result0[jX][1])):
                        Simplified_Result0[jX][1] = copy.deepcopy(Simplified_Result0[jY][1])
                        Simplified_Result0[jX][0] = Simplified_Result0[jY][0] + relative_sign * Simplified_Result0[jX][0]
                        Simplified_Result0[jY][0] = 0
                        Simplified_Result0[jY][1] = 0
        Simplified_Result = [item for item in Simplified_Result0 if item != [0, 0]]
        return Simplified_Result
```

To simplify at the level of isospin symmetry, we first present our results in a clear and readable form. The idea is as follows

- Instead of identifying quarks by their positions and flavors, we now identify them based on the position of the hadron they belong to and their position within this hadron. Additionally to that we assign hadrons on the sink by 1 and hadrons on the source by 0. The notation for a given quark takes then the form:

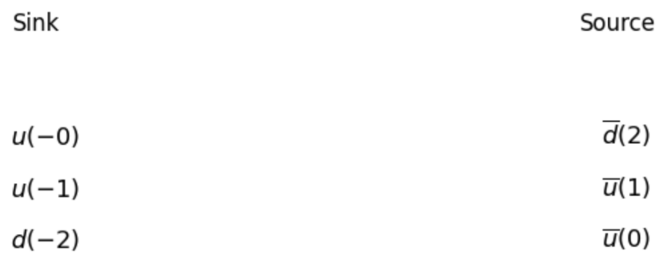
  $$[0 \text{ or } 1, \text{position of the hadron}, \text{position of the quark inside the hadron}]$$

  For example, if the first hadron on the sink is a proton, then the three quarks inside of it are represented as:

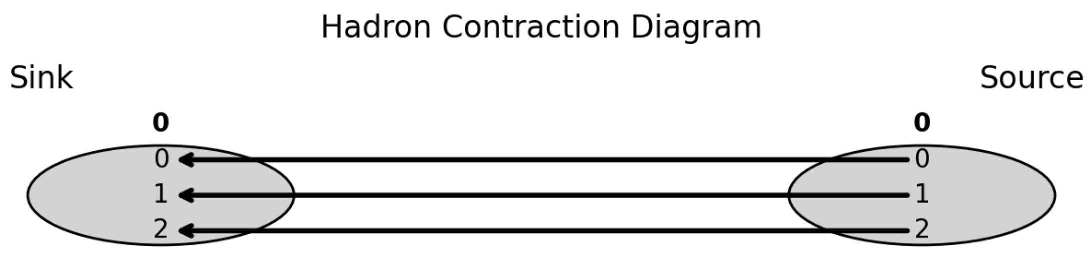
  $$\begin{aligned}&\mathcal{N} = u\,u\,d \rightarrow q^0 q^1 q^2\\ &q^0 = [1,0,0]\ ,\quad q^1 = [1,0,1]\ ,\quad q^2 = [1,0,2]\ .\end{aligned}$$

Before proceeding to the final step of the simplification, we summarize the general procedure using the example of the proton-to-proton correlator:

- The two nucleons and the quarks inside of them are presented at the beginning by



- We then introduce the notation in terms of hadrons, and obtain the following result for the contractions: $[\,[\,[[1,0,0],[0,0,0]],[[1,0,1],[0,0,1]],[[1,0,2],[0,0,2]]\,],2]$. Diagrammatically, this result takes the following form:



7. Now we can do the simplification with the isospin symmetry, since our notation now does not distinguish between up and down quarks. We define now a simplification code, which is similar to the one given above, with the following arguments:
   - The list of all topologies, together with their numerical factors, that we want to simplify $result\_to\_simplify\_F$.
   - The relative sign between the two topologies that reduce to each other is determined by Eq. (5).
   - The list that contains all the information about the quark pairs we want to simplify—including their location, hadron position, and quark position—is called *HQ_List*. This list represents the most important part of the code, as it determines how many simplifications we want to perform simultaneously. Each element in *HQ_List* has the following structure:

$$[\text{ sink or source},\ \text{hadron position},\ [q_1,q_2]\ ]\,,$$

where q_1 and q_2 are the quark positions in the corresponding hadron that we want to exchange with each others. Here are some examples how this list can look like for the correlator describing $2\Lambda^0 \mapsto 2\Lambda^0$:

```
#One pair only (There are 4 options)
HQ_List = [[1, 0, [0, 1]]]
#Two pairs at the same time (There are 6 options)
HQ_List = [[1, 0, [0, 1]], [0, 1, [0, 1]]]
#Three pairs at the same time (There are 4 options)
HQ_List = [[1, 0, [0, 1]], [1, 1, [0, 1]], [0, 1, [0, 1]]]
#Four pairs at the same time (There is one option)
HQ_List = [[1, 0, [0, 1]], [1, 1, [0, 1]], [0, 1, [0, 1]], [0, 0, [0, 1]]]
```

Now we present how the simplification function looks like

```
def Sim_KD(HQ_List, result_to_simplify_F, relativ_sign):
    Simplified_Result0 = copy.deepcopy(result_to_simplify_F)
    lenR = len(result_to_simplify_F)
```

```python
    #One Hadron To Simplify
    if len(HQ_List) == 1:
        #Information about the quark pair
        location = HQ_List[0][0]
        hadron = HQ_List[0][1]
        q1 = HQ_List[0][2][0]
        q2 = HQ_List[0][2][1]
        for jX in range(lenR):
            for jY in range(lenR):
                if jX < jY:
                    re_construct = []
                    for jZ in result_to_simplify_F[jY][0]:#jZ represents a quark contraction pair, i.e. jZ = [ [0/1, hadronX, quarkX], [0/1, hadronY, quarkY] ]
                        recN1 = [] #This list is for reconstructing the zeroth element in jZ by exchanging the quark pair
                        recN2 = []#This list is for reconstructing the first element in jZ by exchanging the quark pair
                        if (jZ[0][0] == location) and (jZ[0][1] == hadron) and (jZ[0][2] == q1):
                            recN1 = [location, hadron, q2]
                        elif (jZ[0][0] == location) and (jZ[0][1] == hadron) and (jZ[0][2] == q2):
                            recN1 = [location, hadron, q1]
                        else:
                            recN1 = jZ[0]
                        if (jZ[1][0] == location) and (jZ[1][1] == hadron) and (jZ[1][2] == q1):
                            recN2 = [location, hadron, q2]
                        elif (jZ[1][0] == location) and (jZ[1][1] == hadron) and (jZ[1][2] == q2):
                            recN2 = [location, hadron, q1]
                        else:
                            recN2 = jZ[1]
                        re_construct.append([recN1, recN2])
                    if Simplified_Result0[jY][1] != 0 and Simplified_Result0[jX][1] != 0 and set(to_tuple(re_construct)) == set(to_tuple(Simplified_Result0[jX][0])):#We check here whether two topologies are equal to each others and proceed similarly to the previous simplification code
                        Simplified_Result0[jX][0] = copy.deepcopy(Simplified_Result0[jY][0])
                        Simplified_Result0[jX][1] = Simplified_Result0[jY][1] + relativ_sign * Simplified_Result0[jX][1]
                        Simplified_Result0[jY][1] = 0
        Simplified_Result = [item for item in Simplified_Result0 if item[1] != 0]
        return Simplified_Result
    #We exchange here two quark pairs. This part of the code is similar to the previous one but we add to it the exchanging of one further pair
    if len(HQ_List) == 2:
        #Information about the first quark pair
        location1 = HQ_List[0][0]
        hadron1 = HQ_List[0][1]
        q1 = HQ_List[0][2][0]
        q2 = HQ_List[0][2][1]
        #Information about the second quark pair
        location2 = HQ_List[1][0]
        hadron2 = HQ_List[1][1]
        p1 = HQ_List[1][2][0]
        p2 = HQ_List[1][2][1]
        for jX in range(lenR):
            for jY in range(lenR):
                if jX < jY:
                    re_construct = []
                    for jZ in result_to_simplify_F[jY][0]:
                        recN1 = []
                        recN2 = []
                        #Here, we exchange two quark pairs, while everything else remains the same as in the one-quark-pair case
                        #First quark pair:
                        if (jZ[0][0] == location1) and (jZ[0][1] == hadron1) and (jZ[0][2] == q1):
                            recN1 = [location1, hadron1, q2]
                        elif (jZ[0][0] == location1) and (jZ[0][1] == hadron1) and (jZ[0][2] == q2):
                            recN1 = [location1, hadron1, q1]
                        #Second quark pair:
                        elif (jZ[0][0] == location2) and (jZ[0][1] == hadron2) and (jZ[0][2] == p1):
                            recN1 = [location2, hadron2, p2]
                        elif (jZ[0][0] == location2) and (jZ[0][1] == hadron2) and (jZ[0][2] == p2):
                            recN1 = [location2, hadron2, p1]
                        else:
                            recN1 = jZ[0]
                        #
                        #First quark pair:
                        if (jZ[1][0] == location1) and (jZ[1][1] == hadron1) and (jZ[1][2] == q1):
                            recN2 = [location1, hadron1, q2]
                        elif (jZ[1][0] == location1) and (jZ[1][1] == hadron1) and (jZ[1][2] == q2):
                            recN2 = [location1, hadron1, q1]
                        #Second quark pair:
                        elif (jZ[1][0] == location2) and (jZ[1][1] == hadron2) and (jZ[1][2] == p1):
                            recN2 = [location2, hadron2, p2]
                        elif (jZ[1][0] == location2) and (jZ[1][1] == hadron2) and (jZ[1][2] == p2):
                            recN2 = [location2, hadron2, p1]
                        else:
                            recN2 = jZ[1]
                        re_construct.append([recN1, recN2])
                    if Simplified_Result0[jY][1] != 0 and Simplified_Result0[jX][1] != 0 and set(to_tuple(re_construct)) == set(to_tuple(Simplified_Result0[jX][0])):
                        Simplified_Result0[jX][0] = copy.deepcopy(Simplified_Result0[jY][0])
                        Simplified_Result0[jX][1] = Simplified_Result0[jY][1] + relativ_sign * Simplified_Result0[jX][1]
                        Simplified_Result0[jY][1] = 0
        Simplified_Result = [item for item in Simplified_Result0 if item[1] != 0]
        return Simplified_Result
    #Three hadrons to simplify
    if len(HQ_List) == 3:
    #...
    #Four hadrons to simplify
    if len(HQ_List) == 4:
    #...
    #Five hadrons to simplify
    if len(HQ_List) == 5:
    #...
    #Six hadrons to simplify
    if len(HQ_List) == 6:
    #...
```