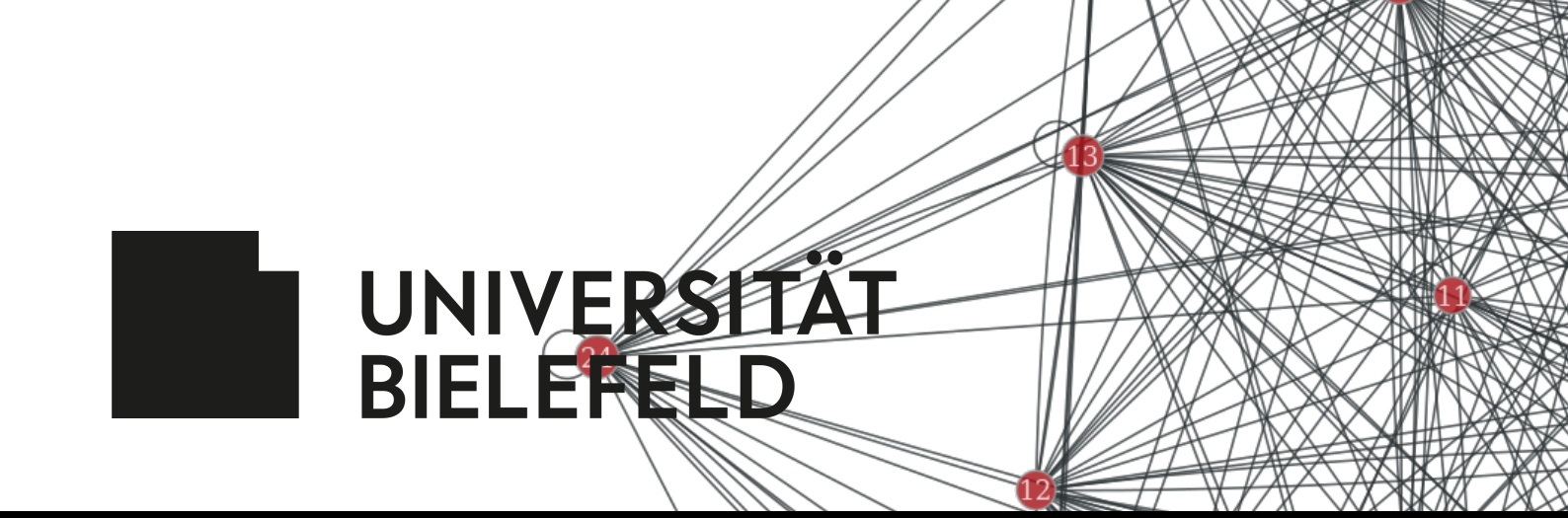
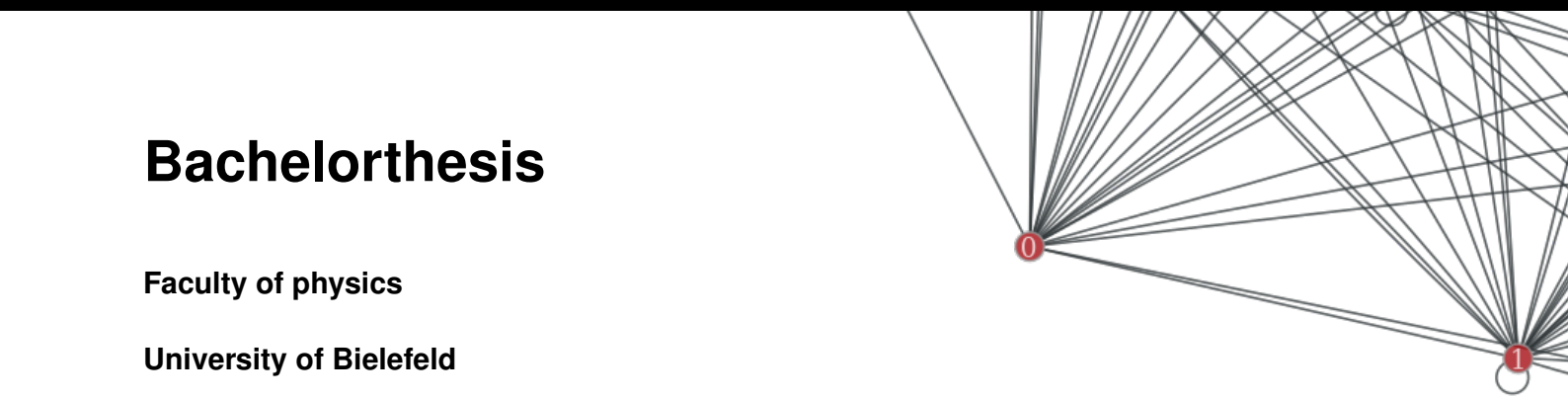


UNIVERSITÄT BIELEFELD

On generalized coupling coefficients of the special orthogonal group



Bachelorthesis

Faculty of physics

University of Bielefeld

submitted by:

Oscar Werner

matriculation number: 2864851

Supervisor and first reviewer: Prof. Dr. Dietrich Bödeker

Second reviewer: Rasmus Nielsen

Bielefeld, 27/10/2023

Contents

1. Introduction

2. Basics

2.1. Group definition

Because the concept of a *group* builds the foundation of this Bachelor thesis, it is critical to first give a precise definition of this term. This is important because the way the term *group* is used colloquially differs from the meaning that physicist and mathematicians attribute to it. In the following chapter firstly the axioms that are used to rigorously define a *group* are presented, after that a few examples can be found that hopefully convey, why the concept is of great use in many areas of physics.

Definition:

A *group* is a nonempty set G of elements $\{g_\alpha\}$, together with a binary composition "$\circ$", that fulfills the following four axioms: [Zee16]

1. **Closure**: if to elements g_α, g_β of G are composed, the result $g_\alpha \circ g_\beta$ is also an element of G
2. **Associativity**: the composition action is associative, that is: $(g_\alpha \circ g_\beta) \circ g_\gamma = g_\alpha \circ (g_\beta \circ g_\gamma)$
3. **Existence of the identity**: there is one special element I, called the identity, in G which has the special property that composing I with any element in G results in the same element : $I \circ g_\alpha = g_\alpha \circ I = g_\alpha$
4. **Existence of the inverse**: for every group element g_α there has to exist a unique group element g_α^{-1}, so that the composition of these two elements results in the identity: $g_\alpha \circ g_\alpha^{-1} = I$

So strictly speaking a group is a pair $(G, \circ)$ of a underlying set and a operation on this set, that takes in two inputs and spits out another member of the set. In the following, this notation will be slightly abused and the group is simply referred to as G.
To distinguish different elements of G the label α is used above. For the moment it is not further specified what this label looks like, it can be finite, countably infinite or even continuous. Examples for each of these three options are presented in the following:

2.2. Finite groups

Finite groups have application amongst other in theoretical solid state physics, where they are used for describing the point symmetries of a crystal lattice. [Czy08] They also find application in the study of elementary particles for describing their internal (finite) symmetries and several other areas of physics. (see e.g.: [Geo00, Chapter 1], [Ish+10], [Lom59], [GL12])
Since the main part of the thesis deals with continuous groups,only two examples are presented.

2.2.1. Ex.: Cyclic group of order 4: $\mathbf{Z}_4$

One of the most simple examples of a finite group is the set of complex numbers $\{1, -1, i, -i\}$ together with the usual multiplication. It is very easy to check that all the axioms from chapter 2.1 are fulfilled. The number of elements in any finite group is called *order* of the group, which is 4 in our case. It is also noteworthy that although here the composition is commutative, that is $g_\alpha \circ g_\beta = g_\beta \circ g_\alpha$ for every group element g_α, g_β, this is not required in general. Groups that have a commutative composition law are called *abelian*.

2.2.2. Ex.: The symmetric group of degree 3: $\mathbf{S_3}$

One of the most important families of finite groups is S_n, the so called symmetric group of degree n. It is the set of the bijective functions on a set of n objects on itself and is of order $n!$.[1] Such a function is called a *permutation* and is said to be *automorph*. By looking at a specific example, namely $n = 3$, many interesting features already emerge, that can be generalised to symmetric groups of higher degrees, other finite groups and even continuous groups.
The goal is two collect all the different permutation that can be done to a set of three objects $X = \{1, 2, 3\}$. For that a commonly used notation is the following: One particular permutation is written using two lines, in the first line the elements of X, below each element the picture under the specific permutation are listed. In this notation the elements of S_3 are the following [BC79]:

$$\begin{aligned} I &= \begin{pmatrix} 1 & 2 & 3 \\ 1 & 2 & 3 \end{pmatrix}, \qquad \sigma_1 = \begin{pmatrix} 1 & 2 & 3 \\ 2 & 3 & 1 \end{pmatrix}, \qquad \sigma_2 = \begin{pmatrix} 1 & 2 & 3 \\ 3 & 1 & 2 \end{pmatrix}, \\ \tau_1 &= \begin{pmatrix} 1 & 2 & 3 \\ 1 & 3 & 2 \end{pmatrix}, \qquad \tau_2 = \begin{pmatrix} 1 & 2 & 3 \\ 3 & 2 & 1 \end{pmatrix}, \qquad \tau_3 = \begin{pmatrix} 1 & 2 & 3 \\ 2 & 1 & 3 \end{pmatrix} \end{aligned} \tag{2.1}$$

The element σ_1 for example can be read as: "1" gets mapped on "2", "2" gets mapped on "3" and "3" gets mapped on "1". It might cause some confusion that ther are now two sets X and S_3. To resolve this, it is helpful to view the elements of S_3 as transformations applied to the set X. This viewpoint of group elements as transformations on a given object, actually is a very common strategy to visualize any given group, continuous or discrete.[Zee16]

[1] A function which maps onto itself is called a endomorphism, for it to be bijective every element of the set has to be mapped to exactly one member of the set (not necessarily a different one)[Tre07]

Now of course it has to be specified, what the composition looks like for the elements listed above. For that a specific example is looked at, where by convention the right permutation is carried out first:

$$\sigma_1 \circ \tau_1 = \begin{pmatrix} 1 & 2 & 3 \\ 2 & 3 & 1 \end{pmatrix} \circ \begin{pmatrix} 1 & 2 & 3 \\ 1 & 3 & 2 \end{pmatrix} = \begin{pmatrix} 1 & 2 & 3 \\ 2 & 1 & 3 \end{pmatrix} = \tau_3 \tag{2.2}$$

This result can be obtained as follows: τ_1 tells that "1" is mapped on itself, σ_1 then maps "1" on "2" so overall "1" gets mapped on "2". "2" gets mapped by τ_1 on "3", which is subsequently mapped on "1" by σ_1.
One interesting property emerges, by looking at the same two elements σ_1, τ_1 and composing them in flipped order:

$$\tau_1 \circ \sigma_1 = \begin{pmatrix} 1 & 2 & 3 \\ 1 & 3 & 2 \end{pmatrix} \circ \begin{pmatrix} 1 & 2 & 3 \\ 2 & 3 & 1 \end{pmatrix} = \begin{pmatrix} 1 & 2 & 3 \\ 3 & 2 & 1 \end{pmatrix} = \tau_2 \tag{2.3}$$

The result of the composition depends on the order of the elements involved. Like mentioned before, the group definition doesn't demand the operation to be commutative. As a result there are groups which elements don't commute. Those groups are called *non-abelian* and S_3 is the smallest example of such a group, with only $3! = 6$ elements.
Although it is relatively easy to see how two elements of S_3 are composed, the procedure is not very intuitive. There is however a very nice visual way to look at S_3 that relies on the interpretation of group objects as transformations on the set X. For that the three objects in X are interpreted as the three vertices of an equilateral triangle, then the elements of S_3 can be viewed as the rotations and reflections that transfer the triangle onto itself. If two groups are obtained by one another by just renaming or reinterpreting the elements like it was done here, the two groups are said to be *isomorphic* to one another. It is fairly obvious that renaming the objects doesn't change the structure of the group which is after all the thing that characterizes it, so two isomorphic groups really are basically identical.
Back to S_3 and it's interpretation as transformations on an equilateral triangle. The six possible actions are shown in the following figure:

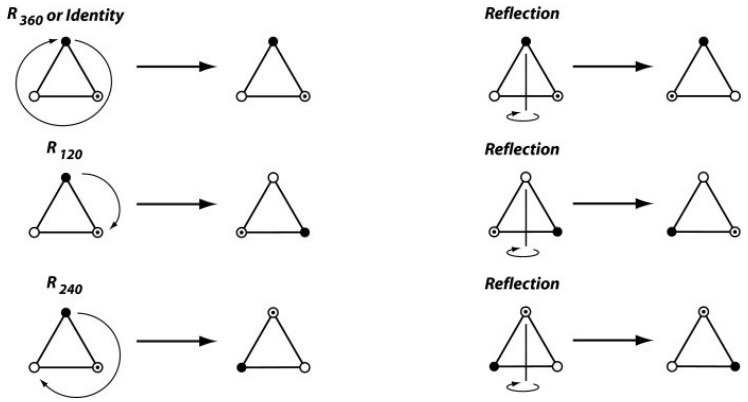


Figure 2.1.: All possible transformation that transfer the equilateral triangle onto itself, corresponding to the 6 elements of S_3 [SK11]

As one can see, the group splits into two parts, the reflections on the right hand side and the rotations on the left hand side of the figure above. One can easily check that the group elements I, σ_1, σ_2 in 2.1 correspond to the rotations, while τ_1, τ_2, τ_3 represent the reflections of the triangle. In this picture the composition of two elements of S_3 simply becomes the back to back execution of any two rotations/reflections. This also makes the non-commutativity of the group apparent; it makes a difference to first reflect along a given axis and then rotate versus doing it the other way around.

2.3. Countably infinite groups

Although the concept of a group with infinitely many members can seem overwhelming at first, there are a few easy examples. Because these groups don't play any major role in the thesis, again only two examples are presented.

2.3.1. Ex.: Additive group of integers

The integers together with addition as the composition law forms a group. Again it can be easily checked that all the axioms are fulfilled; 0 is the neutral element, for each element n there is an inverse element $-n$, the group is closed and the composition law is associative.

2.3.2. Ex.: Multiplicative group of rational numbers

Note that the integers together with multiplication don't form a group because the inverse of any element n, which is $\frac{1}{n}$ is not contained within the set of integers. The rational numbers however have the property that the multiplicative inverse is contained within themselves. So the set of rational numbers $\left\{\frac{n}{m}\right\}$ together with multiplication forms a group.

2.4. Continuous groups

The concept of continuous groups is needed if discrete indices don't suffice to characterize a given group element. Instead continuous indices are used to label the different group elements.

2.4.1. Ex.: Rotations in a plane: SO(2)

The symmetry group of a circle in 2-dimensional Euclidean space forms a group, known as the *special ortogonal group in 2 dimensions* or shortly *SO(2)*. The different group elements are labelled by the angle of rotation, the composition is simply the back-to-back execution of two 2-d rotations which is obviously again a 2-d rotation. As is known from various physic problems, rotations by an angle of ϑ can be represented by a $2 \otimes 2$ matrix:

$$R(\vartheta) = \begin{pmatrix} \cos(\vartheta) & \sin(\vartheta) \\ -\sin(\vartheta) & \cos(\vartheta) \end{pmatrix} \tag{2.4}$$

The set of matrices $\{R(\vartheta), \vartheta \in [0, 2\pi)\}$ are all the matrices that are orthogonal, that is $R(\vartheta)^T \cdot R(\vartheta) = \mathbb{1}$ and have determinant of 1. These two condition ensure that lengths and angles between vectors in the plane are preserved like it can be expected by a rotation.[Zee16]

2.4.2. Ex.: General linear group: GL(n,$\mathbb{R}/\mathbb{C}$)

As was shown in the example above a set of matrices together with ordinary matrix multiplication can be seen as a group, as long as all the matrices have an inverse, or formulated differently a nonzero determinant. This group is called the *General Linear Group* and denoted by GL(n,$\mathbb{R}/\mathbb{C}$) depending on weather the matrices have real or complex components.[2]

[2] even more general one could look at all linear bijective maps from any vector space $\mathcal{V}$ onto itself. Then this set of maps also fulfills the group axioms and is known as the automorphism group of $\mathcal{V}$ or short Aut($\mathcal{V}$). For the special case of $\mathcal{V} = \mathbb{R}^n/\mathbb{C}^n$, Aut($\mathcal{V}$) is isomorphic to GL(n,$\mathbb{R}/\mathbb{C}$). [Böh11]

2.5. Representation theory

In the last chapter it was mentioned that the symmetry group of a circle can be *represented* by a set of matrices $\{R(\vartheta)\}$. Although it was quite intuitive what was meant by that, there is a precise meaning for the term *representation* of a group G. The basic idea is to create a correspondence between each group element g_α and a $d \otimes d$ matrix $D(g_\alpha)$, where d is known as the *dimension* of the representation. Then these matrices are said to *represent* the group G if

$$D(g_\alpha) \cdot D(g_\beta) = D(g_\alpha \circ g_\beta) \tag{2.5}$$

for any two group elements g_α, g_β. Formally the representation then is the map from the group onto the General Linear Group $GL(\mathcal{V})$, with $\mathcal{V}$ being either $\mathbb{R}^n$ or $\mathbb{C}^n$. Condition (2.5) requires this map to be homomorphic, i.e. it has to preserve the group operation.
In physics the line between a given group and a representation of this group often blurs; for example when talked about a 2-d rotation, a physicists will mostly have the matrices given in (2.4) in mind.

2.5.1. Trivial representation and faithfulness

Note that (2.5) allows the creation of a 1-dimensional representation by associating every group element with the number 1. This works for any group and is known as the *trivial* representation. It's obvious that all the information that is contained in the group is lost by representing it trivially. Considering this, a natural thing to do is analyzing if different group elements are always mapped onto different matrices, that is if the representation is injective. If that is the case, the representation is called *faithful* [Tre07]. In this case, the group structure is preserved and no information is lost.

2.5.2. Ex.: The representations of S_3

To make things clearer, next the different possible representations of the group S_3 that was introduced in chapter 2.2.2 are presented. Like mentioned above every group, and therefore also S_3, have a one-dimensional trivial representation defined by $D^{(1)}(g) = 1 \ \forall g \in S_3$. The symmetric group also has a second one-dimensional representation $D^{(1')}$ that is given by the *signum* of any group element. To understand this signum one can make use of the fact that any given permutation can be written as a product of so called *transpositions*; i.e. an exchange of two elements of the underlying set X. The signum-function then maps any permutation onto 1 or -1 depending on weather it can be obtained by an even or uneven number of transpositions respectively[DL15]. For S_3 this results in:

$$D^{(1')}(I) = D^{(1')}(\sigma_1) = D^{(1')}(\sigma_2) = 1 \tag{2.6}$$

$$D^{(1')}(\tau_1) = D^{(1')}(\tau_2) = D^{(1')}(\tau_3) = -1 \tag{2.7}$$

In the interpretation of S_3 as transformations on the equilateral triangle, each reflection is represented by -1, while the two rotations and the identity are represented by 1.

Since the triangle lives in the 2-dimensional plane, and any linear tranformation of that plane can be described by a $2 \otimes 2$ matrix, a 2-dimensional representation of S_3 basically suggests itself. For the rotation part one simply has to evaluate the matrices in eq. (2.4) for $\vartheta = \frac{2\pi}{3}$ and $\vartheta = \frac{4\pi}{3}$.

$$D^{(2)}(I) = \begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix}, \qquad D^{(2)}(\sigma_1) = \frac{1}{2}\begin{pmatrix} -1 & -\sqrt{3} \\ \sqrt{3} & -1 \end{pmatrix}, \qquad D^{(2)}(\sigma_2) = \frac{1}{2}\begin{pmatrix} -1 & \sqrt{3} \\ -\sqrt{3} & -1 \end{pmatrix} \tag{2.8}$$

The three matrices representing the reflections are given subsequently without derivation:

$$D^{(2)}(\tau_1) = \begin{pmatrix} 1 & 0 \\ 0 & -1 \end{pmatrix}, \qquad D^{(2)}(\tau_2) = \frac{1}{2}\begin{pmatrix} -1 & -\sqrt{3} \\ -\sqrt{3} & 1 \end{pmatrix}, \qquad D^{(2)}(\tau_3) = \frac{1}{2}\begin{pmatrix} -1 & \sqrt{3} \\ \sqrt{3} & 1 \end{pmatrix} \tag{2.9}$$

2.5.3. Equivalent representations

Of course the basis in which the representation matrices are written in is free to choose. As it is often the case for physics problems, a basis change can make a seemingly hard problem a lot easier. How to perform this basis change is well known from linear algebra; the matrix D, written in the new basis is given by $S^{-1}DS$ with S inhibiting all the information about the basis change. If two representations D', D are related by such a similarity transformation, that is $D'(g) = S^{-1}D(g)S \ \forall g$, they are called equivalent. [Zee16]; [DL15]

2.5.4. Reducible and irreducible representations

To understand the concept of reducible and irreducible representations again the example of S_3 is looked at. S_3 has two 1-dimensional representations $D^{(1)}$ and $D^{(1')}$ as well as a 2-dimensional one $D^{(2)}$. Using these, it is possible to create the following 3-dimensional representation:

$$D(g) = \begin{pmatrix} D^{(2)}(g) & 0 \\ 0 & D^{(1)}(g) \end{pmatrix} \tag{2.10}$$

It is easy to check that this new representation inherits the requirement (2.5) from $D^{(2)}$ and $D^{(1)}$. Nevertheless this 3-dimensional representation shouldn't count as a proper new representation; it is just a direct sum of smaller representations usually written as $D(g) = D^{(2)}(g) \oplus D^{(1)}(g)$. Whenever the representation matrix is of this block-diagonal form for every group element g it is called *reducible*, when not it is called *irreducible*. In the example (2.10) it was very easy to see that all matrices have this block-diagonal form, after all it was constructed to have exactly this form. However it isn't always this easy to spot whether a representation is reducible or not. Thinking back to chapter 2.5.3 it is possible to construct an equivalent representation by performing a similarity transformation. After this basis change the block diagonal form is most likely gone, depending on the exact form of the change of basis matrix. In fact an example of this was already presented with

the two-dimensional representation of $SO(2)$ in (2.4). These $2 \otimes 2$ matrix can be diagonalized for any value of ϑ in the following way:

$$\underbrace{\begin{pmatrix} e^{-i\vartheta} & 0 \\ 0 & e^{i\vartheta} \end{pmatrix}}_{R'(\vartheta)} = \underbrace{\begin{pmatrix} i & -i \\ 1 & 1 \end{pmatrix}}_{S^{-1}} \cdot \underbrace{\begin{pmatrix} \cos(\vartheta) & \sin(\vartheta) \\ -\sin(\vartheta) & \cos(\vartheta) \end{pmatrix}}_{R(\vartheta)} \cdot \underbrace{\frac{1}{2}\begin{pmatrix} -i & 1 \\ i & 1 \end{pmatrix}}_{S} \tag{2.11}$$

So it can be concluded that the matrices from (2.4), that are usually used to describe rotations in the plane form a reducible representation of $SO(2)$. Note that the diagonalization can only be done using complex entries in the change of basis matrix S.
This result can be generalized to other abelian groups. A well known fact from linear algebra is that two matrices are simultaneously diagonalizable if and only if they commute. [Fis09] Since the representation matrices of abelian groups commute, which is demanded by 2.5, it is possible to diagonalize the representation matrices for every group element in an abelian group. So any representation is of block diagonal form with blocks of size 1. In conclusion every irreducible representation of an abelian group is 1-dimensional and in general complex.

2.5.5. Restriction to a subgroup

In the last chapter the concept or irreducible representations was introduced. There it was explained that a representation $D(g)$ is reducible if it can be block-diagonalized for every $g \in G$. It is fairly obvious that if G has a proper subgroup H with elements $h \in H$, $D(h)$ is a representation of H. To see this fact just ignore all the matrices that represent elements that are in G but not in H, the requirement of H to be a subgroup guarantees that this new set of matrices $D(h)$ is closed under matrix multiplication. However, while the representation of G was irreducible, this is in general no longer the case if the view is restricted to H. This makes sense because the view is limited to fewer matrices, so in most cases it is possible to block-diagonalize this matrices and end up with a direct sum of smaller irreducible representations of H

Ex.: $\mathbf{SO(3) \to SO(2)}$

One group where the restriction to a subgroup can be easily understood is SO(3), the group of rotations in 3-dimensional space. An irreducible 3-dimensional representation $D^{(3)}$ of SO(3) is given by the following set of matrices labelled by 3 angles $\vartheta_1, \vartheta_2, \vartheta_3$. In the following expression abbreviations are used: $\sin(\vartheta_i) = \mathrm{s}(\vartheta_i)$, $\cos(\vartheta_i) = \mathrm{c}(\vartheta_i)$

$$D^{(3)}(\vartheta_1, \vartheta_2, \vartheta_3) = \begin{pmatrix} \mathrm{c}(\vartheta_1)\mathrm{c}(\vartheta_3) - \mathrm{c}(\vartheta_2)\mathrm{s}(\vartheta_1)\mathrm{s}(\vartheta_3) & -\mathrm{c}(\vartheta_1)\mathrm{s}(\vartheta_3) - \mathrm{c}(\vartheta_2)\mathrm{s}(\vartheta_1)\mathrm{c}(\vartheta_3) & \mathrm{s}(\vartheta_1)\mathrm{s}(\vartheta_2) \\ \mathrm{s}(\vartheta_1)\mathrm{c}(\vartheta_3) + \mathrm{c}(\vartheta_2)\mathrm{c}(\vartheta_1)\mathrm{s}(\vartheta_3) & -\mathrm{s}(\vartheta_1)\mathrm{s}(\vartheta_3) + \mathrm{c}(\vartheta_2)\mathrm{c}(\vartheta_1)\mathrm{c}(\vartheta_3) & -\mathrm{c}(\vartheta_1)\mathrm{s}(\vartheta_2) \\ \mathrm{s}(\vartheta_3)\mathrm{s}(\vartheta_2) & \mathrm{c}(\vartheta_3)\mathrm{s}(\vartheta_2) & \mathrm{c}(\vartheta_2) \end{pmatrix} \tag{2.12}$$

A subgroup of SO(3) is SO(2) which can be interpreted as all the rotations around the z-axis. The restriction of $D^{(3)}$ to SO(2) is achieved by fixing $\vartheta_2 = 0$ and $\vartheta_1 = 0$. $D^{(3)}$ then has the following form:

$$D^{(3)}(\vartheta_3) = \begin{pmatrix} \mathrm{c}(\vartheta_3) & -\mathrm{s}(\vartheta_3) & 0 \\ \mathrm{s}(\vartheta_3) & \mathrm{c}(\vartheta_3) & 0 \\ 0 & 0 & 1 \end{pmatrix} \tag{2.13}$$

Obviously since the matrix now is of block-diagonal form it is no irreducible representation any more. Over the complex numbers this representation can even by completely diagonalized like it was shown in chapter 2.5.4:

$$D^{(3)}(\vartheta_3) = \begin{pmatrix} e^{-i\vartheta_3} & 0 & 0 \\ 0 & e^{i\vartheta_3} & 0 \\ 0 & 0 & 1 \end{pmatrix} \tag{2.14}$$

These three irreps of SO(2) can be labelled by m, the prefactor of the angle in the exponent. This in turn means that $D^{(3)}$ contains the irreps with labels $m = -1, 0, 1$. By restriction to SO(2), $D^{(3)}$ is now written as a sum of 1-dimensional irreps of SO(2), which reflects the fact that SO(2) is an abelian group. This makes it possible to label the basis states of the vector space $\mathcal{V}$ that $D^{(3)}$ is acting on by the irrep label m of SO(2). This labelling of basis states will be used later.

2.6. The Lie-Algebra

To explain the concept of a Lie-Algebra and the generators that this algebra is spanned by, it is very instructive to first look at a specific example. For that again the matrix (2.4) representing a rotation in the plane by an angle ϑ is chosen. The basic idea behind generators and their connection to a specific group is to split up the total rotation by ϑ into many small rotations that are executed one after another. If the number of small rotations approaches infinity, the small angle $\Delta\vartheta$ gets infinitesimally small.

2.6.1. Ex.: SO(2)

Following this approach a representation of an infinitesimal rotation in 2 dimension is needed. For that $\cos(\vartheta)$ and $\sin(\vartheta)$ get written as a Taylor series and all factors of order $\Delta\vartheta^2$ or higher are omitted :

$$\begin{pmatrix} \cos(\Delta\vartheta) & \sin(\Delta\vartheta) \\ -\sin(\Delta\vartheta) & \cos(\Delta\vartheta) \end{pmatrix} = \begin{pmatrix} 1 & \Delta\vartheta \\ -\Delta\vartheta & 1 \end{pmatrix} = \underbrace{\begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix}}_{I} + i\,\Delta\vartheta \underbrace{\begin{pmatrix} 0 & -i \\ i & 0 \end{pmatrix}}_{\mathcal{J}} \tag{2.15}$$

So in summary a group element of $SO(2)$ infinitesimally close to the identity is given by $R(\Delta\vartheta) = I + i\,\Delta\vartheta\mathcal{J}$, where $\mathcal{J}$ is known as the generator of $SO(2)$. A factor of i is factored out in the last step just by convention. Like described above a group element with a large angle ϑ is obtained by executing the infinitesimal rotation infinitely often. Strictly speaking the following limit is taken:

$$R(\vartheta) = \lim_{N\to\infty} \left[R\left(\Delta\vartheta\right)\right]^N = \lim_{N\to\infty} \left[I + i\Delta\vartheta\mathcal{J}\right]^N$$

By choosing the infinitesimal angle $\Delta\vartheta$ to be $\frac{\vartheta}{N}$ it is ensured that the total rotation angle is indeed ϑ. Together with the identity $e^x = \lim_{N\to\infty}(1 + \frac{x}{N})^N$ [Zee16] the following important relation is obtained:

$$R(\vartheta) = e^{i\vartheta\mathcal{J}} \tag{2.16}$$

2.6.2. Manifolds and tangent spaces

There are of course constrains to when the construction of a group by the exponential map as presented above works. Firstly the group of course has to be continuous, otherwise it wouldn't be possible to Taylor-expand a group element. But for a group element to be expressed as in (2.16), the group also has to fulfill the topological and analytical criterion of being a so called differentiable manifold. A manifold is informally a topological space that locally looks like $\mathbb{R}^n$. A strict

mathematical definition won't be presented here, but can be found e.g. in [Lug21] or [Sch80]. The group operation can then be seen as a function that takes two inputs from the manifold and maps them onto another point on the same manifold, as required by group closure. In a Lie-group this function has to be differentiable on the whole manifold. The mathematical details are subject to the field of differential geometry, and won't be part of this work. For further information see e.g. [Lug21] or [RS22].
So in the previous paragraph it was established that a Lie-group can be seen as a point on a multi-dimensional smooth generalized surface. The *dimension* of the Lie-group is the dimension of the manifold, or phrased differently the number of parameters needed to specify any group element. Furthermore in chapter 2.6.1 it was shown how the group can be expressed as a exponential map of a specific set of matrices. But how do these two ideas can be connected? To understand this relationship, let's remember that we Taylor-expanded a group element infinitesimally close to the identity. This expansion can be generalized to any Lie-group using the notion of a tangent space. To construct this tangent space at any point x on the manifold, one has to look at curves γ on the manifold, parameterized by $t \in \mathbb{R}$, that pass through x at $t = 0$. The tangent vector a of this curve γ at x then is part of the tangent space at the point x. a is obtained by taking the derivative of the curve with respect to the parameter t and evaluating this derivative at $t = 0$. The details of how to take this derivative are again spared here and can be found in [Lug21], [Sch80] or [RS22].
In the figure below two examples of smooth manifolds together with the tangent space at a point x are shown.

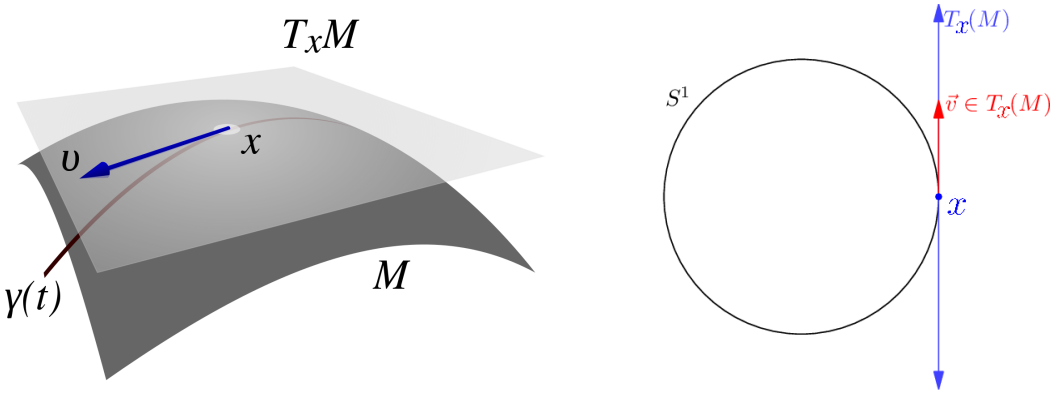


Figure 2.2.: tangent space at point x to an unspecified manifold (left), and to S^1 the manifold of SO(2) (right). [TN08][Che15]

On the right hand side of the figure above the manifold of SO(2), the so called 1-sphere which is the 1-dimensional boundary of a 2-dimensional circle is shown. The fact that it is 1-dimensional reflects the fact that one needs only one parameter, namely ϑ to specify a group element. The 1-d tangent space at a point x is also shown and denoted by $T_x(M)$.
For one- and two-dimensional manifolds it is often easy to visualize the manifold together with it's tangent space, for groups of higher dimension this becomes almost impossible.

With all this basic differential geometry handled, it is now possible to state the following:
The associated Lie-algebra $\mathfrak{g}$ to any given Lie-group G is the tangent space at the identity. With this Lie-algebra it is possible to reconstruct a Lie-group by the exponential map, similarly to the example of $SO(2)$ presented above (see [Jev11] or [Lug21] for a proof of this). Because of this, it is instructive and often easier to investigate the group indirectly by looking at properties of the associated algebra $\mathfrak{g}$. It turns out that $\mathfrak{g}$ is real a vector space that is also closed under the *commutator*[3]: $[A, B] = AB - BA$ [Jev11]. This vector space has the same dimension as the underlying manifold and the basis-elements of it are called *generators* often denoted by T_a. This means every element A of the Lie-algebra can be written as a linear combination of these generators with a factor of i pulled out again by convention to be explained later.

$$A(\vartheta_1, \vartheta_2, ...) = i \sum_{a=1}^{\dim(g)} \vartheta_a T_a \tag{2.17}$$

This allows any group element $g(\vartheta_1, \vartheta_2, ...)$ to be written in terms of the generalized angles ϑ_a and the generators T_a with the exponential map:

$$g(\vartheta_1, \vartheta_2, ...) = \exp\left(i \sum_{a=1}^{\dim(\mathfrak{g})} \vartheta_a T_a \right) \tag{2.18}$$

This is clearly a generalization of the example of SO(2) shown in (2.16).

2.6.3. Commutation relation of the Lie-algebra

It was already mentioned above, that the Lie-algebra $\mathfrak{g}$ is not only a vector space, but also has another operation that it is closed under, namely the commutator for matrix Lie groups or the Lie-bracket $[A, B]$ for abstract Lie-groups. To understand the special role that this commutator plays, the product of two group elements g_A, g_B is expressed through the exponential map of elements $A, B \in \mathfrak{g}$. Here the so called ***Baker-Campbell-Hausdorff-formula*** [Zee16] is used:

$$\exp(A) \cdot \exp(B) = \exp\left(A + B + \frac{1}{2}[A, B] + \frac{1}{12}[A, [A, B]] + \dots \right) \tag{2.19}$$

Here "..." stands for an infinitely long sum of nested commutators. The closure of the algebra under the commutator ensures, that the product of two group elements can also be expressed through the exponential map of a Lie-algebra element. The commutator is therefore the natural composition inside $\mathfrak{g}$, that contains information about how the composition on the associated group elements

[3] more general if one defines an abstract bilinear, antisymmetric operation, that fulfills the Jacobi-identity, the Lie-algebra is closed under it. The commutator of two matrices fulfills these properties, so it is a specific example of this Lie-bracket when dealing with matrix Lie-groups like it is mosten often the case in physics

looks like. Furthermore, since $\mathfrak{g}$ is a vector space it is sufficient to evaluate the commutator on two arbitrary basis elements T_a, T_b. The result $[T_a, T_b]$ can always be written as a linear combination of the generators T_c since these generators form a complete set of basis vectors:

$$[T_a, T_b] = i \sum_{c=1}^{\dim(\mathfrak{g})} f_{abc} T_c \tag{2.20}$$

The coefficients f_{abc} are called *structure constants*. They determine the structure of $\mathfrak{g}$ under the commutator which in turn fixes the group multiplication law as described above.[Zee16]

2.6.4. Generalization to SO(3) and SO(N)

In the case of SO(2) the exact form of the Lie-algebra was quite simple because there is only one generator namely $\mathcal{J}$. Looking back at the definition of SO(N) as the set of real orthogonal matrices of size N that have a determinant of 1, the associated Lie-algebra $\mathfrak{so}$(N) can be constructed in the following way. First a member of SO(N) infinitesimally close to the identity is written as $R \simeq I + A$, where A is a member of $\mathfrak{so}$(N). Then by omitting factors of A^2 or higher the condition $R^T R = I$ becomes:

$$\begin{aligned} I \simeq (I + A)^T (I + A) \simeq (I + (A^T + A)) \\ \Rightarrow A^T = -A \end{aligned} \tag{2.21}$$

So any element A of $\mathfrak{so}$(N) has to be an antisymmetric matrix. Furthermore we can express A as a linear combination of generators T_a as in (2.17).
The convention of pulling out a factor of i introduced earlier results in the requirement that for T_a to span $\mathfrak{so}$(N) it has to be a complex hermitean N $\otimes$ N matrix. This conventional usage of complex hermitean matrices instead of real antisymmetric ones as generators, comes from the fact that hermitean matrices, because of their real expectation values play a crucial role in quantum mechanics [Sch13]. So in order to find the generators of SO(N) it is necessary to find a basis for complex hermitean N $\otimes$ N matrices. For $N = 3$ this basis consists of the following three matrices:

$$\mathcal{J}_x = \begin{pmatrix} 0 & 0 & 0 \\ 0 & 0 & -i \\ 0 & i & 0 \end{pmatrix} \qquad \mathcal{J}_y = \begin{pmatrix} 0 & 0 & i \\ 0 & 0 & 0 \\ -i & 0 & 0 \end{pmatrix} \qquad \mathcal{J}_z = \begin{pmatrix} 0 & -i & 0 \\ i & 0 & 0 \\ 0 & 0 & 0 \end{pmatrix} \tag{2.22}$$

A general antisymmetric 3x3 matrix A, which as stated before is part of the Lie-algebra $\mathfrak{so}$(N), can therefore be written as $A = i\left(\vartheta_x \mathcal{J}_x + \vartheta_y \mathcal{J}_y + \vartheta_z \mathcal{J}_z\right)$, which allows a generic group element of

SO(3) to be constructed in the following way::

$$R(\vartheta_n) = \exp\left(i \sum_n \vartheta_n \mathcal{J}_n\right) \qquad \text{with } n = x, y, z \tag{2.23}$$

For higher dimensions the construction of the generators can be easily generalised. Each element above the diagonal can be chosen to be i or $-i$, which fixes the rest of the matrix by the hermiticity requirement. So the number of generators, also known as the dimension of the associated algebra $\mathfrak{so}$(N), is given by the elements of a N $\otimes$ N matrix above the diagonal:

$$\dim(\mathfrak{so}(N)) = \frac{N^2 - N}{2} = \frac{N(N-1)}{2} \tag{2.24}$$

An element of SO(N), depending on $\frac{N(N-1)}{2}$ generalized angles, can then be written in the same way as in (2.23) with n ranging from 0 to $\frac{N(N-1)}{2}$.

2.6.5. Lie-Algebra representations

In general the Lie-Algebra is an abstract vector space with the previously mentioned composition $[A, B]$, the so called Lie-bracket. Physicist however often do not clearly distinguish between the abstract elements of the algebra and their representations which are a set of $d \otimes d$ matrices. This is the reason why in the previous chapter the Lie-algebra was introduced as a set of matrices, with the Lie-bracket realised by the matrix commutator. An additional reason for this is the fact that many groups like GL(N), SO(N) or SU(N) are defined as a set of matrices with some extra condition. Analogous to group representations, a Lie-Algebra representations $D_\mathfrak{g}$ is a homomorphic map into $GL(\mathbb{R}^n/\mathbb{C}^n)$, here this map has to preserve the Lie-bracket:

$$[D_\mathfrak{g}(g_i), D_\mathfrak{g}(g_j)] = D_\mathfrak{g}([g_i, g_j]) \tag{2.25}$$

Via the exponential map, this representation of the Lie-algebra directly delivers a representation of the associated Lie-group. From this, most of the concepts from the theory of group representations, presented in chapter 2.5 can be transferred to Lie-Algebra representations. For example the concepts of faithfulness and equivalence or irreducibility of a representation of a Lie-Algebra can be adopted. A Lie-Algebra representation can also be restricted to a subalgebra similarly as it was described in chapter 2.5.5 for group representations.

2.7. Tensor product of representations and coupling coefficients

In the following chapter the concept of tensor products of representations shall be established. Two reasons for the importance of this product representations are presented in *An Introduction to Tensors and Group Theory for Physicists* by Jevanjee [Jev11]:

> The tensor product of representations is important for several reasons: First, almost all representations of interest can be viewed as tensor product of other, more basic representations. Second, tensor products are ubiquitous in quantum mechanics (since they represent the addition of degrees of freedom), so we better know how they interact with representations.

An example for the second mentioned reason is the addition of angular momentum in quantum mechanics, the mathematics of which can be described exactly by tensor products of representations of the symmetry group SO(3).
These reasons should give enough motivation to investigate product representations further.

2.7.1. Definition and Properties

The basic idea behind the tensor product is to create a new vector space $\mathcal{V} \otimes \mathcal{W}$ from to given vector spaces $\mathcal{V}$, $\mathcal{W}$. This new vector space contains product vectors $v \otimes w \in \mathcal{V} \otimes \mathcal{W}$ with $v \in \mathcal{V}$ and $w \in \mathcal{W}$. For the operation $\otimes$ to be called a product, it should be bilinear and associative [Jev11]:

$$\begin{aligned} (v_1 + v_2) \otimes w &= v_1 \otimes w + v_2 \otimes w \\ v \otimes (w_1 + w_2) &= v \otimes w_1 + v \otimes w_2 \\ c \cdot (v \otimes w) = (c \cdot v) \otimes w &= v \otimes (c \cdot w), \quad c \in \mathbb{R}/\mathbb{C} \\ (v_1 \otimes v_2) \otimes v_3 &= v_1 \otimes (v_2 \otimes v_3) \end{aligned} \tag{2.26}$$

Basis

A basis for $\mathcal{V} \otimes \mathcal{W}$ can be obtained by taking the tensor product of the basis vectors of the underlying vector spaces. To be precise let $\mathcal{V}$ be a d_1-dimensional vector space with basis $\{e_i\}$ and $\mathcal{W}$ be d_2-dimensional with basis $\{f_j\}$, then $\{e_i \otimes f_j\}$ forms a basis of $\mathcal{V} \otimes \mathcal{W}$ with dimension $d = d_1 \cdot d_2$.

Operators

Using two linear operators A_1 and A_2 acting on vector spaces $\mathcal{V}$ and $\mathcal{W}$ respectively it is possible to create a new linear operator $A_1 \otimes A_2$ acting on the product space:

$$(A_1 \otimes B_1)(v \otimes w) \equiv (A_1 v) \otimes (B_1 w) \tag{2.27}$$

with multiplication law:

$$(A_1 \otimes B_1)(A_2 \otimes B_2) = (A_1 A_2) \otimes (B_1 B_2) \tag{2.28}$$

A proof that the operator defined in this way is unique and well defined and the multiplication law holds as claimed, can be found in *An Elementary Introduction to Groups and Representations* by Hall [Hal00].
It is important to mention that not every linear operator on the product space $\mathcal{V} \otimes \mathcal{W}$ can be written as a product of linear operators acting on $\mathcal{V}$ and $\mathcal{W}$.
In the next chapter the tensor product of operators will be used to create a new group representations from two known representations.

2.7.2. Tensor product representations

Recalling that group elements g of a given Lie-group G can be represented by linear operators $D(g)$ acting on a vector space $\mathcal{V}$, one might suggest that it is possible to create new representations of the same group by the previously described tensor product. In fact it is easy to check that given two representations $D^{(d_1)}$ and $D^{(d_2)}$ of dimensions d_1 and d_2, one obtains a new $(d_1 \cdot d_2)$-dimensional representation $D^{(d_1 \cdot d_2)} = D^{(d_1)} \otimes D^{(d_2)}$ of G via the tensor product. The representation condition (2.5) can be checked using the tensor multiplication law (2.28) as well as the fact that $D^{(d_1)}$ and $D^{(d_2)}$ are representations:

$$\begin{aligned} D^{(d_1 \cdot d_2)}(g_1) \cdot D^{(d_1 \cdot d_2)}(g_2) &= \left(D^{(d_1)}(g_1) \otimes D^{(d_2)}(g_1)\right) \cdot \left(D^{(d_1)}(g_2) \otimes D^{(d_2)}(g_2)\right) \\ &= \left((D^{(d_1)}(g_1) \cdot D^{(d_1)}(g_2)\right) \otimes \left((D^{(d_2)}(g_1) \cdot D^{(d_2)}(g_2)\right) \\ &= \left((D^{(d_1)}(g_1 \circ g_2)\right) \otimes \left((D^{(d_2)}(g_1 \circ g_2)\right) = D^{(d_1 \cdot d_2)}(g_1 \circ g_2) \end{aligned} \tag{2.29}$$

The creation of new representations by two smaller ones was already described in section 2.5.4 where the direct sum was discussed. There it was also described that the direct sum of two irreducible representation is reducible by construction. For the direct product the situation isn't so simply anymore. In general the tensor product representation of two irreps is no irrep anymore, which means it can be written as a direct sum of smaller representations; the remaining work consists of determine these summands, together with the so called multiplicity m_{d_i}, the number describing how often each irrep is contained, and the change of basis matrix S performing the block diagonalization of $D^{(d \cdot d')}(g)$.

$$D^{(d)}(g) \otimes D^{(d')}(g) = S^{-1} \cdot \begin{pmatrix} D^{(d_1)}(g) & 0 & \dots & 0 \\ 0 & D^{(d_2)}(g) & \dots & 0 \\ \vdots & \vdots & \ddots & 0 \\ 0 & 0 & 0 & D^{(d_m)}(g) \end{pmatrix} \cdot S \tag{2.30}$$

The d_i in the block-diagonal matrix label the irreducible representation and specify the dimension of the irrep. Since the total dimension of the tensor product representation is also given by the product of the two dimension d, d' the following identity can be stated:

$$d \cdot d' = \sum_i m_{d_i} d_i \tag{2.31}$$

Like mentioned before a single irrep can appear more than once in which case the multiplicity of that particular irrep is greater than 1.

Lie-algebra product representations

The corresponding algebra representation of a Lie-algebra element $A \in \mathfrak{g}$ to the product representation $D^{(d)} \otimes D^{(d')}$ of the associated Lie-group G is given by the following expression:

$$\left(D_{\mathfrak{g}}^{(d)} \otimes D_{\mathfrak{g}}^{(d')}\right)(A) = D_{\mathfrak{g}}^{(d)}(A) \otimes \mathbb{1}_{d'} + \mathbb{1}_d \otimes D_{\mathfrak{g}}^{(d)}(A) \tag{2.32}$$

A proof for this can be found in [Jev11].

2.7.3. Coupling-coefficients

The above mentioned block-diagonalization of the tensor product representation of a given Lie-group G is called *Clebsch-Gordon expansion*. In this expansion the productspace $\mathcal{V}^{(d)} \otimes \mathcal{V}^{(d')}$ gets decomposed into irreducible subspaces $\mathcal{V}^{(d_i)}$ on which the irreps $D^{(d_i)}$ act on. An analogous formulation of this decomposition described in equation (2.30) is the following [Böh11]:

$$D^{(d)}(g) \otimes D^{(d')}(g) \cong \bigoplus_{d_i} (d\, d' | d_i)\, D^{(d_i)}(g) \tag{2.33}$$

In this equation the sum runs over each irrep label d_i; the multiplicity m_{d_i} of each irrep d_i in the product expansion is written as $(d\, d'|d_i)$.
There are now two equivalent forms of the product representation $D^{(d)} \otimes D^{(d')}$, one in the natural product basis $\left\{e_i^d \otimes e_j^{d'} | i = 1, \ldots, d;\ j = 1, \ldots, d'\right\}$, the other one in the reduced basis $\left\{u_k^{(d_i,\rho)} | k = 1, \ldots, d_i;\ \rho = 1, \ldots, m_{d_i};\ d_i = d_1, \ldots, d_m\right\}$ where the product representation is block-diagonal. This basis is labelled by the irrep label d_i, with the irrep $D^{(d_i)}$ acting on the invariant subspace $\mathcal{V}^{(d_i)}$, the multiplicity label s of this irrep and the label j of the basis of $\mathcal{V}^{(d_i)}$.
The matrix S described in equation (2.30) transforms between these two bases

$$u_k^{(d_i,\rho)} = S \cdot \left(e^d \otimes e^{d'}\right) = \sum_i \sum_j \left(\begin{array}{cc|cc} d & d' & d_i & \rho \\ i & j & k & \end{array}\right) e_i^d \otimes e_j^{d'} \tag{2.34}$$

with the elements of S being called *coupling-coefficients*:

$$C^{d,\,d',\,d_i}_{i\,j,k\,\rho} = \left(\begin{array}{cc|cc} d & d' & d_i & \rho \\ i & j & k & \end{array}\right) \tag{2.35}$$

If both bases are orthonormal, the change of basis matrix S is unitary, $S^\dagger = S^{-1}$, which allows the following formulation of the inverse transformation of (2.34). In it the basis elements in the product space are expressed in terms of the basis elements of irreducible representations[Böh11]:

$$e^d_i \otimes e^{d'}_j = \sum_{d_i} \sum_{\rho} \sum_{k} \left(\begin{array}{cc|cc} d & d' & d_i & \rho \\ i & j & k & \end{array}\right)^* u^{(d_i,\rho)}_k \tag{2.36}$$

One final thing to mention is that the coupling-coefficients can only be determined up to a phasefactor $\exp(it)$ with $t \in \mathbb{R}$.

3. Computation of coupling coefficients

The following chapter follows closely the paper "Racah's method for general subalgebra chains: Coupling coefficients of SO(5) in canonical and physical bases" by Caprio, Sviratcheva, and McCoy [CSM10]. In it, a method for the computation of the previously introduced coupling coefficients is described. Furthermore an implementation of this method in python shall be presented.

3.1. Method

The method aims at computing the coupling coefficients that arise when creating a tensor product representations $D^{(d)} \otimes D^{(d)}$ of two irreps $D^{(d)}$ and $D^{(d')}$ of a Lie-group G. These coupling coefficients describe the transformation between the product basis and the reduced basis as shown in chapter 2.7.3. In [CSM10] the following alternative version of (2.34) is used:

$$\left| \begin{array}{c} \Gamma_1 \ \Gamma_2 \\ \rho\Gamma \\ a\Lambda \\ \lambda \end{array} \right\rangle = \sum_{\substack{a_1\Lambda_1 \ a_2\Lambda_2 \\ \lambda_1 \quad \lambda_2}} \left(\begin{array}{cc|c} \Gamma_1 & \Gamma_2 & \rho\Gamma \\ a_1\Lambda_1 & a_2\Lambda_2 & a\Lambda \\ \lambda_1 & \lambda_2 & \lambda \end{array} \right) \left| \begin{array}{cc} \Gamma_1 & \Gamma_2 \\ a_1\Lambda_1 & a_2\Lambda_2 \\ \lambda_1 & \lambda_2 \end{array} \right\rangle \tag{3.1}$$

In contrast to eq. (2.34) the basis elements of each invariant subspace are written labelled through irrep labels Λ of a subalgebra H together with labels λ that label basis states in each invariant subspace that D^{Λ} acts on. This labelling was explained in chapter 2.5.5 with the example of $G = \mathrm{SO}(3)$ and $H = \mathrm{SO}(2)$. There the basis states of $\mathbb{R}^3$ that the irrep $D^{(\Gamma=3)}$ of SO(3) acts on, where labelled by the irrep label $\Lambda = m$ of SO(2). Because SO(2) is an abelian group, it's irreps are all one-dimensional and no additional label λ is needed there to distinguish the basis states in each invariant subspace. This changes when the considered subalgebra H is non-abelian.
Besides the irrep labels Γ and Λ in eq. (3.1) two multiplicity labels ρ and a are introduced. The outer multiplicity ρ for $G \otimes G \to G$ was already seen in (2.34) and comes into play when in the Clebsch-Gordon decomposition some irreps appear multiple times. The inner multiplicity a resolves the branching for $G \to H$. This is needed when by restriction of an irrep Γ of G to H, an irrep Λ of H emerges more than once. Outer multiplicity of H for $H \otimes H \to H$ shall be assumed to be non present.
The search for the coupling coefficients $\left(\begin{array}{cc|c} \Gamma_1 & \Gamma_2 & \rho\Gamma \\ a_1\Lambda_1 & a_2\Lambda_2 & a\Lambda \\ \lambda_1 & \lambda_2 & \lambda \end{array} \right)$ can be strongly simplified by Racah's

factorization Lemma [Rac49]:

$$\left(\begin{array}{cc|c} \Gamma_1 & \Gamma_2 & \rho\Gamma \\ a_1\Lambda_1 & a_2\Lambda_2 & a\Lambda \\ \lambda_1 & \lambda_2 & \lambda \end{array}\right) = \left(\begin{array}{cc|c} \Lambda_1 & \Lambda_2 & \Lambda \\ \lambda_1 & \lambda_2 & \lambda \end{array}\right) \cdot \left(\begin{array}{cc|c} \Gamma_1 & \Gamma_2 & \rho\Gamma \\ a_1\Lambda_1 & a_2\Lambda_2 & a\Lambda \end{array}\right) \tag{3.2}$$

With this the coupling coefficients of interest can be expressed as a product of a coupling coefficient of H and a *reduced coupling coefficient* that doesn't depend on the λ-labels anymore. For the following method to work, the coupling coefficients for the subalgebra H that embody all the λ-dependence have to be known beforehand.
In the paper [CSM10] the authors start with a bra-ket version of eq.(2.32) for a generator $T^{\Lambda_T}_{\lambda_T}$ of G:

$$\begin{aligned}\left\langle \begin{array}{cc} \Gamma_1 & \Gamma_2 \\ a_1\Lambda_1 & a_2\Lambda_2 \\ \lambda_1 & \lambda_2 \end{array} \middle| T^{\Lambda_T}_{\lambda_T} \middle| \begin{array}{c} \Gamma_1\Gamma_2 \\ \rho\Gamma \\ a\Lambda \\ \lambda \end{array} \right\rangle &= \left\langle \begin{array}{cc} \Gamma_1 & \Gamma_2 \\ a_1\Lambda_1 & a_2\Lambda_2 \\ \lambda_1 & \lambda_2 \end{array} \middle| T^{\Lambda_T\ (1)}_{\lambda_T} \middle| \begin{array}{c} \Gamma_1\Gamma_2 \\ \rho\Gamma \\ a\Lambda \\ \lambda \end{array} \right\rangle \\ &\quad + \left\langle \begin{array}{cc} \Gamma_1 & \Gamma_2 \\ a_1\Lambda_1 & a_2\Lambda_2 \\ \lambda_1 & \lambda_2 \end{array} \middle| T^{\Lambda_T\ (2)}_{\lambda_T} \middle| \begin{array}{c} \Gamma_1\Gamma_2 \\ \rho\Gamma \\ a\Lambda \\ \lambda \end{array} \right\rangle \end{aligned} \tag{3.3}$$

The factorization lemma together with the Wigner-Eckart Theorem[1] is then used to reformulate eq.(3.3) and receive the following equation where only reduced coupling coefficients, reduced matrix elements and recoupling coefficients of H appear:

$$\begin{aligned} &\sum_a \left\langle \begin{array}{c} \Gamma \\ a\Lambda \end{array} \middle\| T^{\Lambda_T} \middle\| \begin{array}{c} \Gamma \\ a'\Lambda' \end{array} \right\rangle \left(\begin{array}{cc|c} \Gamma_1 & \Gamma_2 & \rho\Gamma \\ a_1\Lambda_1 & a_2\Lambda_2 & a\Lambda \end{array}\right) = \\ &\sum_{a'_1\Lambda'_1} \Phi\left(\Lambda_1\Lambda_2;\Lambda\right)\Phi\left(\Lambda'_1\Lambda_2;\Lambda'\right) \begin{bmatrix} \Lambda_2 & \Lambda'_1 & \Lambda' \\ \Lambda_T & \Lambda & \Lambda_1 \end{bmatrix} \left\langle \begin{array}{c} \Gamma_1 \\ a_1\Lambda_1 \end{array} \middle\| T^{\Lambda_T} \middle\| \begin{array}{c} \Gamma_1 \\ a'_1\Lambda'_1 \end{array} \right\rangle \left(\begin{array}{cc|c} \Gamma_1 & \Gamma_2 & \rho\Gamma \\ a'_1\Lambda'_1 & a_2\Lambda_2 & a'\Lambda' \end{array}\right) \\ &\quad + \sum_{a'_2\Lambda'_2} \begin{bmatrix} \Lambda_1 & \Lambda'_2 & \Lambda' \\ \Lambda_T & \Lambda & \Lambda_2 \end{bmatrix} \left\langle \begin{array}{c} \Gamma_2 \\ a_2\Lambda_2 \end{array} \middle\| T^{\Lambda_T} \middle\| \begin{array}{c} \Gamma_2 \\ a'_2\Lambda'_2 \end{array} \right\rangle \left(\begin{array}{cc|c} \Gamma_1 & \Gamma_2 & \rho\Gamma \\ a_1\Lambda_1 & a'_2\Lambda'_2 & a'\Lambda' \end{array}\right) \end{aligned} \tag{3.4}$$

[1] This theorem states that general matrix elements of irreducible tensor operators can be expressed as products of coupling coefficients and *reduced matrix elements*. See [CSM10], [Böh11] or [Zee16] for further information

For each choice of $a_1\Lambda_1, a_2\Lambda_2, \Lambda$ and $a'\Lambda'$ one obtains a different relation between reduced coupling coefficients using (3.4). If there are N possible choices for these four labels, a linear, homogeneous system of equations in N unknowns can be formulated in the following way[CSM10]:

$$\underbrace{\begin{bmatrix} \vdots \\ \cdots\ a_{ki}\ \cdots \\ \vdots \end{bmatrix}}_{\equiv A} \begin{bmatrix} C_1 \\ \vdots \\ C_N \end{bmatrix} = \begin{bmatrix} 0 \\ \vdots \\ 0 \end{bmatrix} \tag{3.5}$$

Coupling coefficient $\xrightarrow{(a_1\Lambda_1 a_2\Lambda_2 a\Lambda)}$

Racah relation $\downarrow_{(a_1\Lambda_1 a_2\Lambda_2 \Lambda a'\Lambda')}$

With this, the computation of the coupling coefficients for $\Gamma_1 \otimes \Gamma_2 \to \rho\Gamma$ comes down to creating the matrix A using (3.4) and determining the null space of it. This null space is one-dimensional in the case where $\Gamma_1 \otimes \Gamma_2 \to \Gamma$ is multiplicity free. The proper normalization of a vector $\vec{C^a}$ in the nullspace can be achieved by dividing $\vec{C^a}$ by the length $\mathcal{N} = \sqrt{\vec{C^a} \cdot \vec{C^a}}$ with the special inner product:

$$\vec{C^a} \cdot \vec{C^b} \equiv \sum_{\substack{i \\ (\text{same } a\Lambda)}} C_i^a \cdot C_i^b \tag{3.6}$$

Additionally to the normalization an overall phase for every $\vec{C^a}$ can be freely chosen.
If there is an outer multiplicity present, the null-space might be higher dimensional. In this case one has to perform a Gram-Schmidt procedure, with the inner product defined in (3.6), to obtain a orthonormal basis for the null-space.
This basis is only determined only to within a unitary transformation. This will be important later when comparing the coupling coefficients from an example in [CSM10] to the coefficients calculated by the implementation in python presented in section 3.2.

3.1.1. Ex.: SO(5) coupling coefficients

A special attention in the paper [CSM10] is paid to the algebra of the special orthogonal group $\mathfrak{so}(5)$ with it's subalgebra $\mathfrak{so}(4) \sim \mathfrak{so}(3) \otimes \mathfrak{so}(3)$. The computation of the coupling coefficients for $\mathfrak{so}(5)$ is demonstrated and a few examples for the coupling of specific irreps are shown. The python implementation in section 3.2 allows the calculation of $\mathfrak{so}(5)$ coupling coefficients.

Generators

The Lie algebra $\mathfrak{so}(5)$ is spanned by 2 independent $\mathfrak{so}(3)$ algebras in addition with 4 generators that connect these subalgebras. This results in the following expression of $\mathfrak{so}(5)$ as the linear span $\mathcal{L}$ of a total of 10 generators:

$$\mathfrak{so}(5) = \mathcal{L}\left(X_1, X_2, X_3, Y_1, Y_2, Y_3, T_{++}, T_{+-}, T_{-+}, T_{--}\right) \tag{3.7}$$

The generators X_i, Y_i fulfill the usual $\mathfrak{so}(3)$ commuation relations, the remaining commutation relations can be found in [CSM10].

Irrep labels

A $\mathfrak{so}(5)$-irrep is labelled by a tuble $\Gamma = (R\,S)$ of two numbers with values $S = 0, \frac{1}{2}, \ldots$ and $R = S, S + \frac{1}{2}, \ldots$. Similarly an irrep of $\mathfrak{so}(4)$ is also labelled by a 2-tuple $\Lambda = (XY)$ with $X = 0, \frac{1}{2}, \ldots$ and $Y = 0, \frac{1}{2}, \ldots$ being $\mathfrak{so}(3)$ irrep labels. Since this subalgebra is not abelian the additional labels $\lambda = (M_X\, M_Y)$ with $M_X = -X, \ldots, X-1, X$ and $M_Y = -Y, \ldots, Y-1, Y$ are needed to label the basis states. The Clebsch-Gordon Expansion for the inner tensor product of two $\mathfrak{so}(4)$-irreps: $(X_1\, Y_1) \otimes (X_2\, Y_2) \to (X\, Y)$ produces no outer multiplicities. $(X\, Y)$ is contained if $X \in \{|X_1 - X_2|, |X_1 - X_2| + 1, \ldots, X_1 + X_2\}$ and $Y \in \{|Y_1 - Y_2|, |Y_1 - Y_2| + 1, \ldots, Y_1 + Y_2\}$. This lack of outer multiplicity was an assumption that was made at the beginning of section 3.1.

Recoupling coefficients

Because of the isomorphism $\mathfrak{so}(4) \sim \mathfrak{so}(3) \otimes \mathfrak{so}(3)$, the recoupling coefficients for $\mathfrak{so}(4)$ can be written as a product of $\mathfrak{so}(3)$ Wigner 6j Symbols:

$$\begin{aligned}\begin{bmatrix} (X_1\, Y_1) & (X_2\, Y_2) & (X_{12}\, Y_{12}) \\ (X_3\, Y_3) & (X\, Y) & (X_{23}\, Y_{23}) \end{bmatrix} &= (-1)^{X_1+X_2+X_3+X}(-1)^{Y1+Y2+Y3+Y}\sqrt{2X_{12}+1} \\ &\cdot\sqrt{2X_{23}+1}\sqrt{2Y_{12}+1}\sqrt{2Y_{23}+1}\begin{Bmatrix} X_1 & X_2 & X_{12} \\ X_3 & X & X_{23} \end{Bmatrix}\begin{Bmatrix} X_1 & X_2 & X_{12} \\ X_3 & X & X_{23} \end{Bmatrix}\end{aligned} \tag{3.8}$$

This Wigner 6j Symbol plays a role in the coupling of 3 angular momenta and can be written as a product of three 3j symbols which in turn are related to the ordinary $\mathfrak{so}(3)$ Clebsch-Gordon coefficients. For further information on this see *Quantum Theory of Angular Momentum* by Varshalovich, Moskalev, and Khersonskii [VMK88].

Phase factor

The phase factor Φ incurred when interchanging columns in the $\mathfrak{so}(4)$ coupling coefficient is given the following expression:

$$\Phi((X_1\,Y_1),(X_2\,Y_2),(X\,Y)) = (-1)^{X_1+X_2-X}(-1)^{Y_1+Y_2-Y} \tag{3.9}$$

$(RS) \to (XY)$ branching

When restricting a $\mathfrak{so}(5)$-irrep (RS) to the subalgebra $\mathfrak{so}(4)$, the $\mathfrak{so}(4)$-irrep $(X\,Y)$ is contained if the following branching conditions are fulfilled:

$$\begin{aligned} X &= R - \frac{n}{2} - \frac{m}{2} \\ Y &= S + \frac{n}{2} - \frac{m}{2} \\ \text{with} \quad 0 \le n \le 2(R-S) \quad &\text{and} \quad 0 \le m \le 2S \end{aligned} \tag{3.10}$$

This branching is of importance because the two sum in eq.(3.4) run over all $\mathfrak{so}(4)$-irreps that are contained in the corresponding $\mathfrak{so}(5)$-irrep.

Reduced matrix elements

The last thing that is needed to handle eq.(3.4) for the special case of $\mathfrak{so}(5)$ are the reduced matrix elements for the subalgebra $\mathfrak{so}(4)$. These matrix elements of the generators that are in $\mathfrak{so}(5)$ but not in $\mathfrak{so}(4)$ are given by the following expressions [CSM10]:

$$\left\langle \begin{matrix}(R\,S)\\(X+\frac{1}{2}\,Y+\frac{1}{2})\end{matrix} \left\| T^{(\frac{1}{2}\,\frac{1}{2})} \right\| \begin{matrix}(R\,S)\\(X\,Y)\end{matrix} \right\rangle = \frac{\left[\begin{matrix}(R+S-X-Y)(R+S+X+Y+3)\\ \cdot(S-R+X+Y+1)(R-S+X+Y+2)\end{matrix}\right]^{1/2}}{2\sqrt{2X+2}\sqrt{2Y+2}}$$

$$\left\langle \begin{matrix}(R\,S)\\(X+\frac{1}{2}\,Y-\frac{1}{2})\end{matrix} \left\| T^{(\frac{1}{2}\,\frac{1}{2})} \right\| \begin{matrix}(R\,S)\\(X\,Y)\end{matrix} \right\rangle = \frac{\left[\begin{matrix}(R+S-X+Y+1)(R+S+X-Y+2)\\ \cdot(R-S-X+Y)(R-S+X-Y+1)\end{matrix}\right]^{1/2}}{2\sqrt{2X+2}\sqrt{2Y}}$$

$$\left\langle \begin{matrix}(R\,S)\\(X-\frac{1}{2}\,Y+\frac{1}{2})\end{matrix} \left\| T^{(\frac{1}{2}\,\frac{1}{2})} \right\| \begin{matrix}(R\,S)\\(X\,Y)\end{matrix} \right\rangle = \frac{\left[\begin{matrix}(R+S-X+Y+2)(R+S+X-Y+1)\\ \cdot(R-S-X+Y+1)(R-S+X-Y)\end{matrix}\right]^{1/2}}{2\sqrt{2X}\sqrt{2Y+2}} \tag{3.11}$$

$$\left\langle \begin{matrix}(R\,S)\\(X-\frac{1}{2}\,Y-\frac{1}{2})\end{matrix} \left\| T^{(\frac{1}{2}\,\frac{1}{2})} \right\| \begin{matrix}(R\,S)\\(X\,Y)\end{matrix} \right\rangle = \frac{-\left[\begin{matrix}(R+S-X-Y+1)(R+S+X+Y+2)\\ \cdot(S-R+X+Y)(R-S+X+Y+1)\end{matrix}\right]^{1/2}}{2\sqrt{2X}\sqrt{2Y}}$$

3.2. Implementation in python

The paper [CSM10] applies the method presented above to calculate the coupling coefficients for the algebra $G = \mathfrak{so}(5)$ with $H = \mathfrak{so}(4)$ as the subalgebra. One of the exercises for this bachelor thesis was to modify the preexisting code by Rasmus Nielsen and include the Gram-Schmidt procedure. For this it was also necessary to programm the special inner product described in eq. (3.6).
I also programmed all the coupling coefficient computation from scratch, with heavy inspiration from the preexisting code by Rasmus. This helped me a lot in understanding the code and the method in general, while also improving my programming skills.
The complete code is attached in A.1. This is the version by Rasmus, with my involvement starting with the function `scalar_product`.

The basic structure of the code:

1. Create a python version of the coupling relation (3.4): `Coupling_Conditions`
   a) Recoupling coefficients: `ReCouplingCoefficients`
   b) phase factors: `PhaseFactor`
   c) reduced matrix elements: `MatrixElements`
2. Collect all Coupling relations in the matrix A from (3.5): `Linear_System_Matrix`
3. Calculate the normalized coupling coefficients: `CouplingCoefficients`
   a) calculate a basis for the nullspace of A
   b) apply the Gram-Schmidt-procedure to this basis: `gram_schmidt`
      i. special inner product (3.6): `scalar_product`

The details of each function mentioned above shall not be explained here. These can be looked at in the code that is attached in section A.1.

3.2.1. Ex.1: 1d-nullspace $\left(\frac{1}{2}\,\frac{1}{2}\right) \otimes \left(\frac{1}{2}\,0\right) \to \left(\frac{1}{2}\,0\right)$

In the paper [CSM10] two specific examples of $\mathfrak{so}(5)$ coupling are presented. With these it is possibel to check whether the code works as expected.
In the first example the following irrep-labels are chosen:
$\Gamma_1 = (R_1\, S_1) = \left(\frac{1}{2}\,\frac{1}{2}\right),\ \Gamma_2 = (R_2\, S_2) = \left(\frac{1}{2}\,0\right),\ \Gamma = (R\, S) = \left(\frac{1}{2}\,0\right)$
Using eq. (3.10) the following branching $\Gamma \to \Lambda$ occurs when restricting $\mathfrak{so}(5)$ to $\mathfrak{so}(4)$:

$$\underbrace{\left(\frac{1}{2}\,\frac{1}{2}\right)}_{\mathfrak{so}(5)-\text{label}} \to \underbrace{\left(\frac{1}{2}\,\frac{1}{2}\right) \oplus \left(0\,0\right)}_{\mathfrak{so}(4)-\text{labels}} \tag{3.12}$$

$$\underbrace{\left(\frac{1}{2}\,0\right)}_{\mathfrak{so}(5)-\text{label}} \to \underbrace{\left(\frac{1}{2}\,0\right) \oplus \left(0\,\frac{1}{2}\right)}_{\mathfrak{so}(4)-\text{labels}} \tag{3.13}$$

Together with the coupling condition of $\mathfrak{so}(4)$, mentioned in section 3.1.1 this results in four coupling coeffcients that need to be determined:

$$c_1 \equiv \left(\begin{array}{cc|c} \left(\frac{1}{2}\,\frac{1}{2}\right) & \left(\frac{1}{2}\,0\right) & \left(\frac{1}{2}\,0\right) \\ (0\,0) & \left(0\,\frac{1}{2}\right) & \left(0\,\frac{1}{2}\right) \end{array}\right) \tag{3.14}$$

$$c_2 \equiv \left(\begin{array}{cc|c} \left(\frac{1}{2}\,\frac{1}{2}\right) & \left(\frac{1}{2}\,0\right) & \left(\frac{1}{2}\,0\right) \\ \left(\frac{1}{2}\,\frac{1}{2}\right) & \left(\frac{1}{2}\,0\right) & \left(0\,\frac{1}{2}\right) \end{array}\right) \tag{3.15}$$

$$c_3 \equiv \left(\begin{array}{cc|c} \left(\frac{1}{2}\,\frac{1}{2}\right) & \left(\frac{1}{2}\,0\right) & \left(\frac{1}{2}\,0\right) \\ (0\,0) & \left(\frac{1}{2}\,0\right) & \left(\frac{1}{2}\,0\right) \end{array}\right) \tag{3.16}$$

$$c_4 \equiv \left(\begin{array}{cc|c} \left(\frac{1}{2}\,\frac{1}{2}\right) & \left(\frac{1}{2}\,0\right) & \left(\frac{1}{2}\,0\right) \\ \left(\frac{1}{2}\,\frac{1}{2}\right) & \left(0\,\frac{1}{2}\right) & \left(\frac{1}{2}\,0\right) \end{array}\right) \tag{3.17}$$

The lower row in each coupling coefficient shows the involved $\mathfrak{so}(4)$-irreps. This triplet of $\mathfrak{so}(4)$-labels also labels the columns of the matrix A from eq. (3.5) as shown in the following figure:

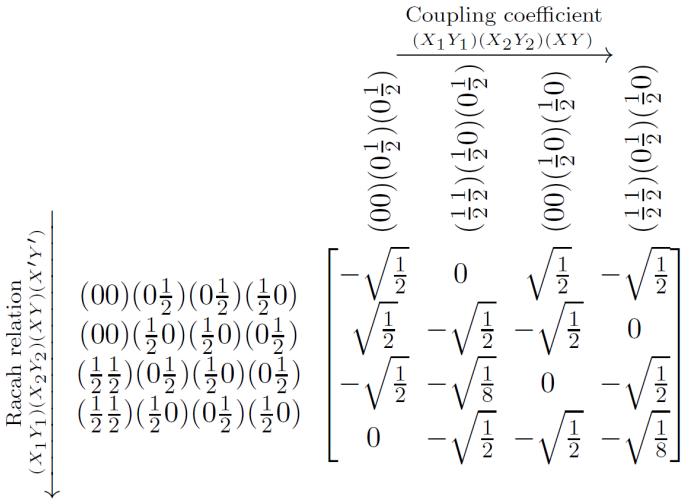


Figure 3.1.: coefficient matrix A used to determine the $\mathfrak{so}(5) \supset \mathfrak{so}(4)$ coupling coefficient for $\left(\frac{1}{2}\,\frac{1}{2}\right) \otimes \left(\frac{1}{2}\,0\right) \to \left(\frac{1}{2}\,0\right)$. [CSM10]

The rows of A correspond to a single coupling relation. They are labelled by four $\mathfrak{so}4$-labels $(X_1\,Y_1), (X_2\,Y_2), (X\,Y), (X'\,Y')$ where the first three labels have to fulfill the same branching/coupling conditions discussed above. The additional label $(X'\,Y')$ has to be connected to $(X\,Y)$ by a generator $T^{\left(\frac{1}{2}\,\frac{1}{2}\right)}$ of $\mathfrak{so}(5)$ that is not contained in $\mathfrak{so}(4)$: $(X'\,Y') \otimes \left(\frac{1}{2}\,\frac{1}{2}\right) \to (X\,Y)$. Only if this condition is fulfilled the matrix elements from 3.1.1 are non-vanishing and a nonnull coupling condition is obtained.
This matrix A is created with the corresponding python function `Linear_System_Matrix`. The order of the column-labels is fixed by `Indexing` and differs from the order used in the paper and shown in fig.3.1. Furthermore the code includes some coupling relations with $\mathfrak{so}(4)$ irreps $\Lambda' = (X'\,Y')$ that are not contained in the $\mathfrak{so}(5)$-irrep $\Gamma = \left(\frac{1}{2}\,0\right)$. This violation of branching conditions $\Gamma \nrightarrow \Lambda'$ is needed in some cases to get enough coupling relations to make sure that the system of equations is solvable.
The only thing left to do is to calculate the nullspace of A using `scipy.linalg.null_space`. Since in this case there is no outer multiplicity ρ present the nullspace is one-dimensional. The last step then is to scale the vector spanning the nullspace, so that it is normalized with respect to the inner product (3.6). The resulting normalized nullvector calculated by the python code is the following

$$\begin{bmatrix} c_1 \\ c_2 \\ c_3 \\ c_4 \end{bmatrix} \approx \begin{bmatrix} -0.447214 \\ -0.894427 \\ 0.447214 \\ 0.894427 \end{bmatrix} \tag{3.18}$$

which matches with the nullvector $\left[-\sqrt{\frac{1}{5}} \; -\sqrt{\frac{4}{5}} \; \sqrt{\frac{1}{5}} \; \sqrt{\frac{4}{5}}\right]$ from the paper [CSM10].

3.2.2. Ex.2: 2d-nullspace $(1\,0) \otimes \left(1\,\frac{1}{2}\right) \to \left(1\,\frac{1}{2}\right)$

In the first example with no outer multiplicity, the coupling coefficients from the python code matched with those from the paper [CSM10] right away. If however there is an outer multiplicity, that is, the irrep Γ in focus appears multiple times in the direct product representation $\Gamma_1 \otimes \Gamma_2$, the situation is not so simple anymore.
In the following section an example with an outer multiplicity of two is shown and the coupling coefficients calculated by the python code are compared to those presented in [CSM10]. For that the following irrep labels are chosen:
$\Gamma_1 = (R_1\, S_1) = (1\,0)\,,\ \Gamma_2 = (R_2\, S_2) = \left(1\,\frac{1}{2}\right)\,,\ \Gamma = (R\, S) = \left(1\,\frac{1}{2}\right)$
The `Indexing` function in the code can be used to see how many coupling coefficients are involved. In this specific example there are 18 of those with each of them labelled by a different combination of $\mathfrak{so}(4)$ labels that fulfill the branching and coupling conditions explained in the previous chapters. The output of the `Indexing` function is shwon in the following figure:

0	list	3	[(1.0, 0.0), (1.0, 0.5), (1.0, 0.5)]
1	list	3	[(1.0, 0.0), (1.0, 0.5), (0.0, 0.5)]
2	list	3	[(1.0, 0.0), (0.5, 0.0), (0.5, 0.0)]
3	list	3	[(1.0, 0.0), (0.5, 1.0), (0.5, 1.0)]
4	list	3	[(1.0, 0.0), (0.0, 0.5), (1.0, 0.5)]
5	list	3	[(0.5, 0.5), (1.0, 0.5), (0.5, 0.0)]
6	list	3	[(0.5, 0.5), (1.0, 0.5), (0.5, 1.0)]
7	list	3	[(0.5, 0.5), (0.5, 0.0), (1.0, 0.5)]
8	list	3	[(0.5, 0.5), (0.5, 0.0), (0.0, 0.5)]
9	list	3	[(0.5, 0.5), (0.5, 1.0), (1.0, 0.5)]
10	list	3	[(0.5, 0.5), (0.5, 1.0), (0.0, 0.5)]
11	list	3	[(0.5, 0.5), (0.0, 0.5), (0.5, 0.0)]
12	list	3	[(0.5, 0.5), (0.0, 0.5), (0.5, 1.0)]
13	list	3	[(0.0, 1.0), (1.0, 0.5), (1.0, 0.5)]
14	list	3	[(0.0, 1.0), (0.5, 0.0), (0.5, 1.0)]
15	list	3	[(0.0, 1.0), (0.5, 1.0), (0.5, 0.0)]
16	list	3	[(0.0, 1.0), (0.5, 1.0), (0.5, 1.0)]
17	list	3	[(0.0, 1.0), (0.0, 0.5), (0.0, 0.5)]

Figure 3.2.: list of triplets of $\mathfrak{so}(4)$ labels that meet the coupling and branching criteria. They label the coupling coefficients that are calculated by the python code.

The matrix A whose nullspace will determine the coupling coefficients then has 18 columns and 44 rows, with each row representing a specific coupling relation. In the next figure a cutout with 11 columns and 18 rows of this matrix calculated via python is shown:

	0	1	2	3	4	5	6	7	8	9	10
0	0.7906	0	-0.6455	0	0	-0.4082	0	0	0	0	0
1	0	1.061	-0.7906	0	0	0.5	0	0	0	0	0
2	0.6124	0	0	-0.5	0	0	-0.7071	0	0	0	0
3	0	-1.369	0	-0.6124	0	0	0.866	0	0	0	0
4	-0.559	0	0	0	-1.061	0	0	-1	0	0	0
5	1.118	0	-1.369	0	-1.061	0	0	0.5	0	0	0
6	0	-0.7906	1.061	0	0	0	0	0	-0.866	0	0
7	0.25	0	0	0	-0.7906	0	0	0	0	-0.5774	0
8	-0.5	0	0	0.6124	-0.7906	0	0	0	0	0.2887	0
9	0	0.3536	0	0.7906	0	0	0	0	0	0	-0.5
10	0	0	0.6124	0	0.7906	0	0	0	0	0	0
11	0	0	0	-0.7906	0.6124	0	0	0	0	0	0
12	-0.3536	0	0	0	0	0	0	-0.7906	0	0.2041	0
13	0.3536	0	0	0	0	0	0	-0.7906	0	-0.6124	0
14	-0.7071	0	0	0	0	0	0.6124	-0.3953	0	0.1021	0
15	0.7071	0	0	0	0	-1.369	0	-0.3953	0	-0.3062	0
16	0	-0.5	0	0	0	0	0.7906	0	-0.6847	0	0.1768
17	0	0.5	0	0	0	1.061	0	0	-0.6847	0	-0.5303

Figure 3.3.: Cutout of the linear system matrix for the second example with an outer multiplicty of 2.

This matrix has a 2-dimensional nullspace, a basis for this is calculated with `scipy.linalg.null_space`. Afterwards the Gram-Schmidt procedure (`gram_schmidt`) is applied to this basis with the use of the special inner product (`scalar_product`). The vectors of the resulting ONB consist of the coupling coefficients of interest. This procedure is embedded in the function `CouplingCoefficients` which references `gram_schmidt`. The two normalized coupling-coefficient vectors as calculated by the python code is shown at the top of the following figure, on the bottom the coupling coefficients from [CSM10] are listed:

$\vec{c_0} =$	-0.145	-0.649	-0.525	0.637	0.375	0.548	-0.576	-0.317	-0.0496	-0.576	0.548	-0.0496	-0.317	0.637	0.375	-0.649	-0.145	-0.525
$\vec{c_1} =$	-0.755	-0.0824	-0.517	-0.37	0.0476	-0.643	-0.392	0.371	-0.558	-0.392	-0.643	-0.558	0.371	-0.37	0.0476	-0.0824	-0.755	-0.517

$$\vec{p_0} = \left[\begin{array}{cccc|cccc|cccc|cccc} \sqrt{\frac{1}{5}} & \sqrt{\frac{3}{10}} & \sqrt{\frac{1}{2}} & 0 & 0 & \sqrt{\frac{3}{10}} & \sqrt{\frac{1}{2}} & \sqrt{\frac{1}{5}} & 0 & \sqrt{\frac{8}{15}} & -\sqrt{\frac{1}{6}} & \sqrt{\frac{1}{10}} & \sqrt{\frac{1}{5}} & -\sqrt{\frac{1}{6}} & \sqrt{\frac{1}{10}} & 0 & \sqrt{\frac{8}{15}} \\ -\sqrt{\frac{12}{35}} & -\sqrt{\frac{1}{70}} & \sqrt{\frac{3}{14}} & -\sqrt{\frac{3}{7}} & -\sqrt{\frac{3}{7}} & -\sqrt{\frac{1}{70}} & \sqrt{\frac{3}{14}} & -\sqrt{\frac{12}{35}} & \sqrt{\frac{1}{7}} & -\sqrt{\frac{2}{35}} & -\sqrt{\frac{1}{14}} & -\sqrt{\frac{27}{70}} & \sqrt{\frac{12}{35}} & \sqrt{\frac{12}{35}} & -\sqrt{\frac{1}{14}} & -\sqrt{\frac{27}{70}} & \sqrt{\frac{1}{7}} & -\sqrt{\frac{2}{35}} \end{array}\right]$$

$$\vec{p_1} =$$

Figure 3.4.: comparison between the two orthonormal coupling coefficient vectors from the python code (top) and from the paper [CSM10] (bottom)

The labelling of each coefficient returned by the code is given in fig. 3.2, in [CSM10] a different order of the $\mathfrak{so}(4)$ irrep label triplets is used. That's why it is hard to compare the coefficients directly. Nevertheless one can see right away that they don't match as one vector from the paper involves four zeros which don't show up in the coefficients produced by the python code. This was

already mentioned at the end of section 3.1 when it was stated that the ONB is only determined up to a unitary transformation. This is intuitively clear for a 2-d real vector space where there are an infinite number of possible orthonormal bases connected with each other by a rotation of the plane which is mathematically described by an orthogonal matrix, the real-valued pendant to a unitary matrix. To test if both vectors in fig.3.4 are actually equivalent one has to determine the matrix that transforms between the two bases and check whether it's orthogonal. The transformation between the two bases looks as follows:

$$\begin{bmatrix} \vec{p_0} \\ \vec{p_1} \end{bmatrix} = \begin{bmatrix} m_{00} & m_{01} \\ m_{10} & m_{11} \end{bmatrix} \begin{bmatrix} \vec{c_0} \\ \vec{c_1} \end{bmatrix} \tag{3.19}$$

With this equation the elements of the transformation matrix can be determined using the fact that the basis from the code $\{\vec{c_i}\}$ is an ONB with respect to the special inner product:

$$\begin{aligned} \vec{p_0} \cdot \vec{c_0} &= (m_{00}\vec{c_0} + m_{01}\vec{c_1}) \cdot \vec{c_0} = m_{00} \\ \vec{p_0} \cdot \vec{c_1} &= (m_{00}\vec{c_0} + m_{01}\vec{c_1}) \cdot \vec{c_1} = m_{01} \\ \vec{p_1} \cdot \vec{c_0} &= (m_{10}\vec{c_0} + m_{11}\vec{c_1}) \cdot \vec{c_0} = m_{10} \\ \vec{p_1} \cdot \vec{c_1} &= (m_{10}\vec{c_0} + m_{11}\vec{c_1}) \cdot \vec{c_1} = m_{11} \end{aligned} \tag{3.20}$$

Using the formulas above, the transformation matrix can be calculated if both bases are known. As a double check the paper basis should be obtained by applying this transformation to the code basis:

$$\begin{bmatrix} \vec{p_0} \\ \vec{p_1} \end{bmatrix} = \begin{bmatrix} 0.126 & -0.992 \\ 0.992 & 0.126 \end{bmatrix} \cdot \begin{bmatrix} \vec{c_0} \\ \vec{c_1} \end{bmatrix} = \begin{bmatrix} 0.73 & -3.34e\text{-}16 & 0.447 & 0.447 & -3.69e\text{-}16 & 0.707 & 0.316 & -0.408 & 0.548 & 0.316 & 0.707 & 0.548 & -0.408 & 0.447 & 9.99e\text{-}17 & -8.55e\text{-}17 & 0.73 & 0.447 \\ -0.239 & -0.655 & -0.586 & 0.586 & 0.378 & 0.463 & -0.621 & -0.267 & -0.12 & -0.621 & 0.463 & -0.12 & -0.267 & 0.586 & 0.378 & -0.655 & -0.239 & -0.586 \end{bmatrix}$$

Figure 3.5.: coupling coefficients from the paper [CSM10] calculated by applying an orthogonal transformation to the coupling coefficient vectors obtained through the python code

It's easy to see that the transformation matrix is in fact orthogonal, so both bases are orthonormal and span the same 2-d space. This makes the coupling coefficients contained in the two bases equivalent. It should be meintioned that the two vectors from fig. 3.5 don't exactly match the ones shown in fig. 3.4 due to numerical errors and different labelling schemes.

4. Fusion coefficients for fuzzy harmonics

The so called *fuzzy approach* poses an alternative to lattice field theory for dealing with quantum fields in the strong coupling regime. This is necessary because for the strong nuclear interaction pertubative methods fail to give reliable results, inter alia, because of arising ultraviolet divergences. In the fuzzy approach the underlying manifold, describing the spacetime the quantum field lives on, gets replaced by a *fuzzy manifold*, where at sufficiently small lengths κ the coordinates of a point become non-commuting operators. Those operators are generators of a underlying non-commutative algebra of functions. Like shown many times before these generators can be represented by matrices which then play the role of scalar fields in an approximation to field theory. [MO03]
With this replacement the position of a particle at scales smaller than κ has no longer a well defined meaning since the three space coordinates can no longer be simultaneously diagonalized. The fuzzy manifold and the associated non-commuative algebra have to approach the classical manifold and it's algebra of functions at scales larger than κ to recover the idea of a precise location of the particle. [Mad92] In contrast to regularization by lattice field theory, the fuzzy approach can preserve symmetries and topological features.[Ydr01].
In the following section the two manifolds S^2,the 2-sphere, and S^4,the 4-sphere, with their corresponding $\mathfrak{so}(3)$ and $\mathfrak{so}(5)$ algebras shall be fuzzyfied. The basis functions on the fuzzy versions of the classical manifolds are called *fuzzy harmonics*. For larger scales these fuzzy harmonics should approach the classical spherical harmonics which form a basis for all function on S^2 and S^4 respectively.

4.1. Fuzzy harmonics on S^2

The defining properties of the $\mathfrak{so}3$ fuzzy harmonics $\hat{Y}_l^m$ are given in the following with the use of the three generators $\mathcal{J}_x$, $\mathcal{J}_y$ and $\mathcal{J}_z$:

$$[\mathcal{J}_z, \hat{Y}_l^m] = m\hat{Y}_l^m \qquad , \qquad \sum_{i=x,y,z} [\mathcal{J}_i, [\mathcal{J}_i, \hat{Y}_l^m]] = l(l+1)\hat{Y}_l^m \tag{4.1}$$

The fuzzy harmonics are labelled by the irrep label l of $\mathfrak{so}(3)$ and the state-label m with $-l \leq m \leq l$. This results in $2l+1$ fuzzy harmonics for each irrep l of $\mathfrak{so}(3)$

Furthermore the $\hat{Y}_l^m$ are orthonormal in the following sense:

$$\mathrm{tr}\left[\hat{Y}_l^m(\hat{Y}_{l'}^{m'})^\dagger\right] = \delta_{l,l'}\,\delta_{m,m'} \tag{4.2}$$

They also fulfill the following conjugation property:

$$(\hat{Y}_l^m)^\dagger = (-1)^m\,\hat{Y}_l^{-m} \tag{4.3}$$

$\hat{Y}_l^m$ can also be represented by a matrix with dimension k alternatively expressed by the label $s = \frac{k-1}{2}$.
Counting all fuzzy harmonics with irrep labels $0 \leq l \leq k-1$ gives the following result:

$$\sum_{l=0}^{k-1}(2l+1) = k^2 \tag{4.4}$$

Since all these $\hat{Y}_l^m$ are orthonormal to each other, they form a basis on the space of complex $k \times k$ matrices. This allows the product of any two fuzzy harmonics to be written as a linear combination of fuzzy harmonics:

$$\hat{Y}_{l_1}^{m_1}\,\hat{Y}_{l_2}^{m_2} = \sum_{l_3=0}^{k-1}\sum_{m_3=-l_3}^{l_3} F_{l_1,m_1;l_2,m_2}^{l_3,m_3}\hat{Y}_{l_3}^{m_3} \tag{4.5}$$

The expansion coefficients $F_{l_1,m_1;l_2,m_2}^{l_3,m_3}$ are called fusion coefficients and can be obtained by using orthonormality of $\hat{Y}_l^m$ and evaluating the trace explicitly. The details of this can be found in [CITE!!!], a similar derivation for the $\mathfrak{so}(5)$ case is also presented later in greater detail. The result for the $\mathfrak{so}(3)$ fuzzy harmonics fusion coefficients is given in the following with the use of the Wigner 3j symbols [1][Mat23]

$$\begin{aligned} F_{l_1,m_1;l_2,m_2}^{l_3,m_3} &= (-1)^{\sum_{i=1}^3 l_i+m_3}\prod_{i=1}^3\sqrt{2l_i+1} \\ &\times\sum_{m_i'=-s}^{s}(-1)^{3s+\sum_{i=1}^3 m_i'}\begin{pmatrix} s & l_1 & s \\ m_2' & m_1 & -m_1'\end{pmatrix}\begin{pmatrix} s & l_2 & s \\ m_3' & m_2 & -m_2'\end{pmatrix}\begin{pmatrix} s & l_3 & s \\ m_1' & -m_3 & -m_3'\end{pmatrix} \end{aligned} \tag{4.6}$$

The implementation of this formula in python can be found in section A.2.1.

[1]The 3j symbols constitute an alternative way of writing coupling coefficients with the following relation between 3j symbols and the Clebsch-Gordon coefficients: $\begin{pmatrix} j_1 & j_2 & j_3 \\ m_1 & m_2 & m_3\end{pmatrix} \equiv \frac{(-1)^{j_1-j_2-m_3}}{\sqrt{2j_3+1}}\langle j_1,m_1;j_2,m_2|j_3,(-m_3)\rangle$

4.1.1. Ex.: 2d-representation of $\hat{Y}_l^m$

In the following subsection the fusion coefficients calculated using eq. (4.6) are tested with a 2-dimensional representation of the fuzzy harmonics $\hat{Y}_l^m$. For that, firstly the 2-dimensional representation of the generators of $\mathfrak{so}(3)$ are presented:

$$\mathcal{J}_x \mathrel{\widehat{=}} \begin{pmatrix} 0 & \frac{1}{2} \\ \frac{1}{2} & 0 \end{pmatrix}, \qquad \mathcal{J}_y \mathrel{\widehat{=}} \begin{pmatrix} 0 & -\frac{i}{2} \\ \frac{i}{2} & 0 \end{pmatrix}, \qquad \mathcal{J}_z \mathrel{\widehat{=}} \begin{pmatrix} \frac{1}{2} & 0 \\ 0 & -\frac{1}{2} \end{pmatrix} \tag{4.7}$$

These 2-d representations are of course essentially the famous *Pauli-matrices*, scaled such that the commutator relations of $\mathfrak{so}(3)$ are fulfilled.
Next the 2d-representation of the relevant fuzzy harmonics are shown:

$$\begin{gathered} Y_0^0 \mathrel{\widehat{=}} \begin{pmatrix} \frac{1}{\sqrt{2}} & 0 \\ 0 & \frac{1}{\sqrt{2}} \end{pmatrix} \\ Y_1^0 \mathrel{\widehat{=}} \begin{pmatrix} \frac{1}{\sqrt{2}} & 0 \\ 0 & -\frac{1}{\sqrt{2}} \end{pmatrix}, \qquad Y_1^{-1} \mathrel{\widehat{=}} \begin{pmatrix} 0 & 0 \\ i & 0 \end{pmatrix}, \qquad Y_1^1 \mathrel{\widehat{=}} \begin{pmatrix} 0 & i \\ 0 & 0 \end{pmatrix} \end{gathered} \tag{4.8}$$

These representations fulfill the eigenvalue equations (4.1), are orthonormalized (4.2) and have the conjugation property (4.3). This allows the testing of the fusion coefficient code, firstly for the following product:

$$Y_1^1 \cdot Y_1^{-1} \mathrel{\widehat{=}} \begin{pmatrix} -1 & 0 \\ 0 & 0 \end{pmatrix} \mathrel{\widehat{=}} \frac{-1}{\sqrt{2}} Y_0^0 + \frac{-1}{\sqrt{2}} Y_1^0 \tag{4.9}$$

This specific product is also written as a linear combination of the matrices from eq.(4.8) which fixes the following two fusion coefficients:

$$\begin{aligned} F^{0,0}_{1,1;1,-1} &= \frac{-1}{\sqrt{2}} \\ F^{1,0}_{1,1;1,-1} &= \frac{-1}{\sqrt{2}} \end{aligned} \tag{4.10}$$

The implementation in python indeed reproduces these two coefficients.
The presented example is also treated in the python code attached in the appendix A.2.1.
Additionally in this code all possible products are compared to the fusion expansion. This verifies the fusion coefficient formula (4.6) as well as the implementation of it for the 2-dimensional case with $s = \frac{1}{2}$. This provides a good indication that everything also works out for different values of s.

4.2. Fuzzy harmonics on S^4

Similarly to the $\mathfrak{so}(3)$ case, the $\mathfrak{so}(5)$ fuzzy harmonics fulfill certain eigenvalue equations which involve the 10 generators $\{T_a\}$ presented in 3.1.1. The corresponding eigenvalues label the $\mathfrak{so}(5)$ fuzzy harmonics just like in the $\mathfrak{so}(3)$ case:

$$\begin{aligned}
\sum_{a=1}^{10}[T_a,[T_a,\hat{Y}^{RS}_{X,Y,m_X,m_Y}]] &= 2\,[R(R+2)+S(S+1)]\;\hat{Y}^{RS}_{X,Y,m_X,m_Y}\\
\sum_{i=1}^{3}[X_i,[X_i,\hat{Y}^{RS}_{X,Y,m_X,m_Y}]] &= X(X+1)\,\hat{Y}^{RS}_{X,Y,m_X,m_Y}\\
[X_3,\hat{Y}^{RS}_{X,Y,m_X,m_Y}] &= m_X\,\hat{Y}^{RS}_{X,Y,m_X,m_Y}\\
\sum_{i=1}^{3}[Y_i,[Y_i,\hat{Y}^{RS}_{X,Y,m_X,m_Y}]] &= Y(Y+1)\,\hat{Y}^{RS}_{X,Y,m_X,m_Y}\\
[Y_3,\hat{Y}^{RS}_{X,Y,m_X,m_Y}] &= m_Y\,\hat{Y}^{RS}_{X,Y,m_X,m_Y}
\end{aligned} \tag{4.11}$$

The $\hat{Y}^{RS}_{X,Y,m_X,m_Y}$ are orthonormal:

$$\mathrm{tr}\left[\hat{Y}^{RS}_{X,Y,m_X,m_Y}(\hat{Y}^{R'S'}_{X',Y',m_{X'},m_{Y'}})^\dagger\right] = \delta_{R,R'}\,\delta_{S,S'}\,\delta_{X,X'}\,\delta_{Y,Y'}\,\delta_{m_X,m_{X'}}\,\delta_{m_Y,m_{Y'}} \tag{4.12}$$

and have the following conjugation properties:

$$\begin{aligned}
(\hat{Y}^{RS}_{X,Y,m_X,m_Y})^\dagger &= (-1)^{1+R-S+X+Y+m_X+m_Y}\hat{Y}^{RS}_{X,Y,-m_X,-m_Y}\\
(\hat{Y}^{RS}_{X,Y,m_X,m_Y})^* &= -(\hat{Y}^{RS}_{X,Y,m_X,m_Y})
\end{aligned} \tag{4.13}$$

The dimension of a particular $\mathfrak{so}(5)$ irrep $(R\,S)$ determines the number of fuzzy harmonics with label $(R\,S)$ [GV18]:

$$\dim_{\mathfrak{so}(5)}(R,S) = \frac{(2R+2S+3)(2R-2S+1)(2R+1)(2S+2)}{6} \tag{4.14}$$

$\hat{Y}^{RS}_{X,Y,m_X,m_Y}$ can again be represented by a k-dimensional matrix; in the following the label n will be used to characterize the dimension of the matrices. The two labels are related through the following equation:

$$k = \frac{(n+3)(n+2)(n+1)}{6} \tag{4.15}$$

To create an ONB for $k\times k$ matrices out of the $\mathfrak{so}(5)$ fuzzy harmonics, all $\hat{Y}^{RS}_{X,Y,m_X,m_Y}$ with $0\le R-S\le R+S\le n$ where R,S are either both integer or both half integers have to be

included. To see that this way a square number of fuzzies is obtained can be proven by induction. For that $\Phi(n)$ shall describe the number of fuzzies which fulfill the condition mentioned above.

assumption:

$$\Phi(n) = \frac{((n+1)(n+2)(n+3))^2}{36} = k^2$$

initial case:

$$\Phi(0) = \frac{((0+1)(0+2)(0+3))^2}{36} = 1^2$$

induction step:

$$\begin{aligned}\Phi(n+1) &= \Phi(n) + \dim(n+1,0) + \dim\left(n+\frac{1}{2},\frac{1}{2}\right) + \ldots + \dim\left(\frac{n+1}{2},\frac{n+1}{2}\right) \\ &= \Phi(n) + \sum_{k=-2}^{n-1} \dim\left(n-\frac{m}{2}, 1+\frac{m}{2}\right) \\ &= \frac{((n+1)(n+2)(n+3))^2}{36} + \frac{(2n+5)(n^2+5n+6)^2}{12} \\ &= \frac{((n+2)(n+3)(n+4))^2}{36} = k'^2\end{aligned} \tag{4.16}$$

The possible values for the matrix dimension are limited to: $k = 1, 4, 10, 20, \ldots$
In summary the set of all fuzzies $\hat{Y}^{RS}_{X,Y,m_X,m_Y}$ with $0 \leq R-S \leq R+S \leq n$ where R, S are either both integer or both half integers form a basis for all complex $k \times k$ matrices which allows again the following fusion expansion. Here the label vector $\vec{L}_i = (R_i, S_i, X_i, Y_i, m_{X_i}, m_{Y_i})$ is introduced.

$$\hat{Y}_{\vec{L_1}} \hat{Y}_{\vec{L_2}} = \sum_{\vec{L_3}} F^{\vec{L_3}}_{\vec{L_1};\vec{L_2}} \hat{Y}_{\vec{L_3}} \tag{4.17}$$

The sum in eq. (4.17) runs over all $(R_3\, S_3)$ values that meet the conditions described above, over all X_3, Y_3 values contained in each $(R_3\, S_3)$ and over all m_{X_3}, m_{Y_3} values that are contained in X_3, Y_3. The fusion coefficients $F^{\vec{L_3}}_{\vec{L_1};\vec{L_2}}$ are obtained by evaluating the following triple trace:

$$F^{\vec{L_3}}_{\vec{L_1};\vec{L_2}} = \mathrm{tr}\left[\hat{Y}_{\vec{L_1}} \hat{Y}_{\vec{L_2}} \left(\hat{Y}_{\vec{L_3}}\right)^\dagger\right] \tag{4.18}$$

The trace is evaluated using the following formula for any operator $\hat{A}$:

$$\mathrm{tr}[\hat{A}] = \sum_{\substack{X,Y \\ m_X,\, m_Y}} \langle R, S, X, Y, m_X, m_Y | \, \hat{A} \, | R, S, X, Y, m_X, m_Y \rangle \tag{4.19}$$

For the calculation the following identity relation is needed with R, S chosen s.t. $\dim(R,S) = k$

$$\mathbb{1} = \sum_{\substack{X,Y \\ m_X, m_Y}} |R,S,X,Y,m_X,m_Y\rangle \langle R,S,X,Y,m_X,m_Y| \tag{4.20}$$

After inserting two identities and using the Wigner-Eckhart theorem [CSM10] as well as the conjugation property (4.3) the following expression for the triple trace is obtained.

$$\begin{aligned}
&\mathrm{tr}\left[\hat{Y}_{\vec{L_1}}\hat{Y}_{\vec{L_2}}\left(\hat{Y}_{\vec{L_3}}\right)^\dagger\right] = (-1)^{1+R_3-S_3+X_3+Y_3+m_{X_3}+m_{Y_3}} \\
&\times \sum_{\substack{X_i',Y_i' \\ m_{X_i'},m_{Y_i'}}} \langle R,S,{X_2}',{Y_2}',m_{X_2'},m_{Y_2'};R_1,S_1,X_1,Y_1,m_{X_1},m_{Y_1}|R,S,{X_1}',{Y_1}',m_{X_1'},m_{Y_1'}\rangle \\
&\qquad \langle R,S,{X_3}',{Y_3}',m_{X_3'},m_{Y_3'};R_2,S_2,X_2,Y_2,m_{X_2},m_{Y_2}|R,S,{X_2}',{Y_2}',m_{X_2'},m_{Y_2'}\rangle \\
&\qquad \langle R,S,{X_1}',{Y_1}',m_{X_1'},m_{Y_1'};R_3,S_3,X_3,Y_3,-m_{X_3},-m_{Y_3}|R,S,{X_3}',{Y_3}',m_{X_3'},m_{Y_3'}\rangle
\end{aligned} \tag{4.21}$$

Now Racah's factorization lemma [Rac49] is exploited. The lemma is responsible for the appearance of the so called reduced matrix element $\rho_i = \rho(R,S,R_i,S_i)$ which will be calculated later. In addition the $\mathfrak{so}(3)$ coupling coefficients are rewritten as 3j-symbols, which gives rise to some additional phase factors.

$$\begin{aligned}
&\mathrm{tr}\left[\hat{Y}_{\vec{L_1}}\hat{Y}_{\vec{L_2}}\left(\hat{Y}_{\vec{L_3}}\right)^\dagger\right] = (-1)^{1+R_3-S_3+X_3+Y_3+m_{X_3}+m_{Y_3}-X_1-X_2-X_3-Y_1-Y_2-Y_3}\,\rho_1\,\rho_2\,\rho_3 \\
&\times \sum_{\substack{X_i',Y_i' \\ m_{X_i'},m_{Y_i'}}} \langle R,S,{X_2}',{Y_2}';R_1,S_1,X_1,Y_1||R,S,{X_1}',{Y_1}'\rangle\ \sqrt{2{X_1}'+1}\,\sqrt{2{Y_1}'+1} \\
&\qquad \langle R,S,{X_3}',{Y_3}';R_2,S_2,X_2,Y_2||R,S,{X_2}',{Y_2}'\rangle\ \sqrt{2{X_2}'+1}\,\sqrt{2{Y_2}'+1} \\
&\qquad \langle R,S,{X_1}',{Y_1}';R_3,S_3,X_3,Y_3||R,S,{X_3}',{Y_3}'\rangle\ \sqrt{2{X_3}'+1}\,\sqrt{2{Y_3}'+1} \\
&\qquad (-1)^{{X_2}'+m_{X_1'}+{Y_2}'+m_{Y_1'}}\begin{pmatrix}{X_2}' & X_1 & {X_1}' \\ m_{X_2'} & m_{X_1} & -m_{X_1'}\end{pmatrix}\begin{pmatrix}{Y_2}' & Y_1 & {Y_1}' \\ m_{Y_2'} & m_{Y_1} & -m_{Y_1'}\end{pmatrix} \\
&\qquad (-1)^{{X_3}'+m_{X_2'}+{Y_3}'+m_{Y_2'}}\begin{pmatrix}{X_3}' & X_2 & {X_2}' \\ m_{X_3'} & m_{X_2} & -m_{X_2'}\end{pmatrix}\begin{pmatrix}{Y_3}' & Y_2 & {Y_2}' \\ m_{Y_3'} & m_{Y_2} & -m_{Y_2'}\end{pmatrix} \\
&\qquad (-1)^{{X_1}'+m_{X_3'}+{Y_1}'+m_{Y_3'}}\begin{pmatrix}{X_1}' & X_3 & {X_3}' \\ m_{X_1'} & -m_{X_3} & -m_{X_3'}\end{pmatrix}\begin{pmatrix}{Y_1}' & Y_3 & {Y_3}' \\ m_{Y_1'} & -m_{Y_3} & -m_{Y_3'}\end{pmatrix}
\end{aligned} \tag{4.22}$$

All that is left to do is to calculate the reduced matrix element ρ_i. For that the following double trace is evaluated explicitly, starting in a similar way as for the triple trace:

$$\begin{aligned}
&\mathrm{tr}\left[\hat{Y}_{\vec{L_1}}\left(\hat{Y}_{\vec{L_2}}\right)^{\dagger}\right] = \rho_1\,\rho_2\,(-1)^{1+R_2-S_2+X_2+Y_2+m_{X_2}+m_{Y_2}+X_1+X_2+Y_1+Y_2} \\
&\times \sum_{X_i{}',Y_i{}'} \sqrt{(2X_1{}'+1)(2Y_1{}'+1)}\,\langle R,S,X_2{}',Y_2{}';R_1,S_1,X_1,Y_1||R,S,X_1{}',Y_1{}'\rangle \\
&\qquad \sqrt{(2X_2{}'+1)(2Y_2{}'+1)}\,\langle R,S,X_1{}',Y_1{}';R_2,S_2,X_2,Y_2||R,S,X_2{}',Y_2{}'\rangle \\
&\times \sum_{m_{X_i{}'},m_{Y_i{}'}} (-1)^{-m_{X_1{}'}-m_{X_2{}'}-m_{Y_1{}'}-m_{Y_2{}'}} \begin{pmatrix} X_2{}' & X_1 & X_1{}' \\ m_{X_2{}'} & m_{X_1} & -m_{X_1{}'} \end{pmatrix} \begin{pmatrix} Y_2{}' & Y_1 & Y_1{}' \\ m_{Y_2{}'} & m_{Y_1} & -m_{Y_1{}'} \end{pmatrix} \\
&\qquad (-1)^{-X_1{}'-X_2{}'-Y_1{}'-Y_2{}'} \begin{pmatrix} X_1{}' & X_2 & X_2{}' \\ m_{X_1{}'} & -m_{X_2} & -m_{X_2{}'} \end{pmatrix} \begin{pmatrix} Y_1{}' & Y_2 & Y_2{}' \\ m_{Y_1{}'} & -m_{Y_2} & -m_{Y_2{}'} \end{pmatrix}
\end{aligned} \tag{4.23}$$

Next the symmetry relations and the completeness of the 3j-symbols are exploited ([Mat23]). For the reduced $\mathfrak{so}(5)$ coupling coefficients one could pursue a similar approach to completely eliminate the sum over $\mathfrak{so}(4)$ labels. The symmetry relations unfortunately aren't as approachable as it is the case for the 3j-symbols. They can be found in Appendix 2 of [Hec65]. Since the calculation of the reduced $\mathfrak{so}(5)$ coupling coefficients and the summation over them is done easily with the python code, here the sum is left as is. After some algebra and setting $\vec{L_2} = \vec{L_1}$ the following expression is obtained:

$$\begin{aligned}
&\mathrm{tr}\left[\hat{Y}_{\vec{L_1}}\left(\hat{Y}_{\vec{L_1}}\right)^{\dagger}\right] = {\rho_1}^2\,(-1)^{1+R_1-S_1-X_1-Y_1}\,\frac{1}{(2X_1+1)(2Y_1+1)} \\
&\times \sum_{X_i{}',Y_i{}'} (-1)^{-X_1{}'-Y_1{}'}\sqrt{(2X_1{}'+1)(2Y_1{}'+1)}\,\langle R,S,X_1{}',Y_1{}';R_1,S_1,X_1,Y_1||R,S,X_2{}',Y_2{}'\rangle \\
&\qquad (-1)^{X_2{}'+Y_2{}'}\sqrt{(2X_2{}'+1)(2Y_2{}'+1)}\,\langle R,S,X_2{}',Y_2{}';R_1,S_1,X_1,Y_1||R,S,X_1{}',Y_1{}'\rangle
\end{aligned} \tag{4.24}$$

Since the fuzzy harmonics are orthonormalized, this double trace should be equal to 1 which allows the formulation of an explicit expression for ρ_1:

$$\begin{aligned}
&\rho_1 = (-1)^{\frac{1}{2}(-1-R_1+S_1+X_1+Y1)}\sqrt{(2X_1+1)(2Y_1+1)} \\
&\Bigg(\sum_{X_i{}',Y_i{}'} (-1)^{-X_1{}'-Y_1{}'}\sqrt{(2X_1{}'+1)(2Y_1{}'+1)}\,\langle R,S,X_1{}',Y_1{}';R_1,S_1,X_1,Y_1||R,S,X_2{}',Y_2{}'\rangle \\
&\quad (-1)^{X_2{}'+Y_2{}'}\sqrt{(2X_2{}'+1)(2Y_2{}'+1)}\,\langle R,S,X_2{}',Y_2{}';R_1,S_1,X_1,Y_1||R,S,X_1{}',Y_1{}'\rangle\Bigg)^{-\frac{1}{2}}
\end{aligned} \tag{4.25}$$

4.2.1. Ex.: 4d-representation of $\hat{Y}_{\vec{L}}$

To test the code version of formula (4.22) which can be found in section A.2.2 the 4-dimensional representation of $\hat{Y}_{\vec{L}}$ is looked at. The 10 generators of $\mathfrak{so}(5)$ have the following 4-d representations:

$$X_1 = \begin{pmatrix} 0 & \frac{1}{2} & 0 & 0 \\ \frac{1}{2} & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \end{pmatrix} X_2 = \begin{pmatrix} 0 & \frac{-i}{2} & 0 & 0 \\ \frac{i}{2} & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \end{pmatrix} X_3 = \begin{pmatrix} \frac{1}{2} & 0 & 0 & 0 \\ 0 & \frac{-1}{2} & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \end{pmatrix}$$

$$Y_1 = \begin{pmatrix} 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & \frac{1}{2} \\ 0 & 0 & \frac{1}{2} & 0 \end{pmatrix} Y_2 = \begin{pmatrix} 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & \frac{-i}{2} \\ 0 & 0 & \frac{i}{2} & 0 \end{pmatrix} Y_3 = \begin{pmatrix} 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & \frac{1}{2} & 0 \\ 0 & 0 & 0 & \frac{-1}{2} \end{pmatrix} \tag{4.26}$$

$$T_{++} = \begin{pmatrix} 0 & 0 & 0 & \frac{-i}{2} \\ 0 & 0 & 0 & 0 \\ 0 & \frac{-i}{2} & 0 & 0 \\ 0 & 0 & 0 & 0 \end{pmatrix} \quad T_{+-} = \begin{pmatrix} 0 & 0 & \frac{-1}{2} & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & \frac{1}{2} & 0 & 0 \end{pmatrix}$$

$$T_{-+} = \begin{pmatrix} 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & \frac{1}{2} \\ \frac{-1}{2} & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \end{pmatrix} \quad T_{--} = \begin{pmatrix} 0 & 0 & 0 & 0 \\ 0 & 0 & \frac{-i}{2} & 0 \\ 0 & 0 & 0 & 0 \\ \frac{-i}{2} & 0 & 0 & 0 \end{pmatrix}$$

Here the X_i and Y_i generators span two separate $\mathfrak{so}(3)$ algebras that each fulfill the usual $\mathfrak{so}(3)$ commutation relation. Together these two algebras form the algebra of $\mathfrak{so}(4) \sim \mathfrak{so}(3) \times \mathfrak{so}(3)$. The 4 additional generators $T_{\pm\pm}$ connect these two subalgebras.
The dimension of 4 fixes $n = 1$. The fuzzy harmonics that span the whole 16-dimensional space of 4×4 matrices have $\mathfrak{so}(5)$ labels $(R, S) \in \{(0, 0), (0.5, 0.5), (1, 0)\}$ as described in section 4.2. Now the task consists of finding the 4d-fuzzies $\hat{Y}^{RS}_{X,Y,m_X,m_Y}$ with the above mentioned $\mathfrak{so}(5)$ labels that fulfill the relevant eigenvalue equations (4.11), are properly normalized according to eq. (4.12) and have the right conjugation properties (4.13). The sign of each fuzzy is chosen such that the reduced matrix element calculated by using the Wigner-Eckhart theorem $\rho_1 = \frac{\langle \vec{L'} | Y_{\vec{L_1}} | \vec{L} \rangle}{\langle \vec{L'}; \vec{L_1} | \vec{L} \rangle}$ matches with the result from eq. (4.25).
The concrete procedure can be found as comments in the code attached in section A.2.2, here the resulting matrices are presented without a detailed derivation:

$$\hat{Y}^{0\,0}_{0,0,0,0} = \begin{pmatrix} \frac{i}{2} & 0 & 0 & 0 \\ 0 & \frac{i}{2} & 0 & 0 \\ 0 & 0 & \frac{i}{2} & 0 \\ 0 & 0 & 0 & \frac{i}{2} \end{pmatrix} \quad \hat{Y}^{\frac{1}{2}\,\frac{1}{2}}_{0,0,0,0} = \begin{pmatrix} \frac{-i}{2} & 0 & 0 & 0 \\ 0 & \frac{-i}{2} & 0 & 0 \\ 0 & 0 & \frac{i}{2} & 0 \\ 0 & 0 & 0 & \frac{i}{2} \end{pmatrix}$$

$$\hat{Y}^{\frac{1}{2}\,\frac{1}{2}}_{\frac{1}{2},\frac{1}{2},\frac{1}{2},\frac{1}{2}} = \begin{pmatrix} 0 & 0 & 0 & \frac{-i}{\sqrt{2}} \\ 0 & 0 & 0 & 0 \\ 0 & \frac{i}{\sqrt{2}} & 0 & 0 \\ 0 & 0 & 0 & 0 \end{pmatrix} \quad \hat{Y}^{\frac{1}{2}\,\frac{1}{2}}_{\frac{1}{2},\frac{1}{2},\frac{1}{2},-\frac{1}{2}} = \begin{pmatrix} 0 & 0 & \frac{i}{\sqrt{2}} & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & \frac{i}{\sqrt{2}} & 0 & 0 \end{pmatrix}$$

$$\hat{Y}^{\frac{1}{2}\,\frac{1}{2}}_{\frac{1}{2},\frac{1}{2},-\frac{1}{2},\frac{1}{2}} = \begin{pmatrix} 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & \frac{-i}{\sqrt{2}} \\ \frac{-i}{\sqrt{2}} & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \end{pmatrix} \quad \hat{Y}^{\frac{1}{2}\,\frac{1}{2}}_{\frac{1}{2},\frac{1}{2},-\frac{1}{2},-\frac{1}{2}} = \begin{pmatrix} 0 & 0 & 0 & 0 \\ 0 & 0 & \frac{i}{\sqrt{2}} & 0 \\ 0 & 0 & 0 & 0 \\ \frac{-i}{\sqrt{2}} & 0 & 0 & 0 \end{pmatrix}$$

$$\hat{Y}^{1\,0}_{0,1,0,-1} = \begin{pmatrix} 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & -i & 0 \end{pmatrix} \quad \hat{Y}^{1\,0}_{0,1,0,0} = \begin{pmatrix} 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & \frac{-i}{\sqrt{2}} & 0 \\ 0 & 0 & 0 & \frac{i}{\sqrt{2}} \end{pmatrix}$$

$$\hat{Y}^{1\,0}_{0,1,0,1} = \begin{pmatrix} 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & i \\ 0 & 0 & 0 & 0 \end{pmatrix} \quad \hat{Y}^{1\,0}_{1,0,-1,0} = \begin{pmatrix} 0 & 0 & 0 & 0 \\ -i & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \end{pmatrix} \tag{4.27}$$

$$\hat{Y}^{1\,0}_{1,0,0,0} = \begin{pmatrix} \frac{-i}{\sqrt{2}} & 0 & 0 & 0 \\ 0 & \frac{i}{\sqrt{2}} & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \end{pmatrix} \quad \hat{Y}^{1\,0}_{1,0,1,0} = \begin{pmatrix} 0 & i & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \end{pmatrix}$$

$$\hat{Y}^{1\,0}_{\frac{1}{2},\frac{1}{2},\frac{1}{2},\frac{1}{2}} = \begin{pmatrix} 0 & 0 & 0 & \frac{i}{\sqrt{2}} \\ 0 & 0 & 0 & 0 \\ 0 & \frac{i}{\sqrt{2}} & 0 & 0 \\ 0 & 0 & 0 & 0 \end{pmatrix} \quad \hat{Y}^{1\,0}_{\frac{1}{2},\frac{1}{2},\frac{1}{2},-\frac{1}{2}} = \begin{pmatrix} 0 & 0 & \frac{-i}{\sqrt{2}} & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & \frac{i}{\sqrt{2}} & 0 & 0 \end{pmatrix}$$

$$\hat{Y}^{1\,0}_{\frac{1}{2},\frac{1}{2},-\frac{1}{2},\frac{1}{2}} = \begin{pmatrix} 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & \frac{i}{\sqrt{2}} \\ \frac{-i}{\sqrt{2}} & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \end{pmatrix} \quad \hat{Y}^{1\,0}_{\frac{1}{2},\frac{1}{2},-\frac{1}{2},-\frac{1}{2}} = \begin{pmatrix} 0 & 0 & 0 & 0 \\ 0 & 0 & \frac{-i}{\sqrt{2}} & 0 \\ 0 & 0 & 0 & 0 \\ \frac{-i}{\sqrt{2}} & 0 & 0 & 0 \end{pmatrix}$$

These 16 $\mathfrak{so}(5)$ fuzzy harmonics are saved in the code attached in sec.A.2.2 using the dictionary `fuzzy_dict` with the corresponding eigenvalues (R, S, X, Y, m_X, m_Y) as the associated keys.
To test the fusion coefficient formula and the python implementation of it, first the following product is looked at:

$$\hat{Y}^{1\,0}_{1,0,1,0} \cdot \hat{Y}^{1\,0}_{1,0,-1,0} = \begin{pmatrix} 0 & i & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \end{pmatrix} \cdot \begin{pmatrix} 0 & 0 & 0 & 0 \\ -1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \end{pmatrix} = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \end{pmatrix} \tag{4.28}$$

The resulting product can be expressed as a linear combination of 3 fuzzy harmonics:

$$\begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \end{pmatrix} = \frac{i}{2}\hat{Y}^{\frac{1}{2}\,\frac{1}{2}}_{0,0,0,0} - \frac{i}{2}\hat{Y}^{0\,0}_{0,0,0,0} + \frac{i}{\sqrt{2}}\hat{Y}^{1\,0}_{1,0,0,0} \tag{4.29}$$

This in turn fixes 3 $\mathfrak{so}(5)$ fusion coefficients that the code can be tested on:

$$\begin{aligned} F^{\frac{1}{2},\frac{1}{2},0,0,0,0}_{(1,0,1,0,1,0);(1,0,1,0,-1,0)} &= \frac{i}{2} \\ F^{0,0,0,0,0,0}_{(1,0,1,0,1,0);(1,0,1,0,-1,0)} &= \frac{-i}{2} \\ F^{1,0,1,0,0,0}_{(1,0,1,0,1,0);(1,0,1,0,-1,0)} &= \frac{i}{\sqrt{2}} \end{aligned} \tag{4.30}$$

The python implementation indeed reproduces the correct values for these 3 choices of $\vec{L_1}$,$\vec{L_2}$ and $\vec{L_3}$. All that remains to to is verify that the code returns the correct values for all possible products of two fuzzy harmonics. For this the program loops through every possible combination of two fuzzies, evaluates their product and compares the resulting matrix to the fusion expansion on the right hand side of eq. (4.17). This test also executed successfully, which verifies the fusion coefficient formula and it's implementation for the 4-d example.
Like in the $\mathfrak{so}(3)$ case, this gives a pretty good indication that the fusion coefficient formula also works for higher values of n.

5. Conclusion

A. Complete python implementation

A.1. Coupling coefficients of SO(5)

```
import numpy as np
from numpy import sqrt
from scipy.linalg import null_space
from time import perf_counter
from sympy.physics.wigner import wigner_6j

def MatrixElements(R, S, X, Y, X_T, Y_T):
    """Matrix-elements for T_{+1/2,+1/2},T_{-1/2,+1/2},
        T_{+1/2,-1/2},T_{-1/2,-1/2}"""
    if X_T == 1/2 and Y_T == 1/2:
        # < (R, S), (X+1/2,Y+1/2) | T_{+1/2, +1/2} | (R, S), (X, Y) >
        N = sqrt((R+S-X-Y)*(R+S+X+Y+3)*(S-R+X+Y+1)*(R-S+X+Y+2))
        D = 2*sqrt(2*(X+1/2)+1)*sqrt(2*(Y+1/2)+1)
        return N / D
    elif X_T == 1/2 and Y_T == -1/2:
        # < (R, S), (X+1/2,Y-1/2) | T_{+1/2, -1/2} | (R, S), (X, Y) >
        N = sqrt((R+S-X+Y+1)*(R+S+X-Y+2)*(R-S-X+Y)*(R-S+X-Y+1))
        D = 2*sqrt(2*(X+1/2)+1)*sqrt(2*(Y-1/2)+1)
        return N / D
    elif X_T == -1/2 and Y_T == 1/2:
        # < (R, S), (X-1/2,Y+1/2) | T_{-1/2, +1/2} | (R, S), (X, Y) >
        N = sqrt((R+S-X+Y+2)*(R+S+X-Y+1)*(R-S-X+Y+1)*(R-S+X-Y))
        D = 2*sqrt(2*(X-1/2)+1)*sqrt(2*(Y+1/2)+1)
        return N / D
    elif X_T == -1/2 and Y_T == -1/2:
        # < (R, S), (X-1/2, Y-1/2) | T_{-1/2, -1/2} | (R, S), (X, Y) >
        N = -sqrt((R+S-X-Y+1)*(R+S+X+Y+2)*(S-R+X+Y)*(R-S+X+Y+1))
        D = 2*sqrt(2*(X-1/2)+1)*sqrt(2*(Y-1/2)+1)
        return N / D

def ReCouplingCoefficients(X1,Y1,X2,Y2,X12,Y12,X3,Y3,X,Y,X23,Y23):
    try:
        return float((-1)**(X1+X2+X3+X) * (-1)**(Y1+Y2+Y3+Y) *
                sqrt(2*X12+1)*sqrt(2*X23+1)*wigner_6j(X1,X2,X12,X3,X,X23)
                *sqrt(2*Y12+1)*sqrt(2*Y23+1)*wigner_6j(Y1,Y2,Y12,Y3,Y,Y23))
    except ValueError:
        return 0
```

```
def PhaseFactor(X1, Y1, X2, Y2, X, Y):
    return (-1)**(X1+X2-X) * (-1)**(Y1+Y2-Y)

def IsValid(X1, Y1, X2, Y2, X, Y):
    """Condition for (X1,Y1) x (X2,Y2) --> (X, Y) for SO(4)"""
    X_steps = (X1 + X2) - abs(X1 - X2)
    Y_steps = (Y1 + Y2) - abs(Y1 - Y2)
    for i in range(int(X_steps) + 1):
        for j in range(int(Y_steps) + 1):
            X_p = abs(X1 - X2) + i; Y_p = abs(Y1 - Y2) + j
            if X == X_p and Y == Y_p:
                return True
    return False

def Indexing(R1, S1, R2, S2, R, S):
    """Gives the order of coupling-coefficients,
        as returned by CouplingCoefficients()"""
    # List to store valid triplets of SO(2) irrep labels
    ordering = []
    # Loop through all the SO(4) irreps in (R1, S1), (R2, S2), (R, S)
    for k1 in range(int(2 * (R1 - S1)) + 1):
        for l1 in range(int(2 * S1) + 1):
            for k2 in range(int(2 * (R2 - S2)) + 1):
                for l2 in range(int(2 * S2) + 1):
                    for k in range(int(2 * (R - S)) + 1):
                        for l in range(int(2 * S) + 1):
                            # Construct the SO(4) irrep labels
                            Z1 = R1 - k1 / 2 - l1 / 2
                            W1 = S1 + k1 / 2 - l1 / 2
                            Z2 = R2 - k2 / 2 - l2 / 2
                            W2 = S2 + k2 / 2 - l2 / 2
                            Z = R - k / 2 - l / 2
                            W = S + k / 2 - l / 2
                            # Checks if:
                            # (Z1, W1) x (Z2, W2) --> (Z, W)
                            if IsValid(Z1, W1, Z2, W2, Z, W):
                                # Append ((P1,Q1), (P2,Q2), (P,Q))
                                ordering.append([(Z1,W1),(Z2,W2),(Z,W)])
    # Return the ordering list for valid SO(4) triplets
    return ordering

def Coupling_Conditions(R1,S1,X1,Y1,R2,S2,X2,Y2,R,S,X,Y,X_s,Y_s,X_T,Y_T):
    """All relations for given (X1, Y1) and (X2, Y2) in the lattice
    first choose a set of three dummy SO(4) irrep labels:
    ((Z1,W1), (Z2,W2), (Z,W)), given the following branching conditions:
    (R1,S1) --> (Z1,W1) , (R2,S2) --> (Z2,W2) , (R,S) --> (Z,W)
    Secondly, we choose SO(4) irrep labels ((X1_s,Y1_s), (X2_s,Y2_s)),
    given the following branching conditions:
    (X_T,Y_T) x (X1_s,Y1_s)-->(X1,Y1), (X_T,Y_T) x (X2_s,Y2_s)-->(X2,Y2)"""
```

```
# List to store relation between reduced cc (as a row of the matrix)
    relation = []
#Loop through all the SO(4) irreps in (R1,S1), (R2,S2), (R,S)
    for k1 in range(int(2 * (R1 - S1)) + 1):
        for l1 in range(int(2 * S1) + 1):
            for k2 in range(int(2 * (R2 - S2)) + 1):
                for l2 in range(int(2 * S2) + 1):
                    for k in range(int(2 * (R - S)) + 1):
                        for l in range(int(2 * S) + 1):
#Construct the SO(4) irrep labels
                            Z1 = R1 - k1 / 2 - l1 / 2
                            W1 = S1 + k1 / 2 - l1 / 2
                            Z2 = R2 - k2 / 2 - l2 / 2
                            W2 = S2 + k2 / 2 - l2 / 2
                            Z = R - k / 2 - l / 2
                            W = S + k / 2 - l / 2
#Checks if: (Z1,W1) x (Z2,W2)-->(Z,W)
                            if IsValid(Z1, W1, Z2, W2, Z, W):
#Variable indicating whether a match has been found
                                conditionMet = False

#Checks if the coupling coefficient:
#((R1,S1),(Z1,W1);(R2,S2),(Z2,W2)|(R,S),(Z,W))
#match the coupling coefficient:
#((R1,S1),(X1,Y1);(R2,S2),(X2,Y2)|(R,S),(X,Y))
                                if (Z1,W1)==(X1,Y1)\
                                and (Z2,W2)==(X2,Y2)\
                                and (Z,W)==(X,Y):
#coefficient for the cc:
#((R1,S1),(X1,Y1);(R2,S2),(X2,Y2)|(R, S), (X, Y)) is:
#<(R,S),(X,Y)|T_(X_T,Y_T)|(R,S),(X_s,Y_s)>
                                    relation.append(\
                                    MatrixElements(R,S,X_s,Y_s,X_T,Y_T))
# Match has been found
                                    conditionMet = True
# Loop over all T_(X1_T,Y1_T):
                                for X1_T in [1/2,-1/2]:
                                    for Y1_T in [1/2,-1/2]:
# <(R1,S1),(X1,Y1)|T_(X_T,Y_T) ~ <(R1,S1),(X1-X_T, Y1-Y_T)|
# Note: we should loop over all (X1_s, Y1_s), consistent with:
# (X_T,Y_T)x(X1_s,Y1_s)-->(X1,Y1)
                                        X1_s = X1 - X1_T
                                        Y1_s = Y1 - Y1_T

# Checks if the cc: ((R1,S1),(Z1,W1);(R2,S2),(Z2,W2)|(R,S),(Z,W))
# match the cc: ((R1,S1),(X1_s,Y1_s);(R2,S2),(X2,Y2)|(R,S),(X_s,Y_s))
                                        if (Z1,W1)==(X1_s,Y1_s) and \
                                        (Z2,W2)==(X2,Y2) and \
                                        (Z,W)==(X_s,Y_s):
```

```
136 # The coefficient for the cc:
137 # ((R1,S1),(X1_s,Y1_s)(R2,S2),(X2,Y2)|(R,S),(X_s,Y_s)) is:
138 # <(R1,S1),(X1,Y1)| T_(X_T, Y_T)|(R1,S1),(X1_s,Y1_s)>
139                                             relation.append(\
140                                             -PhaseFactor(\
141                                             X1, Y1, X2, Y2, X, Y)\
142                                             *PhaseFactor(\
143                                             X1_s,Y1_s,X2,Y2,X_s,Y_s)\
144                                             *ReCouplingCoefficients(\
145                                             X2, Y2,X1_s,Y1_s,X_s,Y_s\
146                                             ,1/2, 1/2, X, Y, X1, Y1)\
147                                              *MatrixElements(\
148                                              R1,S1,X1_s,Y1_s,X1_T,Y1_T))
149 # Match has been found
150                                             conditionMet = True
151
152 # Loop over all T_(X2_T, Y2_T):
153                                 for X2_T in [1/2, -1/2]:
154                                     for Y2_T in [1/2, -1/2]:
155 #<(R2,S2),(X2,Y2)|T_(X_T, Y_T)~<(R2, S2),(X2-X_T,Y2-Y_T)|
156 #Note: we should loop over all (X2_s, Y2_s),
157 #consistent with:(X_T,Y_T)x(X2_s,Y2_s)-->(X2,Y2)
158                                         X2_s = X2 - X2_T
159                                         Y2_s = Y2 - Y2_T
160
161 #Checks if the cc:((R1,S1),(Z1,W1);(R2,S2),(Z2,W2)|(R,S),(Z,W))
162 #match the cc:((R1,S1),(X1,Y1);(R2,S2),(X2_s,Y2_s)|(R,S),(X_s,Y_s))
163                                         if (Z1,W1)==(X1,Y1)\
164                                         and (Z2, W2)==(X2_s,Y2_s)\
165                                         and (Z,W)==(X_s, Y_s):
166 # The coefficient for the cc:
167 #((R1,S1),(X1,Y1);(R2,S2),(X2_s,Y2_s)|(R,S),(X_s,Y_s)) is:
168 #<(R2,S2),(X2,Y2)|T_(X_T,Y_T)|(R2, S2),(X2_s, Y2_s)>
169                                             relation.append(\
170                                             -ReCouplingCoefficients(\
171                                             X1,Y1, X2_s, Y2_s, X_s, Y_s\
172                                             ,1/2, 1/2, X, Y, X2, Y2)
173                                             *MatrixElements(\
174                                             R2,S2,X2_s,Y2_s,X2_T,Y2_T))
175 # Match has been found
176                                             conditionMet = True
177
178 # The cc:((R1, S1), (Z1, W1); (R2, S2), (Z2, W2) | (R, S), (Z, W))
179 # does not match any coupling coefficient in the current relation
180                                 if not conditionMet:
181                                     relation.append(0)
182
183 # return the coefficients of the current relation
184     return relation
```

```
185 def Linear_System_Matrix(R1, S1, R2, S2, R, S):
186     """Computes the non normalized coupling-coefficients
187     First choose a set of four SO(4) irrep labels:
188     ((X1,Y1),(X2,Y2),(X,Y),(X_s,Y_s)), and a generator in SO(5)
189     but not in SO(4): T_(X_T,Y_T) with X_T,Y_T =+1/2,-1/2,
190     given the following branching conditions:
191     (R1,S1)-->(X1,Y1), (R2,S2)-->(X2,Y2), (R,S)-->(X_s,Y_s)
192     (X1,Y1) x (X2,Y2)-->(X,Y), (X_T,Y_T) x (X_s,Y_s)-->(X, Y)
193     Note: we must allow (R, S) -/-> (X, Y), in order to get a
194     sufficient number of relations whenever (R, S) = (0, 0)
195     """
196 # List to store relations between reduced cc (as a matrix)
197     relations = []
198 # Loop through all the SO(4) irreps in (R1,S1), (R2,S2), (R,S)
199     for n1 in range(int(2 * (R1 - S1)) + 1):
200         for m1 in range(int(2 * S1) + 1):
201             for n2 in range(int(2 * (R2 - S2)) + 1):
202                 for m2 in range(int(2 * S2) + 1):
203                     for n_s in range(int(2 * (R - S)) + 1):
204                         for m_s in range(int(2 * S) + 1):
205 
206 # Loop over all T_(X_T, Y_T):
207                             for X_T in [1/2, -1/2]:
208                                 for Y_T in [1/2, -1/2]:
209 
210 # Construct the SO(4) irrep labels:
211                                     X1 = R1 - n1 / 2 - m1 / 2
212                                     Y1 = S1 + n1 / 2 - m1 / 2
213                                     X2 = R2 - n2 / 2 - m2 / 2
214                                     Y2 = S2 + n2 / 2 - m2 / 2
215                                     X_s = R - n_s / 2 - m_s / 2
216                                     Y_s = S + n_s / 2 - m_s / 2
217 # T_(X_T, Y_T) |(R,S), (X_s,Y_s)> ~ |(R,S), (X_s+X_T, Y_s+Y_T)>
218 # loop over all (X,Y), consistent with: (X_T,Y_T)x(X_s,Y_s)-->(X, Y)
219 
220                                     X = X_s + X_T
221                                     Y = Y_s + Y_T
222 
223 # Checks if: (X1, Y1) x (X2, Y2) --> (X, Y)
224                                     if IsValid(X1, Y1, X2, Y2, X, Y):
225 
226 # Get the relation between reduced cc for the SO(4) irrep labels:
227 #((X1,Y1), (X2,Y2), (X,Y), (X_s,Y_s))
228                                         relation=Coupling_Conditions(\
229                                         R1,S1,X1,Y1,R2,S2,X2,Y2,R,S,\
230                                         X,Y,X_s,Y_s,X_T,Y_T)
231                                         relations.append(relation)
232 # Return the list of all relation between reduced cc, as a matrix
233     return relations
```

```
234 def scalar_product(vec1,vec2,R1,S1,R2,S2,R,S,X_S,Y_S):
235     """ XS,Y_S should be chosen such that (R,S)-->(X_S,Y_S)
236     vec1 and vec2 should be of same length
237     """
238     scalar_product=0
239     RS_lst=Indexing(R1, S1, R2, S2, R, S)
240     i=0
241     while i<len(vec1):
242         if RS_lst[i][2]==(X_S,Y_S):
243             scalar_product+=vec1[i]*vec2[i]
244         i+=1
245     return scalar_product
246 
247 def gram_schmidt(vec_list,R,S,R1,S1,R2,S2):
248     """
249     Input:vec_list:list of vectors the gram-schmidt procedure is done to
250     Returns:new_vec_list: new ONB calculated through gram-schmidt
251     """
252     i=0
253     new_vec_list=[]
254     while i<len(vec_list):
255         s=0
256         for j in range(i):
257             s+=scalar_product(new_vec_list[j],vec_list[i],\
258             R1,S1,R2,S2,R,S,R,S)*new_vec_list[j]
259         new_vec_list.append(vec_list[i]-s)
260         new_vec_list[i]=new_vec_list[i]/np.sqrt(scalar_product(\
261         new_vec_list[i],new_vec_list[i],R1,S1,R2,S2,R,S,R,S))
262         i+=1
263     return new_vec_list
264 
265 def CouplingCoefficients(R1, S1, R2, S2, R, S):
266     """Computes the normalized coupling coefficients"""
267 # Get the Linear system matrix for the
268 # particular choices of (R1, S1), (R2, S2), (R, S)
269     M = np.array(Linear_System_Matrix(R1, S1, R2, S2, R, S))
270 
271 # Compute the null vector of the Linear system matrix
272 # This will be the non-normalized coupling-coefficients
273 # Note: in case the outer multiplicity of (R, S) is not 1,
274 # there will be multiple null vectors.These run through
275 # the Gram-Schmidt procedure to obtain the coupling-coefficients
276 
277     NullVector = null_space(M)
278     nullvec_list=[]
279     for i in range(np.ma.size(NullVector,1)):
280         nullvec_list+=[NullVector[:,i]]
281     nullvec_list=gram_schmidt(nullvec_list,R,S,R1,S1,R2,S2)
282     return nullvec_list
```

```
def ONB_test(nullvec_basis,R1,S1,R2,S2,R,S):
    """
    creates a square matrix with same size as the dim of the nullspace.
    It is supposed to be the unity matrix if "nullvec_basis" is an ONB
    """
    coupling_coefficients=nullvec_basis
    dim=len(coupling_coefficients)
    ONB_test=np.zeros((dim,dim))
    for i in range(dim):
        for j in range(dim):
            ONB_test[i,j]=scalar_product(coupling_coefficients[i],\
            coupling_coefficients[j],R1,S1,R2,S2,R,S,R,S)
    return ONB_test

# first example of irrep labels with a 1-D nullspace
# R1=1; S1=1; R2=1; S2=1; R=1; S=1
# third example of irrep labels with a 4-D nullspace
# R1=2; S1=1; R2=2; S2=1; R=2; S=1

#the following section deals with the 2-D example:
R1=1; S1=0; R2=1; S2=0.5; R=1; S=0.5
#since there is a choice of basis for the 2-D nullspace,
#the cc from the code and from the paper differ
#if both are valid ONBs one can express one in terms of the other
#paper basis: p=[p_0,p_1]; code basis: c=[c_0,c_1]
#transformation between the two: p= mat*c, with mat a 2x2 unitary matrix
#the elements of this matrix are calculated by taking the scalar product
#of the possible combinations of the basis vectors.
#this is meant as a check that the code-basis spans the same space
#as the paper basis and is also an ONB
#to calculate the scalar product it is sufficient
#only to look at the coefficients that share the same (X,Y) label
#here (X,Y)=(0,0.5) is chosen and 2 subcvectors of c_0,c_1
#are created where (X,Y)==(0,0.5) is fulfilled
I=Indexing(R1, S1, R2, S2, R, S)
code_basis=CouplingCoefficients(R1, S1, R2, S2, R, S)

sub_0=[]
i=0
while i<len(I):
    if I[i][2]==(0,0.5):
        sub_0.append(code_basis[0][i])
    i+=1
sub_1=[]
i=0
while i<len(I):
    if I[i][2]==(0,0.5):
        sub_1.append(code_basis[1][i])
    i+=1
```

```
#small_index is created to keep track of the labels, taken over from I
small_index=[]
i=0
Index=Indexing(R1, S1, R2, S2, R, S)
while i<len(Index):
    if Index[i][2]==(0,0.5):
        small_index.append(Index[i])
    i+=1
#next the subvectors of the paper-basis with
#(X,Y)==(0,0.5) are manually created
p_0=np.zeros(4); p_0[0]=0; p_0[1]=np.sqrt(3/10)
p_0[2]=np.sqrt(1/2);p_0[3]=np.sqrt(1/5)

p_1=np.zeros(4); p_1[0]=-np.sqrt(3/7); p_1[1]=-np.sqrt(1/70)
p_1[2]=np.sqrt(3/14); p_1[3]=-np.sqrt(12/35)

#finally the components of the matrix are calculated
#here the normal scalar product is used since we are
#already in a subspace where (X,Y) is fixed

mat=np.zeros((2,2))
mat[0,0]=np.dot(sub_0,p_0)
mat[1,0]=np.dot(sub_0,p_1)
mat[0,1]=np.dot(sub_1,p_0)
mat[1,1]=np.dot(sub_1,p_1)

#this matrix is indeed orthogonal as expected
#like mentioned above the full cc from the paper can now be obtained by
#multiplying this 2x2 matrix with the coefficients from the code:
#[p_0,p_1] = mat * [c_o,c_1]

paper_basis=np.dot(mat, code_basis)
paper_basis=list(paper_basis)

#at the end it is checked if both basis are really an ONB:
ONB_test_paper=ONB_test(paper_basis,R1,S1,R2,S2,R,S)
ONB_test_code=ONB_test(code_basis, R1, S1, R2, S2, R, S)
```

A.2. Fusion coefficients for fuzzy harmonics

A.2.1. Fuzzy harmonics on S^2

```
import numpy as np
import so3_cc
from sympy.physics.wigner import wigner_3j

def fusion(l3,m3,l1,m1,l2,m2,s):
    exponent=complex(l1+l2+l3+m3)
    phase1=(-1)**exponent
    result=phase1*np.sqrt(2*l1+1)*np.sqrt(2*l2+1)*np.sqrt(2*l3+1)
    summe=0
    for n1 in np.arange(-s,s+1,1, dtype=float):
        for n2 in np.arange(-s,s+1,1, dtype=float):
            for n3 in np.arange(-s, s+1,1, dtype=float):
                exponent2=complex(3*s+n1+n2+n3)
                phase2=(-1)**exponent2
                summe+=phase2*complex(wigner_3j(s, l1, s, n2, m1, -n1))\
                    *complex(wigner_3j(s, l2, s, n3, m2, -n2))\
                    *complex(wigner_3j(s, l3, s, n1, -m3, -n3))
    result=result*summe
    return result

#testing the fusion code on the 2d example
s=0.5
#manually implementing the pauli matrices:
sigma_0=np.identity(2,dtype=complex)
sigma_1=np.zeros((2,2),dtype=np.complex64)
sigma_1[0,1]=1; sigma_1[1,0]=1
sigma_2=np.zeros((2,2),dtype=np.complex64)
sigma_2[0,1]=-1j; sigma_2[1,0]=1j
sigma_3=np.zeros((2,2),dtype=complex)
sigma_3[0,0]=1; sigma_3[1,1]=-1

#SO(3) generators
t_1=1/2*sigma_1; t_2=1/2*sigma_2;t_3=1/2*sigma_3
generator_dict={1:t_1,2:t_2,3:t_3}
#two helping functions:
def commutator(a,b):
    return (np.dot(a, b)-np.dot(b, a))
def levi_civita(i,j,k):
    if i==j or j==k or i==k:
        return 0
    elif (i,j,k)==(1,2,3) or (i,j,k)==(2,3,1) or (i,j,k)==(3,1,2):
        return 1
    else:
        return -1
```

```
#generator testing:

gen_test=True
for i1,gen1 in list(generator_dict.items()):
    for i2,gen2 in list(generator_dict.items()):
        comm=commutator(gen1, gen2)
        summe=0
        for i3,gen3 in list(generator_dict.items()):
            summe+=1j*levi_civita(i1, i2, i3)*gen3
        if not np.array_equal(comm, summe)==True:
            gen_test=False

print("generator comm rel=",gen_test)

#the commutation relation of SO(3) are fulfilled

#fuzzy harmonics for l=0 m=0 s=1/2
Y_0=1/np.sqrt(2)*sigma_0
#fuzzy harmonics for l=1 m=1 s=1/2
Y_plus=1j/2*(sigma_1+1j*sigma_2)
#fuzzy harmonics for l=1 m=-1 s=1/2
Y_minus=1j/2*(sigma_1-1j*sigma_2)
#fuzzy harmonics for l=1 m=0 s=1/2
Y_3=1/np.sqrt(2)*sigma_3

fuzzy_dict={(0,0):Y_0,(1,0):Y_3,(1,-1):Y_minus,(1,1):Y_plus}

#testing of l eigenvalue equation:

fuzzy_l_test=True
for (l,m),fuzzy in list(fuzzy_dict.items()):
    summe=0
    for t,gen_t in list(generator_dict.items()):
        summe+=commutator(gen_t, commutator(gen_t, fuzzy))
    if not np.array_equal(summe, l*(l+1)*fuzzy):
        fuzzy_l_test=False

print("l-eigenvalue eq=",fuzzy_l_test)

#testing of m eigenvalue equation:

fuzzy_m_test=True
for (l,m),fuzzy in list(fuzzy_dict.items()):
    if not np.array_equal(commutator(t_3,fuzzy), m*fuzzy):
        fuzzy_m_test=False

print("m-eigenvalue eq=",fuzzy_m_test)

#both defining eigenvalue equations of the fuzzy harmonics are fulfilled
```

```
#fusion testing for multiplicaton of l=1,m=1; l=1,m=-1 fuzzy harmonics
l1=1; l2=1; m1=1; m2=-1

fusion_prod_test1=np.dot(Y_plus,Y_minus)

#the result of the multiplication can be written as a linear combination
#of Y_0 and Y_3 with coefficients -1/sqrt(2) and -1/sqrt(2)
fusion_summe_test1=-1/np.sqrt(2)*Y_0-1/np.sqrt(2)*Y_3

#This fixes the following two fusion coefficients:

#for l3=1 m3=0 the coefficient should be -1/sqrt(2)
l3=1; m3=0

fusion_coefficient_test1=fusion1(l3, m3, l1, m1, l2, m2, s)

#for l3=0 m3=0 the coefficient should be -1/sqrt(2)
l3=0; m3=0

fusion_coefficient_test2=fusion1(l3, m3, l1, m1, l2, m2, s)

#both is the case !!

#now lets check all possible products

fusion_test=True
for (l1,m1),fuzzy1 in list(fuzzy_dict.items()):
    for (l2,m2),fuzzy2 in list(fuzzy_dict.items()):
        prod=np.dot(fuzzy1,fuzzy2)
        summe=0
        for (l3,m3),fuzzy3 in list(fuzzy_dict.items()):
            summe+=fusion1(l3, m3, l1, m1, l2, m2, s)*fuzzy3
        if not np.allclose(prod, summe)==True:
            fusion_test=False

print("fusion coefficients=",fusion_test)
```

A.2.2. Fuzzy harmonics on S^4

```
import numpy as np
import so5_cc_numpy
from sympy.physics.wigner import wigner_3j
from sympy.physics.quantum.cg import CG
import math

def reduced_cc(R1,S1,X1,Y1,R2,S2,X2,Y2,R3,S3,X3,Y3,m):
    """
    Returns the reduced so(5) coupling coefficient
    <R1,S1,X1,Y1;R2,S2,X2,Y2||R3,S3,X3,Y3>
    from the coupling coefficient code  so5_cc.numpy

    if the nullspace is multi-dimensional there are multiple choices for the
    reduced coupling coefficient. This choice is realized with the variable m

    """
    triplet=[(X1,Y1),(X2,Y2),(X3,Y3)]
    cc=so5_cc_numpy.CouplingCoefficients(R1,S1,R2,S2,R3,S3)[m]
    Indexing_test=so5_cc_numpy.Indexing(R1,S1,R2,S2,R3,S3)
    i=0
    match_found=False
    while i<len(cc):
        if triplet==Indexing_test[i]:
            match_found=True
            return cc[i]
        i+=1
    if match_found==False:
        return 0

def all_SO4(R,S):
    all_SO4=[]
    for n in np.arange(0,2*(R-S)+1,1):
        for m in np.arange(0,2*S+1,1):
            X=R-n/2-m/2
            Y=S+n/2-m/2
            all_SO4.append((X,Y))
    return all_SO4

def dim(R,S):
    return ((2*R+2*S+3)*(2*R-2*S+1)*(2*S+1)*(2*R+2))/6

#testing on the 4-d representation R=0.5,S=0
R=0.5;S=0;m=0

#if we would deal with so(5) irreps with outer multiplicity, we would have
#to take extra care of the variable "m" and possibly sum over it
```

```
#first define the 10 generators of so(5):

L_12=np.zeros((4,4),dtype=complex)
L_12[0,0]=1/2; L_12[1,1]=-1/2; L_12[2,2]=1/2; L_12[3,3]=-1/2
gen_dict={(1,2):L_12}

L_13=np.zeros((4,4),dtype=complex)
L_13[0,1]=1/2; L_13[1,0]=1/2; L_13[2,3]=1/2; L_13[3,2]=1/2
gen_dict[(1,3)]=L_13

L_23=np.zeros((4,4),dtype=complex)
L_23[0,1]=-1j/2; L_23[1,0]=1j/2; L_23[2,3]=-1j/2; L_23[3,2]=1j/2
gen_dict[(2,3)]=L_23

L_24=np.zeros((4,4),dtype=complex)
L_24[0,1]=-1/2; L_24[1,0]=-1/2; L_24[2,3]=1/2; L_24[3,2]=1/2
gen_dict[(2,4)]=L_24

L_25=np.zeros((4,4),dtype=complex)
L_25[0,3]=1/2; L_25[1,2]=1/2; L_25[2,1]=1/2; L_25[3,0]=1/2
genr_dict[(2,5)]=L_25

L_14=np.zeros((4,4),dtype=complex)
L_14[0,1]=-1j/2; L_14[1,0]=1j/2; L_14[2,3]=1j/2; L_14[3,2]=-1j/2
gen_dict[(1,4)]=L_14

L_15=np.zeros((4,4),dtype=complex)
L_15[0,3]=1j/2; L_15[1,2]=-1j/2; L_15[2,1]=1j/2; L_15[3,0]=-1j/2
gen_dict[(1,5)]=L_15

L_34=np.zeros((4,4),dtype=complex)
L_34[0,0]=1/2; L_34[1,1]=-1/2; L_34[2,2]=-1/2; L_34[3,3]=1/2
gen_dict[(3,4)]=L_34

L_35=np.zeros((4,4),dtype=complex)
L_35[0,2]=-1/2; L_35[1,3]=1/2; L_35[2,0]=-1/2; L_35[3,1]=1/2
gen_dict[(3,5)]=L_35

L_45=np.zeros((4,4),dtype=complex)
L_45[0,2]=1j/2; L_45[1,3]=1j/2; L_45[2,0]=-1j/2; L_45[3,1]=-1j/2
gen_dict[(4,5)]=L_45

gen_dict[(2,1)]=-L_12; gen_dict[(3,1)]=-L_13
gen_dict[(4,1)]=-L_14; gen_dict[(5,1)]=-L_15
gen_dict[(3,2)]=-L_23; gen_dict[(4,2)]=-L_24
gen_dict[(5,2)]=-L_25; gen_dict[(4,3)]=-L_34
gen_dict[(5,3)]=-L_35; gen_dict[(5,4)]=-L_45
gen_dict[(1,1)]=gen_dict[(2,2)]=gen_dict[(3,3)]=\
gen_dict[(4,4)]=gen_dict[(5,5)]=np.zeros((4,4),dtype=complex)
```

```
97
98  gen_dict_so3xso3={"X2":(gen_dict[(2,3)]+gen_dict[(1,4)])/2 \
99              ,"X1":-(gen_dict[(3,1)]+gen_dict[(2,4)])/2 \
100             ,"X3":(gen_dict[(1,2)]+gen_dict[(3,4)])/2 \
101             ,"Y2":(gen_dict[(2,3)]-gen_dict[(1,4)])/2 \
102             ,"Y1":-(gen_dict[(3,1)]-gen_dict[(2,4)])/2 \
103             ,"Y3":(gen_dict[(1,2)]-gen_dict[(3,4)])/2 \
104             ,"T_plus_plus":-(gen_dict[(1,5)]+1j*gen_dict[(2,5)])/2 \
105             ,"T_plus_minus":(gen_dict[(3,5)]+1j*gen_dict[(4,5)])/2 \
106             ,"T_minus_plus":(gen_dict[(3,5)]-1j*gen_dict[(4,5)])/2 \
107             ,"T_minus_minus":(gen_dict[(1,5)]-1j*gen_dict[(2,5)])/2}
108
109 #testing of the generators:
110 def delta(a,b):
111     if a==b:
112         return 1
113     else:
114         return 0
115
116 def comm(a,b):
117     return (np.dot(a, b)-np.dot(b, a))
118
119 gen_test=True
120 for (p,q),G_pq in list(gen_dict.items()):
121     for (r,s),G_rs in list(gen_dict.items()):
122         commutator=comm(G_pq,G_rs)
123         summe=-1j*(delta(q,r)*gen_dict[(p,s)]+delta(p,s)*gen_dict[(q,r)]\
124                   +delta(s,q)*gen_dict[(r,p)]+delta(r,p)*gen_dict[(s,q)])
125         if not np.array_equal(commutator,summe)==True:
126             gen_test=False
127
128 print("generator_testing=",gen_test)
129
130
131 #now define the fuzzy harmonics:
132
133 R1=0; S1=0
134 #the R1=0,S1=0 fuzzy has to commute with all other fuzzys so it has to be
135 #proportional to the 4x4 identity matrix:
136 # ==> X=0 Y=0 ==> mx=0, my=0
137 fuzzy=np.identity(4,dtype=complex)
138 fuzzy_dict={(0,0,0,0,0,0):fuzzy}
139
140 R1=1;S1=0
141 # the 10 R1=1,S1=0 fuzzies should be directly proportional to the 10 so(5) gens
142
143 # first a specific example
144
145 fuzzy1=(gen_dict_so3xso3["X1"]+1j*gen_dict_so3xso3["X2"])/2
```

```
Casimir=0
for (g_i,g_j) in [(1,2),(1,3),(1,4),(1,5),(2,3),(2,4),(2,5),(3,4),(3,5),(4,5)]:
         Casimir+=comm(gen_dict[(g_i,g_j)],comm(gen_dict[(g_i,g_j)],fuzzy1))
Casimir_check=2*(R1*(R1+2)+S1*(S1+1))*fuzzy1

X_square=comm(gen_dict_so3xso3["X1"],comm(gen_dict_so3xso3["X1"],fuzzy1)) \
        +comm(gen_dict_so3xso3["X2"],comm(gen_dict_so3xso3["X2"], fuzzy1)) \
        +comm(gen_dict_so3xso3["X3"],comm(gen_dict_so3xso3["X3"], fuzzy1))
X1=1; X_check=X1*(X1+1)*fuzzy1

Y_square=comm(gen_dict_so3xso3["Y1"],comm(gen_dict_so3xso3["Y1"], fuzzy1)) \
        +comm(gen_dict_so3xso3["Y2"],comm(gen_dict_so3xso3["Y2"], fuzzy1)) \
        +comm(gen_dict_so3xso3["Y3"],comm(gen_dict_so3xso3["Y3"], fuzzy1))
Y1=0; Y_check=Y1*(Y1+1)*fuzzy1

X_3=comm(gen_dict_so3xso3["X3"], fuzzy1)
Y_3=comm(gen_dict_so3xso3["Y3"], fuzzy1)
m_X1=-1; m_Y1=0
X_3_check=m_X1*fuzzy1; Y_3_check=m_Y1*fuzzy1

#fuzzy1 has labels(1,0,1,0,-1,0)

def Casimir_test(fuzzy,R,S):
    Casimir=0
    for (g_i,g_j) in [(1,2),(1,3),(1,4),(1,5),\
                      (2,3),(2,4),(2,5),(3,4),(3,5),(4,5)]:
            Casimir+=comm(gen_dict[(g_i,g_j)],comm(gen_dict[(g_i,g_j)], fuzzy))
    if np.array_equal(Casimir, 2*(R*(R+2)+S*(S+1))*fuzzy)==True:
            #print("Casimir^2 eigenvalue eq. fulfilled")
            return True
    else:
            #print("Casmimir^2 eigenvalue equation not fulfilled")
            return False

def X_Y_test(fuzzy,X,Y):
    X_square=comm(gen_dict_so3xso3["X1"],comm(gen_dict_so3xso3["X1"], fuzzy)) \
            +comm(gen_dict_so3xso3["X2"],comm(gen_dict_so3xso3["X2"], fuzzy)) \
            +comm(gen_dict_so3xso3["X3"],comm(gen_dict_so3xso3["X3"], fuzzy))
    Y_square=comm(gen_dict_so3xso3["Y1"],comm(gen_dict_so3xso3["Y1"], fuzzy)) \
            +comm(gen_dict_so3xso3["Y2"],comm(gen_dict_so3xso3["Y2"], fuzzy)) \
            +comm(gen_dict_so3xso3["Y3"],comm(gen_dict_so3xso3["Y3"], fuzzy))
    if np.array_equal(Y_square,Y*(Y+1)*fuzzy)==True\
            and np.array_equal(X_square,X*(X+1)*fuzzy)==True:
        return True
        #print("X^2 and Y^2 eigenvalue equation fulfilled")
    else:
        return False
        #print("X^2 and Y^2 eigenvalue equation not fulfilled")

```

```
def mx_test(fuzzy,m_X):
    X_3=commutator(generator_dict_so3xso3["X3"], fuzzy)
    if np.array_equal(X_3,m_X*fuzzy)==True:
        return True;#print("X_3 eigenvalue eq. fulfilled")
    else:
        return False;#print("X_3 eigenvalue equation not fulfilled")

def my_test(fuzzy,m_Y):
    Y_3=commutator(generator_dict_so3xso3["Y3"], fuzzy)
    if np.array_equal(Y_3,m_Y*fuzzy)==True:
        return True;#print("Y_3 eigenvalue eq. fulfilled")
    else:
        return False;#print("Y_3 eigenvalue equation not fulfilled")

#testing all eigenvalue equationa of a given the fuzzy harmonic:
def fuzzy_test(fuzzy,R,S,X,Y,m_X,m_Y):
    if Casimir_test(fuzzy, R, S)==True and X_Y_test(fuzzy, X, Y)==True \
        and mx_test(fuzzy, mx) and my_test(fuzzy, my)==True:
        return True
    else:
        return False

#we need the ladder operator version of the 10 generators:
gen_ladder_dict={"X3":gen_dict_so3xso3["X3"],"Y3":gen_dict_so3xso3["Y3"]\
                ,"X+":(gen_dict_so3xso3["X1"]+1j*gen_dict_so3xso3["X2"])/2\
                ,"X-":(gen_dict_so3xso3["X1"]-1j*gen_dict_so3xso3["X2"])/2\
                ,"Y+":(gen_dict_so3xso3["Y1"]+1j*gen_dict_so3xso3["Y2"])/2\
                ,"Y-":(gen_dict_so3xso3["Y1"]-1j*gen_dict_so3xso3["Y2"])/2\
                ,"T++":-(gen_dict[(1,5)]+1j*gen_dict[(2,5)])/2\
                ,"T+-":(gen_dict[(3,5)]+1j*gen_dict[(4,5)])/2\
                ,"T-+":(gen_dict[(3,5)]-1j*gen_dict[(4,5)])/2\
                ,"T--":(gen_dict[(1,5)]-1j*gen_dict[(2,5)])/2}

#now i want to systematically go through all 10 generators in gen_ladder_dict,
#check if the Casimir^2 is fulfilled, then find out the X,Y,m_X,m_Y values:

for gen_name in list(gen_ladder_dict.keys()):
    fuzzy=gen_ladder_dict[gen_name]
    if Casimir_test(fuzzy, R, S)==True:
        print(gen_name, "Casimir^2 test fulfilled")
    for (X,Y) in all_SO4(R, S):
        if X_Y_test(fuzzy, X, Y)==True:
            print("X=",X); print("Y=",Y)
            for m_X in np.arange(-X, X+1):
                for m_Y in np.arange(-Y,Y+1):
                    if mx_test(fuzzy,m_X)==True and my_test(fuzzy,m_Y)==True:
                        print("m_X=",m_X); print("m_Y=",m_Y)
                        fuzzy_dict[(R,S,X,Y,m_X,m_Y)]=fuzzy

```

```
R1=0.5; S1=0.5
#the R1=0.5 S1=0.5 fuzzies are given by the remaining tensor product np.kron
# of pauli matrices sigma_1,sigma_2,sigma_3 plus the identity sigma_0
#that haven't shown up in gen_dict

sigma_0=np.identity(2,dtype=complex)
sigma_1=np.zeros((2,2),dtype=np.complex64)
sigma_1[0,1]=1; sigma_1[1,0]=1
sigma_2=np.zeros((2,2),dtype=np.complex64)
sigma_2[0,1]=-1j; sigma_2[1,0]=1j
sigma_3=np.zeros((2,2),dtype=complex)
sigma_3[0,0]=1; sigma_3[1,1]=-1

#make a function that loops through all possible combinations
#of sigma_i,sigma_j pairs,
#calculate their kronecker product, check wheater it is contained in gen_dict
#and return the key if that is the case
#if the tensor product of sigma_i, sigma_j is not contained
#add it to a new dictionary kron_dict

kron_dict={}
for i,sigma_i in [(0,sigma_0),(1,sigma_1),(2,sigma_2),(3,sigma_3)]:
    for j,sigma_j in [(0,sigma_0),(1,sigma_1),(2,sigma_2),(3,sigma_3)]:
        kron=0.5*np.kron(sigma_i,sigma_j)
        match_found=False
        for key,gen in list(gen_dict.items()):
            if np.array_equal(kron,gen)==True:
                match_found=True
                #print("the tensor product of sigma_",i,"and sigma_",j,
                            "is equal to the generator L_",key)
        if match_found==False:
            kron_dict[(i,j)]=kron

#first lets delete the kron(sigma_0,sigma_0) matrix
#it is the 4x4 identity matrix and is already included as
#the R1=0;S1=1;X1=0;Y1=0;m_X1=0;m_Y1=0 fuzzy harmonic in fuzzy_dict
del(kron_dict[(0,0)])

#now we have to create ladder operators similar to the ones for R=1;S=0.
#they get stored in the list kron_ladder

kron_ladder=[(kron_dict[(1,0)]-1j*kron_dict[(2,3)])/2,\
             (kron_dict[(1,0)]+1j*kron_dict[(2,3)])/2,\
             (kron_dict[(2,2)]+1j*kron_dict[(2,1)])/2,\
             (kron_dict[(2,2)]-1j*kron_dict[(2,1)])/2,kron_dict[(3,0)]]

#now like before we loop through all the ladder operators
#and find out the corresponding X1,Y1,m_X1,m_Y1 values
#and store them in the fuzzy_dict
```

```
for i,fuzzy in enumerate(kron_ladder):
    if Casimir_test(fuzzy, R1, S1)==True:
        print("kron_",i, "Casimir^2 test fulfilled")
    for (X1,Y1) in all_SO4(R1, S1):
        if X_Y_test(fuzzy, X1, Y1)==True:
            print("X1=",X1); print("Y1=",Y1)
            for m_X1 in np.arange(-X1, X1+1):
                for m_Y1 in np.arange(-Y1,Y1+1):
                    if mx_test(fuzzy,m_X1)==True and my_test(fuzzy,m_Y1)==True:
                        print("m_x=",m_X1); print("m_y=",m_Y1)
                        fuzzy_dict[(R1,S1,X1,Y1,m_X1,m_Y1)]=fuzzy


#now we have all 4d representations of the 16 fuzzy harmonics in fuzzy_dict
#and can test the eigenvalue equations on all of them

fuzzy_ev_test=True
for (R1,S1,X1,Y1,m_X1,m_Y1),fuzzy in list(fuzzy_dict.items()):
    if not fuzzy_test(fuzzy, R1, S1, X1, Y1, m_X1, m_Y1)==True:
        fuzzy_ev_test=False
print("eigenvalue equations=", fuzzy_ev_test)

#reduced matrix element formula:
#a sum over all "m" would have to be included if there is outer multiplicity

R=0.5;S=0
red_mat_el_formula={}
for (R1,S1,X1,Y1,m_X1,m_Y1),fuzzy in fuzzy_dict.items():
    summe=0
    for (X1_p,Y1_p) in all_SO4(R,S):
        for (X2_p,Y2_p) in all_SO4(R,S):
            summe+=(-1)**(complex(X2_p-X1_p+Y2_p-Y1_p))*np.sqrt(2*X1_p+1)*\
                np.sqrt(2*Y1_p+1)*np.sqrt(2*X2_p+1)*np.sqrt(2*Y2_p+1)*\
                reduced_cc(R,S,X2_p,Y2_p,R1,S1,X1,Y1,R,S,X1_p,Y1_p,0)*\
                reduced_cc(R,S,X1_p,Y1_p,R1,S1,X1,Y1,R,S,X2_p,Y2_p,0)
    RHS=(-1)**(complex(1+R1-S1-X1-Y1))*1/((2*X1+1)*(2*Y1+1))*summe
    red_mat_el_formula[(R1,S1,X1,Y1,m_X1,m_Y1)]=1/(np.sqrt(RHS))

#now onto normalization:
def fuzzy_size(R,S,X,Y,m_X,m_Y):
    fuzzy_matrix=np.matrix(fuzzy_dict[(R,S,X,Y,m_X,m_Y)])
    return np.trace(np.dot(fuzzy_matrix, fuzzy_matrix.H))

#normalization on a single fuzzy

# check on R1=0,S1=0,X1=0,Y1=0,m_X1=0,m_Y1=0 fuzzy
R1=0;S1=0;X1=0;Y1=0;m_X1=0;m_Y1=0
print(fuzzy_size(R1, S1, X1, Y1, m_X1, m_Y1))

```

```
# the product of fuzzy and fuzzy^dagger has a trace of 4
# division by sqrt(4)=2  ==> fuzzy properly normalized
fuzzy_dict[(R1,S1,X1,Y1,m_X1,m_Y1)]=1/2*fuzzy_dict[(R1,S1,X1,Y1,m_X1,m_Y1)]
print(fuzzy_size(R1, S1, X1, Y1, m_X1, m_Y1))

#loop through all fuzzys and divide by np.trace(Y*Y^dagger)
for (R1,S1,X1,Y1,m_X1,m_Y1),fuzzy in list(fuzzy_dict.items()):
    fuzzy_dict[(R1,S1,X1,Y1,m_X1,m_Y1)]=fuzzy_dict[(R1,S1,X1,Y1,m_X1,m_Y1)]\
    *1/(np.sqrt(fuzzy_size(R1,S1,X1,Y1,m_X1,m_Y1)))

#check weather the fuzzys are properly normalized

norm_test=True
for (R1,S1,X1,Y1,m_X1,m_Y1),fuzzy in list(fuzzy_dict.items()):
    if not math.isclose(np.real(fuzzy_size(R1,S1,X1,Y1,m_X1,m_Y1)),1,\
                                                    rel_tol=1e-4)==True:
        norm_test=False
        print(R1,S1,X1,Y1,m_X1,m_Y1)
print("normalization=",norm_test)

#conjugation properties

def conjugation_test_1(R,S,X,Y,m_X,m_Y):
    LHS=np.conjugate(fuzzy_dict[(R,S,X,Y,m_X,m_Y)])
    RHS=-fuzzy_dict[(R,S,X,Y,m_X,m_Y)]
    if np.array_equal(LHS, RHS)==True:
        return True
    else:
        return False

def conjugation_test_2(R,S,X,Y,m_X,m_Y):
    fuzzy_matrix_transpose=np.matrix(fuzzy_dict[(R,S,X,Y,m_X,m_Y)]).H
    fuzzy_matrix_compare=(-1)**complex(1+R-S+X+Y+m_X+m_Y)*\
                    np.matrix(fuzzy_dict[(R,S,X,Y,-m_X,-m_Y)])
    if np.array_equal(fuzzy_matrix_transpose,fuzzy_matrix_compare)==True:
        return True
    else:
        return False

#start with conjugation 1 on a single fuzzy:

#R1=0,S1=0,X1=0,Y1=0,m_X1=0,m_Y1=0 fuzzy ==>conjugation property 1 doesn't hold:
R1=0;S1=0;X1=0;Y1=0;m_X1=0;m_Y1=0
print("conj property 1 on ",(R1,S1,X1,Y1,m_X1,m_Y1),"=",\
        conjugation_test_1(R1 , S1, X1, Y1, m_X1, m_Y1))
#however if we multiply it by 1j it does:
fuzzy_dict[(R1,S1,X1,Y1,m_X1,m_Y1)]=1j*fuzzy_dict[(R1,S1,X1,Y1,m_X1,m_Y1)]
print("conj property 1 on ",(R1,S1,X1,Y1,m_X1,m_Y1),"=",\
        conjugation_test_1(R1 , S1, X1, Y1, m_X1, m_Y1))
```

```
#run through all the fuzzys in fuzzy_dict and check conjugation property 1
#if it's not fulfilled multiply the corresponding fuzzy by 1j
for (R1,S1,X1,Y1,m_X1,m_Y1),fuzzy in list(fuzzy_dict.items()):
    if not conjugation_test_1(R1, S1, X1, Y1, m_X1, m_Y1)==True:
        fuzzy_dict[(R1,S1,X1,Y1,m_X1,m_Y1)]*=1j

#conjugation 2 on a single fuzzy:

R1=0.5;S1=0.5;X1=0.5;Y1=0.5;m_X1=-0.5;m_Y1=0.5
print("conj property 1 on ",(R1,S1,X1,Y1,m_X1,m_Y1),"=",\
        conjugation_test_1(R1 , S1, X1, Y1, m_X1, m_Y1))
print("conj property 2 on ",(R1,S1,X1,Y1,m_X1,m_Y1),"=",\
        conjugation_test_2(R1 , S1, X1, Y1, m_X1, m_Y1))

#however if we multiply Y_(R1,S1,X1,Y1,m_X1,m_Y1) by -1 it does:
fuzzy_dict[(R1,S1,X1,Y1,m_X1,m_Y1)]=-1*fuzzy_dict[(R1,S1,X1,Y1,m_X1,m_Y1)]
fuzzy_dict[(R1,S1,X1,Y1,-m_X1,-m_Y1)]=1*fuzzy_dict[(R1,S1,X1,Y1,-m_X1,-m_Y1)]

print("conj property 1 on ",(R1,S1,X1,Y1,m_X1,m_Y1),"=",\
        conjugation_test_1(R1 , S1, X1, Y1, m_X1, m_Y1))
print("conj property 2 on ",(R1,S1,X1,Y1,m_X1,m_Y1),"=",\
        conjugation_test_2(R1 , S1, X1, Y1, m_X1, m_Y1))

#run through all the fuzzys in fuzzy_dict and check conjugation property 2
#if it's not fulfilled we multiply Y_(R1,S1,X1,Y1,m_X1,m_Y1) by -1
for (R1,S1,X1,Y1,m_X1,m_Y1),fuzzy in list(fuzzy_dict.items()):
    if not conjugation_test_2(R1, S1, X1, Y1, m_X1, m_Y1)==True:
        fuzzy_dict[(R1,S1,X1,Y1,m_X1,m_Y1)]*=-1

#now lets check if all fuzzy fulfill the conjugation properties:

conj_test=True
for (R1,S1,X1,Y1,m_X1,m_Y1),fuzzy in list(fuzzy_dict.items()):
    if not conjugation_test_1(R1, S1, X1, Y1, m_X1, m_Y1)==True:
        conj_test=False
print("conjugation property 1=", conj_test)

conj_test=True
for (R1,S1,X1,Y1,m_X1,m_Y1),fuzzy in list(fuzzy_dict.items()):
    if not conjugation_test_2(R1, S1, X1, Y1, m_X1, m_Y1)==True:
        conj_test=False
        print((R1,S1,X1,Y1,m_X1,m_Y1))
print("conjugation property 2=", conj_test)


# still a phase choice of +1/-1 for each fuzzy
# it should be chosen such that the reduced matrix elements from
#the wigner eckhart theorem and from the formula match

```

```
# getting the fuzzy_red_mat_el directly from the wigner eckhart theorem
#first storing the vectors with their (R,S,X,Y,mx_my) eigenvalues

# 4d column vector with labels R=0.5, S=0, X=0.5, m_X=-0.5, Y=0, m_Y=0
vec0=np.array([0,1,0,0])

#X eigenvalue equations:
S_X3=np.dot(gen_dict_so3xso3["X3"],vec0)
S_X_square=np.dot(np.dot(gen_dict_so3xso3["X3"],gen_dict_so3xso3["X3"]),vec0)+ \
        np.dot(np.dot(gen_dict_so3xso3["X2"],gen_dict_so3xso3["X2"]),vec0)+ \
        np.dot(np.dot(gen_dict_so3xso3["X1"],gen_dict_so3xso3["X1"]),vec0)

#Y eigenvalue equations:
S_Y3=np.dot(gen_dict_so3xso3["Y3"],vec0)
S_Y_square=np.dot(np.dot(gen_dict_so3xso3["Y3"],gen_dict_so3xso3["Y3"]),vec0)+ \
        np.dot(np.dot(gen_dict_so3xso3["Y2"],gen_dict_so3xso3["Y2"]),vec0)+ \
        np.dot(np.dot(gen_dict_so3xso3["Y1"],gen_dict_so3xso3["Y1"]),vec0)

#SO(5) casimir square operator eigenvalue equation:

S_Casimir=np.zeros(4,dtype=complex)
for (g_i,g_j) in [(1,2),(1,3),(1,4),(1,5),(2,3),(2,4),(2,5),(3,4),(3,5),(4,5)]:
    S_Casimir+=np.dot(np.dot(gen_dict[(g_i,g_j)],gen_dict[(g_i,g_j)]), vec0)

#all eigenvalue equations are fulfilled
#the remaining vectors:
#R=0.5, S=0, X=0.5, m_X=0.5, Y=0, m_Y=0
vec1=np.array([1,0,0,0])
#R=0.5, S=0, X=0, m_X=0, Y=0.5, m_Y=0.5
vec2=np.array([0,0,1,0])
#R=0.5, S=0, X=0, m_X=0, Y=0.5, m_Y=-0.5
vec3=np.array([0,0,0,1])

vec_dict={(0.5,0,0.5,0,-0.5,0):vec0,(0.5,0,0.5,0,0.5,0):vec1,\
            (0.5,0,0,0.5,0,0.5):vec2,(0.5,0,0,0.5,0,-0.5):vec3}

#now evaluating the reduced matrix element using the wigner eckhart theorem
#and storing them in a dictionary
#{[(R1,S1,X1,Y1,m_X1,m_Y1),(R,S,X,Y,m_X,m_Y),\
#            (R',S',X',Y',m_X',m_Y')]:red_mat_el_we}

red_mat_el_we={}
for (R1,S1,X1,Y1,m_X1,m_Y1),fuzzy in fuzzy_dict.items():
    for (R_p,S_p,X_p,Y_p,m_X_p,m_Y_p),ket_vec in vec_dict.items():
        for (R,S,X,Y,m_X,m_Y),bra_vec in vec_dict.items():
            LHS=np.dot(bra_vec,np.dot(fuzzy,ket_vec))
            RHS=reduced_cc(R_p,S_p,X_p,Y_p,R1,S1,X1,Y1,R,S,X,Y,0)\
              *complex(CG(X_p, m_X_p, X1, m_X1, X, m_X).doit())\
              *complex(CG(Y_p, m_Y_p, Y1, m_Y1, Y, m_Y).doit())
```

```
            if LHS==0:
                red_mat_el_we[(R1,S1,X1,Y1,m_X1,m_Y1),(R,S,X,Y,m_X,m_Y),\
                        (R_p,S_p,X_p,Y_p,m_X_p,m_Y_p)]=0
            else:
                red_mat_el_we[(R1,S1,X1,Y1,m_X1,m_Y1),(R,S,X,Y,m_X,m_Y),\
                        (R_p,S_p,X_p,Y_p,m_X_p,m_Y_p)]=LHS/RHS

#observation: the value of the red_mat_el for a fixed (R1,S1,X1,Y1,m_X1,m_Y1),
#if nonzero doesn't depend on X,Y,mx,my,X_p,Y_P,mx_p,my_p

#pick out this value for each fuzzy and stor it in a smaller dict:

red_mat_el_we_small={}

for [(R1,S1,X1,Y1,m_X1,m_Y1),(R,S,X,Y,m_X,m_Y),(R_p,S_p,X_p,Y_p,m_X_p,m_Y_p)],\
    red_mat_el in red_mat_el_we.items():
    if not red_mat_el==0:
        red_mat_el_we_small[(R1,S1,X1,Y1,m_X1,m_Y1)]=red_mat_el

#now choose the +1/-1 prefactor such that phases of reduced matrix elements
#from wigner eckhart and formula match

for (R1,S1,X1,Y1,m_X1,m_Y1),fuzzy in fuzzy_dict.items():
    if not np.angle(red_mat_el_we_small[(R1, S1, X1, Y1, m_X1, m_Y1)])\
            ==np.angle(red_mat_el_formula[(R1, S1, X1, Y1, m_X1, m_Y1)]):
        fuzzy_dict[(R1,S1,X1,Y1,m_X1,m_Y1)]*=-1

#now recalculate the red_mat_el_we from the wigner eckhart theorem:
red_mat_el_we={}
for (R1,S1,X1,Y1,m_X1,m_Y1),fuzzy in fuzzy_dict.items():
    for (R_p,S_p,X_p,Y_p,m_X_p,m_Y_p),ket_vec in vec_dict.items():
        for (R,S,X,Y,m_X,m_Y),bra_vec in vec_dict.items():
            LHS=np.dot(bra_vec,np.dot(fuzzy,ket_vec))
            RHS=reduced_cc(R_p,S_p,X_p,Y_p,R1,S1,X1,Y1,R,S,X,Y,0)\
              *complex(CG(X_p, m_X_p, X1, m_X1, X, m_X).doit())\
              *complex(CG(Y_p, m_Y_p, Y1, m_Y1, Y, m_Y).doit())
            if LHS==0:
                red_mat_el_we[(R1,S1,X1,Y1,m_X1,m_Y1),(R,S,X,Y,m_X,m_Y),\
                        (R_p,S_p,X_p,Y_p,m_X_p,m_Y_p)]=0
            else:
                red_mat_el_we[(R1,S1,X1,Y1,m_X1,m_Y1),(R,S,X,Y,m_X,m_Y),\
                        (R_p,S_p,X_p,Y_p,m_X_p,m_Y_p)]=LHS/RHS

red_mat_el_we_small={}
for [(R1,S1,X1,Y1,m_X1,m_Y1),(R,S,X,Y,m_X,m_Y),(R_p,S_p,X_p,Y_p,m_X_p,m_Y_p)],\
    red_mat_el in red_mat_el_we.items():
    if not red_mat_el==0:
        red_mat_el_we_small[(R1,S1,X1,Y1,m_X1,m_Y1)]=red_mat_el
#all phases match !!!
```

```
def fusion_so5(R1,S1,X1,Y1,m_X1,m_Y1,R2,S2,X2,Y2,m_X2,m_Y2,\
                                    R3,S3,X3,Y3,m_X3,m_Y3,R,S,m):
    """returns the fusion coefficient F^(L3)_(L1,L2) with L=(R,S,X,Y,m_X,m_Y)
    R,S determine the dimension of the representation of the fuzzy harmonics
    the variable m labels the different reduced coupling coefficients
    if there is outer multiplicity present
    In this case the label "m" would have to be taken extra care of """
    phase1=(-1)**complex(1+R3-S3+m_X3+m_Y3-X1-Y1-X2-Y2)
    pref1=red_mat_el_formula[(R1,S1,X1,Y1,m_X1,m_Y1)]*\
            red_mat_el_formula[(R2,S2,X2,Y2,m_X2,m_Y2)]*\
            red_mat_el_formula[(R3,S3,X3,Y3,-m_X3,-m_Y3)]
    summe=0
    for (X1p,Y1p) in all_SO4(R, S):
        for (X2p,Y2p) in all_SO4(R, S):
            for (X3p,Y3p) in all_SO4(R, S):
                red_cc_prod=reduced_cc(R,S,X2p,Y2p,R1,S1,X1,Y1,R,S,X1p,Y1p,m)\
                            *reduced_cc(R,S,X3p,Y3p,R2,S2,X2,Y2,R,S,X2p,Y2p,m)\
                            *reduced_cc(R,S,X1p,Y1p,R3,S3,X3,Y3,R,S,X3p,Y3p,m)
                if not red_cc_prod==0:
                    for m_X1p in np.arange(-X1p,X1p+1):
                        for m_Y1p in np.arange(-Y1p,Y1p+1):
                            for m_X2p in np.arange(-X2p,X2p+1):
                                for m_Y2p in np.arange(-Y2p,Y2p+1):
                                    for m_X3p in np.arange(-X3p,X3p+1):
                                        for m_Y3p in np.arange(-Y3p,Y3p+1):
                                            pref2=np.sqrt((2*X1p+1)*\
                                                (2*X2p+1)*(2*X3p+1)*(2*Y1p+1)*\
                                                (2*Y2p+1)*(2*Y3p+1))
                                            phase2=(-1)**complex(X1p+X2p+X3p+\
                                                 Y1p+Y2p+Y3p+mx1p+mx2p+mx3p\
                                                 +my1p+my2p+my3p)
                                            j_prod=complex(wigner_3j(X2p,X1,X1p\
                                                    ,mx2p,mx1,-mx1p))\
                                                *complex(wigner_3j(X3p,X2,X2p\
                                                    ,mx3p,mx2,-mx2p))\
                                                *complex(wigner_3j(X1p,X3,X3p\
                                                    ,mx1p,-mx3,-mx3p))\
                                                *complex(wigner_3j(Y2p,Y1,Y1p\
                                                    ,my2p,my1,-my1p))\
                                                *complex(wigner_3j(Y3p,Y2,Y2p\
                                                    ,my3p,my2,-my2p))\
                                                *complex(wigner_3j(Y1p,Y3,Y3p\
                                                    ,my1p,-my3,-my3p))
                                            summe+=pref2*phase2*\
                                                    red_cc_prod*j_prod
                else:
                    summe+=0
    result=pref1*phase1*summe
    return result
```

```
#now onto the fusion coefficient testing
R=0.5;S=0;m=0
#first the multiplication of two arbitrary fuzzys
R1=1;S1=0;X1=1;Y1=0;m_X1=1;m_Y1=0
R2=1;S2=0;X2=1;Y2= 0;m_X2= -1;m_Y2= 0

fusion_prod1=np.dot(fuzzy_dict[(R1,S1,X1,Y1,m_X1,m_Y1)],\
                        fuzzy_dict[(R2,S2,X2,Y2,m_X2,m_Y2)])
fusion_summe1=0.5j*fuzzy_dict[(0.5,0.5,0,0,0,0)]-0.5j*fuzzy_dict[(0,0,0,0,0,0)]\
                +1j/np.sqrt(2)*fuzzy_dict[(1,0,1,0,0,0)]

#this means fusion_so5(1,0,1,0,1,0,1,0,1,0,-1,0,0.5,0.5,0,0,0,0) has to be -0.5j
R3=0.5;S3=0.5;X3=0;Y3=0;m_X3=0;m_Y3=0
print("fusion_cc_1=",np.round(fusion_so5(R1,S1,X1,Y1,m_X1,m_Y1,\
        R2,S2,X2,Y2,m_X2,m_Y2,R3,S3,X3,Y3,m_X3,m_Y3,R,S,m),15))
R3=1;S3=0;X3=1;Y3=0;mx3=0;my3=0
print("fusion_cc_2=",np.round(fusion_so5(R1, S1, X1, Y1, mx1, my1,\
        R2,S2,X2,Y2,m_X2,m_Y2,R3,S3,X3,Y3,m_X3,m_Y3,R,S,m),15))
R3=0;S3=0;X3=0;Y3=0;mx3=0;my3=0
print("fusion_cc_3=",np.round(fusion_so5(R1, S1, X1, Y1, mx1, my1,\
        R2,S2,X2,Y2,m_X2,m_Y2,R3,S3,X3,Y3,m_X3,m_Y3,R,S,m),15))

#another example of multiplication of two fuzzies
R1=1;S1=0;X1=1;Y1=0;m_X1=0;m_Y1=0
R2=1;S2=0;X2=0.5;Y2= 0.5;m_X2=0.5;m_Y2=0.5

fusion_prod2=np.dot(fuzzy_dict[(R1,S1,X1,Y1,m_X1,m_Y1)],\
                fuzzy_dict[(R2,S2,X2,Y2,m_X2,m_Y2)])
fusion_summe2=0
for (R3,S3,X3,Y3,m_X3,m_Y3),fuzzy3 in fuzzy_dict.items():
    fusion_summe2+=fusion_so5(R1,S1,X1,Y1,m_X1,m_Y1,R2,S2,X2,Y2,m_X2,m_Y2\
                        ,R3,S3,X3,Y3,m_X3,m_Y3,R,S,m)*fuzzy3

#test all fusion coefficients:
fusion_test=True
for (R1,S1,X1,Y1,m_X1,m_Y1),fuzzy1 in fuzzy_dict.items():
    for (R2,S2,X2,Y2,m_X2,m_Y2),fuzzy2 in fuzzy_dict.items():
        prod=np.dot(fuzzy1,fuzzy2)
        summe=0
        for (R3,S3,X3,Y3,m_X3,m_Y3),fuzzy3 in fuzzy_dict.items():
            summe+=fusion_so5(R1,S1,X1,Y1,m_X1,m_Y1,R2,S2,X2,Y2,m_X2,m_Y2\
                    ,R3,S3,X3,Y3,m_X3,m_Y3,R,S,m)*fuzzy3
        if not np.allclose(prod, summe,rtol=1e-4,atol=1e-7)==True:
            fusion_test=False
            print([(R1,S1,X1,Y1,m_X1,m_Y1),(R2,S2,X2,Y2,m_X2,m_Y2)],"wrong")
        else:
            print([(R1,S1,X1,Y1,m_X1,m_Y1),(R2,S2,X2,Y2,m_X2,m_Y2)],"correct")
print("complete fusion testing=",fusion_test)
#test succesful !!!
```

Bibliography

[kil19] kilian. *Graph theory 101*. 2019. URL: https://bionerdnotes.wordpress.com/2019/08/14/graph-theory-101/ (visited on 08/06/2023).

[Zee16] Anthony Zee. *Group theory in a Nutshell for Physicists*. Princeton, New Jersey 08540: Princeton University Press, 2016. ISBN: 978-0-691-16269-0.

[Czy08] Gerd Czycholl. *Theoretische Festkörperphysik. Von den klassischen Modellen zu modernen Forschungsthemen*. Deutsch. Springer-Verlag Berlin Heidelberg, 2008. ISBN: 978-3-540-74789-5.

[Geo00] Howard Georgi. *Lie Algebras In Particle Physics: from Isospin To Unified Theories*. CRC Press, 2000. DOI: 10.1201/9780429499210.

[Ish+10] Hajime Ishimori et al. "Non-Abelian Discrete Symmetries in Particle Physics". In: *Progress of Theoretical Physics Supplement* 183 (2010), pp. 1–163. DOI: 10.1143/ptps.183.1.

[Lom59] Justin Paul Lomont. *Applications of Finite Groups*. Academic Press, 1959. ISBN: 978-1-4832-3132-7. DOI: 10.1016/C2013-0-12379-3.

[GL12] Walter Grimus and Patrick Otto Ludl. "Finite flavour groups of fermions". In: *Journal of Physics A: Mathematical and Theoretical* 45.23 (May 2012), p. 233001. DOI: 10.1088/1751-8113/45/23/233001.

[Tre07] Hans-Werner Trebin. *Gruppentheoretheoretische Methoden in der Physik. Skript zur Vorlesung*. Deutsch. 2007.

[BC79] Benjamin Baumslag and Bruce Chandler. *Gruppentheorie [Theory and Problems of Group Theory]. Theorie und Anwendung*. Deutsch. Trans. by Bernhard Thomas. McGraw-Hill Book Company GmbH, 1979. ISBN: 0-07-092026-6.

[SK11] Yoland Savriama and Chris Klingenberg. "Beyond bilateral symmetry: Geometric morphometric methods for any type of symmetry". In: *BMC Evolutionary Biology* 11.280 (Sept. 2011). DOI: 10.1186/1471-2148-11-280.

[Böh11] Manfred Böhm. *Lie-Gruppen und Lie-Algebren in der Physik*. German. Springer-Verlag Berlin Heidelberg, 2011. ISBN: 978-3-642-20378-7.

[DL15] Oliver Deiser and Caroline Lasser. *Erste Hilfe in Linearer Algebra*. Deutsch. Springer-Verlag Berlin Heidelberg, 2015. ISBN: 978-3-642-41626-2. URL: https://www.aleph1.info/?call=Puc&permalink=ela1.

[Fis09] G. Fischer. *Lineare Algebra*. Springer-Verlag Berlin Heidelberg, 2009. ISBN: 9783834809964. DOI: 10.1007/978-3-658-03945-5.

[Lug21] Gabriel Lugo. *Differential Geometry in Physics*. University of North Carolina Wilmington, 2021. ISBN: 9781469669250. DOI: 10.5149/9781469669267.

[Sch80] Bernard Schutz. *Geometrical Methods of Mathematical Physics*. Cambridge University Press, 1980. ISBN: 978-0-521-29887-2. DOI: 10.1017/CBO9781139171540.

[RS22] Joel W. Robbin and Dietmar A. Salamon. *Introduction to Differential Geometry*. Springer Berlin Heidelberg, 2022. ISBN: 9783662643396. DOI: 10.1007/978-3-662-64340-2.

[TN08] TN. *File:Tangentialvektor.svg*. 2008. URL: https://commons.wikimedia.org/wiki/File:Tangentialvektor.svg (visited on 05/23/2023).

[Che15] Evan Chen. *Constructing the Tangent and Cotangent Space*. 2015. URL: https://blog.evanchen.cc/2015/10/04/constructing-the-tangent-and-cotangent-space/ (visited on 05/24/2023).

[Jev11] Nadir Jevanjee. *An Introduction to Tensors and Group Theory for Physicists*. Springer Berlin Heidelberg, 2011. ISBN: 978-0-8176-4714-8. DOI: 10.1007/978-0-8176-4715-5.

[Sch13] Franz Schwabl. *Quantenmechanik: Eine Einführung*. Springer-Verlag Berlin Heidelberg, 2013. ISBN: 9783662096291. DOI: 10.1007/978-3-540-73675-2.

[Hal00] Brian C. Hall. *An Elementary Introduction to Groups and Representations*. 2000. URL: https://arxiv.org/pdf/math-ph/0005032.pdf.

[CSM10] M. A. Caprio, K. D. Sviratcheva, and A. E. McCoy. "Racah's method for general subalgebra chains: Coupling coefficients of SO(5) in canonical and physical bases". In: *Journal of Mathematical Physics* 51.9 (Sept. 2010), p. 093518. DOI: 10.1063/1.3445529.

[Rac49] Giulio Racah. "Theory of Complex Spectra. IV". In: *Phys. Rev.* 76 (9 Nov. 1949), pp. 1352–1365. DOI: 10.1103/PhysRev.76.1352.

[VMK88] D.A. Varshalovich, A.N. Moskalev, and V.K. Khersonskii. *Quantum Theory of Angular Momentum*. World Scientific, 1988. DOI: 10.1142/0270.

[MO03] Julieta Medina and Denjoe O'Connor. "Scalar field theory on fuzzyiS/isup4/sup". In: *Journal of High Energy Physics* 2003.11 (Nov. 2003), pp. 051–051. DOI: 10.1088/1126-6708/2003/11/051.

[Mad92] John Andrew Madore. "The fuzzy sphere". In: *Classical and Quantum Gravity* 9.1 (Jan. 1992), p. 69. DOI: 10.1088/0264-9381/9/1/008.

[Ydr01] Badis Ydri. “Fuzzy Physics”. PhD thesis. Syracuse University, Oct. 2001. URL: https://www.researchgate.net/publication/47454379_Fuzzy_Physics.

[Mat23] Wolfram Mathworld. *Wigner 3j-Symbol*. 2023. URL: https://mathworld.wolfram.com/Wigner3j-Symbol.html (visited on 10/13/2023).

[GV18] Aleix Gimenez Grau and Matthias Volk. “One-Loop One-Point Functions in -Non-Supersymmetric AdS/dCFT”. PhD thesis. University of Kopenhagen, Aug. 2018.

[Hec65] Karl Hecht. “Some simple R5 Wigner coefficients and their application”. In: *Nuclear Physics* 63.2 (1965), pp. 177–213. ISSN: 0029-5582. DOI: 10.1016/0029-5582(65)90338-X.