Calculation of Clebsch-Gordan coefficients via Weyl group symmetry

Lukas Everding

München 2011

Calculation of Clebsch-Gordan coefficients via Weyl group symmetry

Lukas Everding

Bachelorarbeit
an der Fakultät für Physik
der Ludwig–Maximilians–Universität
München

vorgelegt von
Lukas Everding
aus Hannover

München, den 12.08.2011

Gutachter: Prof. Dr. Jan von Delft
Tag der mündlichen Prüfung: 02.09.2011

Contents

1 Introduction

1.1 Aims

The aim of this Bachelor thesis was to write a C++ library that allows the calculation of Clebsch-Gordan coefficients for the decomposition of an arbitrary coupling of irreducible representations (irreps) of SU(N) into a direct sum of irreps. A program that computes these coefficients has recently been developed[1]. However, the code described here uses an improved algorithm that accelerates the calculation by a factor of order $N!$. To achieve this, it exploits a symmetry called Weyl-group symmetry.

1.2 Clebsch-Gordan coefficients

Clebsch-Gordan coefficients are required for the decomposition of a tensor *product* of representations of two irreducible representations $\mathbb{V}^S$ and $\mathbb{V}^{S'}$ into a *sum* of irreducible representations. In physics one usually encounters them in atomic physics, e.g. spin-orbit-interaction or coupling of spins and angular momenta. In this case, the Clebsch-Gordan coefficients for SU(2) are needed. They are the expansion coefficients for a change from a tensor product basis to a coupled basis.

$$|M'', \alpha\rangle = \sum_{M,M'} C_{M,M'}^{M'',\alpha} |M \otimes M'\rangle \tag{1.1}$$

Here M'' is a state of the new basis. It is a linear combination of old states which are given in the tensor product basis of the irreps M and M'. The coefficients $C_{M,M'}^{M'',\alpha}$ in the sum are the Clebsch-Gordan coefficients that are to be determined. The index α describes the outer multiplicity of the state, which does not occur in SU(2), but has to be considered for general SU(N) and will be explained in detail later.
Systems with higher SU(N) symmetry appear for example in the standard model (SU(3)) or in quantum impurity[2] models of solid states. Such systems can be simplified significantly using the Wigner-Eckart theorem. This is where Clebsch-Gordan coefficients are required. They are essential to exploit the Wigner-Eckart theorem. This theorem says that the Hamiltonian is block-diagonal and the Clebsch-Gordan coefficients determine which matrix elements do not vanish. Furthermore, by introducing irreps as new quantum numbers to the states, the dimension of the Hamiltonian can be reduced. It is necessary to keep only one representative state of the respective carrier space for each irrep that appears, but the full Hamiltonian can still be reconstructed from the reduced one. This drastically speeds up numerical calculations, because the dimension of the Hamiltonian can be reduced significantly. To use the theorem it is crucial to know the Clebsch-Gordan coefficients explicitly. For higher irrep dimensions and higher SU(N) the computation of these coefficients becomes very time consuming. Thus, we propose a new algorithm to decrease computation times. It uses the fact that SU(N)carrier space states can be classified according to $N!$ so called Weyl chambers. Those Weyl chambers

can be mapped onto each other by elements from the Weyl group. Explicit calculation of the Clebsch-Gordan coefficients is required for *only one* of those chambers. All other coefficients can be obtained by very fast symmetry operations.

1.3 Outline

We will start by giving a brief introduction into the mathematical framework of the problem and fixing notation.Then we provide a short revision of the old algorithm for the calculation of Clebsch-Gordan coefficients for SU(N) which is still required as a basis for our proposed algorithm. The main part in chapters 3 and 4 is dedicated to the detailed description of how the new algorithm works and how it was implemented. The C++ code can be found in chapter 6.

2 Theoretical background

This chapter is following the line of argument given by Alex et al. in the papers [1] and [3]. We will give a short recapitulation of the definitions that are used. For further details, please see the references.

2.1 SU(N) group and corresponding Lie algebra

SU(N) is the *special unitary group* of degree N. It consists of all $N \times N$ unitary matrices with determinant 1 and the operation matrix multiplication and often appears in physics, when the system has a continuous symmetry. Working with the group SU(N) directly is comparatively difficult as there is no easy parametrization of the group elements that would suit our purposes. We work with the corresponding algebra, instead. This Lie algebra, denoted $\mathfrak{su}(N)$, has the properties of a vector space and consists of all traceless anti-Hermitian $n \times n$ matrices. Since the group SU(N) is simply connected, many properties of the algebra and the group are identical; in particular, they have the same Clebsch-Gordan coefficients.
A basis, that is suitable for our purposes, is given by

$$J_z^{(l)} = \frac{1}{2}(E^{l,l} - E^{l+1,l+1}), \tag{2.1a}$$

$$J_+^{(l)} = E^{l,l+1}, \tag{2.1b}$$

$$J_-^{(l)} = E^{l+1,l} \tag{2.1c}$$

and their commutation relations

$$E^{p,q} = [J_-^{p-1}, [J_-^{p-2}, ...[J_-^{q+1}, J_-^{q}]...]] \qquad \text{for} \quad p > q, \tag{2.2a}$$

$$E^{p,q} = [J_+^{p}, [J_+^{p+1}, ...[J_+^{q-2}, J_+^{q-1}]...]] \qquad \text{for} \quad p < q. \tag{2.2b}$$

with the single-entry matrices $E^{m,n} = E^{m,n}_{a,b} = \delta_{ma}\delta_{nb}$. That way all traceless anti-Hermitian matrices are accessible. As one can see, every element of the basis can be found if the $J_\pm^{(l)}$ and $J_z^{(l)}$are known. The matrices $J_\pm^{(l)}$ will later be understood as raising and lowering operators and play an important role for the calculation of the Clebsch-Gordan coefficients.

2.2 Irreps, states and weights

2.2.1 Labeling of irreps and states

The task is to decompose a tensor product of two irreducible representations (irreps) of the SU(N) group. To label the irreps and the basis of their carrier spaces, we use a labeling scheme proposed by Gelfand and Tsetlin[4],[5]: each irrep from SU(N) is defined via an irrep weight (i-weight), which is a sequence of N non-increasing integers (with N from SU(N))

$$S = (m_{1,N}, m_{2,N} \dots m_{N,N}). \tag{2.3}$$

If the elements of two i-weights differ only by a constant value, then they describe the same irrep ($(m_{1,N}, m_{2,N} \dots m_{N,N}) \mathrel{\widehat{=}} (m_{1,N} + c, m_{2,N} + c \dots m_{N,N} + c)$, $c \in \mathbb{Z}$). This fact is used to "normalize" the i-weights by defining $m_{N,N} = 0$.
A basis of the carrier space of an irrep can be written as so called Gelfand-Tsetlin-patterns (GT-pattern). These patterns are orthonormal basis states of the carrier space (for two states described by patterns p and p': $\langle p|p'\rangle = \delta_{p,p'}$) and have the following form:

$$M = \begin{pmatrix} m_{1,N} & m_{2,N} & \dots & m_{N,N} \\ & m_{1,N-1} & \dots & m_{N-1,N-1} \\ & \ddots & & \iddots \\ & & m_{1,2} \quad m_{2,2} & \\ & & m_{1,1} & \end{pmatrix} \tag{2.4}$$

The first row of the pattern is the respective i-weight. In order to be valid a pattern has to satisfy the betweenness condition:

$$m_{k,l} \leq m_{k,l-1} \leq m_{k+1,l} \tag{2.5}$$

which limits the number of allowed pattern per i-weight. The dimension of the carrier space corresponds to the number of valid patterns. It can be calculated directly via:

$$\dim(S) = \prod_{1 \leq k \leq k' \leq N} \left(1 + \frac{m_{k,N} - m_{k',N}}{k' - k}\right) \tag{2.6}$$

Pattern weights

Each Gelfand-Tsetlin pattern M, that is a basis state of an irrep, is also assigned a weight $w^M = (w_1^M, w_2^M, ..., w_N^M)$, that is a sequence of integers. It consists of the difference between the row sums of the pattern

$$w_l^M = \sum\nolimits_{k=1}^{l} m_{k,l} - \sum\nolimits_{k=1}^{l-1} m_{k,l-1} \tag{2.7}$$

and is called pattern weight (p-weight). However, a p-weight does not define a basis state unambiguously, because different weights can have the same p-weight as long as their row sums are identical, e.g.

$$w\left(\begin{pmatrix} 2 & 1 & 0 \\ & 1 & 1 \\ & & 1 \end{pmatrix}\right) = w\left(\begin{pmatrix} 2 & 1 & 0 \\ & 2 & 0 \\ & & 1 \end{pmatrix}\right) = (1,1,1) \tag{2.8}$$

The number of different states that have the same p-weight is called the *inner multiplicity* of that p-weight.

2.2.2 Decomposition of an irrep product

Littlewood Richardson rule

The (hard to proof, though easy to apply) Littlewood-Richardson rule states how to decompose the product of two irreps into a direct sum of irreps. We will not give a proof here, but explain merely how it works:
Begin by writing down all GT-patterns M of one of the coupled irreps S. Now construct an auxiliary pattern B from each GT-pattern. Its entries $b_{k,l}$ are $b_{k,l} = m_{k,l} - m_{k,l-1}$ with $m_{k,0} \equiv 0$. Now, take the i-weight of the other irrep S' and modify its entries according to the following rules:

1. Take the i-weight of the other irrep S': $(m'_1, \ldots, m'_N)$
2. Add $b_{k,l}$ to m'_l for each row k, starting with $k = 1$ and $l = N$ (ascending k, descending l)
3. If the i-weight violates the condition that its entries must be non-increasing at any step, discard the i-weight
4. If all entries of B have been taken care of, the resulting irrep appears in the decomposition of the coupling

Note that an irrep can appear more than once in a decomposition. Therefore, we label the irreps with an additional index α which we call *outer multiplicity*.

2.3 Raising and lowering operator

The action of a raising (or lowering) operator $J_{\pm}^{(l)}$ (defined via the single entry matrices) on a GT-pattern produces a linear combination of the patterns that differ from the original pattern by exactly ± 1 in exactly one entry in row l[6]. The linear combination contains only patterns that are valid. It can be thought of as addition of

$$M^{\pm} = \begin{pmatrix} 0 & 0 & \ldots & 0 \\ & 0 & \ldots & 0 \\ & \ddots & & \iddots \\ & & \pm 1^{k,l} & \\ & 0 & & 0 \\ & & 0 & \end{pmatrix} \tag{2.9}$$

for every k. (plus sign for raising operator, minus sign for lowering operator). All patterns in this linear combination have the same i-weight and p-weight. The p-weight differs from the p-weight of the original pattern only on two positions, since it is constructed by the differences of the row sums: w_l^M is raised by 1 and w_{l+1}^M is lowered by 1 for a raising operator and vice versa for a lowering operator.

$$J_{\pm}^{(l)}(w_1, w_2, \ldots, w_l, w_{l+1}, \ldots, w_N) = (w_1, w_2 \ldots, w_l \pm 1, w_{l+1} \mp 1, \ldots, w_N) \tag{2.10}$$

So the raising/lowering operator just modifies two neighboring integers in the p-weight. The action of a general single entry matrix on the p-weight is

$$E^{m,n}(w_1, w_2, \ldots, w_m, \ldots, w_n, \ldots, w_N) = (w_1, w_2 \ldots, w_m + 1, \ldots, w_n - 1, \ldots, w_N) \quad (2.11)$$

The coefficients in front of the patterns in the linear combination are given by

$$\langle M - M^{k,l}|J_-^{(l)}|M\rangle = \left(-\frac{\prod_{k'=1}^{l+1}(m_{k',l+1} - m_{k,l} + k - k' + 1)\prod_{k'=1}^{l-1}(m_{k',l-1} - m_{k,l} + k - k')}{\prod_{\substack{k'=1 \\ k'\neq k}}^{l}(m_{k',l} - m_{k,l} + k - k' + 1)(m_{k',l} - m_{k,l} + k - k')} \right)^{\frac{1}{2}} \quad (2.12)$$

for the raising operator and

$$\langle M + M^{k,l}|J_+^{(l)}|M\rangle = \left(-\frac{\prod_{k'=1}^{l+1}(m_{k',l+1} - m_{k,l} + k - k')\prod_{k'=1}^{l-1}(m_{k',l-1} - m_{k,l} + k - k' - 1)}{\prod_{\substack{k'=1 \\ k'\neq k}}^{l}(m_{k',l} - m_{k,l} + k - k')(m_{k',l} - m_{k,l} + k - k' - 1)} \right)^{\frac{1}{2}} \quad (2.13)$$

for the lowering operator.
Equation (2.14) shows an example of the raising operator acting on a basis state of a SU(3) irrep with i-weight $(2, 1, 0)$:

$$J_+^{(2)} \begin{pmatrix} 2 & 1 & 0 \\ & 1 & 0 \\ & & 0 \end{pmatrix} = \alpha \begin{pmatrix} 2 & 1 & 0 \\ & 2 & 0 \\ & & 0 \end{pmatrix} + \beta \begin{pmatrix} 2 & 1 & 0 \\ & 1 & 1 \\ & & 0 \end{pmatrix}. \quad (2.14)$$

The p-weight $p = (0, 1, 2)$ is changed to $p' = (0, 2, 1)$. β equals zero because the pattern violates the betweenness condition and is therefore invalid. α can be determined from equation (2.12).

2.3.1 Weight diagrams

A complete set of p-weights belonging to all states of a certain irrep can be visualized by representing the p-weights as points in an $N-1$-dimensional coordinate system. The p-weights have to obey the relation

$$\sum_i w_i = \sum_j m_{j,N} \quad (2.15)$$

, i.e. the sum over all entries in the p-weight has to be equal to the sum over the entries in the respective i-weight. Thus, one loses one degree of freedom of the p-weight entries, making them visualizable in $N-1$ dimensional space. A possible (non-unique) choice for the position vector of a p-weight is given by

$$W_z = \left(\frac{1}{2}(w_1^M - w_2^M), \frac{1}{2}(w_2^M - w_3^M), \ldots, \frac{1}{2}(w_{N-1}^M - w_N^M) \right) \quad (2.16)$$

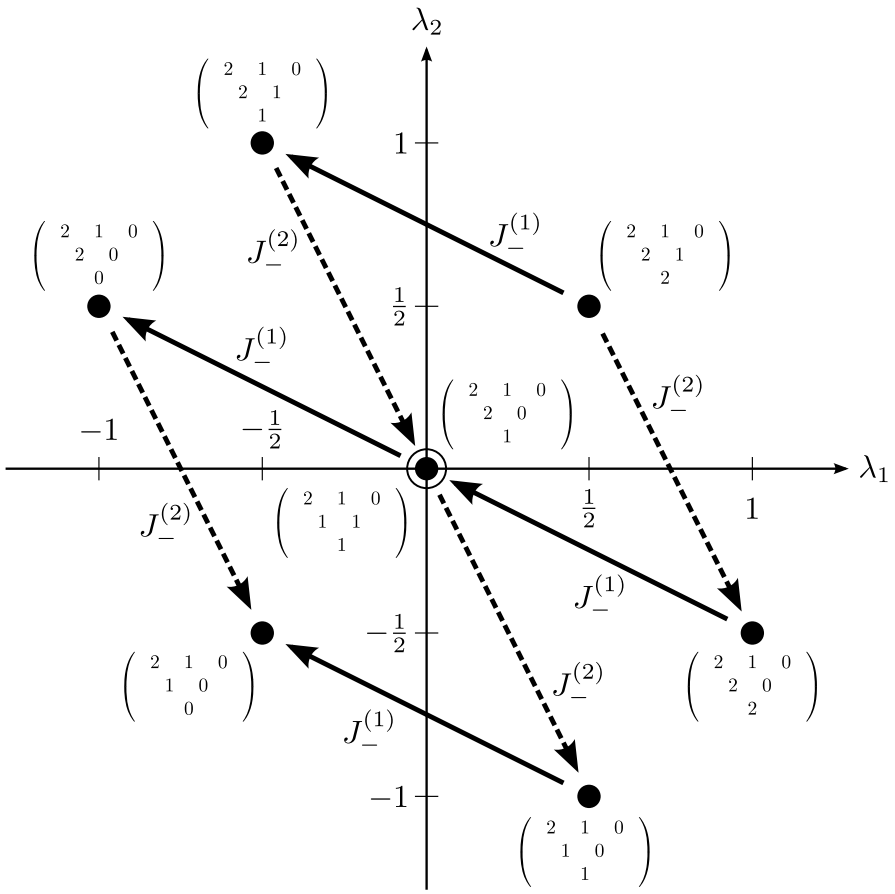

Figure 2.1: A SU(3) weight diagram with i-weight (2,1,0). Action of the lowering operators shown as arrows. The circle around the middle dot indicates that two states belong two this p-weight (Courtesy of Arne Alex)

After assigning each p-weight a vector with this equation, one can think of them as points in a *weight space*. Note that each position can be occupied by multiple states, and will be if the inner multiplicity of the p-weight is larger than 1.

2.4 Weyl group

The Weyl group for SU(N) is isomorphic to the group of permutations. The action of an element from the Weyl group can be understood as a permutation of the entries in a p-weight. The weight space of the p-weights can be divided in separate *Weyl chambers* by Weyl borders: a Weyl border is defined via a set of p-weights that is not changed if you apply a certain permutation from the Weyl group to it. So at least two entries have to be equal. P-weights containing only different integers cannot lie on a Weyl border, because there is no permutation that leaves the weight invariant. A border forms a hyper plane in the weight space. The borders belonging to every permutation taken together divide the space of weights in $N!$ *Weyl chambers* as there are $N!$ different permutations in the Weyl group. Note that you

can also define a Weyl border if no p-weight lies on it. In that case, it just runs between weights. Each Weyl chamber consists of different p-weights. The elements of the Weyl group map each p-weight in a Weyl chamber onto another p-weight in a different Weyl chamber. For our calculation of the Clebsch-Gordan coefficients we focus on one specific chamber, the so-called *dominant* Weyl chamber. The remaining coefficients can be obtained via symmetry operations.
P-weights that fulfill the following condition lie in the dominant Weyl chamber:

$$w_1 \geq w_2 \geq ... \geq w_n \tag{2.17}$$

Equation (2.17) is the definition of the dominant Weyl chamber. It can be mapped onto all other chambers via an application of a permutation what will be exploited later. Note that you can also define the Weyl chambers as sets of states. We do not distinguish between the Weyl chamber of p-weights and the Weyl chamber of states, because if one set is known the other one can be constructed.

2.4.1 Permutations

A permutation is a rearranging of the elements of an ordered set. It can be written as a row vector that maps every element from its original position to a new one, i.e. $(\sigma(1), \sigma(2), \sigma(3)...\sigma(N))$. $\sigma(j)$ is the number of the position to which the jth element is shifted. The dimension of a vector representing a permutation and the cardinality of the reordered set must be the same, as each element has to be placed somewhere. A way of writing down the application of a permutation is by a $2\times N$-matrix with the set in the first row and the permutation in the second, such that it becomes easily visible where each element is mapped to. The result is again a row vector with permuted elements. The permutation $\pi = (2, 3, 5, 1, 4)$ applied to a weight $w = (3, 3, 2, 2, 0)$ maps for example the 3 from position 1 to position 2, the 3 from position 2 to position 3, the 2 from position 3 to position 4 and so on:

$$\begin{pmatrix}\text{weight}\\ \text{permutation}\end{pmatrix} = \pi(w) = \begin{pmatrix}3 & 3 & 2 & 2 & 0\\ 2 & 3 & 5 & 1 & 4\end{pmatrix} = \begin{pmatrix}2 & 3 & 3 & 0 & 2\end{pmatrix} = \begin{pmatrix}\text{result}\end{pmatrix} \tag{2.18}$$

However, if you want to apply a permutation to a state given as GT patterns, it is necessary to find a representation matrix for the permutation. The shape of this matrix depends on the SU(N) and the irrep to which the basis state belongs. We will explain how this is done in the next chapter.
There are two important subsets of permutations regarding p-weights:

1. the *stabiliser*. It contains all permutations that do not change a given weight: $\beta(w) = w$, for any β from the stabiliser of p-weight w.

2. the *connectors*. A connector is the lexicographically smallest permutation that maps one weight onto another. Since there may be more than one permutation that connects two weights, we choose the smallest one as representative of those permutations. Any permutation that is not element of the connectors can be written as unique composition of a permutation of the connectors and the stabiliser: $\pi = \tilde{\kappa}\beta$ with a connector $\tilde{\kappa}$ and β from the stabiliser.

A permutation that swaps only the position of two elements and leaves the rest invariant is called a transposition. When denoting transpositions, we will only write down the elements that interchange positions, e.g. the transposition $(1, 2, 4, 3, 5)$ will be denoted as $(4, 3)$ (or equivalently $(3, 4)$).

2.4.2 Weyl basis

For the application of the Weyl group symmetry, we have to modify the labeling scheme. The states are now labeled by two labels: a pattern D from the dominant Weyl chamber and a connector κ, that permutes the pattern to the chamber that it belongs to: $|\kappa, D\rangle$. The patterns that lie in the dominant Weyl chamber are labeled with the identity permutation $|\mathbb{1}, D\rangle$.

2.4.3 Action of raising and lowering operator in the Weyl basis

We know how $J^{(l)}{}_{\pm}$ act on a pattern in the GT basis. Now we want to work out how they act on a state given in the new basis. So we pull the permutation in front of the operator to evaluate the action of it on a GT state:

$$J_{\pm}|\kappa, D\rangle = \kappa J'_{\pm}|\mathbb{1}, D\rangle, \qquad J'_{\pm} = \kappa^{-1} J_{\pm} \kappa. \tag{2.19}$$

$j'_{\pm}$ can be explicitly calculated if the representation matrices of the permutation κ and the $J_{\pm}$ are known. $J'_{\pm}$ is found to be a matrix that is identical to $J_{\pm}$ except for permutations of rows and columns. Thus it too is the representation matrix of a single entry matrix. Applying this to a state of the dominant Weyl chamber $|\mathbb{1}, D\rangle$ yields a linear combination of states with a changed p-weight according to (2.11). In general, these states do not lie in the dominant Weyl chamber. In order to describe them with our labelling scheme, we have to permute them back by inserting a connector and its inverse $\kappa'^{-1}\kappa'$.

$$J^{(l)}_{\pm}|\kappa, D\rangle = \kappa\kappa'^{-1} \left(\kappa' \kappa^{-1} J^{(l)}_{\pm} \kappa |\mathbb{1}, D\rangle \right). \tag{2.20}$$

The weight in brackets lies per construction in the dominant Weyl chamber. $\kappa\kappa'^{-1}$ can be factorized in a connector and a permutation from the stabiliser.

3 Algorithm

The program discussed in this thesis contains newly written code as well as reused code from the old program[1]. In this chapter, we focus on the parts of the code that are genuinely new. For parts that have already been used and just underwent minor changes, please see the description of the old algorithm.

3.1 General outline

In order to calculate the Clebsch-Gordan coefficients for the tensor product decomposition of two given irreps S and S', we first determine all irreps $\{S''\}$ into which the irrep product decomposes. Then we follow this scheme for every arising irrep (referred to as S'' in the following description):

1. Determine the Clebsch-Gordan coefficients of the highest weight state
2. Use the lowering operator to calculate the Clebsch-Gordan coefficients in the dominant Weyl chamber
3. Determine for each state in the dominant Weyl chamber which permutations are connectors
4. Map the states of the dominant Weyl chamber and their coefficients to all other chambers with the connectors

3.2 Irrep product decomposition

The algorithm of the tensor product decomposition is a direct implementation of the Littlewood-Richardson rule. It had already been used in the old program and was just adapted to the new code.

3.3 Highest weight state

To calculate the coefficients in the dominant Weyl chamber we start with the highest weight state of S'': $|\mathbb{1}, H'', \alpha\rangle$. Every raising operator $J_+^{(l)}$, $l = 1 \dots N-1$, applied to it gives zero. We use this fact by applying these operators to the decomposition equation of the highest weight state.

$$J_+^{(l)}|\mathbb{1}, H'', \alpha\rangle = J_+^{(l)} \sum_{\substack{D,D' \\ \kappa,\kappa'}} C^{H'',\alpha}_{\kappa,D;\kappa',D'} |\kappa, D\rangle \otimes |\kappa', D'\rangle \tag{3.1}$$

The left hand side gives zero and the right hand side is evaluated as described in chapter (2.4.3). The summation on the right hand side can be restricted to coupling of patterns that obey the relation for the p-weights.

$$p(H'') \neq p(D) + p(D') \rightarrow C^{H'',\alpha}_{\kappa,D;\kappa',D'} = 0 \tag{3.2}$$

If the relation is not fulfilled, the Clebsch-Gordan coefficients are zero (this relation is a generalisation of the well-known SU(2) case, where this relation implies that the sum of spin z-components of the independent spins has to be equal to the z-component of the spin in the new system. The condition states that the elementwise addition of the p-weight entries of the coupled states has to be equal to the p-weight of the highest state). The relation is checked for every state, i.e. every possible combination of a state in the dominant Weyl chamber with a permutation. Those states are stored in a list. Then the raising operators are applied. We obtain a set of irrep product state sums, where each state has a coefficients according to(2.13), multiplied with Clebsch-Gordan coefficients of the highest weight state. This is a set of homogeneous linear equations. Each coefficient in front of the states has to vanish independently from the other ones since the states are linearly independent. That determines the Clebsch-Gordan coefficients for the decomposition of the highest weight state (together with the normalisation condition $\sum_{\kappa,\kappa',D,D'}(C^{H'',\alpha}_{\kappa,D,\kappa',D'})^2 = 1$ and up to a sign choice). To solve the system we determine the nullspace of the matrix. The amount of linearly independent solutions is equal to the outer multiplicity of S''[7].

3.4 Calculation the Clebsch-Gordan coefficients in the dominant Weyl chamber

After we got the Clebsch-Gordan coefficients of the highest weight state, we can calculate the coefficients of all other states in the dominant Weyl chamber. To explain how this is done, we first define 'parent states': a parent state to a state $|\mathbb{1}, D''\rangle$ in the dominant Weyl chamber is a state that generates $|\mathbb{1}, D''\rangle$ if a $J^{(l)}_-$ is applied to it. Since the action of a single entry matrix on a p-weight is known (cf. (2.11)) and the lowering operator corresponds to such a matrix (cf. (2.1a)) one only has to check if the subtraction of the states' p-weights yields $w_{parent} - w_{|\mathbb{1},D''\rangle} = \Delta w = (\ldots, w_l + 1, w_{l+1} - 1, \ldots)$.
To calculate the coefficients, we iterate over all states in the dominant Weyl chamber[7] . If a states has not been visited yet, we search all parent states of it. Applying the right lowering operators to each of them generates a set of linear equations for the Clebsch-Gordan coefficients of $|\mathbb{1}, D''\rangle$ which is solvable assuming that the Clebsch-Gordan coefficients of the parent states are known. In fact, it can happen that we get an overdetermined system, yet it has to be consistently solvable. This is done by the method of least squares. In general, it minimizes $||Ax - b||$ for a matrix vector equation $Ax = b$ which gives zero in our case, since a solution exists. The equations for parent state $|\kappa'', D''\rangle$ are found by:

$$J^{(l)}_-|\kappa'', D''\rangle = \sum_{D''} \alpha_{D''}|\mathbb{1}, D''\rangle = \sum_{D''} \alpha_{D''} \sum_{\kappa,D;\kappa',D',} C^{D''}_{\kappa,D;\kappa',D'}|\kappa, D \otimes \kappa', D'\rangle \tag{3.3}$$

with $\alpha_{D''}$ from (2.12). You can also expand the parent state first:

$$\begin{aligned} J_-^{(l)}|\kappa'', D''\rangle = J_-^{(l)} \left(\sum_{\kappa,D;\kappa',D,} C^{\kappa'',D''}_{\kappa,D;\kappa',D'} |\kappa, D \otimes \kappa', D'\rangle \right) = \\ = \sum_{\kappa,D;\kappa',D'} C^{\kappa'',D''}_{\kappa,D;\kappa',D,} \cdot \sum_{\tilde{\kappa},\tilde{D},\tilde{\kappa}',\tilde{D}'} \beta_{\tilde{\kappa},\tilde{D},\tilde{\kappa}',\tilde{D}'} |\tilde{\kappa}, \tilde{D} \otimes \tilde{\kappa}', \tilde{D}'\rangle \end{aligned} \tag{3.4}$$

If the Clebsch-Gordan coefficients of the parent states $C^{\kappa'',D''}_{\kappa,D;\kappa',D,}$ are not known (i.e. at least one parent state has not been visited), then the algorithm calls itself recursively to determine the Clebsch-Gordan coefficients of the parent states first. That is why it is important to already know the coefficients of the highest weight state. It is the only parent state to some other states and serves as anchor for the recursive calculation.
Since only lowering operators are used, it is impossible to create a loop of dependencies what would make the algorithm break down. Starting from any state, you cannot come back to it by just using lowering operators (e.g. in SU(2): you cannot come back to a state using only one of the ladder operators). By iterating over all states we assure that the coefficients of every state have been calculated.

Example: SU(2) irreps

To understand the principle, let us look at a coupling of SU(2) irreps. They are labelled $|j, m\rangle$ like spins in quantum mechanics for easier understanding: we take one particle with spin $j = 1$ and z component l, $|1, l\rangle$, and one with spin $j' = \frac{1}{2}$ and z-component m' , $|\frac{1}{2}, m`\rangle$. For easier understanding we use standard quantum mechanics notation here which is related to GT pattern via:

$$\begin{pmatrix} 2j & & 0 \\ & j-m & \end{pmatrix} = |j, m\rangle \tag{3.5}$$

The highest weight state is: $|1, 1\rangle \otimes |\frac{1}{2}, \frac{1}{2}\rangle = |\frac{3}{2}, \frac{3}{2}\rangle$.We need not solve a system of linear equations here, since only one state fulfills the condition (3.2): the z-components of the two spins have to add up to $\frac{3}{2}$ and only $|1, 1\rangle \otimes |\frac{1}{2}, \frac{1}{2}\rangle$ has that property. We apply the lowering operator to both sides taking the this property of the operator into account:

$$J_\pm|s, m\rangle\sqrt{(s \pm m + 1)(s \mp m)}|s, m \pm 1\rangle \tag{3.6}$$

The lowering operator acts on a tensor product like

$$J_- = J_-^{(1)} \otimes \mathbb{1} + \mathbb{1} \otimes J_-^{(2)} \tag{3.7}$$

It follows that

$$\begin{aligned} J_- \left(|1, 1\rangle \otimes |\frac{1}{2}, \frac{1}{2}\rangle \right) = \sqrt{2} \cdot |1, 0\rangle \otimes |\frac{1}{2}, \frac{1}{2}\rangle + |1, 1\rangle \otimes 1 \cdot |\frac{1}{2}, -\frac{1}{2}\rangle \\ = \sqrt{3} \cdot |\frac{3}{2}, \frac{1}{2}\rangle = J_-|\frac{3}{2}, \frac{3}{2}\rangle. \end{aligned} \tag{3.8}$$

Therefore

$$|\frac{3}{2}, \frac{1}{2}\rangle = \sqrt{\frac{2}{3}} \cdot |1, 0\rangle \otimes |\frac{1}{2}, \frac{1}{2}\rangle + \sqrt{\frac{1}{3}}|1, 1\rangle \otimes |\frac{1}{2}, -\frac{1}{2}\rangle. \tag{3.9}$$

The Clebsch-Gordan coefficients can now easily be read off. Applying the lowering operator again yields:

$$\begin{aligned} J_-\left(\sqrt{\frac{2}{3}}\cdot|1,0\rangle\otimes|\frac{1}{2},\frac{1}{2}\rangle+\sqrt{\frac{1}{3}}|1,1\rangle\otimes|\frac{1}{2},-\frac{1}{2}\rangle\right)= \\ \frac{2}{\sqrt{3}}\cdot|1,-1\rangle\otimes|\frac{1}{2},\frac{1}{2}\rangle+\sqrt{\frac{2}{3}}\cdot|1,0\rangle\otimes|\frac{1}{2},-\frac{1}{2}\rangle+\sqrt{\frac{2}{3}}\cdot|1,0\rangle\otimes|\frac{1}{2},-\frac{1}{2}\rangle= \\ 2\cdot|\frac{3}{2},-\frac{1}{2}\rangle=J_-\left(|\frac{3}{2},\frac{1}{2}\rangle\right) \end{aligned} \tag{3.10}$$

It follows that

$$|\frac{3}{2},-\frac{1}{2}\rangle=\sqrt{\frac{1}{3}}\cdot|1,-1\rangle\otimes|\frac{1}{2},\frac{1}{2}\rangle+\sqrt{\frac{2}{3}}\cdot|1,0\rangle\otimes|\frac{1}{2},-\frac{1}{2}\rangle. \tag{3.11}$$

The last application yields:

$$\begin{aligned} J_-\left(\sqrt{\frac{1}{3}}\cdot|1,-1\rangle\otimes|\frac{1}{2},\frac{1}{2}\rangle+\sqrt{\frac{2}{3}}\cdot|1,0\rangle\otimes|\frac{1}{2},-\frac{1}{2}\rangle\right)= \\ \sqrt{\frac{1}{3}}\cdot|1,-1\rangle\otimes|\frac{1}{2},-\frac{1}{2}\rangle+\frac{2}{\sqrt{3}}\cdot|1,-1\rangle\otimes|\frac{1}{2},-\frac{1}{2}\rangle= \\ =\sqrt{3}\cdot|\frac{3}{2},-\frac{3}{2}\rangle=J_-\left(|\frac{3}{2},-\frac{1}{2}\rangle\right) \end{aligned} \tag{3.12}$$

and

$$|\frac{3}{2},-\frac{3}{2}\rangle=|1,-1\rangle\otimes|\frac{1}{2},-\frac{1}{2}\rangle \tag{3.13}$$

This scheme can be generalized to higher dimensions. To exploit the Weyl group symmetry it just would have been necessary to calculate the dominant Weyl chamber. It consists here of $|\frac{3}{2},\frac{3}{2}\rangle$ and $|\frac{3}{2},\frac{1}{2}\rangle$. Let us change to the $|\kappa,D\rangle$ basis. The states in the dominant Weyl chamber are $|\mathbb{1},\frac{3}{2}\rangle$ and $|\mathbb{1},\frac{1}{2}\rangle$. The other two states are $|(1,2),\frac{3}{2}\rangle$ and $|(1,2),\frac{1}{2}\rangle$. There are only $2!=2$ permutations in SU(2), the identity and the transposition $\tau=(1,2)$. After we have gotten the Clebsch-Gordan coefficients in the dominant Weyl chamber we can apply the factorization of τ into a connector (itself) and an element of the stabiliser (the identity). Using the formalism described below we obtain the result that we have to multiply the Clebsch-Gordan coefficients with the matrix elements of the representation matrix of the stabiliser. Since this is unity we multiply every coefficient with 1. The coefficients of $|\mathbb{1},\frac{3}{2}\rangle$ and $|\tau,\frac{3}{2}\rangle$ respective of $|\mathbb{1},\frac{1}{2}\rangle$ and $|\tau,\frac{1}{2}\rangle$ are now known to be identical without any further calculation like in the example above. Thus we have reduced our calculations by a factor of $N!$ ($=2$ here).

3.5 Finding the remaining Clebsch-Gordan coefficients via permutations

Having calculated all Clebsch-Gordan coefficients in the dominant Weyl chamber we can now exploit the action of the Weyl group, i.e. permuting the p-weights to find all other coefficients very fast. For arbitrary $|\kappa'',D''\rangle$, we apply κ'' to $|\mathbb{1},D''\rangle=\sum_{\kappa,D;\kappa',D'}C^{\mathbb{1},D''}_{\kappa,D;\kappa',D'}|\kappa,D\otimes\kappa',D,\rangle$.

$$\kappa''\left(\sum_{\kappa,D;\kappa',D'} C^{\mathbb{1},D''}_{\kappa,D;\kappa',D'}|\kappa, D\otimes\kappa', D,\rangle\right) = \sum_{\kappa,D;\kappa',D'} C^{\mathbb{1},D''}_{\kappa,D;\kappa',D'}(\kappa''\kappa\otimes\kappa''\kappa')|\mathbb{1}, D\otimes\mathbb{1}, D'\rangle \tag{3.14}$$

Then decompose $\kappa''\kappa$ and $\kappa''\kappa'$ into a connector and a stabiliser:

$$\kappa''\kappa = \tilde{\kappa}\beta \quad , \qquad \kappa''\kappa' = \tilde{\kappa}'\beta' \tag{3.15}$$

Now (3.14) can be written as:

$$\kappa''|\mathbb{1}, D''\rangle = \sum_{\kappa,D;\kappa',D'} C^{\mathbb{1},D''}_{\kappa,D;\kappa',D'}(\tilde{\kappa}\otimes\tilde{\kappa'})\cdot(\beta\otimes\beta')|\mathbb{1}, D\otimes\mathbb{1}, D'\rangle \tag{3.16}$$

And since β, β' are from the stabiliser we can say with certainty that they will not change the state. We calculate explicit representation matrices for them and the equation becomes:

$$\kappa''|\mathbb{1}, D''\rangle = \sum_{\kappa,D;\kappa',D'} C^{\mathbb{1},D''}_{\kappa,D;\kappa',D'} \sum_{\tilde{D},\tilde{D}'} U^{\beta}_{\tilde{D},D} U^{\beta'}_{\tilde{D}',D'}|\tilde{\kappa}, \tilde{D}\otimes\tilde{\kappa}', \tilde{D}'\rangle \tag{3.17}$$

The Clebsch-Gordan coefficient $C^{\kappa'',D''}_{\tilde{\kappa},\tilde{D};\tilde{\kappa},\tilde{D}'}$ is then:

$$C^{\kappa'',D''}_{\tilde{\kappa},\tilde{D};\tilde{\kappa},\tilde{D}'} = \sum_{\kappa,D;\kappa',D'} C^{\mathbb{1},D''}_{\kappa,D;\kappa',D'} \sum_{\tilde{D},\tilde{D}'} U^{\beta}_{\tilde{D},D} U^{\beta'}_{\tilde{D}',D'} \tag{3.18}$$

As one can see only knowledge of the Clebsch-Gordan coefficients in the dominant Weyl chamber and the representation matrices of the stabiliser permutation is required to calculate every Clebsch-Gordan coefficient for a given irrep S''.

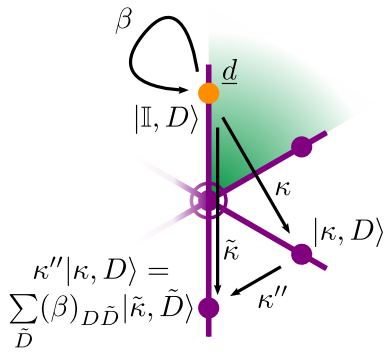

Figure 3.1: Schematic illustration of the application of κ'' to $|\kappa, D\rangle$. κ maps a dominant state to a state outside of the dominant Weyl chamber, κ' maps it to yet another one. They are decomposed into $\tilde{\kappa}$ that links the dominant state directly to the new one and β that does not change the weight. (Courtesy of Arne Alex)

4 Routines

This chapter describes the program routines that have been newly developed from scratch.

4.1 Finding the representation matrix of a permutation

To calculate the representation matrix of a permutation it is useful to decompose the permutation into transpositions first, because finding the representation matrix of a transposition is comparatively easy. The representation matrix of the permutation is then obtained via multiplication of the transposition matrices.

4.1.1 Decomposing permutations

To find a representation matrix of a transposition it is necessary to calculate several exponentials of matrices (see below). Therefore, we decompose every permutation only into transpositions of the form $(1, n)$. In general this makes the decomposition contain more transpositions than necessary, but should speed up the calculation since we only need to perform d (computationally intensive) calculations of representation matrices instead of $\frac{d^2-d}{2}$, where d is the dimension of the representation matrix.
In the implementation of the decomposition a given permutation is converted to the identity step by step, where every step is a transposition. The program starts with the element at the last position N. It checks if this element is already the correct one. If it is not, the algorithm finds the position x of the element that belongs to the last position in the identity, and swaps its entry with position number one. So, the first transposition is $(1, x)$. Next, it swaps the first position with the last, making the next transposition $(1, N)$. Now the the element on position N is correct. Then it continues with the next position $N-1$. So the maximal number of transposition is less than $2N$.

Example

Decomposition of $\pi = (2, 3, 1, 5, 4)$: there is a 4 at the last position, where a 5 is supposed to be. So the 5 is swapped from position 4 to position 1 and then position 1 is swapped with position 5. The permutation is now $\pi' = (4, 3, 1, 2, 5)$. The transpositions $\tau_1 = (1, 4)$ and $\tau_2 = (1, 5)$ were split off. At position 4 is a 2. Therefore, it is swapped with position 1. At position 3 is 1, so position 2, where the 3 is, is swapped with position 1 and afterwards position 1 with position 3. So the permutation can be decomposed into five transpositions τ_i $(i = 1 \dots 5)$. Their order has to be respected (i.e. they do not commute): $(1, 4) \circ (1, 5) \circ (1, 4) \circ (1, 2) \circ (1, 3)$.

4.1.2 Representation matrix

A representation matrix of a transposition $\tau = (m, n)$ can be written as:

$$\mathrm{Rep}(\tau) = e^{i\pi E^{m,m}} e^{E^{n,m}} e^{-E^{m,n}} e^{E^{n,m}} \tag{4.1}$$

with the single-entry matrices $E^{m,n}$ ($E^{m,n}_{a,b} = \delta_{ma}\delta_{nb}$) of the Lie algebra. To show this, we examine the matrix elements:

$$\left(e^{i\pi E^{m,m}} e^{E^{n,m}} e^{-E^{m,n}} e^{E^{n,m}}\right)_{j,k} \tag{4.2}$$

We look at the individual exponentials first. By Taylor expansion, we find

$$\left(e^{i\pi E^{m,m}}\right)_{j,p} = \delta_{j,p}\delta_{j,m} - 2\delta_{j,p}\delta_{j,m} \tag{4.3a}$$

$$\left(e^{E^{n,m}}\right)_{p,q} = \delta_{p,q} + \delta_{p,n}\delta_{q,m} \tag{4.3b}$$

$$\left(e^{-E^{m,n}}\right)_{q,r} = \delta_{q,r} - \delta_{p,m}\delta_{r,n} \tag{4.3c}$$

$$\left(e^{E^{n,m}}\right)_{r,k} = \delta_{r,k} + \delta_{r,n}\delta_{k,m}. \tag{4.3d}$$

Substituting these equations into (4.2) yields

$$\begin{aligned}\sum_p \sum_q \sum_r \left(\delta_{j,p} - 2\delta_{j,p}\delta_{j,m}\right)\left(\delta_{p,q} + \delta_{p,n}\delta_{q,m}\right)\left(\delta_{q,r} - \delta_{p,m}\delta_{r,n}\right)\left(\delta_{r,k} + \delta_{r,n}\delta_{k,m}\right) = \\ \delta_{j,k} - \delta_{j,m}\delta_{k,m} - \delta_{j,n}\delta_{k,n} + \delta_{j,m}\delta_{k,n} + \delta_{j,n}\delta_{k,m}\end{aligned} \tag{4.4}$$

This is exactly the form of a transposition matrix for a transposition (m, n).
Now, let us use equation (4.1) to calculate the representation matrices of transpositions in the group. Remember, that

$$J^{(l)}_{+} = E^{l,l+1} \tag{4.5}$$

$$J^{(l)}_{-} = E^{l+1,l} \tag{4.6}$$

So, with the representation matrices of the raising and lowering operators and the commutator relations stated in equation (2.2a) and (2.2b), we can construct any $E^{1,n}$ representation (with $n \neq 1$). The raising and lowering operators are obtained by evaluating the action of the respective operator on all patters that form the basis of the irrep carrier space. The coefficients that arise for valid patterns are explicitly calculated via (2.13) and (2.12) and stored in a matrix.
Having a representation of the $E^{1,n}$, we can exponentiate them. $E^{p,q}$ is nilpotent for $p \neq q$. So it is not possible to exponentiate by diagonalization. To get the exponentials of those matrices, we use Taylor expansion instead. The expansion breaks down fairly quickly as an N-dimensional upper-trigonal matrix with zeros on its diagonal is of maximal nilpotency rank N (i.e. $A^N = 0$, for a nilpotent matrix A of rank N). The last three exponentials of (4.1) are calculated that way. To avoid an introduction of complex numbers, the action of the first matrix is implemented differently: the representation matrix of $E^{1,1}$ is diagonal with the $m_{1,1}$ of each pattern on the diagonal[6]. Since those entries are integers, the multiplication with $i\pi$ can only yield an odd or even number times $i\pi$. Therefore, exponentiation of this matrix gives a diagonal matrix with only 1 and -1 as entries. The program changes explicitly the signs of the rows in the product of the second three matrices where there is a -1 on the diagonal.

Example:

If you want to find e.g. the representation matrix of a transposition $(1,3)$ for the irrep with i-weight $(2,1,0)$, you have to determine $E^{1,3}$:

$$E^{1,3} = [J_+^1, J_+^2] = \begin{pmatrix} 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ -\sqrt{\frac{3}{2}} & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ \sqrt{\frac{1}{2}} & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & -1 & 0 & 0 & 0 & 0 \\ 0 & 0 & \sqrt{\frac{3}{2}} & 0 & -\sqrt{\frac{1}{2}} & 0 & 0 & 0 \end{pmatrix} \tag{4.7}$$

in conclusion, we can compute a representation matrix of an arbitrary permutation. Therefore we decompose it into transpositions of form $(1,n)$, calculate the representation matrices of every transposition of that form and multiply the matrices with respect to the order of the decomposition.

4.2 Connectors

Permutations of the weight can be divided into different classes. A permutation class contains every permutation that has the same effect when applied to a p-weight (so the members of the classes depend strongly on the p-weight). For uniqueness, it is necessary to pick a single, distinct representative of each class. So if you have an arbitrary permutation and a p-weight, you need to determine its class and find the respective representative. The representative we chose here is the numerically smallest member of the class, e.g. from the set of permutations $\{(2,1,3),(2,3,1)\}$ it would be $(2,1,3)$ since $213 < 231$. We call such representatives *connectors*.

To find a connector, the permutation is applied to the weight. Then the numerically smallest permutation with the same effect is constructed. Therefore, it is checked at which positions in the permuted weight the first identical elements of the original weight appear. Those positions appended in the 'smallest' possible order form the first part of the connector. Then this procedure is repeated for all entries in the weight that differ in value in descending order.

Let's take the weight $w = (3,3,2,2,0)$ and the permutation $\pi = (2,3,5,1,4)$ as an example. The permutation applied to the weight gives $\pi(w) = (2,3,3,0,2)$ as a permuted weight. Now we are looking for the numerically lowest permutation that does the same. Since the first entries of the dominant weight is the highest value, we focus first on the value 3. It appears on position 2 and 3, so position 1 has to be mapped to 2 and position 2 has to be mapped to 3, giving $\rho = (2,3,\pi_3,\pi_4,\pi_5)$ as intermediate result for the connector. We continue with the next smaller element of the original weight: 2. The first 2 now appears at position 1, so $\pi_3 = 1$. The second 2 appears at 5, so $\pi_4 = 5$ leaving $\pi_5 = 4$. It follows that $\rho = (2,3,1,5,4)$ is the representative of the class, to which the permutation π belongs (for the given weight w).

Any permutation that belongs neither to the connectors nor to the stabiliser can be split up into two permutations of which one is a connector and the other one is from the stabiliser.

The set of connectors depends on the p-weight. The trivial examples are: the set of connectors of a weight with mutually differing entries is the set of all permutations of the elements, whereas the set contains only the identity for a p-weight whose entries are all equal.
For the calculation of the Clebsch-Gordan coefficients it is necessary to identify all connectors that map a given p-weight to all other Weyl chambers. The algorithm for an arbitrary p-weight works as follows: first, the p-weight is sorted such that its entries are in ascending order. This is the lexicographically smallest (or just: 'smallest') possible p-weight. Then, this p-weight is compared with the original p-weight. The smallest permutation that converts the original weight to the smallest p-weight is determined. It is not stored directly, instead we use an indexing scheme(see below) and store its index in a vector. Then the smallest p-weight is increased, i.e. its entries are sorted in order of the next larger p-weight. Now, the smallest permutation for that conversion is determined again and its index is stored in the vector. This is done for every different permutation of the p-weight entries; the total number of connectors per weight is given by the multinomial coefficients

$$\frac{N!}{\prod_{i=0}^{\gamma} N_i!} \tag{4.8}$$

where N is the irrep dimension, N_i is the number of p-weight entries with value i and γ the weight's largest entry.

Finding the back connector

By applying the raising operator conjugated with permutation κ, $J_+'^{(l)} = \kappa^{-1} J_+^{(l)} \kappa$, to a state we generally leave the dominant Weyl chamber. To describe the arising state with our labelling scheme, we have to permute it back into the dominant Weyl chamber. To find the necessary connector, we use the fact that $J_+'^{l}$ is also a single entry matrix. The conjugation of $E^{p,q}$ is easy to evaluate, since $\kappa^{-1} E^{p,q} \kappa = E^{\kappa^{-1}(p),\kappa^{-1}(q)}$ and its action onto a weight is well known (cf. (2.11)). So, after evaluating this relation the generated weight is permuted back to the form of a dominant Weyl chamber weight. The permutation that does this is the back connector.

4.3 Indexing permutations

For convenience, we store permutations not as a whole, but tag them with an index. The indexing scheme labels the permutations with an integer in lexicographically ascending order: the identity is labelled as first permutation with index $\mathrm{Ind}(\mathbb{1}) = 0$, while the permutation $\pi_{N,max} = (N, N-1, \ldots, 2, 1)$ has index $\mathrm{Ind}(\pi_{N,max}) = N! - 1$. To determine the index of a a permutation π with N elements, we interpret the row vector $\pi = (\pi_1, \pi_2, \ldots, \pi_N)$ describing the permutation as a number. The number system in which it is given is somewhat special as the basis of each consecutive digit contains one number less than the basis of the number prior to it. The size of the basis decreases because a picked number cannot be chosen again, reducing the available numbers by one after each digit. Additionally, when determining the actual numerical value of a permutation entry, it is important to check which *smaller* numbers have already been picked. The amount of these numbers has to be subtracted from the considered entry (when you regard a number given in a basis $B = \{1, 4, 6, 7\}$, 4 is the third smallest element and stands for the value $4 - 3 = 1$, where 3 is the cardinality of the

set of missing numbers $M = \{0, 2, 3\}$. Note, that one must also take care of 0 as smallest element).
To index a given permutation we take the first element from the permutation and multiply it with $N-1$, where N is the number of elements of the permutation. Then we add the next element subtracting the amount of numerically smaller elements that have already been used, because these elements are not part of the basis anymore. Then we multiply with $N-2$, which corresponds to the decreased size of basis etc. until we reach the last element, which always stands for zero as you can regard it as given in a basis of size 1.

Example

First, let us look at the translation of an octal number into a decimal one to get a better understanding of the basis issue. The octal number 15324_8 is translated into a decimal one like this:

$$1 \cdot N^4 + 5 \cdot N^3 + 3 \cdot N^2 + 2 \cdot N + 4 = (((1 \cdot N + 5) \cdot N + 3) \cdot N + 2) \cdot N + 4 \overset{N=8}{=} 6868_{10} \quad (4.9)$$

In this translation, the number gets multiplied with $N = 8$ at each step, because the octal basis does not shrink. Each picked number is again available after usage. Now, let us look at the permutation $(2, 4, 1, 3, 0)$ with $N = 5$. Its index is determined via

$$\begin{aligned} &\mathrm{Ind}\,[(2, 4, 1, 3, 0)] = \\ &(((2 \cdot (N-1) + (4-1)) \cdot (N-2) + 1) \cdot (N-3) + (3-2)) \cdot (N-4) + 0 \overset{N=5}{=} 69 \end{aligned} \quad (4.10)$$

We start with the first entry, multiply it with the size of the base $N - 1 = 4$, then add the next element 4 minus the number of elements smaller than 4 that have already been taken. In this case $4 - 1 = 3$, since 2 has already been removed from the basis. Then we multiply with $N - 2 = 3$, the size of the new basis, and so on.

4.4 Finding all states in the dominant Weyl chamber

At some stages in the calculations we have to iterate over all states that lie in the dominant Weyl chamber. Thus, it is necessary to know which states do. This task is split up into two steps: First, we determine all possible p-weights in the chamber. Then, we construct all states for every p-weight.
The p-weights in the dominant Weyl chamber have to obey two relations: their entries must not be ascending and the sum over their entries is fixed. The highest weight state's p-weight is identical to the i-weight of the irrep. We start with this weight and process it by adding 1 to the last entry w_n and subtracting 1 from w_{n-1}. This leaves the entry sum invariant. If the generated weight obeys the conditions it is stored. Otherwise, the prior step is reversed and we proceed with w_{n-1} and w_{n-2}. If w_2 and w_1 are reached and no valid weight has been generated, the spacing between the manipulated entries is increased, i.e. it is checked if incrementing of w_n and decrementing of w_{n-2} up to incrementing w_3 and decrementing w_1 yield a valid weight. If they do, we start with spacing 1 again. If they do not, the spacing is increased until w_1 gets lowered and w_n raised. Once that returns an invalid weight, we have

determined all weights in the dominant Weyl chamber. As an example let us examine the irrep with i-weight $i = (4, 2, 1)$:
We start with $w_2 - 1$ and $w_3 + 1$ and gain $(4, 1, 2)$ which does not fulfill the conditions. It is reset and we continue with $w_1 - 1$ and $w_2 + 1$. That yields $(3, 3, 1)$ which is valid and hence lies in the dominant Weyl chamber. Now, we start at the beginning with $w_2 - 1$ and $w_3 + 1$ and obtain the next valid weight $(3, 2, 2)$. Applying again $w_2 - 1$ and $w_3 + 1$, we get an invalid one, $(3, 1, 3)$. After resetting and applying $w_1 - 1$ and $w_2 + 1$ we again get an invalid one $(2, 3, 2)$. Since we arrived at w_1 we increase the spacing by one. The next step then is $w_1 - 1$ and $w_3 + 1$ which also results in an invalid weight. Since, the spacing is maximal for a three element p-weight, we cannot go on. The p-weight that lie in the dominant Weyl chamber of the irrep $(4, 2, 1)$ are: $(4, 2, 1), (3, 3, 1)$ and $(3, 2, 2)$.
Having found all p-weights we now want to construct all states belonging to them. To do this, we calculate the row sums of the state pattern first. They are uniquely defined via (2.7). Then the pattern that has the maximal entries (starting from the left hand side) is constructed by merely inserting the largest possible values as entries with respect to the row sums and the betweenness condition. Using this as an anchor we construct the other patterns by varying the rows of the pattern in the same way we varied the p-weights in the calculation above, since the rows have to obey the same relations: the entries must not ascend and the row sum is fixed. We start with the second line from bottom (the bottom row is invariant, since it contains just one entry and its sum is fixed) and replace it with the next possible sequence of non-ascending numbers with fixed row sum. If a variation yields a pattern that is valid regarding the betweenness condition, we store it. When no valid pattern can be found, we proceed with varying the next line above until we reach the first row which must also not be varied, since it is the i-weight. Let us go back to the example irrep $i = (4, 2, 1)$ and take $w = (3, 2, 2)$ as p-weight. First we construct the row sum vector from the p-weight (eq. (2.7)): $r = (3, 5, 7)$. Then the maximal pattern M as described above is:

$$M = \begin{pmatrix} 4 & 2 & 1 \\ 4 & 1 & \\ 3 & & \end{pmatrix} \tag{4.11}$$

$m_{2,1}$ is chosen to be 4, the largest number that respects the betweenness condition. $m_{2,2}$ then has to be 1, because the row sum of the second row must be 5. $m_{1,1}$ is 3, since the row sum of the first row is 3. To find the next pattern we vary the second row and get

$$\begin{pmatrix} 4 & 2 & 1 \\ 3 & 2 & \\ 3 & & \end{pmatrix} \tag{4.12}$$

which is a valid pattern. There is no other variation of the second row that respects the conditions, so we have to start varying the row above it to find new ones. But this one is the first row with the i-weight in it. We are not allowed to change it. Hence the two resulting states are the only ones lying in the dominant Weyl chamber with p-weight $p = (3, 2, 2)$ for the irrep $(4, 2, 1)$. For higher N in SU(N) or larger i-weights there can be significantly more states in the dominant Weyl chamber.

5 Outlook

Currently, we use a self-written C++ matrix class for our matrix operations with few operations included from LAPACK. The matrix size increases quadratically with the carrier space dimensions d of the irreps, because the representation matrices are $d \times d$. For SU(N) with large N or large indices of the irreps, d grows rapidly and with it the required memory space (e.g. for the transposition representation matrices) and operation time (e.g. for exponentiation) on the computer. However, the matrices are very sparse, i.e. most of their entries are 0. A further increase in computation speed could be achieved by implementing an optimized class for matrix handling which takes the properties of the matrices into account.
Concluding, we state that with this new algorithm it is possible to compute Clebsch-Gordan coefficients even for very large SU(N) (like SU(20)) efficiently. In contrast, the old program is not capable of handling SU(N) of that order. Hence, the new program should fully meet the requirements at the chair where the Clebsch-Gordan coefficients are mainly needed for NRG and DMRG calculations[8][9].
Finally, the publication of a paper that gives a brief overview over the algorithm in a scientific journal is planned and we may also install a web interface for the computation of Clebsch-Gordan coefficients that uses the new algorithm.

6 Source Code

This chapter presents the code that has been written. It consists of multiple classes:

1. `matrix`
2. `irrep`
3. `pattern`
4. `permut`

At the beginning of each section there is a brief description of the functions of these classes.

Matrix class (header and source file)

Creates objects `weyl::matrix` that is capable of standard operations (addition, multiplication, commutator) and the more advanced matrix operation that are required (exponentiation, finding the nullspace and least square methods for the equation $Ax - b = \min$).

```
#ifndef WEYL_MATRIX_H_
#define WEYL_MATRIX_H_

// C headers
#include <cassert>

// C++ headers
#include <ostream>
#include <vector>

// weyl headers
// none (delete this line if you add some)

namespace weyl {
// guarantees: matrix is initialized with zeros
    class matrix {
    public:
        // create matrix initialized with zeros
        matrix(int rows, int cols);

        // square matrix
        explicit matrix(int dim);

        // access elements
        // i = 0, ..., rows - 1
        // j = 0, ..., cols - 1
        double& operator()(int i, int j);
        const double& operator()(int i, int j) const;

        // convert to human-readable form
        operator std::string() const;

        // return shape of this matrix
        int rows() const;
        int cols() const;
```

```
        int dim() const;

        // matrix algebra (does what you expect it to do)
        matrix operator-() const;
        matrix operator+(const matrix& other) const;
        matrix operator-(const matrix& other) const;
        matrix operator*(double coeff) const;
        matrix operator*(const matrix& other) const;

        // convenience function for ab - ba
        matrix commutator(const matrix& other) const;

        // matrix exponential, aborts series after 'order' terms
        matrix exp(int order = 256) const;

        // find column vectors x such that Ax = 0
        // eps determines which singular values are considered zero
        // returns a matrix X such that A * X == 0
        matrix nullspace(double eps = 1e-9) const;

        // find minimum-norm solution to AX = B
        // returns a matrix X
        matrix least_sq(const matrix& rhs) const;

    private:
        std::vector<double> elem;
        int rows_, cols_;
    };
}

// inline code follows

inline double& weyl::matrix::operator()(int i, int j) {
    return elem[i * cols_ + j];
}

inline const double& weyl::matrix::operator()(int i, int j) const {
    return elem[i * cols_ + j];
}

inline int weyl::matrix::rows() const {
    return rows_;
}

inline int weyl::matrix::cols() const {
    return cols_;
}

inline int weyl::matrix::dim() const {
    assert(rows_ == cols_);
    return rows_;
}

#endif // WEYL_MATRIX_H_
```

```
// this file implements the declarations from the following file
#include "matrix.h"

// C headers
#include <cassert>

// C++ headers
#include <sstream>
#include <string>
#include <vector>
```

```
// weyl headers
// none (delete this line if you add some)

// declarations of LAPACK routines

extern "C" void dgesvd_(const char* JOBU,
                        const char* JOBVT,
                        const int* M,
                        const int* N,
                        double* A,
                        const int* LDA,
                        double* S,
                        double* U,
                        const int* LDU,
                        double* VT,
                        const int* LDVT,
                        double* WORK,
                        const int* LWORK,
                        int *INFO);

extern "C" void dgels_(const char* TRANS,
                       const int* M,
                       const int* N,
                       const int* NRHS,
                       double* A,
                       const int* LDA,
                       double* B,
                       const int* LDB,
                       double* WORK,
                       const int* LWORK,
                       int *INFO);

// implementation of weyl::matrix follows

weyl::matrix::matrix(int dim) : elem(dim * dim, 0), rows_(dim), cols_(dim) {}

weyl::matrix::matrix(int rows, int cols) : elem(rows * cols, 0), rows_(rows), cols_(cols) {}

weyl::matrix::operator std::string() const {
    std::ostringstream os;

    for (int i = 0; i < rows_; ++i) {
        os << (i > 0 ? "  " : "[ ");
        for (int j = 0; j < cols_; ++j) {
            os.width(12);
            os << (*this)(i, j);
        }
        os << (i + 1 < rows_ ? "\n" : " ]");
    }

    return os.str();
}

weyl::matrix weyl::matrix::operator-() const {
    matrix result(rows_, cols_);
    std::vector<double>::iterator result_it = result.elem.begin();

    for (std::vector<double>::const_iterator it = elem.begin(); it != elem.end(); ++it) {
        *result_it = -*it;
        ++result_it;
    }

    return result;
}
```

```
weyl::matrix weyl::matrix::operator+(const weyl::matrix& other) const {
    assert(cols_ == other.cols_);
    assert(rows_ == other.rows_);

    matrix result(rows_, cols_);
    std::vector<double>::iterator result_it = result.elem.begin();

    for (std::vector<double>::const_iterator it = elem.begin(), jt = other.elem.begin();
            it != elem.end() && jt != other.elem.begin(); ++it, ++jt) {
        *result_it = *it + *jt;
        ++result_it;
    }

    return result;
}

weyl::matrix weyl::matrix::operator-(const weyl::matrix& other) const {
    assert(cols_ == other.cols_);
    assert(rows_ == other.rows_);

    weyl::matrix result(rows_, cols_);
    std::vector<double>::iterator result_it = result.elem.begin();

    for (std::vector<double>::const_iterator it = elem.begin(), jt = other.elem.begin();
            it != elem.end(); ++it, ++jt) {
        *result_it = *it - *jt;
        ++result_it;
    }

    return result;
}

weyl::matrix weyl::matrix::operator*(double coeff) const {
    matrix result(rows_, cols_);
    std::vector<double>::iterator result_it = result.elem.begin();

    for (std::vector<double>::const_iterator it = elem.begin(); it != elem.end(); ++it) {
        *result_it = coeff * (*it);
        ++result_it;
    }

    return result;
}

weyl::matrix weyl::matrix::operator*(const weyl::matrix& other) const {
    assert(cols_ == other.rows_);

    matrix result(rows_, other.cols_);

    for (int i = 0; i < rows_; ++i) {
        for (int j = 0; j < other.cols_; ++j) {
            for (int k = 0; k < cols_; ++k) {
                result(i,j) += (*this)(i, k) * other(k, j);
            }
        }
    }

    return result;
}

weyl::matrix weyl::matrix::commutator(const weyl::matrix& other) const {
    return (*this) * other - other * (*this);
}

weyl::matrix weyl::matrix::exp(int order) const {
    assert(rows_ == cols_);
```

```
    matrix result(rows_, cols_);
    matrix power(rows_, cols_);

    for (int i = 0; i < rows_; ++i) power(i, i) = 1.0;
    for (int i = 0; i < order; ++i) {
        result = result + power;
        power = (*this) * power * (1.0 / (i + 1));
    }

    return result;
}

weyl::matrix weyl::matrix::nullspace(double eps) const {
    double dummy;
    double* A = new double[rows_ * cols_];
    double* S = new double[std::min(rows_, cols_)];
    double* VT = new double[cols_ * cols_];
    double* WORK = &dummy;
    int LWORK = -1;
    int INFO;

    // copy this into A
    for (int i = 0; i < rows_; ++i) {
        for (int j = 0; j < cols_; ++j) {
            A[i + j * rows_] = (*this)(i, j);
        }
    }

    // call DGESVD to find out size of WORK
    dgesvd_("N",
            "A",
            &rows_,
            &cols_,
            A,
            &rows_,
            S,
            NULL,
            &rows_,
            VT,
            &cols_,
            WORK,
            &LWORK,
            &INFO);
    assert(INFO == 0);

    LWORK = *WORK;
    WORK = new double[LWORK];

    // do the SVD
    dgesvd_("N",
            "A",
            &rows_,
            &cols_,
            A,
            &rows_,
            S,
            NULL,
            &rows_,
            VT,
            &cols_,
            WORK,
            &LWORK,
            &INFO);
    assert(INFO == 0);
```

```
    // create result from S and VT
    int dim_nullspace = cols_ - std::min(rows_, cols_);
    for (int i = std::min(rows_, cols_) - 1; i >= 0 && S[i] < eps; --i) {
        ++dim_nullspace;
    }

    matrix result(cols_, dim_nullspace);
    for (int i = 0; i < cols_; ++i) {
        for (int j = 0; j < dim_nullspace; ++j) {
            result(i, j) = VT[cols_ - dim_nullspace + j + i * cols_];
        }
    }

    delete[] WORK;
    delete[] VT;
    delete[] S;
    delete[] A;

    return result;
}

weyl::matrix weyl::matrix::least_sq(const weyl::matrix& rhs) const {
    double dummy;
    double* A = new double[rows_ * cols_];
    double* B = new double[rhs.rows_ * rhs.cols_];
    double* WORK = &dummy;
    int LWORK = -1;
    int INFO;

    assert(rows_ == rhs.rows_);

    // make copies of this and rhs
    for (int i = 0; i < rows_; ++i) {
        for (int j = 0; j < cols_; ++j) {
            A[i + j * rows_] = (*this)(i, j);
        }
    }

    for (int i = 0; i < rhs.rows_; ++i) {
        for (int j = 0; j < rhs.cols_; ++j) {
            B[i + j * rhs.rows_] = rhs(i, j);
        }
    }

    // call DGELS to obtain proper LWORK
    dgels_("N",
           &rows_,
           &cols_,
           &rhs.cols_,
           A,
           &rows_,
           B,
           &rhs.rows_,
           WORK,
           &LWORK,
           &INFO);
    assert(INFO == 0);

    LWORK = *WORK;
    WORK = new double[LWORK];

    // real work happens here
    dgels_("N",
           &rows_,
           &cols_,
           &rhs.cols_,
```

```
            A,
            &rows_,
            B,
            &rhs.rows_,
            WORK,
            &LWORK,
            &INFO);
    assert(INFO == 0);

    // initialize result with return value of DGELS
    matrix result(cols_, rhs.cols_);
    for (int i = 0; i < cols_; ++i) {
        for (int j = 0; j < rhs.cols_; ++j) {
            result(i, j) = B[i + j * rhs.rows_];
        }
    }

    delete[] WORK;
    delete[] B;
    delete[] A;

    return result;
}
```

Irrep class (header and source file)

Creates objects `weyl::irrep` that contain the i-weight and methods to handle it.

```
#ifndef WEYL_IRREP_H_
#define WEYL_IRREP_H_

// C headers
// none (delete this line if you add some)

// C++ headers
#include <string>
#include <vector>

// weyl headers
#include "matrix.h"

namespace weyl {
    template<int N> class irrep {
    public:
        // allocate enough space, does not create a valid irrep label
        irrep();

        // create irrep with given index
        /* explicit */ irrep(int index);

        // access elements, k = 1, ..., N
        int &operator()(int k);
        const int &operator()(int k) const;

        // convert to human-readable form
        operator std::string() const;

        // returns the index of this irrep (0, 1, 2, ...)
        int index() const;

        // return number of states in this irrep
        int dimension() const;

        // representation matrices of raising/lowering operators
        // l = 1, ..., N-1
        matrix lowering_operator(int l) const;
        matrix raising_operator(int l) const;

        // representation matrices of single-entry matrices
        // p, q = 1, ..., N
        matrix single_entry_matrix(int p, int q) const;

        // decompose product of two irreps
        std::vector<irrep> operator*(const irrep& other) const;

    private:
        std::vector<int> elem;
    };
}

// inline code follows

template<int N> inline int& weyl::irrep<N>::operator()(int k) {
    return elem[k - 1];
}

template<int N> inline const int& weyl::irrep<N>::operator()(int k) const {
    return elem[k - 1];
}

#endif // WEYL_IRREP_H_
```

```
// this file implements the declarations from the following file
#include "irrep.h"

// C headers
// none (delete this line if you add some)

// C++ headers
#include <sstream>
#include <string>
#include <vector>

// weyl headers
#include "binomial.h"
#include "matrix.h"
#include "pattern.h"


template<int N> weyl::irrep<N>::irrep() : elem(N) {}

template<int N> weyl::irrep<N>::irrep(int index) : elem(N) {
    for (int i = 0; index > 0 && i < N; ++i) {
        for (int j = 1; weyl::binomial(N - i - 1 + j, N - i - 1) <= index; j <<= 1) {
            elem[i] = j;
        }

        for (int j = elem[i] >> 1; j > 0; j >>= 1) {
            if (weyl::binomial(N - i - 1 + (elem[i] | j), N - i - 1) <= index) {
                elem[i] |= j;
            }
        }

        index -= weyl::binomial(N - i - 1 + elem[i]++, N - i - 1);
    }
}

template<int N> weyl::irrep<N>::operator std::string() const {
    std::ostringstream result;

    result << "[";
    for (int i = 0; i < N; ++i) {
        if (i > 0) result << ", ";
        result << elem[i];
    }
    result << "]";

    return result.str();
}

template<int N> int weyl::irrep<N>::index() const {
    int result = 0;

    for (int i = 0; elem[i] > elem[N - 1]; ++i) {
        result += weyl::binomial(N - i - 1 + elem[i] - elem[N - 1] - 1, N - i - 1);
    }

    return result;
}

template<int N> int weyl::irrep<N>::dimension() const {
    long long numerator = 1, denominator = 1;

    for (int i = 1; i < N; ++i) {
        for (int j = 0; i + j < N; ++j) {
            numerator *= elem[j] - elem[i + j] + i;
```

```
            denominator *= i;
        }
    }

    return numerator / denominator;
}

template<int N> weyl::matrix weyl::irrep<N>::lowering_operator(int l) const {
    int dim = dimension();
    matrix result(dim);
    pattern<N> pat(*this, 0);

    // loop invariant: i == pattern.index()
    for (int i = 0; i < dim; ++i, ++pat) {
        for (int k = 1; k <= l; ++k) {
            // pattern still valid if we decrease entry (k,l)?
            if ((k >= l || pat(k, l) - 1 >= pat(k, l - 1))
                    && pat(k, l) - 1 >= pat(k + 1, l + 1)) {
                --pat(k, l);
                int h = pat.index();
                ++pat(k, l);
                result(h, i) = pat.lowering_coeff(k, l);
            }
        }
    }

    return result;
}

template<int N> weyl::matrix weyl::irrep<N>::raising_operator(int l) const {
    int dim = dimension();
    matrix result(dim);
    pattern<N> pat(*this, 0);

    // loop invariant: i == pattern.index()
    for (int i = 0; i < dim; ++i, ++pat) {
        for (int k = 1; k <= l; ++k) {
            // pattern still valid if we increase entry (k,l)?
            if ((k <= 1 || pat(k, l) + 1 <= pat(k - 1, l - 1))
                    && pat(k, l) + 1 <= pat(k, l + 1)) {
                ++pat(k, l);
                int h = pat.index();
                --pat(k, l);
                result(h, i) = pat.raising_coeff(k, l);
            }
        }
    }

    return result;

}

template<int N> weyl::matrix weyl::irrep<N>::single_entry_matrix(int p, int q) const {
    if (p < q) {
        matrix result = raising_operator(p);
        for (int i = p + 1; i < q; ++i) result = result.commutator(raising_operator(i));
        return result;
    } else if (p > q) {
        matrix result = lowering_operator(q);
        for (int i = q + 1; i < p; ++i) result = lowering_operator(i).commutator(result);
        return result;
    }

    // p == q not implemented
    assert(false);
    return matrix(1);
```

```
}

template<int N> std::vector<weyl::irrep<N> > weyl::irrep<N>::operator*(const weyl::irrep<N>& other) c
    weyl::pattern<N> low(*this, 0), high(*this, 0);
    weyl::irrep<N> trial(other);
    std::vector<irrep> result;
    int k = 1, l = N;

    do {
        while (k <= N) {
            --l;
            if (k <= l) {
                low(k, l) = std::max(high(k + N - l, N), high(k, l + 1) + trial(l + 1) + trial(l));
                high(k, l) = high(k, l + 1);
                if (k > 1 && high(k, l) > high(k - 1, l - 1)) {
                    high(k, l) = high(k - 1, l - 1);
                }
                if (l > 1 && k == l && high(k, l) > trial(l - 1) - trial(l)) {
                    high(k, l) = trial(l - 1) - trial(l);
                }
                if (low(k, l) > high(k, l)) {
                    break;
                }
                trial(l + 1) += high(k, l + 1) - high(k, l);
            } else {
                trial(l + 1) += high(k, l + 1);
                ++k;
                l = N;
            }
        }

        if (k > N) {
            result.push_back(trial);
        } else {
            ++l;
        }

        while (k != 1 || l != N) {
            if (l == N) {
                l = --k - 1;
                trial(l + 1) -= high(k, l + 1);
            } else if (low(k, l) < high(k, l)) {
                --high(k, l);
                ++trial(l + 1);
                break;
            } else {
                trial(l + 1) -= high(k, l + 1) - high(k, l);
            }
            ++l;
        }
    } while (k != 1 || l != N);

    return result;
}


// make sure the linker finds the following classes
template class weyl::irrep<1>;
template class weyl::irrep<2>;
template class weyl::irrep<3>;
template class weyl::irrep<4>;
template class weyl::irrep<5>;
template class weyl::irrep<6>;
template class weyl::irrep<7>;
template class weyl::irrep<8>;
template class weyl::irrep<9>;
```

Pattern class (header and source file)

Creates objects `weyl::pattern` and provides a member function `index()` to map patterns to the natural numbers. It has overloaded operators `operator++();` and `operator--();` to find the pattern with the next larger or smaller index and can generate the coefficients of raising and lowering operators stated in (2.13) and (2.12) with `lowering_coeff` and `raising_coeff`.

```
#ifndef WEYL_PATTERN_H_
#define WEYL_PATTERN_H_

// C headers
// none (delete this line if you add some)

// C++ headers
#include <string>
#include <vector>

// weyl headers
#include "irrep.h"

namespace weyl {
    template<int N> class pattern {
    public:
        // allocate enough space, does not produce a valid pattern
        pattern();

        // create pattern with given index in irrep
        // index = 0, ..., irrep.dimension()-1
        pattern(const irrep<N>& irrep, int index);

        // return index inside of the irrep
        int index() const;

        // access elements; l = 1, ..., N; k = 1, ..., l
        int &operator()(int k, int l);
        const int &operator()(int k, int l) const;

        // convert to human-readable form
        operator std::string() const;

        // set pattern to the lexicographically next/previous one
        // return false if already at last/first pattern
        // if "false", no longer contains a valid pattern
        bool operator++();
        bool operator--();

        // matrix element of raising/lowering operator J^{(l)}
        // between this + M^{k,l} and this pattern
        double lowering_coeff(int k, int l) const;
        double raising_coeff(int k, int l) const;

        // return weight
        std::vector<int> weight() const;

    private:
        std::vector<int> elem;
    };
};

template<int N> int& weyl::pattern<N>::operator()(int k, int l) {
    return elem[(N * (N + 1) - l * (l + 1)) / 2 + k - 1];
}

template<int N> const int& weyl::pattern<N>::operator()(int k, int l) const {
    return elem[(N * (N + 1) - l * (l + 1)) / 2 + k - 1];
```

```
}

#endif // WEYL_PATTERN_H_
```

```
// this file implements the declarations from the following file
#include "pattern.h"

// C headers
#include <cassert>
#include <cmath>

// C++ headers
#include <algorithm>
#include <sstream>
#include <string>
#include <vector>

// weyl headers
#include "irrep.h"


template<int N> weyl::pattern<N>::pattern() : elem(N * (N + 1) / 2) {}

template<int N> weyl::pattern<N>::pattern(const weyl::irrep<N>& irrep, int index) : elem(N * (N + 1)
    for (int k = 1; k <= N; ++k) (*this)(k, N) = irrep(k);

    for (int l = N - 1; l >= 1; --l) {
        for (int k = 1; k <= l; ++k) {
            (*this)(k, l) = (*this)(index < 0 ? k : k + 1, l + 1);
        }
    }

    if (index < 0) {
        while (++index < 0) {
            bool b = --*this;
            assert(b);
        }
    } else {
        while (index--- > 0) {
            bool b = ++*this;
            assert(b);
        }
    }
}

template<int N> int weyl::pattern<N>::index() const {
    int result = 0;

    for (typename weyl::pattern<N> p(*this); --p; ++result) {}

    return result;
}

template<int N> weyl::pattern<N>::operator std::string() const {
    std::ostringstream result;

    result << "[";
    for (int l = N; l >= 1; --l) {
        for (int k = 1; k <= l; ++k) {
            if (k > 1) result << ",␣";
            result << (*this)(k, l);
        }
        if (l > 1) result << ";␣";
    }
    result << "]";
```

```
    return result.str();
}

template<int N> bool weyl::pattern<N>::operator++() {
    int k = 1, l = 1;

    while (l < N && (*this)(k, l) == (*this)(k, l + 1)) {
        if (--k == 0) k = ++l;
    }

    if (l == N) return false;
    ++(*this)(k, l);

    while (k != 1 || l != 1) {
        if (++k > l) {
            k = 1;
            --l;
        }

        (*this)(k, l) = (*this)(k + 1, l + 1);
    }

    return true;
}

template<int N> bool weyl::pattern<N>::operator--() {
    int k = 1, l = 1;

    while (l < N && (*this)(k, l) == (*this)(k + 1, l + 1)) {
        if (--k == 0) k = ++l;
    }

    if (l == N) return false;
    --(*this)(k, l);

    while (k != 1 || l != 1) {
        if (++k > l) {
            k = 1;
            --l;
        }

        (*this)(k, l) = (*this)(k, l + 1);
    }

    return true;
}

template<int N> double weyl::pattern<N>::lowering_coeff(int k, int l) const {
    double result = 1.0;

    for (int i = 1; i <= l + 1; ++i) {
        result *= (*this)(i, l + 1) - (*this)(k, l) + k - i + 1;
    }

    for (int i = 1; i <= l - 1; ++i) {
        result *= (*this)(i, l - 1) - (*this)(k, l) + k - i;
    }

    for (int i = 1; i <= l; ++i) {
        if (i == k) continue;
        result /= (*this)(i, l) - (*this)(k, l) + k - i + 1;
        result /= (*this)(i, l) - (*this)(k, l) + k - i;
    }

    return std::sqrt(-result);
}
```

```
129
template<int N> double weyl::pattern<N>::raising_coeff(int k, int l) const {
    double result = 1.0;

    for (int i = 1; i <= l + 1; ++i) {
134     result *= (*this)(i, l + 1) - (*this)(k, l) + k - i;
    }

    for (int i = 1; i <= l - 1; ++i) {
        result *= (*this)(i, l - 1) - (*this)(k, l) + k - i - 1;
139 }

    for (int i = 1; i <= l; ++i) {
        if (i == k) continue;
        result /= (*this)(i, l) - (*this)(k, l) + k - i;
144     result /= (*this)(i, l) - (*this)(k, l) + k - i  - 1;
    }

    return std::sqrt(-result);
}
149
template<int N> std::vector<int> weyl::pattern<N>::weight() const {
    std::vector<int> result(N);

    for (int prev = 0, l = 1; l <= N; ++l) {
154     int now = 0;

        for (int k = 1; k <= l; ++k) now += (*this)(k, l);
        result[l - 1] = now - prev;
        prev = now;
159 }

    return result;
}

164
// make sure the linker finds the following classes
template class weyl::pattern<1>;
template class weyl::pattern<2>;
template class weyl::pattern<3>;
169 template class weyl::pattern<4>;
template class weyl::pattern<5>;
template class weyl::pattern<6>;
template class weyl::pattern<7>;
template class weyl::pattern<8>;
174 template class weyl::pattern<9>;
```

Permutation class (header and source file)

Creates objects `weyl::permut` and defines the member functions `apply_to(std::vector<int>&)` to permute a weight, `before(const permut&)` and `after(const permut&)` to connect permutations, `inverse()` to find the inverse permutation. It can furthermore decompose itself into transposition of type $(1, x)$ with `transpositions()` and return a representation matrix of its class representative with `normal_form(const std::vector<int>&)` dependent of the i-weight.

```
#ifndef WEYL_PERMUT_H_
#define WEYL_PERMUT_H_

// C headers
// none (delete this line if you add some)

// C++ headers
#include <string>
#include <vector>

// weyl headers
#include "matrix.h"
#include "irrep.h"


namespace weyl {
    // stores a permutation of the numbers 0..N-1
    template<int N> class permut {
    public:
        // allocate enough space, does not create a valid permutation
        permut();

        // copy from vector
        /* explicit */ permut(const std::vector<int>& input);

        // mapping of permutations to numbers 0, ..., N!-1
        explicit permut(int index);
        int index() const;

        // create a transposition a <-> b
        // a == b creates identity
        permut(int a, int b);

        // access elements, k = 0, ..., N-1
        int& operator ()(int k);
        const int& operator ()(int k) const;

        // convert to human-readable form
        operator std::string() const;

        // return inverse permutation
        permut inverse() const;

        // compose two permutations
        // a.before(b)(x) = b(a(x))
        // a.after(b)(x) = a(b(x))
        permut after(const permut& other) const;
        permut before(const permut& other) const;

        // permutes the elements in weight
        // apply_to works in place, applied_to returns a copy
        std::vector<int> applied_to(const std::vector<int>& weight) const;
        void apply_to(std::vector<int>& weight) const;

        // decomposes into transpositions 0 <-> n
```

```
        // original permutation = permut(result[0]).before(permut(result[1])).before(...)
        std::vector<int> transpositions() const;

        // return representation matrix of this permutation in given irrep
        matrix repr_matrix(const irrep<N>& ir) const;

        // return "representative" of the class of this permutation
        // result has the same effect on weight as this, but is lexicographically smallest
        permut normal_form(const std::vector<int>& weight) const;

    private:
        std::vector<int> elem;
    };
}

// inline code follows

template<int N> int& weyl::permut<N>::operator ()(int k) {
    return elem[k];
}

template<int N> const int& weyl::permut<N>::operator ()(int k) const {
    return elem[k];
}

#endif // WEYL_PERMUT_H_
```

```
// this file implements the declarations from the following file
#include "permut.h"

// C headers
#include <cassert>

// C++ headers
#include <algorithm>
#include <sstream>
#include <string>
#include <vector>

// weyl headers
#include "matrix.h"
#include "irrep.h"
#include "pattern.h"


template<int N> weyl::permut<N>::permut() : elem(N) {}

template<int N> weyl::permut<N>::permut(const std::vector<int>& input) : elem(input) {
    assert(static_cast<int>(input.size()) == N);
}

template<int N> weyl::permut<N>::permut(int index) : elem(N) {
    for (int i = N - 1; i >= 0; --i) {
        elem[i] = index % (N - i);
        index /= N - i;
        for (int j = i + 1; j < N; ++j) {
            if (elem[j] >= elem[i]) ++elem[j];
        }
    }
}

template<int N> int weyl::permut<N>::index() const {
    int result = 0;

    for (int seen = 0, i = 0; i < N - 1; seen |= 1 << elem[i++]) {
```

```
            result *= N - i;
            result += elem[i];
            for (int j = seen & ((1 << elem[i]) - 1); j > 0; j &= j - 1) {
                --result;
            }
        }

        return result;
    }

    template<int N> weyl::permut<N>::permut(int a, int b) : elem(N) {
        assert(0 <= a && a < N);
        assert(0 <= b && b < N);
        for (int i = 0; i < N; ++i) elem[i] = i;
        std::swap(elem[a], elem[b]);
    }

    template<int N> weyl::permut<N>::operator std::string() const {
        std::ostringstream result;

        result << "(";
        for (int i = 0; i < N; ++i) {
            if (i > 0) result << ",␣";
            result << elem[i];
        }
        result << ")";

        return result.str();
    }

    template<int N> weyl::permut<N> weyl::permut<N>::inverse() const {
        permut<N> result;
        for (int i = 0; i < N; ++i) result.elem[elem[i]] = i;
        return result;
    }

    template<int N> weyl::permut<N> weyl::permut<N>::before(const weyl::permut<N>& other) const {
        permut<N> result;
        for (int i = 0; i < N; ++i) result.elem[i] = other.elem[elem[i]];
        return result;
    }

    template<int N> weyl::permut<N> weyl::permut<N>::after(const weyl::permut<N>& other) const {
        permut<N> result;
        for (int i = 0; i < N; ++i) result.elem[i] = elem[other.elem[i]];
        return result;
    }

    template<int N> std::vector<int> weyl::permut<N>::applied_to(const std::vector<int>& obj) const {
        std::vector<int> result = obj;
        for (int i = 0; i < N; ++i) result[elem[i]] = obj[i];
        return result;
    }

    template<int N> void weyl::permut<N>::apply_to(std::vector<int>& obj) const {
        std::vector<char> done(N, false);

        for (int i = 0; i < N; ++i) {
            for (int j = i; !done[j]; ) {
                done[j] = true;
                j = elem[j];
                std::swap(obj[i], obj[j]);
            }
        }
    }
```

```
template<int N> std::vector<int> weyl::permut<N>::transpositions() const {
    std::vector<int> perm(elem);
    std::vector<int> result;

    for (int i = perm.size() - 1; i > 0; --i) {
        if (i != perm[i]) {
            for (int k = i - 1; k >= 0; --k) {
                if (perm[k] == i) {
                    // swap searched entry to position 1
                    if (k != 0) {
                        result.push_back(k);
                        perm[k] = perm[0];
                        perm[0] = i;
                    }

                    // then swap first entry with destination position
                    result.push_back(i);
                    perm[0] = perm[i];
                    perm[i] = i;
                }
            }
        }
    }

    return result;
}

template<int N> weyl::matrix weyl::permut<N>::repr_matrix(const weyl::irrep<N>& ir) const {
    int dim = ir.dimension();
    weyl::matrix result(dim, dim);
    for (int i = 0; i < N; ++i) result(i, i) = 1.0;

    std::vector<int> transpos = transpositions();
    for (std::vector<int>::const_iterator it = transpos.begin(); it != transpos.end(); ++it) {
        weyl::matrix a = ir.single_entry_matrix(*it + 1, 1).exp(dim);
        weyl::matrix b = a * (-ir.single_entry_matrix(1, *it + 1)).exp(dim) * a;

        weyl::pattern<N> pat(ir, 0);
        for (int i = 0; i < dim; ++i, ++pat) {
            if (pat(1, 1) % 2 != 0) {
                for (int j = 0; j < dim; ++j) {
                    b(i, j) = -b(i, j);
                }
            }
        }

        result = result * b;
    }

    return result;
}

template<int N> weyl::permut<N> weyl::permut<N>::normal_form(const std::vector<int>& weight) const {
    std::vector<int> new_weight = this->applied_to(weight);
    std::vector<char> taken(N, false);
    permut result;

    for (int i = 0; i < N; ++i) {
        for (int j = 0; true; ++j) {
            if (!taken[j] && new_weight[j] == weight[i]) {
                taken[j] = true;
                result.elem[i] = j;
                break;
            }
        }
    }
```

```
    return result;
}

// make sure the linker finds the following classes
template class weyl::permut<1>;
template class weyl::permut<2>;
template class weyl::permut<3>;
template class weyl::permut<4>;
template class weyl::permut<5>;
template class weyl::permut<6>;
template class weyl::permut<7>;
template class weyl::permut<8>;
template class weyl::permut<9>;
```

Coefficient class (header and source file)

Implements the algorithm, stores the Clebsch-Gordan coefficients and makes them accessible for usage.

```
#ifndef WEYL_COEFF_H_
#define WEYL_COEFF_H_

// C headers
// none (delete this line if you add some)

// C++ headers
#include <utility>
#include <vector>

// weyl headers
// none (delete this line if you add some)


namespace weyl {
    template<int N> class coeff {
    public:
        coeff(const irrep<N>& irrep,
              const irrep<N>& factor1,
              const irrep<N>& factor2);
    };

private:
    typedef std::pair<int, int> state_id;

    std::vector<std::pair<state_id, state_id>> sift_product_states() const;
};

#endif
```

```
// this file implements the declarations from the following file
#include "coeff.h"

// C headers
#include <cassert>

// C++ headers
#include <vector>

// weyl headers
#include "irrep.h"
#include "pattern.h"

// implementation follows
```

```
16  template<int> weyl::coeff::coeff(const weyl::irrep<N>& irrep,
                                     const weyl::irrep<N>& factor1,
                                     const weyl::irrep<N>& factor2) {
    }

21  template<int N> std::vector<std::pair<state_id, state_id> > sift_product_states(const weyl::irrep<N>&
                                                                                    const weyl::irrep<N>&
    }

    // make sure the linker finds the following classes
26  template class weyl::coeff<1>;
    template class weyl::coeff<2>;
    template class weyl::coeff<3>;
    template class weyl::coeff<4>;
    template class weyl::coeff<5>;
31  template class weyl::coeff<6>;
    template class weyl::coeff<7>;
    template class weyl::coeff<8>;
    template class weyl::coeff<9>;
```

Bibliography

[1] A. Alex, M. Kalus, A. Huckleberry, and J. von Delft, "A numerical algorithm for the explicit calculation of SU(n) and SL($n, \mathbb{C}$) Clebsch-Gordan coefficients," *J. Math. Phys.*, vol. 52, p. 023507, 2011.

[2] T. A. Costi, L. Bergqvist, A. Weichselbaum, J. von Delft, T. Micklitz, A. Rosch, P. Mavropoulos, H. Dederichs, F. Mallet, L. Saminadayar, and C. Buerle, "Kondo decoherence: finding the right spin model for iron impurities in gold and silver," *Phys. Rev. Lett.*, vol. 102(5), p. 056802, 2009.

[3] A. Alex, J. von Delft, L. Everding, and P. Littelmann, "Clebsch-gordan coefficients via weyl symmetry group.".

[4] I. M. Gelfand and M. L. Tsetlin, *Matrix elements for the unitary group.* Dokl. Akad. Nauk SSSR 71, 825 and 1017, 1950.

[5] I.M. Gelfand, R. A. Minlos, and Z. Ya. Shapiro, *Representations of the Rotation and Lorentz Group.* New York: Pergamon, 1963.

[6] A. O. Barut and R. Raczka, *Theory of group representations and applications.* Warszawa: PWN-Polish Scientific Publ., 2nd, revised ed., 1986.

[7] A. Alex *private communication*, 2011.

[8] I. P. McCulloch and M. Gulacsi, "The non-abelian density matrix renormalization group algorithm," *Europhys. Lett.*, vol. 57(6), p. 852, 2002.

[9] Ö. A. I. Tóth, C. P. Moca and G. Zaránd, "Density matrix numerical renormalization group for non-abelian symmetries," *Phys. Rev. B*, vol. 78(24), p. 245109, 2008.

7 Acknowledgements

I would like to thank Prof. Dr. Jan von Delft for giving me the opportunity to work on a current research project in his group that has the potential for actual scientific gain.

I would also like to thank my supervisor Arne Alex, who always helped me overcome the many obstacles I faced and who took the time to give me detailed support when I needed it. I also owe much of my understanding of programming to him. Thank you very much!

Finally, I would like to thank Mathias Arens for his continuous linguistic inspiration.

Hiermit versichere ich, dass ich die vorliegende Arbeit selbstständig und ohne fremde Hilfe angefertigt und nur die im Literaturverzeichnis angeführten Quellen verwendet habe.

München, den 12.08.2011