Calculating Clebsch-Gordan Coefficients and Complete Combined State Space for N-Particle Spin Systems

Vandan Pokal[1] **and Peter Alex Mahhov**[2]

[1]**College of Arts and Sciences, NYU, New York, NY**
[2]**NYU Abu Dhabi, Abu Dhabi, UAE**

ABSTRACT

This report describes a method to compute Clebsch-Gordan coefficients for an arbitrary-numbered set of particles with different spins, by using the operator method to calculate the complete set of combined eigenstates of the system.

INTRODUCTION

The addition of angular momentum is an integral part in many areas of physics - from atomic spectroscopy to nuclear and particle collisions (Sakurai and Napolitano, 2020). It also serves as a fundamental building block in quantum computation, where qubits (quantum bits) are single particles with spin-1/2 and n such particles have to be combined into n-qubits. Since a single spin-1/2 particle has 2 states: up or down, then just one combination, a 2-qubit, can already exhibit 4 states. Computing multiple combinations of spin-particles allows the performance of certain large computations much faster than on conventional computers. Furthermore, combining spin angular momentum creates the so called entangled states or Bell states that can influence the measurement on either side, Einstein's famous "spooky action at a distance". These concepts are still being researched and advanced for ever more practical applications.

When spin-angular momentum or spins are combined they add in a specific way that doesn't lose the information about the composite spins. The Clebsch-Gordan coefficients, first discovered by German mathematicians Alfred Clebsch and Paul Gordan, specify the way the composite spins combine. To put it simply it is a basis transformation, one composite and the other irreducible. The Clebsch-Gordan coefficients also give us the probability of finding a specific particle in a particular spin-orientation in the combined state. As the number of particles increases, this task becomes increasingly cumbersome as it involves large matrices and a large number of coefficients. For this reason, computational methods are preferred due to both the ease of use and the significantly faster completion time. In this project, we use spin operators by which we obtain the Clebsch-Gordan coefficients and decomposition of combined states for any number of particles, only limited by time and computational power.

BACKGROUND

Spin Angular Momentum

Spin is an intrinsic characteristic that each fundamental particle possesses. It can be modelled, classically, as the spin about a particle's axis (Griffiths and Schroeter, 2018). However, quantum mechanically there is no such thing as a classical spin, therefore it is as fundamental as the charge, mass, and other such inherent physical properties of a particle. Spin was first observed by Otto Stern and Gerlach in their famous Stern-Gerlach experiments, where they beamed silver atoms through an inhomogeneous magnetic field and found that instead of observing a continuous spectrum they observed discrete values, 1/2 and -1/2, which showed that spin angular momentum is quantized.

Spin matrices

Spin matrices are the matrix form of the Spin operators that measure the spin of a particle in a given basis, such as x, y, or z. For simplicity we consider the spins in the z-basis:

$$\hat{S}_z|sm\rangle = \hbar m|sm\rangle \tag{1}$$

Here m is the spin-projection and s is the spin of the particle.
Each spin has the projection set:

$$s_i \to m_i \in \{s_i, s_i - 1, ... - s_i\} \tag{2}$$

Projecting the right hand side state vector, $|s,m'\rangle$ onto the LHS, one can recover a matrix form of $\hat{S}_z$,

$$S_{z_{mm'}} = \hbar\langle sm'|\hat{S}_z|sm\rangle = \hbar\delta_{mm'}m \tag{3}$$

which is a diagonal matrix:

$$S_z = \hbar \begin{bmatrix} s & & & & \\ & s-1 & & & \\ & & \ddots & & \\ & & & s-(n-2) & \\ & & & & -s \end{bmatrix} \tag{4}$$

Here n is the cardinal number of the projection set for the spin s_i.
The states themselves can be represented by column vectors:

$$|s = \frac{1}{2} m = +\frac{1}{2}\rangle = \begin{bmatrix} 1 \\ 0 \end{bmatrix} \tag{5}$$

$$|s = \frac{1}{2} m = -\frac{1}{2}\rangle = \begin{bmatrix} 0 \\ 1 \end{bmatrix} \tag{6}$$

(5) and (6) are the eigen-vectors of the Spin matrix S_z for spin $\frac{1}{2}$. Which means that the state vectors form an eigen-basis for the Spin matrix for a particular spin in x,y,z basis. From here onwards, multiplication by constant $\hbar$ is implied for all spin matrices.

In a similar way, using the fact that

$$\hat{S}_\pm|sm\rangle = \sqrt{(s \mp m)(s \pm m + 1)}|sm \pm 1\rangle \tag{7}$$

we can compute the raising and lowering matrices:

$$S_{+_{m,m'}} = \delta_{m,m+1}\sqrt{(s-m)(s+m+1)} \tag{8}$$

$$S_{-_{m,m'}} = \delta_{m+1,m}\sqrt{(s+m)(s-m+1)} \tag{9}$$

These are off-diagonal matrices with the entries shown above. The raising and lowering matrices can be used to jump from one state to another, depending on the eigenvalues m. The raising operator takes the state from $|sm\rangle$ to $|sm+1\rangle$.

The calculations above can be replicated for a particle with any value of spin, for example for spin-1:

$$S_-|m=1\rangle = \begin{pmatrix} \sqrt{2} & & \\ & \sqrt{2} & \end{pmatrix} \begin{bmatrix} 1 \\ 0 \\ 0 \end{bmatrix} = \begin{bmatrix} 0 \\ \sqrt{2} \\ 0 \end{bmatrix} \to^{(normalisation)} \begin{bmatrix} 0 \\ 1 \\ 0 \end{bmatrix} = |m=0\rangle \tag{10}$$

However, if this operation is done on the maximum and minimum projection states, the result is 0:

$$S_+|m=+1\rangle = 0 \qquad\qquad S_-|m=-1\rangle = 0 \tag{11}$$

This result agrees with the fact that the basis for S_z for spin-1 is formed by 3 vectors. Since it is a hermitian matrix, the eigenvectors are orthogonal. Only exactly 3 vectors can span the space R^3z. Thus, assuming that we have only one state vector, we can use the raising and lowering operators exhaustively to find the other states. Furthermore, the S_x and S_y matrices can also be derived from the raising and lowering matrices. While we do not use these in the project, this can be easily implemented.

Algebra of Addition of Spin Angular Momentum

Classically we could just add the angular momentum vectors and get the net momentum of the system as $\overrightarrow{L}_{net} = \overrightarrow{L}_1 + \overrightarrow{L}_2$, but in the quantum case the precise value is not determined until the moment of measurement, and when measured, the state collapses. Thus, in that case, the measurement operators combine, in the form (Sakurai and Napolitano, 2020):

$$S^{(12)} = S^{(1)} \oplus S^{(2)} = S^{(1)} \otimes I^{(2)} + I^{(1)} \otimes S^{(2)} \tag{12}$$

where $S^{(1)}$ and $S^{(2)}$ are the spin operators for particle 1 and 2 respectively, and $I^{(1)}$, $I^{(2)}$ are the unit operators in the eigenspace of particle 1 and of particle 2. This treats each eigenspace as a distinct vector space, from which it follows that the composite vector space is the direct sum of the two. This can be generalized for n-particles as:

$$S^{(123..n)} = \bigoplus_i S^{(i)} \tag{13}$$

Since the operator for each particle can be shown in Cartesian component form:

$$S^{(i)} = S_x^{(i)}\hat{x} + S_y^{(i)}\hat{y} + S_z^{(i)}\hat{z} \tag{14}$$

Then the combined S_z operator is:

$$S_z^{(123...n)} = \bigoplus_i S_z^{(i)} \tag{15}$$

The combined operator has a complete set of eigenvectors that span the space but gives no information about how the combined eigen-states decompose into a linear combination of the irreducible states. That is, each combined state can be written as a linear combination Kronecker product of the eigenstates of S_z:

$$|S,M\rangle = \sum_{m=\sum_i m_i} C^{s_1 s_2 s_3 ... s_n}_{m1m2m3m} |s_1 m_1\rangle \otimes |s_2 m_2\rangle \otimes |s_3 m_3\rangle ..... \otimes |s_n m_n\rangle \tag{16}$$

The constants $C^{s_1 s_2 s_3 ... s_n}_{m1m2m3m}$ are the normalised Clebsch-Gordan coefficients that tell us how the combined or coupled states decompose into coupled states or the product states of the eigenstates of irreducible representations of the rotation groups, namely, the S_z matrices.
The decomposition of n-particle systems can become very tedious to do analytically as the number of combinations increases. These combinations are just the linear combinations of uncoupled states that have a sum of projections such that $M = \sum_i m_j^{(i)}$ For example, for spin-$\frac{1}{2}$, all the spin states decompose as :

$$|11\rangle = |\frac{1}{2}\frac{1}{2}\rangle^{(1)} \otimes |\frac{1}{2}\frac{1}{2}\rangle^{(2)} \tag{17}$$

$$|10\rangle = \frac{1}{\sqrt{2}}|\frac{1}{2}\frac{1}{2}\rangle^{(1)} \otimes |\frac{1}{2}-\frac{1}{2}\rangle^{(2)} + \frac{1}{\sqrt{2}}|\frac{1}{2}-\frac{1}{2}\rangle^{(1)} \otimes |\frac{1}{2}\frac{1}{2}\rangle^{(2)} \tag{18}$$

$$|1-1\rangle = |\frac{1}{2}-\frac{1}{2}\rangle^{(1)} \otimes |\frac{1}{2}-\frac{1}{2}\rangle^{(2)} \tag{19}$$

$$|00\rangle = \frac{1}{\sqrt{2}}|\frac{1}{2}\frac{1}{2}\rangle^{(1)} \otimes |\frac{1}{2}-\frac{1}{2}\rangle^{(2)} - \frac{1}{\sqrt{2}}|\frac{1}{2}-\frac{1}{2}\rangle^{(1)} \otimes |\frac{1}{2}\frac{1}{2}\rangle^{(2)} \tag{20}$$

These are 4 combined states in a linear combination of 4 uncoupled states. Both sets, combined and uncoupled, span the combined space. Hence it can be simply called a basis transformation. The normalised Clebsch-Gordan Coefficients in this case are the coefficients before the Kronecker products. In the next section, we show how to use the raising and lowering operators to produce the complete set of combined states in the form of linear combinations of the irreducible states or the Kronecker product states. In doing so, we can resolve the Clebsch-Gordan coefficients.

THE OPERATOR METHOD: PRODUCING A COMPLETE SET OF COMBINED EIGENSTATES

Total Spin Angular Momentum Space for Spin-1 and Spin-$\frac{1}{2}$ particles

In this section, a complete set of combined states in the form of columns of a matrix are explicitly produced as a linear combination of the irreducible states, or the Kronecker products of basis-vectors of each spin matrix $S_z^{(1)}(s=1)$ and $S_z^{(2)}(s=\frac{1}{2})$.

Step 1

The maximum projection of the non-degenerate combined state($|S_{nd}, M_{max}\rangle$) is always a just one irreducible Kronecker product because there is just such combination of the spin-projections m_i that produces M_{max}. The value of S_{nd} for non-degenerate states is $S = \sum_i s_i$. In this case, $|S_{nd} = \frac{3}{2}, M_{max} = \frac{3}{2}\rangle$.

$$|\frac{3}{2},\frac{3}{2}\rangle = |11\rangle^{(1)} \otimes |\frac{1}{2}\frac{1}{2}\rangle^{(2)} \tag{21}$$

The projection set for spin-1 is $\{1,0,-1\}$. The eigenvector of $S_z(s=1)$ that corresponds to the maximum projection eigenvalue, which are just the columns of a 3×3 identity matrix, $m=1$, can be represented as:

$$|11\rangle = \begin{bmatrix}1\\0\\0\end{bmatrix} \tag{22}$$

The projection set for spin-1/2 is $\{\frac{1}{2},-\frac{1}{2}\}$. The max-eigenvector is:

$$|\frac{1}{2}\frac{1}{2}\rangle = \begin{bmatrix}1\\0\end{bmatrix} \tag{23}$$

Then, the maximum-non-degenerate combined state using kronecker product of (22) and (23) is:

$$|11\rangle \otimes |\frac{1}{2}\frac{1}{2}\rangle = \begin{bmatrix}1\\0\\0\end{bmatrix} \otimes \begin{bmatrix}1\\0\end{bmatrix} = \begin{bmatrix}1\\0\\0\\0\\0\\0\end{bmatrix} \tag{24}$$

Step 2

Constructing the combined lowering operator that operates in the combined space:

$$S_-^{(1)} \oplus S_-^{(2)} = S_-^{(1)} \otimes I^{(2)(2\times 2)} + I^{(2)(3\times 3)} \otimes S_-^{(1)} \tag{25}$$

Which, in matrix form, is:

$$S_-^{(12)} = \begin{pmatrix}0&0&0\\\sqrt{2}&0&0\\0&\sqrt{2}&0\end{pmatrix} \otimes \begin{pmatrix}1&0\\0&1\end{pmatrix} + \begin{pmatrix}1&0&0\\0&1&0\\0&0&1\end{pmatrix} \otimes \begin{pmatrix}0&0\\1&0\end{pmatrix} = \begin{pmatrix}0&0&0&0&0&0\\1&0&0&0&0&0\\\sqrt{2}&0&0&0&0&0\\0&\sqrt{2}&1&0&0&0\\0&0&\sqrt{2}&0&0&0\\0&0&0&\sqrt{2}&1&0\end{pmatrix} \tag{26}$$

Step 3

Operate the combined lowering operator on the maximum non-degenerate state vector and retrieve the non-degenerate state vectors, that is sequentially operate until $S_-^{(12)}|S_{nd}M_{min}\rangle = 0$. For examples,

$$S_-^{(12)}|S_{nd}M_{max}\rangle = \begin{pmatrix}0&0&0&0&0&0\\1&0&0&0&0&0\\\sqrt{2}&0&0&0&0&0\\0&\sqrt{2}&1&0&0&0\\0&0&\sqrt{2}&0&0&0\\0&0&0&\sqrt{2}&1&0\end{pmatrix} \begin{bmatrix}1\\0\\0\\0\\0\\0\end{bmatrix} = \begin{bmatrix}0\\1\\\sqrt{2}\\0\\0\\0\end{bmatrix} \rightarrow^{(normalising)} \begin{bmatrix}0\\\frac{1}{\sqrt{3}}\\\frac{\sqrt{2}}{\sqrt{3}}\\0\\0\\0\end{bmatrix} \tag{27}$$

Thus doing it two more times produces the set of non-degenerate state-vectors:

$$ND = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & \frac{1}{\sqrt{3}} & 0 & 0 \\ 0 & \frac{\sqrt{2}}{\sqrt{3}} & 0 & 0 \\ 0 & 0 & \frac{\sqrt{2}}{\sqrt{3}} & 0 \\ 0 & 0 & \frac{1}{\sqrt{3}} & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} \tag{28}$$

However, these aren't enough to span the entire space of R^6, thus the other 2 must lie in the orthogonal complement of the column-space of ND^T or the Null-space of ND^T , since, all the states are fundamentally orthogonal due to the combined $S_z^{(12)}$ being Hermitian.

Step 4

Solving $ND^T \overrightarrow{x} = 0$, in Row-echeleon form:

$$\begin{pmatrix} 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & \sqrt{2} & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & \frac{1}{\sqrt{2}} & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 \end{pmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \\ x_4 \\ x_5 \\ x_6 \end{bmatrix} = \begin{bmatrix} 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \end{bmatrix} \tag{29}$$

Which gives two vectors after normalisation by taking two cases for the non-pivot columns $x_3 = 0, x_5 = 1$ and $x_3 = 1, x_5 = 0$ in (29) respectively:

$$\begin{bmatrix} 0 & 0 \\ \frac{-\sqrt{2}}{\sqrt{3}} & 0 \\ \frac{1}{\sqrt{3}} & 0 \\ 0 & -\frac{1}{\sqrt{3}} \\ 0 & \frac{\sqrt{2}}{\sqrt{3}} \\ 0 & 0 \end{bmatrix} \tag{30}$$

However, this might be one of the special cases where the correct order of states is obtained. However, in the general case, this is definitely not true, or at least, in the computational case. So, we take the vector obtained by taking the last free variable $x_n = 1$ and other free variables $= 0$ and solving equation like (29). And, using the lowering operator until 0 vector is obtained. In this case,

$$\begin{pmatrix} 0 & 0 & 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 & 0 & 0 \\ \sqrt{2} & 0 & 0 & 0 & 0 & 0 \\ 0 & \sqrt{2} & 1 & 0 & 0 & 0 \\ 0 & 0 & \sqrt{2} & 0 & 0 & 0 \\ 0 & 0 & 0 & \sqrt{2} & 1 & 0 \end{pmatrix} \begin{bmatrix} 0 \\ \frac{-\sqrt{2}}{\sqrt{3}} \\ \frac{1}{\sqrt{3}} \\ 0 \\ 0 \\ 0 \end{bmatrix} = \begin{bmatrix} 0 \\ 0 \\ 0 \\ -\frac{1}{\sqrt{3}} \\ \frac{\sqrt{2}}{\sqrt{3}} \\ 0 \end{bmatrix} \rightarrow S_-^{(12)} \begin{bmatrix} 0 \\ 0 \\ 0 \\ -\frac{1}{\sqrt{3}} \\ \frac{\sqrt{2}}{\sqrt{3}} \\ 0 \end{bmatrix} = 0 \tag{31}$$

For larger cases, we append the these non-degenerates states to the ND matrix as columns at the end. Then, we find the first special solution to the equation like (29), where ND has added columns, then, use the combined lowering operator like (31).Then, repeat the process until the nullspace=0. That is we have enough state vectors to span the entire combined space. Then we square each entry while preserving sign. That is, Hadamard product with itself and consequently, a Hadamard product with the the sign function of itself:

$$TS \circ TS \circ sgn(TS) = \text{Clebsch-Gordan coefficient matrix} \tag{32}$$

An example of that can be seen in Fig 1. (computed result from the program). As, mentioned before, this can be implemented for multiple particles of different spins. The computational implementation process is shown in the following Code-Listings and Program Section:

```
[[ 1.         0.         0.         0.         0.         0.        ]
 [ 0.         0.33333333 0.         0.        -0.66666667 0.        ]
 [ 0.         0.66666667 0.         0.         0.33333333 0.        ]
 [ 0.         0.         0.66666667 0.         0.        -0.33333333]
 [ 0.         0.         0.33333333 0.         0.         0.66666667]
 [ 0.         0.         0.         1.         0.         0.        ]]
```

Figure 1. Total spin space-CGC: $1 \times \frac{1}{2}$

PROGRAM AND CODE LISTS

Spin Matrices

Projection Set for a Given Spin s

```
def get_m(s):
  M = []
  i = s
  while(i >= -s):
    M.append(i)
    i = i-1
  return M
```

S_z

This code produces the S_z matrix for any spin given as argument.

```
def get_Sz(s):
  M = get_m(s)
  Sz = np.zeros((len(M),len(M)))
  for i in range(len(M)):
    Sz[i,i] = M[i]
  return Sz
```

Raising and Lowering matrices: $S_\pm$

This code produces the $S_\pm$ and S_x, S_y matrices for any spin given as argument.

```
def getC_plus(s,m):
  return cmath.sqrt((s-m)*(s+m+1))

def getC_minus(s,m):
  return cmath.sqrt((s+m)*(s-m+1))

def get_S_plus(s):
  M = get_m(s)
  S = np.zeros((len(M),len(M)))
  for i in range(len(M)-1):
    S[i,i+1] = getC_plus(s,M[i+1])
  return S

def get_S_minus(s):
  M = get_m(s)
  S = np.zeros((len(M),len(M)))
  for i in range(len(M)-1):
    S[i+1, i] = getC_minus(s,M[i])
  return S
```

```
def get_S_x(s):
  S_plus = get_S_plus(s)
  S_minus = get_S_minus(s)
  return 0.5*(S_plus + S_minus)

def get_S_y(s):
  S_plus = get_S_plus(s)
  S_minus = get_S_minus(s)
  return -0.5j*(S_plus - S_minus)
```

Combined Spin Matrices

In these functions, the combined spin matrices are generated by using the direct sum expansion discussed in the Background.

$$S^{(1234....n)} = \bigoplus_i S^{(i)} = S^{(1)} \otimes I^{(2)} \otimes ..... + ... + I^{(1)} \otimes ..... \otimes S^{(n)} \tag{33}$$

```
 def correct_S(value, base):
  if base=="x":
    return get_S_x(value)
  elif base=="y":
    return get_S_y(value)
  elif base=="z":
    return get_Sz(value)
  elif base=="plus" or base=="+":
    return get_S_plus(value)
  elif base=="minus" or base=="-":
    return get_S_minus(value)
  else:
    raise Exception("wrong base")

def comb_lowering_matrix_addends(base, spins=spins):
  addends = []
  for i in range(len(spins)): # for each location of S
    if i == 0:
      product = correct_S(spins[0],base)
    else:
      product = np.identity(len(correct_S(spins[0],base)))

    # number of kroenecker products done to get one addend
    for j in range(len(spins)-1):
      if j+1 == i:
        product = kr_product(product, correct_S(spins[j+1],base))
      else:
        product = kr_product(product, np.identity(len(correct_S(spins[j+1],base))))
    addends.append(product)
  return addends

def comb_lowering_matrix(base,spins=spins):
  sum = 0
  addends = comb_lowering_matrix_addends(base,spins)
  for i in range(len(spins)):
    sum += addends[i]
  return sum
```

Complete Spin Space

Maximum Non-degenerate State Vectors

In this function, the maximum degenerate state vector is produced by taking the maximum eigenvalue eigenvector of spin matrix S_z for each particle/spin, and sequentially executing Kronecker product as shown in equation (16), while knowing that the CGC for this state is always 1.

```
def eigenvectors(matrix):
  D,v = np.linalg.eig(matrix)
  v = np.asmatrix(v)
  return v

# Provides the sequential kroenecker product of the eigenvectors of Sz
def seq_kr_of_Sz_evectors(spins=spins):

  # eigenvectors of Sz matrix
  Sz_matrices = []

  for i in range(len(spins)):
    Sz_matrices.append(get_Sz(spins[i]))

  e_matrices = []

  for j in Sz_matrices:
    e_matrices.append(eigenvectors(j))

  # sequential kroenecker: take the kr_product of two values, then the kr_product
  # of the result with a third value, etc until you run out of values

  if len(spins) > 1:
    s = np.asmatrix(e_matrices[0][0]).H
    for i in range(len(spins)-1):
      s = kr_product(s, np.asmatrix(e_matrices[i+1][0]).H)
  return s
```

All Non-degenerate State Vectors

This relates back to step 3 where the combined lowering matrix is applied to the maximum degenerate-vector and the results are normalised until 0 is reached.

```
 def get_max_spin_space(spins=spins):

  normalized_state_vectors = [seq_kr_of_Sz_evectors(spins)]

  state_vector = np.dot(comb_lowering_matrix("minus",spins),seq_kr_of_Sz_evectors(spins))
  norm = np.linalg.norm(state_vector)
  normalized_state_vector = state_vector/norm
  normalized_state_vectors.append(normalized_state_vector)

  while(np.count_nonzero(normalized_state_vector) != 0):
    state_vector = np.dot(comb_lowering_matrix("minus",spins),normalized_state_vector)
    norm = np.linalg.norm(state_vector)
    if (np.count_nonzero(state_vector) == 0):
      break
    normalized_state_vector = state_vector/norm
    normalized_state_vectors.append(normalized_state_vector)

  maximum_spin_space = np.zeros((len(normalized_state_vectors),len(normalized_state_vectors[0])))


  for i in range(len(normalized_state_vectors)):
    for j in range(len(normalized_state_vectors[0])):
      maximum_spin_space[i][j] = normalized_state_vectors[i][j]
  maximum_spin_space = np.asmatrix(maximum_spin_space).H
  return maximum_spin_space
```

Total Combined Spin Space

In this listing we make use of Sympy's nullspace function due to a problem with scipy.linalg.nullspace that produced a mixed up null space basis, from which the state vectors could not be retrieved. Sympy gives a rational basis that keeps the first null solution intact.

This code further applies the combined lowering matrix on the first orthogonal nullvector as in Step 4 to retrieve the 1st degenerate set of state vectors. Then, it appends the found and normalised vectors to the

non-degenerate state matrix as columns. It repeats the process as mentioned in Step 4 until the nullspace is empty.

```
 import sympy

def nullspace(matrix):
  M = sympy.Matrix(matrix)
  K = M.nullspace()
  return np.matrix(K).astype(complex)

def get_total_spin_space(spins=spins):

  max = np.asmatrix(get_max_spin_space(spins=spins))

  nullsp = nullspace(max.H)

  while(np.count_nonzero(nullsp) !=0):
      normalized_vectors = []
      vector = nullsp[0].H
      norm = np.linalg.norm(vector)
      normalized_vector = np.array(vector/norm)
      normalized_vectors.append(normalized_vector)

      while(np.count_nonzero(normalized_vector) != 0):
        vector = np.dot(comb_lowering_matrix("minus",spins=spins),normalized_vector)
        for i in range(len(vector)):
          if (abs(vector[i])<(10**-8)):
            vector[i] = 0
        norm = np.linalg.norm(vector)
        if (np.count_nonzero(vector) == 0):
          break
        normalized_vector = np.array(vector/norm)
        normalized_vectors.append(normalized_vector)
      normalized_vectors = np.column_stack(normalized_vectors)
      new_max = np.hstack((max,normalized_vectors))
      max = new_max
      nullsp = nullspace(max.H)


  total_spin_space = np.asmatrix(max).H
  return total_spin_space
```

Clebsch-Gordan Matrix

This the code for equation (32). It provides a readable ladder ordered matrix of Clebsch-Gordan Coefficients.

```
def C_G_coefficients(spins=spins):
 max = get_total_spin_space(spins=spins).H
 signs = np.sign(max.real)
 squared = np.square(max)
 return np.multiply(squared,signs)
```

Specific Combined State Vector

This code brings out the combined state vector with an input of $|S,M\rangle$, that is, combined spin and its projection. This function, while not called in the main program, would help to extract data from the CG matrices in a way that would facilitate getting state probabilities in the future.

```
def state_column(given_spin, given_m, spins=spins):
 C_G = C_G_coefficients(spins=spins)
 total_spin = np.sum(spins)
 temp = total_spin
 place = 0
 index = 0
```

```
  while(temp != given_spin):
    place += 1
    temp -= 1
    if (temp < 0):
      raise Exception("Spin not present in columns")
  for i in range(place):
    index += (len(get_m(total_spin)))

  for i in (get_m(given_spin)):
    if given_m == i:
      break
    else:
      index += 1
  return C_G[:,index]
```

RESULTS AND EFFICIENCY

The above implementation functioned perfectly, successfully reproducing the Clebsch-Gordan coefficients table as seen in Griffiths and Schroeter (2018) (see Figure 1 above). However, for practical application the computational time increased drastically when using higher spins (see Table 1) or a larger number of particles (see Table 2). This indicates that while correct, the code is not very efficient, most likely due to heavy usage of for loops and while loops in the algorithm.

System [spins]	Calculation time [s]
0.5, 0.5	0.01
1, 0.5	0.01
1, 1	0.04
2, 1	0.08
1, 1, 1	0.56
0.5, 0.5, 0.5, 0.5, 0.5	1.51
2, 1, 1	2.16
5, 3	4.99
4, 2, 1	39.73

Table 1. Example CG coefficient calculation speeds of various n-particle systems.

System [spins]	Calculation time [s]
1, 1	0.04
1, 1, 1	0.56
1, 1, 1, 1	18.4
1, 1, 1, 1, 1	735.8

Table 2. Example CG coefficient calculation speeds of n-particle systems with all spins 1.

Furthermore, the initial usage of a custom Kronecker product algorithm (see Appendix 2) that worked element-wise slowed down the process. Fortunately it was eventually possible to replace it with numpy's matrix-wise implementation without breaking the program, increasing the calculation speed.

This program serves not only as an accompanying tool in tackling problems that involve large sets of spins but also as way to understand the order of degeneracy, which is an emergent property.

Figure 2 and Figure 3 are some examples of the Clebsch-Gordan tables produced by the program.

```
55 coef = C_G_coefficients([0.5,0.5,0.5]).real
56 print(coef)
```

```
[[ 1.          0.          0.          0.          0.          0.
   0.          0.        ]
 [ 0.          0.33333333  0.          0.          0.5         0.
   0.16666667  0.        ]
 [ 0.          0.33333333  0.          0.         -0.5         0.
   0.16666667  0.        ]
 [ 0.          0.          0.33333333  0.          0.          0.
   0.          0.66666667]
 [ 0.          0.33333333  0.          0.          0.          0.
  -0.66666667  0.        ]
 [ 0.          0.          0.33333333  0.          0.          0.5
   0.         -0.16666667]
 [ 0.          0.          0.33333333  0.          0.         -0.5
   0.         -0.16666667]
 [ 0.          0.          0.          1.          0.          0.
   0.          0.        ]]
```

Figure 2. CGC table for $\frac{1}{2} \times \frac{1}{2} \times \frac{1}{2}$

```
55 coef = C_G_coefficients([1.5,0.5]).real
56 print(coef)
```

```
[[ 1.    0.    0.    0.    0.    0.    0.    0.  ]
 [ 0.    0.25  0.    0.    0.    0.75  0.    0.  ]
 [ 0.    0.75  0.    0.    0.   -0.25  0.    0.  ]
 [ 0.    0.    0.5   0.    0.    0.    0.5   0.  ]
 [ 0.    0.    0.5   0.    0.    0.   -0.5   0.  ]
 [ 0.    0.    0.    0.75  0.    0.    0.    0.25]
 [ 0.    0.    0.    0.25  0.    0.    0.   -0.75]
 [ 0.    0.    0.    0.    1.    0.    0.    0.  ]]
```

Figure 3. CGC table for $\frac{3}{2} \times \frac{1}{2}$

FURTHER IMPROVEMENTS

It is evident that the program is not very efficient for large sets of spins. To mitigate the problem, one possibility could have been to use the pandas package to drastically reduce the time-weight on the for and while loops, which can be very inefficient. As of now, we do not have the in-depth knowledge of pandas package to execute this process but it is possible to include it in the future.

Furthermore, we think that the spherical harmonics can be easily added to the program in the same way but would involve solving eigen-value problem and integrals which can be easily solved with the methods of quadrature. Thus, in that way we could produce the combined total angular momentum decomposition for hydrogen atoms.

FINAL NOTES

The work was split evenly between the two authors. We debated and discussed methods equally to produce this project. Therefore, the work split is: 50-50.

REFERENCES

Griffiths, D. J. and Schroeter, D. F. (2018). *Introduction to quantum mechanics*. Cambridge University Press.

Sakurai, J. J. and Napolitano, J. (2020). *Modern quantum mechanics, Second edition*.

APPENDIX 1: USER INTERFACE

This function is initially called to produce the set of n particles and their respective spins from the user.

```
 # Initial function, starting user interface:
def ask_for_spins():
  print("Please input the state:")
  while(True):
    particles = input("What is the number of particles? \n")
    if particles.isnumeric():
      if int(particles) > 1:
        particles = int(particles)
        break
    print("Please input a whole number larger than 1")
  spins = []
  for i in range(particles):
      print("Particle number",i+1)
      spin = (input("Input the spin of the particle: "))
      spin = float(spin)
      spins.append(spin)
  return spins

spins = ask_for_spins()
```

The following is the main code that runs in the very end, after all other functions have been defined. This contains command line functionality for increased usability of the program.

```
# Start of the main function

start = time.time()   # measuring the calculation time
coef = C_G_coefficients(spins)
end = time.time()
rcoef = coef.real     # removes the imaginary component 0 for cleaner display

print("spins:",spins)
print("C_G matrix:")
print(rcoef)

print("Calculation time:",end - start,"seconds")

def listCommands():
  print("List of available Commands:")
  print("help               : Display the list of available commands")
  print("matrix             : Display the Clebsch-Gordan coefficients matrix")
  print("spins              : Display the spins for each particle")
  print("restart            : Restart the Program with new particles  ")
  print("exit               : Exit the Program")

listCommands()

while(True):
  print("Please input your command")
  user_input = input(">")
```

```
  if user_input == "exit" or user_input == "quit":
    break
  elif user_input == "matrix":
    print("C_G matrix:")
    print(rcoef)
  elif user_input == "spins":
    print("spins: ",spins)
  elif user_input == "help":
    listCommands()
  elif user_input == "restart":
    spins = ask_for_spins()
    start = time.time()
    coef = C_G_coefficients(spins)
    end = time.time()
    rcoef = coef.real
    print("spins:",spins)
    print("C_G matrix:")
    print(rcoef)
    print("Calculation time:",end - start,"seconds")
```

APPENDIX 2: ELEMENTWISE ALGORITHM FOR KRONECKER PRODUCT

This was eventually replaced by the function numpy.kron()

```
# Kronecker product
def kr_product(A, B, dtype = complex):
  a = len(A)
  if (isinstance(A[0], float) or isinstance(A[0],int)):
    b  = 1
  else:
    b = len(A[0])
  c = len(B)
  if isinstance(B[0], float) or isinstance(B[0],int):
    d = 1
  else:
    d = len(B[0])

  C = np.zeros((a*c, b*d),dtype=dtype)

  for j in range(d):
    for i in range(b):
      for k in range(a*c):
          C[k, d*i+j] = A[k//c,i] * B[k%c, j]

  return C
```