EFFICIENT AUTOMATED GENERATION OF LOCAL DESCRIPTORS FOR USE IN *AB INITIO* ELECTRONIC STRUCTURE SIMULATIONS

A THESIS IN
Physics

Presented to the Faculty of the University
of Missouri–Kansas City in partial fulfillment of
the requirements for the degree

MASTER OF SCIENCE

by
DUONG THUY HOANG

B. S., University of Missouri - Kansas City, USA, 2020

Kansas City, Missouri
2023

EFFICIENT AUTOMATED GENERATION OF LOCAL DESCRIPTORS FOR USE IN *AB INITIO* ELECTRONIC STRUCTURE SIMULATIONS

Duong Thuy Hoang, Candidate for the Master of Science Degree

University of Missouri–Kansas City, 2023

ABSTRACT

First-principles electronic structure calculations based on density functional theory (DFT) are well-known to have a high computational cost that scales algorithmically as $O(N^3)$, where N is the number of electrons. Reducing that cost is a key goal of the computational materials physics community and machine learning (ML) is viewed as a potential tool for that task. However, ML model training requires an appropriate match between the input descriptors and the target property as well as copious quantities of training data. This thesis presents a computer program that is designed to automate the generation of local atomic environment descriptors for single element systems that may be used for training neural networks to predict the set of electronic potential function coefficients, $\{A_i\}$, that are used within the DFT based orthogonalized linear combination of atomic orbitals (OLCAO) method. Predicting the potential function coefficients rather than explicitly

computing them will dramatically reduce the computational cost. We explore additional research directions by connecting the gap between descriptors used in computer vision and those employed in electronic structure predictions. This approach opens up possibilities for cross-disciplinary knowledge and techniques from computer vision, which may further improve the accuracy and efficiency of electronic structure calculations.

APPROVAL PAGE

The faculty listed below, appointed by the Dean of the School of Science and Engineering, have examined a thesis titled "Efficient Automated Generation of Local Descriptors for use in *Ab Initio* Electronic Structure Simulations" presented by Duong Thuy Hoang, candidate for the Master of Science degree, and certify that in their opinion it is worthy of acceptance.

Supervisory Committee

Paul Rulis, Ph.D., Committee Chair
Physics and Astronomy

Zhu Li, Ph.D., Committee Member
Computer Science; Electrical and Computer Engnieering

Elizabeth Stoddard, Ph.D., Committee Member
Physics and Astronomy

Anthony Caruso, Ph.D., Committee Member
Physics and Astronomy

Contents

ILLUSTRATIONS

TABLES

ACKNOWLEDGEMENTS

To my supervisor, Dr. Rulis, thank you for your invaluable support and guidance throughout my research journey. Your expertise, advice, and mentorship have played a pivotal role in shaping the outcome of my thesis.

To my peers at UMKC, thank you for the feedback and discussions during the Computational Physics Group meetings. You have significantly enhanced my understanding and enriched this entire process.

To my beloved parents, Van Nguyen and Hai Hoang, thank you for your unwavering support and the immense sacrifices you have made. Your integrity and resilience in the face of challenges, as well as the value you place in education and community, have been a constant source of strength and motivation in my life. I am proud to be your daughter and proud of our ancestors' history and our culture.

To my dearest Courtney Osiris Hines, I am grateful to have you in my life. Your love fills my heart with joy and gives me the strength and courage to persevere.

To each and every person I have encountered, thank you for your invaluable contributions to my journey. Your support does not go unnoticed.
Duong Thuy Hoang

CHAPTER 1

INTRODUCTION

1.1 Computational Materials Modeling

The field of materials physics strives to discover fundamental scientific principles and apply them to the creation, refinement, and advancement of cutting-edge structural and functional materials with diverse applications. For instance, the advancement of innovative electrochemical environments for more compact and durable batteries, the design of materials for superconducting qubits [2], and data storage through a magnetic read-and-write head, altering the magnetization of minute regions on a crystalline slice of lithium tantalate [3]. Many of these advancements are shaped by theoretical and methodological developments in materials physics and computational studies of electronic structures.

For decades, the material science field has used a combination of theory, experiment, characterization, computational modeling, and simulation. The interplay between those activities was greatly accelerated with the start of the Materials Genome Initiative for Global Competitiveness in 2011 [4]. The 1998 Nobel Prize was awarded to Walter Kohn for his development of Density Functional Theory (DFT) [5], and to John Pople for his contribu-

tions to computational methods in quantum chemistry [6]. This recognition came 40 years after Erwin Schrödinger published his seminal paper, which marked the inception of wave mechanics [7]. In recent years, methods based on DFT have seen significant advancements. DFT approaches has been progressively supplemented with machine learning (ML) based methods that are typically faster while also maintaining equivalent accuracy. The Roadmap article [8] brings together insights from experts in the field to explore the use of ML in materials science. It sheds light on current and future challenges across a wide range of problems, such as predicting materials properties, constructing force-fields, developing exchange correlation functionals for DFT, solving the many-body problem, and more. Despite the numerous and impressive research that have already taken place, example include the use of active learning techniques in ML to investigate transition-metal complexes and uncover method-insensitive, synthetically accessible chromophores [9], and other studies employing machine-learning-based approaches to assess potential energies for quantum mechanical investigations of ground and excited vibrational states in small molecules [10]. We find ourselves only at the initial stages of a prolonged journey to incorporate ML in material science research.

It's important to recognize that in our pursuit of a sustainable and inclusive future, research must extend beyond the boundaries of material innova-

tion. It must embrace a comprehensive understanding of the environmental, and social implications posed by waste materials on our land and soil, while also actively engaging with the pressing challenges of climate change issues.

1.2 Machine Learning for Electronic Structure Problems

1.2.1 Machine Learning and Its Infancy

The achievements in research that incorporate Machine Learning (ML) into materials science are valuable; however, the application of ML frequently comes with excessive hype and pledges of predicting extraordinary results. It is therefore crucial to acknowledge that many current ML models are trained on existing data points that do not adequately sample the problem space and may therefore not fully align with our intuitive understanding of general laws and physics concepts in trying to solve electronic structure problems with ML. Consequently, it becomes essential to emphasize the significance of the input descriptors, or features, used to train the models and to investigate the reasoning behind the training process to eliminate biases.

ML models heavily depend on the data they are trained on, which implies that their predictions are founded on patterns and relationships observed in the training data. This dependence on existing data presents challenges when integrating new data inputs, potentially rendering the ML model inefficient and unreliable unless a comprehensive understanding of the underlying phys-

ical laws and mathematical principles is achieved.

1.2.2 Intuitive Understanding of Neural Networks

Neural networks (NNs) are machine learning techniques designed to recognize patterns and relationships in data. A NN model is trained on a large dataset, and during training it undergoes iterative parameter updates, refining its ability to make predictions. Supervised learning and unsupervised learning are the two main learning paradigms for neural networks. Supervised learning involves training the neural network on labeled data, where each input has a corresponding target or label (known correct output). The network adjusts its parameters during training to minimize the prediction error by comparing its outputs with the known targets. This approach is commonly used for classification, where the network predicts discrete class labels, and regression, where it predicts continuous values. On the other hand, unsupervised learning trains the network on unlabeled data, aiming to identify patterns and relationships within the data without explicit guidance. Unsupervised learning is often used for clustering, grouping similar data points based on their characteristics, and dimensionality reduction, reducing the number of data features while retaining essential information. There is also a hybrid approach called semi-supervised learning, which combines aspects of both supervised and unsupervised learning, using both labeled and unlabeled

data to enhance performance.

Selecting the appropriate strategy for the study objective relies on understanding how we can access and interpret the organization and extent of our input and output data.

CHAPTER 2

BACKGROUND

2.1 The OLCAO Method

The orthogonalized linear combinations of atomic orbital (OLCAO) method [11] is an efficient first-principles method for calculating the electronic structure of low-symmetry systems, for example, amorphous silicon (a-Si). In this method, all the electrons in a system are taken into account when calculating the interaction between atoms. However, the valence orbitals, which are the outermost electrons involved in chemical bonding, are modified to be orthogonal (mathematically independent) from the core orbitals (electrons closer to the atomic nucleus). By orthogonalizing the valence orbitals against the core, the core orbitals can be effectively eliminated from the eigenvalue problem thereby reducing the computational cost with a negligible impact on the computed results.

Studies have demonstrated that the OLCAO method produces eigenvalues and eigenvectors (quantities that describe the energy states and associated wave functions) with accuracy that is competitively comparable to those obtained using a plane wave basis set [12, 13]. This means that by employing the OLCAO method, we can obtain reliable results while significantly

reducing the computational complexity when dealing with complex material systems containing a large number of atoms and different types of elements.

In OLCAO, the solid state wave function $\psi_{n\mathbf{k}}(\mathbf{r})$ of band index n or energy in the $\mathbf{k} = 0$ and wave vector $\mathbf{k}$ is expanded in terms of Block sums $b_{i\gamma}(\mathbf{k}, \mathbf{r})$

$$\psi_{n\mathbf{k}}(\mathbf{r}) = \sum_{i,\gamma} C^n_{i\gamma}(\mathbf{k}) b_{i\gamma}(\mathbf{k}, \mathbf{r}) \tag{2.1}$$

where γ labels atoms in the cell and γ represents differents type of atoms as well as non-equivalent atoms of the same type within the unit cell. i represents all collective quantum number of the atom (i.e. principal quantum number n and the angular quantum numbers (l, m)).

The Bloch sum in Eq. 2.1 is expressed as the linear combination of atomic orbitals u_j centered at each site

$$b_{i\gamma}(\mathbf{k}, \mathbf{r}) = \left(\frac{1}{\sqrt{N}}\right) \sum_{\nu} e^{i(\mathbf{k}.\mathbf{R}_\nu)} u_i(\mathbf{r} - \mathbf{R}_\nu - \mathbf{t}_\gamma) \tag{2.2}$$

$\mathbf{R}_\nu$ is the lattice vector and $\mathbf{t}_\nu$ is the position of the γ^{th} atom in the cell.

The atomic orbital $u_i(\mathbf{r})$ can be expanded using a basis set of Gaussian-type orbitals (GTOs)

$$u_i(\mathbf{r}) = \left[\sum_{i=1}^{N} A_i r^l e^{(-\alpha_i r^2)}\right] . Y^m_l(\theta, \phi) \tag{2.3}$$

where $Y^m_l(\theta, \phi)$ is real spherical harmonics and N is the total number of

coefficients in set $\{\alpha_i\}$.

The real space charge density of a given crystal then can be written as a sum of atom-centered Gaussian functions [11]

$$\rho_{cry}(\mathbf{r}) = \sum_B \rho_B(\mathbf{r} - \mathbf{t}_B), \quad \rho_B(\mathbf{r}) = \sum_{i=1}^{N} D_i e^{-\alpha_i r^2} \tag{2.4}$$

The total electronic potential function of a crystal, $V_{cry}(\mathbf{r})$, can also be expressed as the sum of atom-centered spherical Gaussian functions with different coefficients.

$$V_{cry}(\mathbf{r}) = \sum_B V_B(\mathbf{r} - \mathbf{t}_B), \quad V_B(\mathbf{r}) = \sum_{i=1}^{N_A} A_i e^{-\alpha_i r^2} \tag{2.5}$$

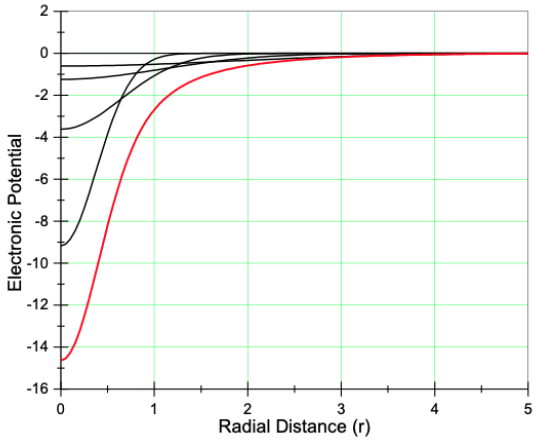
where V_B is the symbolic interpretation of the total potential for each site, and it represents the summation over all the local neighborhood potentials for each central atom i^{th} in that site.

In Figure 1, (a) demonstrate the numerical values of an example set $\{A_i\}$ and the fixed exponential coefficients $\{\alpha_i\}$ for the silicon (Si) potential function. The first four potential functions are plotted - black lines and summed together - red line (b). All Gaussian functions for one Si atomic site are summed together (c).

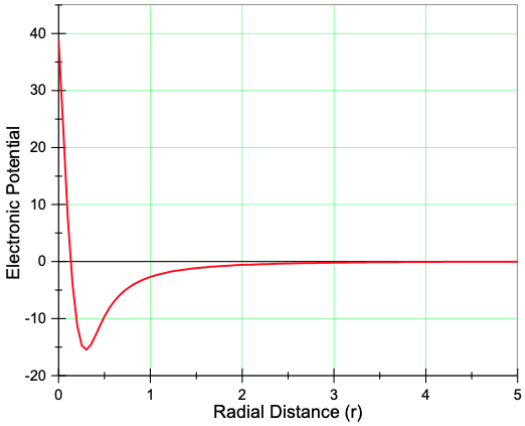
Each exponent coefficient set $\{\alpha_i\}$ is pre-determined for each atom whereas the coeficients set of $\{A_i\}$ are updated in each cycle of the self-consistent field

ID	A_i	α_i
1	$-0.60672619 \times 10^{0}$	0.15000000×10^{0}
2	$-0.12407671 \times 10^{1}$	0.42758111×10^{0}
3	$-0.36166706 \times 10^{1}$	0.12188374×10^{1}
4	$-0.91724811 \times 10^{1}$	0.34743455×10^{1}
5	$-0.16959754 \times 10^{2}$	0.99037634×10^{1}
6	$-0.40745139 \times 10^{2}$	0.28231081×10^{2}
7	0.12733590×10^{2}	0.80473846×10^{2}
8	0.10442423×10^{2}	0.22939397×10^{3}
9	$-0.19109966 \times 10^{-1}$	0.65389686×10^{3}
10	0.54010939×10^{1}	0.18639596×10^{4}
11	$-0.42032006 \times 10^{1}$	0.53132928×10^{4}
12	0.66156017×10^{1}	0.15145758×10^{5}
13	$-0.74064920 \times 10^{1}$	0.43173599×10^{5}
14	0.91870114×10^{1}	0.12306810×10^{6}
15	$-0.92689314 \times 10^{1}$	0.35081063×10^{6}
16	$-0.63220401 \times 10^{1}$	0.10000000×10^{7}

(a) Numerical values of an example set A_i, and fixed set $\{\alpha_i\}$



(b) The first four Gaussian functions



(c) All Gaussian functions for one Si atomic site

Figure 1: Electronic Potential Calculation

(SCF) iterations in accordance with the charge density $\rho(\mathbf{r})$ that was computed in the previous SCF step (see Figure 2). However, if the choice of the fitting set $\{A_i\}$ can be accurately predicted for a given system, then the SCF process can be skipped, satisfying an important requirement of our goal to reduce the computational cost. Optimizing method mentioned in Chapter 2.3

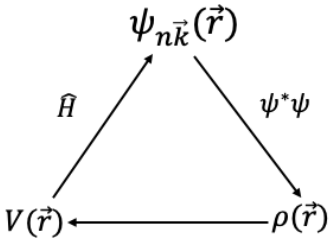


Figure 2: Illustration of SFC process

2.2 Local Atomic Environment

Density is commonly understood as mass per unit volume, but for a local atomic environment that description of compactness or concentration of matters in a given region is a bit misleading. Instead, the total local atomic density of k neighbor atoms within a cutoff R_{cut} radius of a central atom i located at r is (reconstructed based on references [14–16]) expressed as

$$\rho(\mathbf{r}) = \omega_i^{self}\delta(\mathbf{r}) + \sum_{r_{ik}<R_{cut}} f_c(r_{ik})\,\omega_k\delta(\mathbf{r}-\mathbf{r_{ik}}) \tag{2.6}$$

where r_{ik} is the relative distance of neighbor atom k to center atom i, and $r, r_{ik} \in \mathbb{R}^3$. ω_i^{self} is a weight coefficient assigned to the self-contribution of the central atom, while ω_k is a unique weight factor assigned according to the atomic species of atom k, used to differentiate the contribution of different types of neighbor atoms. $f_c(r_{ik}; R_{cut})$ is the cutoff function that ensures a smooth decay for the contribution of neighbor atoms, reaching zero at the cutoff radius R_{cut}. The sum is over all neighbor atoms k within cutoff distance R_{cut}. The dirac delta functions $\delta(r_{ik})$ are used to define the positions of the k neighbor atoms.

Physically, the local atomic density remains invariant under the rotation of atoms in the neighborhood, but mathematically the change in coordinates will produce a different local atomic density function $\rho(\mathbf{r})$, making the function unsuitable for ML input training data. To achieve invariance a more comprehensive approach is required. One intriguing approach is the use of the three-point correlation, bispectrum. This involves mapping the atomic density onto the surface of a four-dimensional unit sphere and expanding the atomic density function using hyper-spherical harmonics $U^j_{mm'}(\phi, \theta, \theta_0)$ as an orthonormal basis set. The neighbor density function is expressed as follows

$$\rho(\mathbf{r}) = \sum_{j=0}^{\infty} \sum_{m,m'=-j}^{j} u^j_{mm'}.U^j_{mm'}(\theta_0, \theta, \phi) \tag{2.7}$$

The expansion coefficients $u^j_{mm'}$ are complex valued, do not remain invariant under rotation and are obtained from,

$$u^j_{mm'} = \left\langle U^j_{mm'} | \rho \right\rangle \tag{2.8}$$

where $U^j_{mm'}$ functions are complex-valued, and are the elements of the unitary transformation matrices corresponding to the rotation of spherical harmonics by an angle θ_0 around the axis defined by $\mathbf{n}(\theta, \phi)$. The symbol "." in Eq. 2.7 indicates the scalar product of the two functions. In the reference Ref. [15], it's crucial to emphasize our objective of using real-valued local descriptors. This involves employing scalar triple products of expansion coefficients that maintain their real-valued nature and rotation invariance, as discussed in Ref. [14].

$$B_(j_1, j_2, j) = \sum_{m_1, m'_1 = -j_1}^{j_1} \sum_{m_2, m'_2 = -j_2}^{j_2} \sum_{m, m' = -j}^{j} (u^j_{mm'})^* \, H^{jmm'}_{j_1 m_1 m'_1, j_2 m_2 m'_2} u^{j_1}_{m_1 m'_1} u^{j_2}_{m_2 m'_2} \tag{2.9}$$

where $H^{jmm'}_{j_1 m_1 m'_1, j_2 m_2 m'_2} \equiv C^{jm}_{j_1 m_1 j_2 m_2} C^{jm'}_{j_1 m'_1 j_2 m'_2}$ is the coupling coefficient for four-dimensional spherical harmonics, analogous to the Clebsch-Gordan coefficients for rotations in three dimensional space.

The derivation of the related functions and variables mentioned in this chapter will be discussed in Chapter 3.

2.3 Local Descriptors

The objective is to generate bispectrum components (as will be discussed in Chapter 3) and use them as input descriptors for the proposed neural network (see Chapter 5.2).

These components have been successfully implemented in various packages and libraries such as the Material Learning Algorithm (MALA) [17] and PyXtall_FF [18], are used for predicting the electronic structure of matter and automating force field generation. Therefore, it makes sense to utilize bispectrum components to optimize calculation of the electronic potential within the framework of the OLCAO method.

We aim to predict the choice of the set of electronic potential coefficients A_i and thereby approximate the potential V_B at each atomic site to find the total crystal potential $V_{cry}(\mathbf{r})$. To achieve this, we must assess whether the input data comes with labels or is unlabeled. If it is labeled, we need to investigate what properties of the input features are associated with these labels. On the other hand, if the data is not labeled, we need to consider whether we should annotate it ourselves and evaluate the efficiency of this process.

To illustrate the concept of labeled input data in supervised learning (see Figure 4), consider the case of an amorphous silicon unit cell with N atoms, Figure 3. Each atom in the unit cell is described by a set of d input

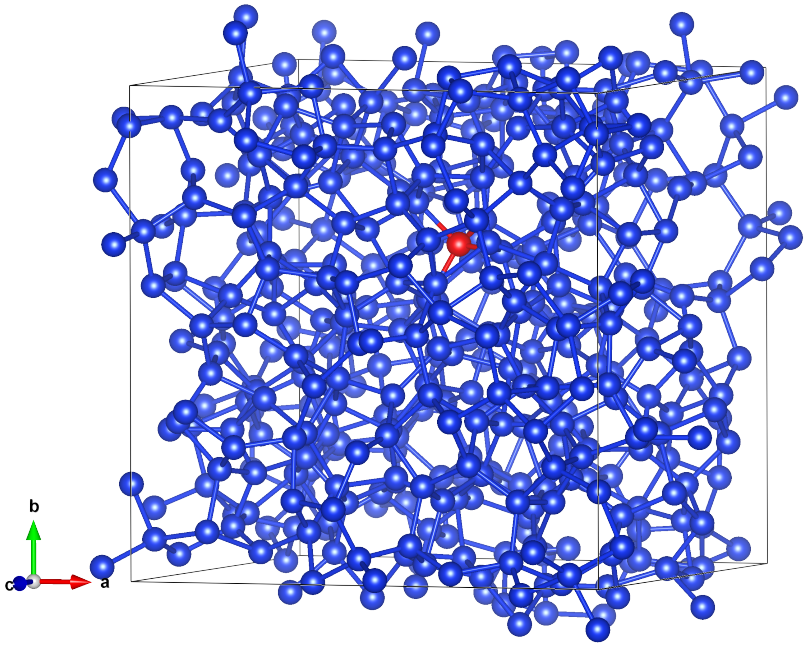


Figure 3: Amorphous silicon unit cell with $N = 340$ atoms

features (independent variables), where each member d of the set x may itself represent one or more different physical features. The set of inputs is a vector $[x_1, x_2, \ldots, x_d]$, and its corresponding output (dependent variable which may also be a set itself) is denoted as y_i. By combining all the feature vectors of each atom, we create the feature matrix A representing all atoms in the cell, with dimensions $[N \times d]$. Each column in matrix A is one specific feature (or set of features) associated with each atomic site and with its dependent y, which could represent the target feature being examined in this thesis (i.e., the OLCAO potential function coefficients set $\{A_i\}$), or label the element in

some way (e.g., amorphous silicon versus crystalline silicon). The variable y for the entire N atoms has dimensions $[1 \times N]$ in Figure 4 if we use target labels.

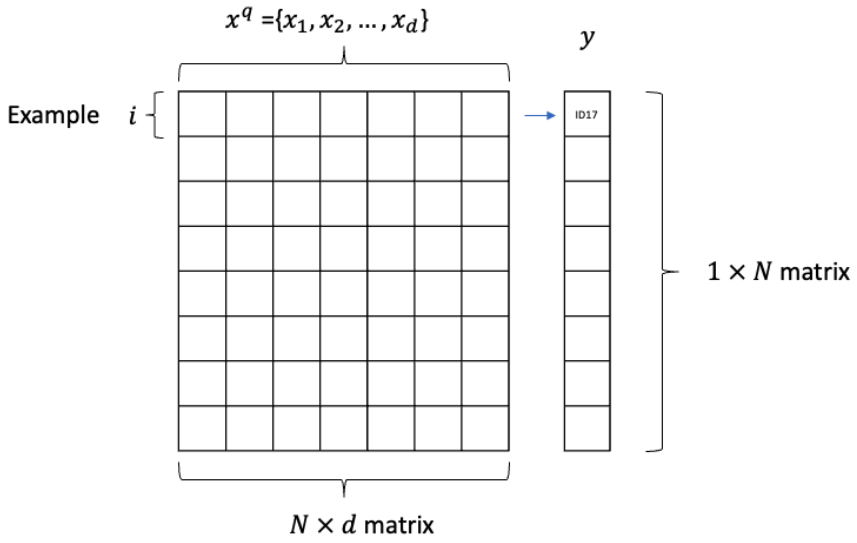


Figure 4: Illustration of input feature vectors

If we focus on one atom (say $i = 17$) colored in red within the amorphous silicon model (see Figure 3). We can represent the input feature vector for this target atom, along with its corresponding label, as $\{x^i_{d_1}, y_i\}$ for the case when training pattern feature $d_1 \equiv \{B\}$ then the total feature set for the entire system with N atoms will be $\sum_{i=1}^{N} \{x^i_B, y_i\}$. In this example, B refers to the set of bispectrum components, which characterizes the structural environment of a central atom and its neighbors. Other patterns, such as bond types $d_2 \equiv \{d_{bond}\}$, chemical elements $d_3 \equiv \{d_{chem}\}$, and so on d_n, can

also be considered.

The key idea is to transform the input data into vector form in order and identify the relationships or patterns between different physical features that differentiate atoms and their neighborhoods within a system, or even between different systems, such as system A (amorphous silicon) and system B (crystalline silicon). This process extracts information from the input features vector and analyzes the eigenvectors and eigenvalues of the studied system (target features).

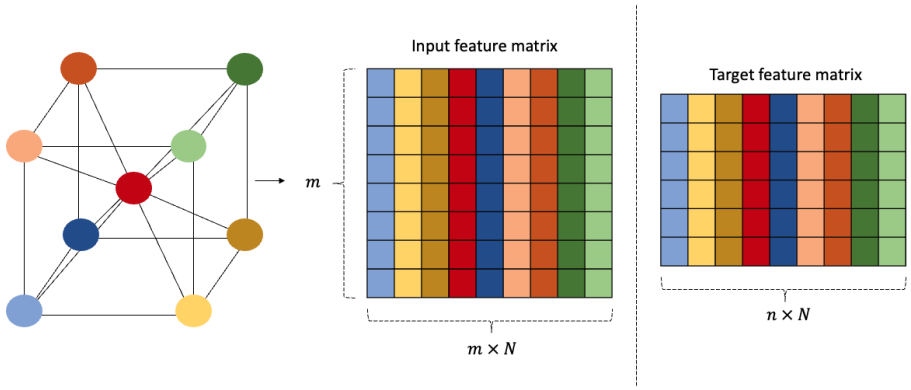


Figure 5: Illustration of bispectrum input feature matrix and target feature matrix A_i

In the case we work with a target feature matrix $y = [n \times N]$, which comprises N sample atoms, and each atom has m different bispectrum components. The values of y are obtained by summing the output sets A_i, acquired through the SCF (Self-Consistent Field) process in OLCAO. The conceptual idea of extracting an input feature matrix for the bispectrum components of

these atoms is illustrated in Figure 5. The input feature matrix has dimensions $[m \times N]$ and consists of feature vectors representing the bispectrum components of each atom.

To understand the relationship between the bispectrum components and the coefficients A_i, we examine the target feature matrix y, which has dimensions $[n \times N]$. Here, n represents the set of output coefficients A_i for each atom from the SFC process in OLCAO. It is essential to investigate this relationship between bispectrum components and potential coefficient A_i. If $n \neq m$, indicating a difference in dimensions between the target feature matrix and the input feature matrix. In such cases, we need to perform dimensionality reduction.

Dimensionality reduction involves a mathematical transformation of data from a high-dimensional space into a lower-dimensional space. The goal is to retain meaningful properties of the original data in the reduced representation, ideally close to its intrinsic dimension. By performing dimensionality reduction, we can effectively analyze and compare the relationship between the bispectrum components and the coefficients A_i despite the differences in dimensionality.

For future studies, we plan to incorporate additional datasets of pure Si models, including amorphous Si, crystalline Si, Si with a passive defect, and Si with a self-interstitial. Additionally, we will implement other models to

compare their efficiency. Each model will use input/target output training pairs consisting of local environment descriptors - bispectrum components (input) - that encode the structure of neighboring atoms relative to the central atom i at a specific point in real space. The target output will be the converged potential functions obtained through the SCF process. To assess and optimize the model's performance during training, we will divide the dataset into training, test, and validation sets. These sets will be used in subsequent iterations of training and validation.

In conclusion, the generation of precise local atomic environment descriptor sets holds significant importance. By integrating bispectrum components, unique values are generated. These values encode the structural characteristics of neighboring atoms in relation to a central atom i at a specific location in real space. This approach ensures the establishment of an accurate atomic structure.

CHAPTER 3

COMPUTATIONAL DETAILS

This chapter specifically presents a computer program that is designed to automate the generation of bispectrum components 2.9

$$B_(j_1, j_2, j) = \sum_{m_1, m_1'=-j_1}^{j_1} \sum_{m_2, m_2'=-j_2}^{j_2} \sum_{m, m'=-j}^{j} (u^j_{mm'})^* \, H^{jmm'}_{j_1 m_1 m_1', j_2 m_2 m_2'} \, u^{j_1}_{m_1 m_1'} \, u^{j_2}_{m_2 m_2'}$$

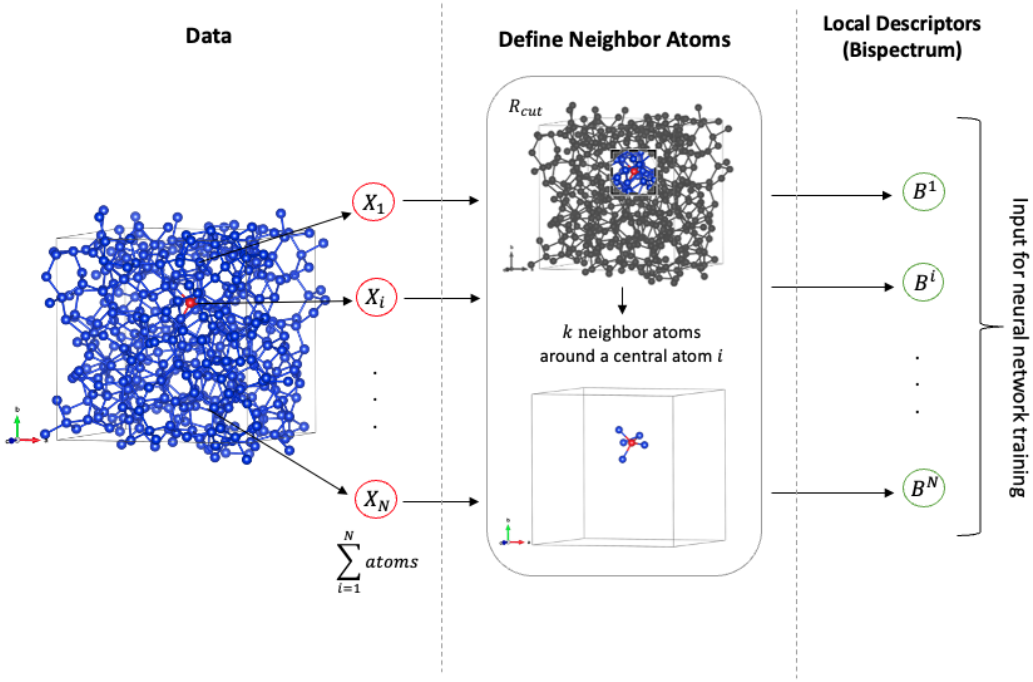


Figure 6: Framework for generating bispectrum coefficients

Figure 6 illustrates the simplified version calculation process for the bis-

pectrum. This process is iterated until it encompasses all atoms, denoted by $\sum_{i=1}^{N}$. The process proceeds from left to right as follows:

Step 1: Extract atomic coordinates $\sum_{i=1}^{N}\{x_i, y_i, z_i\}$ from the input file (e.g. CIF, JSON) for the atoms within the unit cell.

Step 2: Define neighboring atoms around a central atom i and specify angular conversions. The input data comprises a subset $\{X_1\}, ...\{X_N\}$ containing atom structures along with their coordinates, positions in the cell, and other necessary parameters.

Step 3: Compute all key functions in the bispectrum equation and generate a bispectrum component set for each atom.

3.1 Defining Neighbor Atoms

3.1.1 Cartesian Coordinate System Transformation

The relative position of a neighboring atom k with respect to a central atom i can be represented as a point within a three-dimensional sphere of radius r_3. To establish a consistent frame of reference, we conceptually designate the central atom i and position its coordinates (x_i, y_i, z_i) at the center of a singular cell, or away from the edges when a cutoff radius is applied. This approach helps mitigate complexities in picturing the process that may arise from periodic boundary conditions. In the program code, of course, all

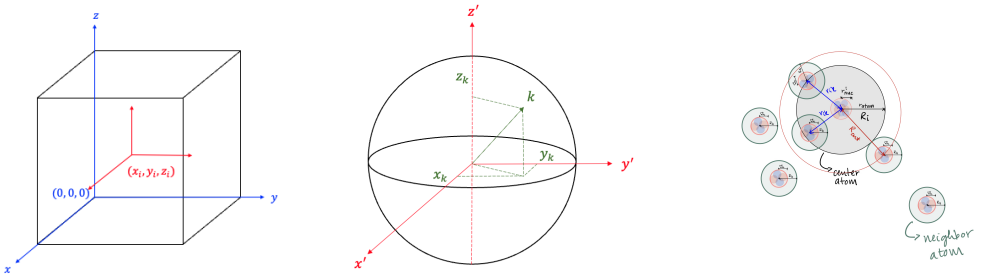
atomic coordinates are simply computed relative to the coordinates of the current target atom.

The original cell has an origin point $O(0, 0, 0)$ and possibly non-orthogonal axes (a, b, c). Each atom is designated with fractional coordinate values that specify the position of the atom along the crystallographic axes. These values range from 0 to 1 and indicate the position of the atom relative to the unit cell. For simplicity, when considering an orthorhombic cell the axes are often referred to using (x, y, z) Cartesian labels.

To calculate the new coordinates of each atom in relation to the central atom i, the coordinates of neighbor atoms k with respect to the new system are calculated.

$$\begin{bmatrix} x_k \\ y_k \\ z_k \end{bmatrix} = \begin{bmatrix} x - x_i \\ y - y_i \\ z - z_i \end{bmatrix} \tag{3.1}$$

Our objective is to perform a transformation of neighbor atoms, along with their associated spatial coordinates (x_k, y_k, z_k), into spherical coordinates (r, θ, ϕ) within a three-dimensional ball. These transformed coordinates will be projected onto a set of points (θ_0, θ, ϕ) that form the surface of the 3-sphere.



(a) (b) (c)

Figure 7: Schematic of coordinate transformation for neighboring atoms.

(a) Coordinate transformation: shifting origin O to the location of center atom i. (b) Defining the position of a neighbor atom k relative to a central atom i as a point within a sphere

. (c) Visualize cut off region around central atom i.

3.1.2 Unit Sphere vs. Unit Ball

We define the concepts of a unit sphere and a unit ball. We refer to the origin of space as O, and the axes in p dimensional space as $\{x_1, x_2, x_3, \cdots, x_p\}$.

A unit sphere can be defined as the set of points in a space that are one unit distance away from the origin

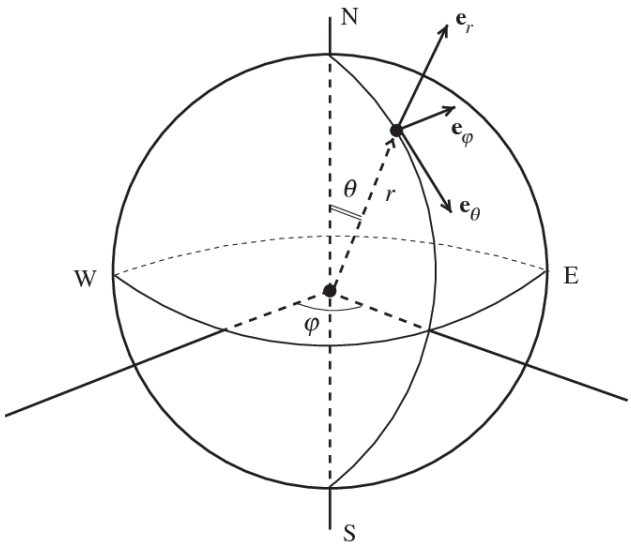


Figure 8: The 2-sphere coordinate system in $\mathbb{R}^3$ [1]

where $\theta \in [0, \pi]$ represents the polar angle, $\phi \in [0, 2\pi]$ represents the azimuthal angle, and r represents the distance from the origin. The North Pole corresponds to $\theta = 0$, and the South Pole corresponds to $\theta = \pi$.

$$\begin{aligned}
\mathbb{S}^0 &:= \{x_1 \in \mathbb{R}^1 \mid x_1^2 = 1\} \\
\mathbb{S}^1 &:= \{x_1, x_2 \in \mathbb{R}^2 \mid x_1^2 + x_2^2 = 1\} \\
\mathbb{S}^2 &:= \{x_1, x_2, x_3 \in \mathbb{R}^3 \mid x_1^2 + x_2^2 + x_3^2 = 1\} \\
\mathbb{S}^3 &:= \{x_1, x_2, x_3, x_4 \in \mathbb{R}^4 \mid x_1^2 + x_2^2 + x_3^2 + x_4^2 = 1\} \\
&\vdots \\
\mathbb{S}^{p-1} &:= \{x_1, x_2, \cdots, x_p \in \mathbb{R}^p \mid x_1^2 + x_2^2 + \cdots + x_p^2 = 1\}
\end{aligned} \tag{3.2}$$

For example, a two dimensional sphere $\mathbb{S}^2$ exists in three-dimensional Euclidean space $\mathbb{R}^3$. A three-dimensional sphere is a hyper-surface, denoted

as $\mathbb{S}^3$, embedded in four-dimensional space, $\mathbb{R}^4$.

A unit ball is obtained by filling the unit sphere. The p-ball is defined as the set of points on or within the $(p-1)$-sphere. Thus, the p-ball contains both the surface and the interior of the $(p-1)$-sphere

$$\begin{aligned}
\mathbb{B}^1 &:= \{x_1 \in \mathbb{R}^1 \mid x_1^2 \leq 1\} \\
\mathbb{B}^2 &:= \{x_1, x_2 \in \mathbb{R}^2 \mid x_1^2 + x_2^2 \leq 1\} \\
\mathbb{B}^3 &:= \{x_1, x_2, x_3 \in \mathbb{R}^3 \mid x_1^2 + x_2^2 + x_3^2 \leq 1\} \\
&\vdots \\
\mathbb{B}^p &:= \{x_1, x_2, \cdots, x_p \in \mathbb{R}^p \mid x_1^2 + x_2^2 + \cdots + x_p^2 \leq 1\}
\end{aligned} \tag{3.3}$$

3.1.3 Spherical Harmonics in Four-Dimensions

To achieve rotational invariance a more comprehensive approach is necessary. One intriguing avenue involves the use of the Fourier transform of the three point correlation function, the bispectrum. We map the atomic density function onto the surface of a four-dimensional unit sphere. Then we employ expansion coefficients to express the atomic density in terms of four-dimensional spherical harmonics

$$u^j_{mm'} = \left\langle U^j_{mm'} | \rho \right\rangle \tag{3.4}$$

To construct the four-dimensional spherical coordinate for $\mathbb{R}^4$ and express

them in terms of the corresponding Cartesian coordinates, we follow the general equations in Ref. [19] for spherical coordinate system in p dimensions

$$\begin{aligned} r = & \sqrt{x_1^2 + x_2^2 + ... + x_p^2} && \in [0, \infty] \\ \phi = & \tan^{-}1\left(\frac{x_2}{x_1}\right) && \in [0, 2\pi] \\ \theta_1 = & \tan^{1}\left(\frac{\sqrt{x_1^2 + x_2^2}}{x_3}\right) && \in [0, \pi] \\ & \vdots \\ \theta_{p-2} = & \tan^{-}1\left(\frac{\sqrt{x_1^2 + x_2^2 + ... + x_{p-1}^2}}{x_p}\right) && \in [0, \pi] \end{aligned} \tag{3.5}$$

where r represents the distance from the origin, ϕ denotes the azimuthal angle, and θ_1 to θ_{p-2} are polar angles, ranging from 0 to π.

To transition from $p-1$ dimensions to p dimensions, the x_p-axis is added perpendicular to the $p-1$ dimensional space. This process is analogous to transitioning from two dimensions to three dimensions by adding the $x_3 \equiv z$ axis perpendicular to the two-dimensional space (the (x, y) plane). The general expression for $\mathbb{R}^p$ and $\mathbb{R}^{p-1}$ are shown in Figure 9.

When projecting the position vector r_p in $\mathbb{R}^p$ onto $\mathbb{R}^{p-1}$, we obtain the

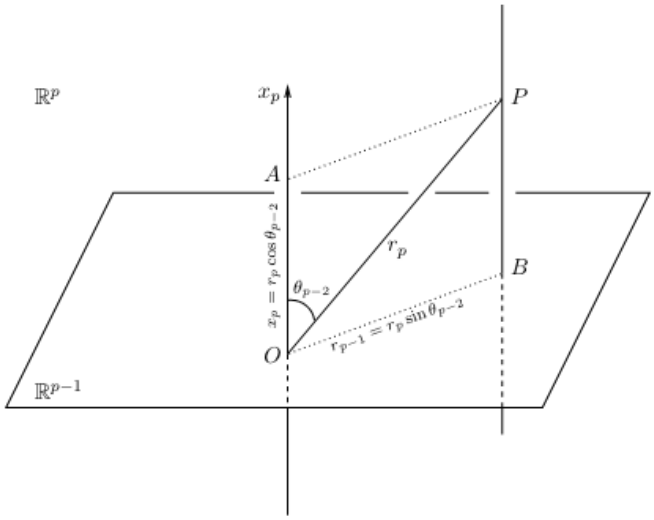


Figure 9: Visualization of $\mathbb{R}^{p-1}$ and $\mathbb{R}^p$

new angular coordinate θ_{p-2} is introduced to determine the location in new direction [19].

radius in the subspace $\mathbb{R}^{p-1}$, given by

$$r_{p-1} = r_p \sin \theta_{p-2} \tag{3.6}$$

We can derive a sequence of projections that transition from a four-

dimensional space to a three-dimensional space with established relationships

$$\begin{aligned}
x_1 &= r \sin\theta_{p-2} \sin\theta_{p-3} \cdots \sin\theta_3 \sin\theta_2 \sin\theta_1 \cos\phi \\
x_2 &= r \sin\theta_{p-2} \sin\theta_{p-3} \cdots \sin\theta_3 \sin\theta_2 \sin\theta_1 \sin\phi \\
x_3 &= r \sin\theta_{p-2} \sin\theta_{p-3} \cdots \sin\theta_3 \sin\theta_2 \cos\theta_1 \\
x_4 &= r \sin\theta_{p-2} \sin\theta_{p-3} \cdots \sin\theta_3 \cos\theta_2 \\
&\vdots \\
x_{p-1} &= r \sin\theta_{p-2} \cos\theta_{p-3} \\
x_p &= r \cos\theta_{p-2}
\end{aligned} \tag{3.7}$$

Imagine a single neighbor atom assigned at a point P with radial distance r_4 in $\mathbb{R}^4$ which thus has spherical coordinates $(r_4, \phi, \theta, \theta_0)$. The vector $\overrightarrow{OP}$ can be written as sum $\overrightarrow{OA} + \overrightarrow{OB}$, where $\overrightarrow{OA}$ lies along z_p and has magnitude $r_4 \; cos \; \theta_0$ while $\overrightarrow{OB}$ lies in the plane and has magnitude

$$r_3 \;=\; r_4 \sin\theta_0 \tag{3.8}$$

and the point B has spherical coordinate $(r_3, \phi, \theta))$

To extend from three dimensions to four dimensions, a fourth axis z_p is introduced, perpendicular to the three-dimensional space. We reconstruct the Cartesian coordinates of the set points (x_k, y_k, z_k) for k neighboring atoms

in $\mathbb{R}^3$, in terms of the radius $r_{p=4}$ in $\mathbb{R}^4$, as follows

$$\begin{aligned} x &= r_4 \sin\theta_0 \sin\theta \cos\phi = r_3 \sin\theta \cos\phi \\ y &= r_4 \sin\theta_0 \sin\theta \sin\phi = r_3 \sin\theta \sin\phi \\ z &= r_4 \sin\theta_0 \cos\theta = r_3 \cos\theta \\ z_p &= r_4 \cos\theta_0 \end{aligned} \tag{3.9}$$

We follow the Eqs. 3.5 to define the set of hyperspherical coordinates $(r_4, \phi, \theta, \theta_0)$ for k neighbor atoms

$$\begin{aligned} r_4 &= \sqrt{x^2 + y^2 + z^2 + z_p^2} && \in [0, \infty] \\ \phi &= \tan^{-1}\left(\frac{y}{x}\right) && \in [0, 2\pi] \\ \theta &= \tan^{-1}\left(\frac{\sqrt{x^2 + y^2}}{z}\right) && \in [0, \pi] \\ \theta_0 &= \tan^{-1}\left(\frac{\sqrt{x^2 + y^2 + z^2}}{z_p}\right) && \in [0, \pi] \end{aligned} \tag{3.10}$$

From Figure 9 θ_0 is the angel between the $x_{p=4} \equiv z_p$ axis and the vector r_4, combines with Eq. 3.10 θ_0 can be expressed as

$$\theta_0 = \cos^{-1}\left(\frac{z_p}{r_4}\right) \tag{3.11}$$

The angle θ_0 can be interpreted an additional angle that enables the mapping of a set of points (r_3, θ, ϕ) in $\mathbb{R}^2$ onto a set of points $(r_4, \theta, \theta_0, \phi)$ in

$\mathbb{R}^4$, with unit length for r_4. It is important to note that r_3 and r_4 represent the radial distances in three dimensions and four dimensions, respectively. When performing the mapping, it becomes necessary to define the fractional value and establish the relationship between these radial distances.

A useful analogy is the mapping of coordinates 2-ball (disk), $\mathbb{B}^2 := \{x_1, x_2 \in \mathbb{R}^2 \mid x_1^2 + x_2^2 \leq 1\}$ exist in two-dimensional Euclidean space $\mathbb{R}^2$ to the surface of a two-dimensional sphere $\mathbb{S}^2$ exists in three-dimensional Euclidean space $\mathbb{R}^3$ (see Eq.3.2). In that case, the angular variable (say ϕ) is the same in both the 2-ball and 2-sphere. The radial coordinate of the 2-ball $0 \leq r \leq R$ then maps to the angular coordinate $0 \leq \theta \leq \pi$. That is, given some angle ϕ, the radial variable of the 2- ball maps to a meridian of a sphere. The same logic applies to the transition from a 3-ball to the surface of a 3-sphere. The radial variable of the 3-ball maps to a "meridian" on the surface of a hyper-surface in four-dimensional space.

For neighbor atoms located at a radial distance r_{ik} from the central atom i within a three-dimensional region, we follow Bartók et al.'s approach [14] to establish a reasonable cutoff radius for mapping the region around a target atom onto the 3-sphere surface containing all the points (θ, θ_0, ϕ). Considering that $\theta_0 \in [0, \pi]$ and $r_{ik} < R_{cut}$, we can express the relationship between

the radial distance r_{ik} within R_{cut} and the polar angle θ_0 by

$$\theta_0 = \theta_0^{max} \frac{r_{ik}}{R_{cut}} = \pi \frac{r_{ik}}{R_{cut}} \tag{3.12}$$

We have thus defined the spherical coordinates (θ_0, θ, ϕ) in terms of the corresponding Cartesian coordinates

$$\begin{aligned}
r_3 &\equiv r_{ik} \\
r_{ik} &= \sqrt{x_k^2 + y_k^2 + z_k^2} \\
\phi &= \tan^{-1}\left(\frac{y_k}{x_k}\right) && \in [0, 2\pi] \\
\theta &= \tan^{-1}\left(\frac{\sqrt{x_k^2 + y_k^2}}{z_k}\right) = \cos^{-1}\left(\frac{z_k}{r_{ik}}\right) && \in [0, \pi] \\
\theta_0 &= \pi \frac{r_{ik}}{R_{cut}} && \in [0, \pi]
\end{aligned} \tag{3.13}$$

If ϕ is in range $[-\pi, \pi]$, conversion is required to satisfied condition in Eq. 3.10

$$\phi_{\text{converted}} = \begin{cases} \phi + 2\pi & \text{if } \phi < 0 \\ \phi & \text{otherwise} \end{cases} \tag{3.14}$$

3.2 Coupling Coefficient $H^{jmm'}_{j_1m_1m'_1,j_2m_2m'_2}$

The coupling coefficients $H^{jmm'}_{j_1m_1m'_1,j_2m_2m'_2}$ is the four- dimensional analouge of Clebsch-Gordan coefficients.

Let angular momentum j_1 and j_2, vector addition $\boldsymbol{j_1}+\boldsymbol{j_2}=\boldsymbol{j}$ with projections m_1 and m_2 on the quantization axis. The Clebsch-Gordan coefficients represent the probability amplitude that $\boldsymbol{j_1}$ and $\boldsymbol{j_2}$ are coupled into a resultant angular momentum $\boldsymbol{j}$ with projection m [20]

$$(j_1, m_1) \otimes (j_2, m_2) \to (j, m) \tag{3.15}$$

the symbol $\otimes$ indicates tensor product of basic vectors $|j_1, m_1\rangle$ of space $\mathscr{H}_{j_1}$ and $|j_2, m_2\rangle$ of space $\mathscr{H}_{j_2}$ which denoted as [21]

$$\{|j_1, m_1; j_2, m_2\rangle \ \equiv |j_1, m_1\rangle \otimes |j_2, m_2\rangle : j1 \geq m_1 \geq -j_1,\ j_2 \geq m_2 \geq -j_2\} \tag{3.16}$$

where $j_1,\ j_2$ are fixed and $m_1,\ m_2$ are running values.

The resulting set of basis vectors, denoted as $|j, m\rangle$, spans from $|j_1 - j_2|$ to $j_1 + j_2$ to ensure that the triangular inequality conditions are satisfied for every j in that set, with m ranging from $-j$ to j.

$$\langle j_1m_1j_2m_2|jm\rangle \neq 0 \iff |j_1 - j_2| \leq j \leq j_1 + j_1 \tag{3.17}$$

The Clebsch-Gordan coefficients are real numbers and act as elements of the unitary matrix that performs direct and inverse transformations between the state vectors $|j_1 m_1 j_2 m_2\rangle$ and $|(j_1 j_2) jm\rangle$

$$\begin{aligned} C^{jm}_{j_1 m_1 j_2 m_2} &\equiv \langle j_1 m_1 j_2 m_2 | (j_1 j_2) jm\rangle = \langle (j_1 j_2) jm | j_1 m_1 j_2 m_2\rangle \\ &\equiv \langle j_1 m_1 j_2 m_2 | jm\rangle = \langle jm | j_1 m_1 j_2 m_2\rangle \end{aligned} \tag{3.18}$$

where basis vectors define [21]

$$|(j_1 j_2) jm\rangle = \sum_{m_1 m_2} C^{jm}_{j_1 m_1 j_2 m_2} |j_1 m_1\rangle \otimes |j_2 m_2\rangle \tag{3.19}$$

The unitary relation is [20]

$$\begin{aligned} \sum_{m_1, m_2} C^{jm}_{j_1 m_1 j_2 m_2} C^{j'm'}_{j_1 m_1 j_2 m_2} &= \delta_{jj'} \delta_{mm'} \\ \sum_{j,m} C^{jm}_{j_1 m_1 j_2 m_2} C^{jm}_{j_1 m'_1 j_2 m'_2} &= \delta_{m_1 m'_1} \delta_{m_2 m'_2} \end{aligned} \tag{3.20}$$

The Clebsch-Gordan coefficients are related to the Wigner $3jm$ symbols [20]

$$C^{jm}_{j_1 m_1 j_2 m_2} = (-1)^{j_1 - j_2 + m} \sqrt{2j+1} \begin{pmatrix} j_1 & j_2 & j \\ m_1 & m_2 & -m \end{pmatrix} \tag{3.21}$$

Therefore, the algebraic form of the Clebsch-Gordan coefficients can be

obtained using Wigner's method of calculation [21]

$$
\begin{aligned}
C^{jm}_{j_1 m_1 j_2 m_2} = \; & \epsilon(j_1, j_2, j)\delta(m_1 + m_2, m) \\
& \times \left[\frac{(2j+1)(j+j_1-j_2)!(j-j_1+j_2)!(j_1+j_2-j)!}{(j+j_1+j_2+1)!}\right]^{\frac{1}{2}} \\
& \times \left[\frac{(j+m)!(j-m)!}{(j_1+m_1)!(j_1-m_1)!(j_2+m_2)!(j_2-m_2)!}\right]^{\frac{1}{2}} \\
& \times \sum_{s=s_{min}}^{s_{max}} \frac{(-1)^{j_2+m_2+s}(j_2+j+m_1-s)!(j_1-m_1+s)!}{s!(j-j_1+j_2-s)!(j+m-s)!(j_1-j_2-m+s)!}
\end{aligned}
\tag{3.22}
$$

where $s_{min} = \max(0, m_1 - j_1, j_2 - j_1 + m)$ and $s_{max} = \min(j_2 + j + m_1, j - j_1 + j_2, j + m)$ such that nowhere does factorial of negative number appear

$$
\begin{aligned}
s_{max} &= \begin{cases} j_2 + j + m_1 - s > 0 & \Leftrightarrow s < j_2 + j + m_1 \\ j - j_1 + j_2 - s > 0 & \Leftrightarrow s < j - j_1 + j_2 \\ j + m - s > 0 & \Leftrightarrow s < j + m \end{cases} \\
\implies s_{max} &= \min(j_2 + j + m_1, j - j_1 + j_2, j + m) \\
s_{min} &= \begin{cases} j_1 - m_1 + s > 0 & \Leftrightarrow s > m_1 - j_1 \\ j_1 - j_2 - m + s > 0 & \Leftrightarrow s > -j_1 + j_2 + m \\ s > 0 \end{cases} \\
\implies s_{min} &= \max(0, m_1 - j_1, j_2 - j_1 + m)
\end{aligned}
\tag{3.23}
$$

and

$$\epsilon(j_1, j_2, j) = \begin{cases} 1 \text{ if } (j_1, j_2, j) \text{ form a triangle} \\ 0 \text{ otherwise} \end{cases} \tag{3.24}$$

The Clebsch-Gordan coefficients for the symmetry group $SO(4)$ are the products of two $SO(3)$ groups, $SO(4) = SO(3) \otimes SO(3)$ [14, 22, 23]

$$H^{jmm'}_{j_1 m_1 m'_1, j_2 m_2 m'_2} \equiv C^{jm}_{j_1 m_1 j_2 m_2} C^{jm'}_{j_1 m'_1 j_2 m'_2} \tag{3.25}$$

The calculation for $C^{jm'}_{j_1 m'_1 j_2 m'2}$ is similar to $C^{jm}_{j_1 m_1 j_2 m_2}$. The corresponding Python function for calculating Clebsch-Gordan coefficients is given in Appendix A.2.

From Eq. 3.20, we can rewrite

$$\sum_{j,m} C^{jm}_{j_1 m_1 j_2 m_2} C^{jm'}_{j_1 m'_1 j_2 m'_2} = \delta_{m_1 m'_1} \delta_{m_2 m'_2} \delta_{m m'} \tag{3.26}$$

The input values for computing Clebsch-Gordan coefficients satisfy the following conditions

(1) $|j_1 - j_2| \leq j \leq j_1 + j_2$ and $j_1 + j_2 + j$ are non-negative integers with $\boldsymbol{j_1 + j_2 = j}$.

(2) $(j + j_1 - j_2)$ and $(j - j_1 + j_2)$ and $(j_1 + j_2 - j)$ are non-negative integers.

(3) j_1, j_2, and j do not exceed a positive integer $J = j_1 + j_2 + j$.

(4) $m_1, m_2, m, m_1', m_2', m'$ are integers or half-integers (positive or negative numbers).

(5) $(j_1 + m_1),\ (j_1 - m_1),\ (j_2 + m_2),\ (j_2 - m_2),\ (j + m),\ (j - m),\ (j_1 + m_1'),\ (j_1 - m_1'),\ (j_2 + m_2'),\ (j_2 - m_2'),\ (j + m'),\ (j - m')$ are non-negative integers.

(6) $m_1 + m_2 = m, m_1' + m_2' = m'$

(7) $|m_1| \leq j_1,\ |m_2| \leq j_2,\ |m| \leq j,\ |m_1'| \leq j_1,\ |m_2'| \leq j_2,\ |m'| \leq j.$

3.3 Rotation Matrix $U^j_{mm'}$

The spherical harmonics are eigenfunctions of the angular part of the Laplace operator, which makes them fundamental in describing the angular behavior of functions in various physical systems. When considering the basis functions on the four-dimensional unit sphere, these can be derived using rotation matrices that involve a rotation angle of $\omega \equiv 2\theta_0$ around the rotation axis $\mathrm{n}(\theta, \phi)$. The angles (θ_0, θ, ϕ) takes values within a specific range

$$0 \leq \theta_0 \leq 2\pi,\ 0 \leq \theta \leq \pi,\ 0 \leq \phi \leq 2\pi \tag{3.27}$$

We follow the equations outlined in Ref. [20] to derive the explicit form

of $U^j_{mm'}(\theta_0, \theta, \phi)$ using Wigner-D functions

$$U^j_{mm'}(\theta_0, \theta, \phi) = \sum_{m''=-j}^{j} D^j_{mm''}(\phi, \theta, -\phi) e^{-im''\theta_0} D_{m''m'}(\phi, -\theta, -\phi) \tag{3.28}$$

$D^j_{mm''}(\phi, \theta, -\phi)$ and $D_{m''m'}(\phi, -\theta, -\phi)$ are calculated using the general explicit forms of the Wigner-D functions, denoted as $D^j_{mm'}(\alpha, \beta, \gamma)$

$$D^j_{mm'}(\alpha, \beta, \gamma) = e^{-im\alpha} d^j_{mm'}(\beta) e^{-im'\gamma} \tag{3.29}$$

The Wigner-D functions are complex and they depend on real Euler angles (α, β, ϕ) and are defined in the domain

$$0 \leq \alpha < \pi,\ 0 \leq \beta \leq \pi,\ 0 \leq \gamma < 2\pi \tag{3.30}$$

The expression for the real function $d^j_{mm'}(\beta)$ is as follows

$$\begin{aligned} d^j_{mm'}(\beta) &= [(j+m)!(j-m)!(j+m')!(j-m')!]^{\frac{1}{2}} \\ &\times \sum_k (-1)^k \frac{\left(cos\frac{\beta}{2}\right)^{2j-2k+m-m'} \left(sin\frac{\beta}{2}\right)^{2k-m+m'}}{k!(j+m-k)!(j-m'-k)!(m'-m+k)!} \end{aligned} \tag{3.31}$$

where k is an integer variable that ranges over all possible values, ensuring the non-negativity of factorial arguments. The summations in Eq. 3.31 consist of $(N+1)$ terms, where N represents the minimum value among

$j+m, j-m, j+m'$, and $j-m'$.

$$
\begin{aligned}
k_{min} &= \begin{cases} j+m-k>0 & \Leftrightarrow k<j-m<j+m \\ j-m'-k>0 & \Leftrightarrow k<j-m'<j+m \end{cases} \\
\implies k_{min} &= \min(j+m, j-m') \\
k_{max} &= \begin{cases} k>0 & \\ m-m'+k>0 & \Leftrightarrow k>m'-m \end{cases} \\
\implies k_{max} &= \max(0, m'-m)
\end{aligned}
\tag{3.32}
$$

3.4 Expansion Coefficients $u^j_{m,m'}$

From Eqs.2.6 and 2.7 , we can express expansion coefficients $u^j_{m,m'}$ as

$$
\begin{aligned}
u^{\mu}_{j,m,m'} &= w^{self}_{\mu_i} U_{j,m,m'}(0,0,0) \\
&\quad + \sum_{r_{ik}<R_{cut}} f_c(r_{ik}; R_{cut}) w_{\mu_k} U^j_{m,m'}(\theta_0, \theta, \phi) \delta_{\mu\mu_k}
\end{aligned}
\tag{3.33}
$$

where $w^{self}_{\mu_i}$ represents the weight coefficient for the self-contribution of the central atom ($w^{self}\mu_i$ can be either 0 or 1), and $w_{\mu k}$ represents the weight of contribution from neighboring atoms to $u^{\mu}_{j,m,m'}$. Since $U^j_{m,m'}(0,0,0) = 0$, we can disregard the first term in Eq. 3.33, resulting in a simplified equation

$$
u^{\mu}_{j,m,m'} = \sum_{r_{ik}<R_{cut}} f_c(r_{ik}; R_{cut}) w_{\mu_k} U^j_{m,m'}(\theta_0, \theta, \phi) \delta_{\mu\mu_k} \tag{3.34}
$$

where $\delta_{\mu\mu_k}$ indicates that only neighbor atoms of element μ contribute to $u^{\mu}_{j,m,m'}$. For single element system we set $w_{\mu_k} = 1$ and $\delta_{\mu\mu_k} = 1$, the expansion coefficients become

$$u^{j}_{mm'} = \sum_{r_{ik}<R_{cut}} f_c(r_{ik}; R_{cut}) U^{j}_{m,m'}(\theta_0, \theta, \phi) \tag{3.35}$$

where

$$f_c(r_{ik}; R_{cut}) = \begin{cases} \left[\cos\ \left(\frac{\pi r_{ik}}{R_{cut}}\right) + 1\right]/2 & r_{ik} \leq R_{cut} \\ 0 & r_i k > R_{cut} \end{cases} \tag{3.36}$$

CHAPTER 4

RESULTS

4.1 Bispectrum Calculation for an Individual Atom

4.1.1 Neighbor Atoms Case Study

To verify the invariant of bispectrum components resulting from neighboring atom rotations, several analyses were conducted (see Appendix B.5). In Figure 6 (a), it can be observed that when there are no neighboring atoms within the cell and the self-contribution $w_{\mu_i}^{self}$ is not taken into account, the bispectrum component $B_{j_1,j_2,j} = 0$

Figures 6 (b - e) depict the same calculation performed for $B_{j_1,j_2,j} = 0.75$ after introducing a neighboring atom and permuting its position. The values of $B_{j_1,j_2,j}$ are also displayed in Figure 6. In the case of two neighboring atoms (f - g), $B_{j_1,j_2,j} = 1.13$, for three neighboring atoms (h - k), $B_{j_1,j_2,j} = 1.28$. These results confirm that the bispectrum components maintain their invariant properties when the number of atoms N remains constant within a cutoff radius R_{cut} which emphasized the inherent properties of local atomic density.

Furthermore, we observed that as more neighboring atoms are added while maintaining the same R_{cut}, the bispectrum exhibits an increasing trend.

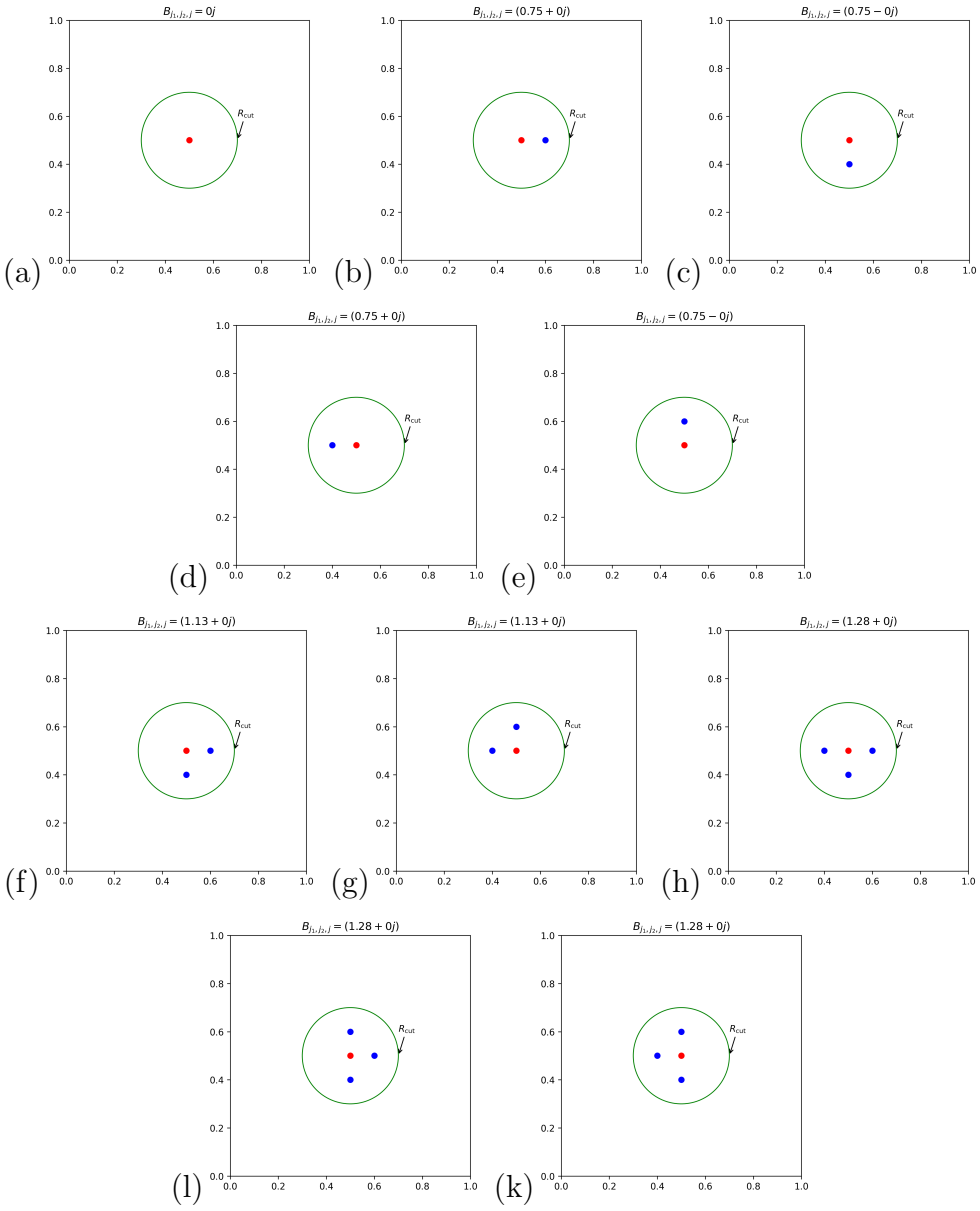


Figure 10: Visualization of the neighbors of center atom i with bispectrum calculation

(a) central atom i in an empty cell, and (b-e) modified position of same element type neighbor atom k in the case of a single neighboring atom, (f-g) two neighboring atoms, and (h-k) three neighboring atoms, Appendix B.5.

This indicates a correlation between the number of neighboring atoms, spacial volume and the magnitude of the bispectrum.

4.1.2 A Central Atom in Amorphous Silicon

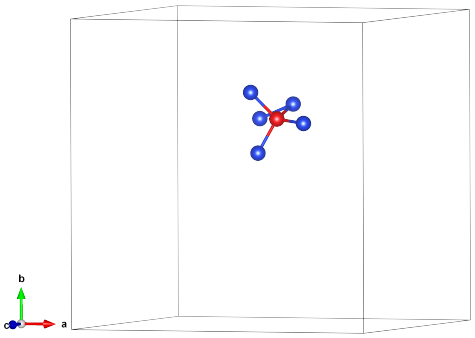
The main objective of our investigation was to assess the bispectrum calculation for a central atom within an amorphous silicon cell. Detailed information regarding the structural arrangement of neighboring atoms in amorphous silicon (a-Si) can be found in Table 1. For a visual representation of the atom arrangement and the runtime execution of the bispectrum calculation, please refer to Figure 11, which illustrates the atom configuration.

We further investigate the symmetric properties of the bispectrum component discussed in Ref. [15]. We examine the behavior of the bispectrum component when interchanging the values of j_1 and j_2, while maintaining a constant angular momentum value of j. Our computer programming findings support that the bispectrum component remains unchanged. Specifically, when interchanging indices j_1, j_2, and j under the restriction $j_2 \leq j_1 \leq j$, our results align with the proof established in [15].

$$\frac{B_{j,j_1,j_2}}{2j+1} = \frac{B_{j1,j_2,j}}{2j_1+1} = \frac{B_{j_2,j,j_1}}{2j_2+1} \tag{4.1}$$

Note: B_{j,j_2,j_1} implies that the first index input j corresponds to the resultant angular momentum. Additional information about the computer program

used in this study can be found in Appendix A.4 and B.4.



(a)

	r_ik	theta_0	theta	phi	w_ik	delta	r_cut
0	0.160620	2.018412	0.699198	0.049713	1.0	1.0	0.25
1	0.122957	1.545120	0.382047	1.253323	1.0	1.0	0.25
2	0.122197	1.535575	1.184880	5.410382	1.0	1.0	0.25
3	0.126199	1.585869	1.255835	1.133789	1.0	1.0	0.25
4	0.119252	1.498569	1.154929	6.181818	1.0	1.0	0.25

Bispectrum coefficient for atom ID17 (9.12-0j)

(b) Execution time for bispectrum calculation 2.15 seconds

Figure 11: Bispectrum coefficient for a center atom i in (a-Si) unit cell

(a) VESTA visualization for a central atom i and its neighbor atoms. (b) Bispectrum calculation and execute running time for i with $j = 5/2, j1 = 3/2, j2 = 1$

Symmetry

Lattice type	P
Space group name	P 1
Space group number	1
Setting number	1

Lattice (Conventional)

a	b	c	α	β	γ
18.73370	18.73370	18.73370	90°	90°	90°

Unit-cell volume = 6574.620 197 Å^3, (a, b, c) unit is Å.

Atomic Positions

No.	**Label**	x	y	z	x_k	y_k	z_k
1	si1_1	0.54433	0.66558	0.55988	0.000000	0.000000	0.000000
2	si1_1	0.44108	0.66044	0.43695	-0.103248	-0.005137	-0.122932
3	si1_1	0.55864	0.70912	0.44579	0.014310	0.043550	-0.114092
4	si1_1	0.47157	0.75231	0.60588	-0.072758	0.086734	0.045996
5	si1_1	0.49355	0.55686	0.59898	-0.050784	-0.108715	0.039094
6	si1_1	0.65286	0.65454	0.60806	0.108528	-0.011039	0.048176

Table 1: Neighbor atoms structure of the central atom i

within $R_{cut} = 0.25$, Appendix B.6

4.2 Bispectrum Calculation for Various Silicon Models

Bispectrum calculations were performed on various models of Silicon (Si), includes amourphous Si with uniform scaling of bond lengths (short, medium, long), crystalline Si, Si with a passive defect, and Si with a self-interstitial. For all systems,($j_1 = 5/2$, $j_2 = 5/2$) and $j_{max} = 5$.)

Figures 13, 14,15, 16, 18 and 20 display side-by-side plots of the bispectrum calculations and the potential function results obtained from the self-consistent field process in the OLCAO method. These visualization aim to see if there is any obvious pattern within the set of bispectrum calculations for each atom in comparison to the potential function associated with each of them. However, there is no clear and straightforward way to visually identify the connections and mathematical relationships between the bispectrum set $\{B_i\}$ and the coefficient set $\{A_i\}$ used in the potential function calculation in figures 13, 14 and 15 for models of amorphous Si with uniform scaling of bond lengths (short, medium, long).

In Figures 18 and 20, an important correlation can be observed between the peaks present in the bispectrum plots and those in the potential function plots. This correlation suggests the possibility of a linear relationship between the coefficients B_i and A_i. It is worth noting that the potential function for each site is expressed as $V_B(\mathbf{r}) = \sum_{i=1}^{N_A} A_i e^{-\alpha_i r^2}$, where N_A represents the number of Gaussian functions, A_i is the amplitude, and α_i influences the

decay behavior of the Gaussian functions.

We observe the peaks in bispectrum plots appear at the same region in potential function plots (see Figures 18 and 20). Since there is lattice distortions (abnormal of atoms arrangement in local neighborhood) in both model of the crystalline silicon with an I4 set of self interstitial silicon (see Figure. 19) and model of silicon with a passive defect (see Figure. 17.

4.2.1 Amorphous Silicon with Uniform Scaling Bond Lengths

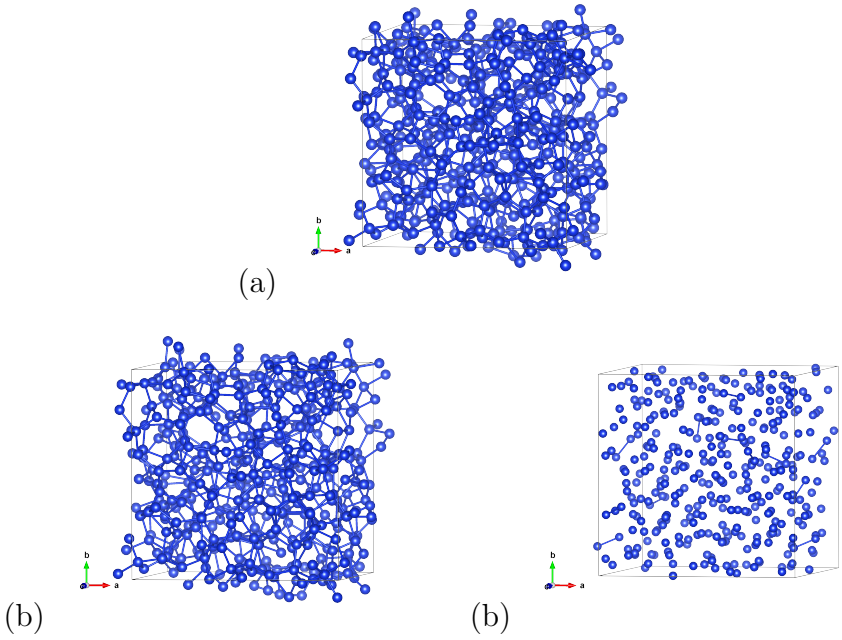


Figure 12: Models of (a-Si) with uniform scaling of bond lengths: short (a), medium (b), long (c), $N_A = 340$ atoms.

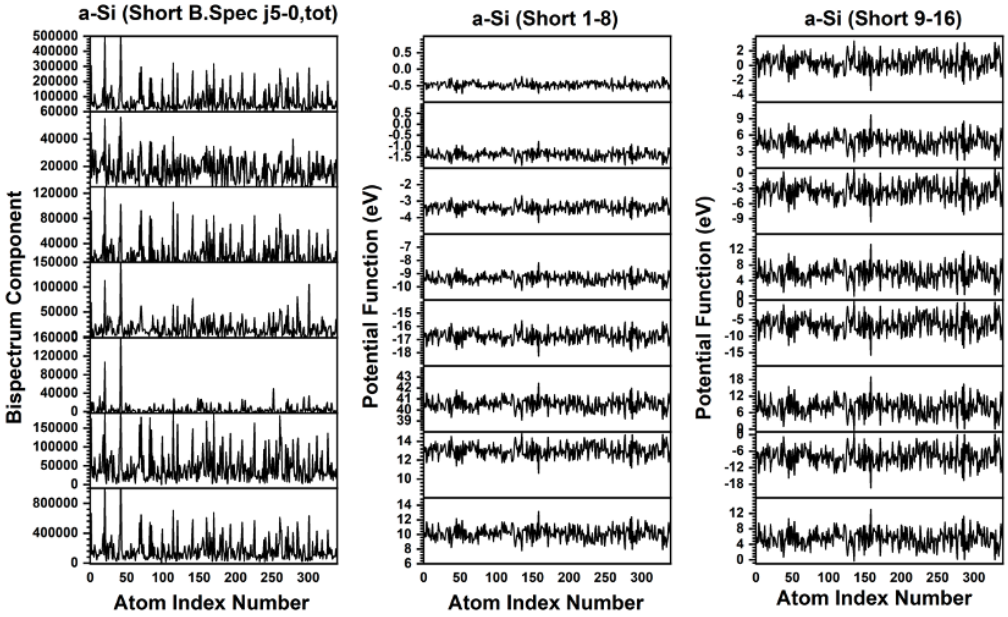


Figure 13: (a-Si) short model - Bispectrum components calculation (left) and SFC calculation of the potential in OLCAO (middle and right).

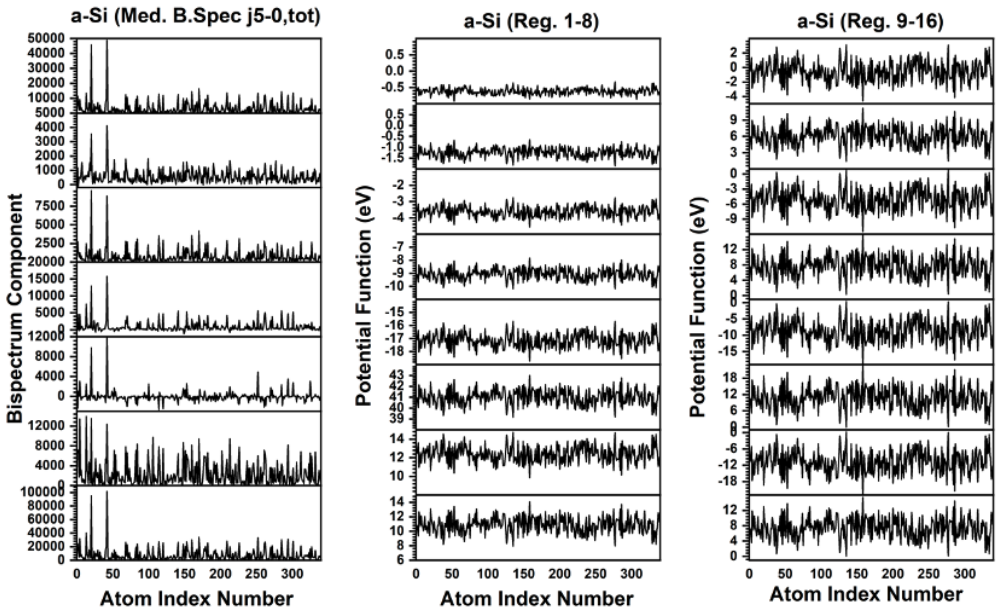


Figure 14: (a-Si) medium model - Bispectrum components calculation (left) and SFC calculation of the potential in OLCAO (middle and right).

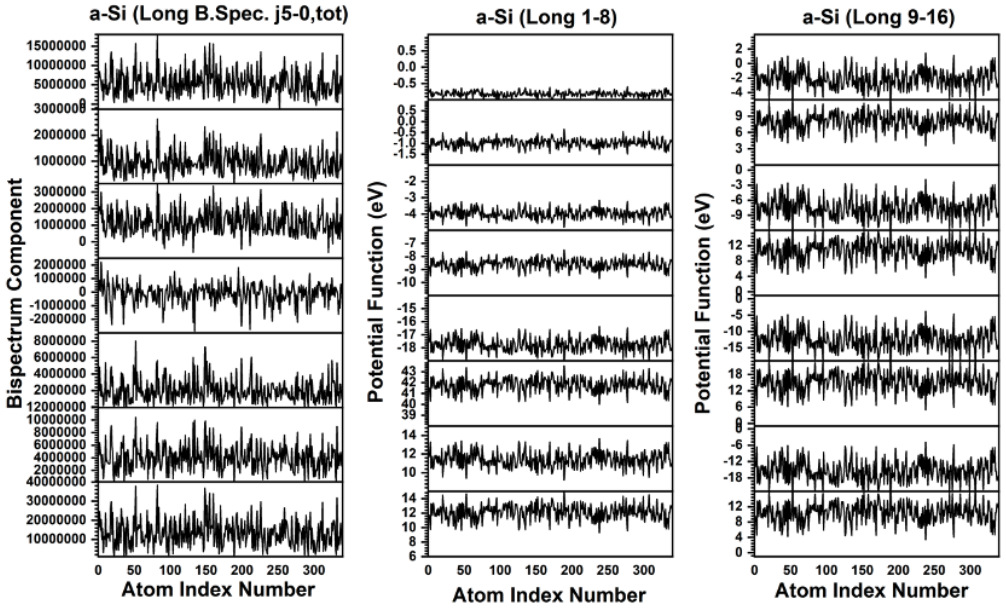


Figure 15: (a-Si) long model - Bispectrum components calculation (left) and SFC calculation of the potential in OLCAO (middle and right).

4.2.2 Crystalline Silicon

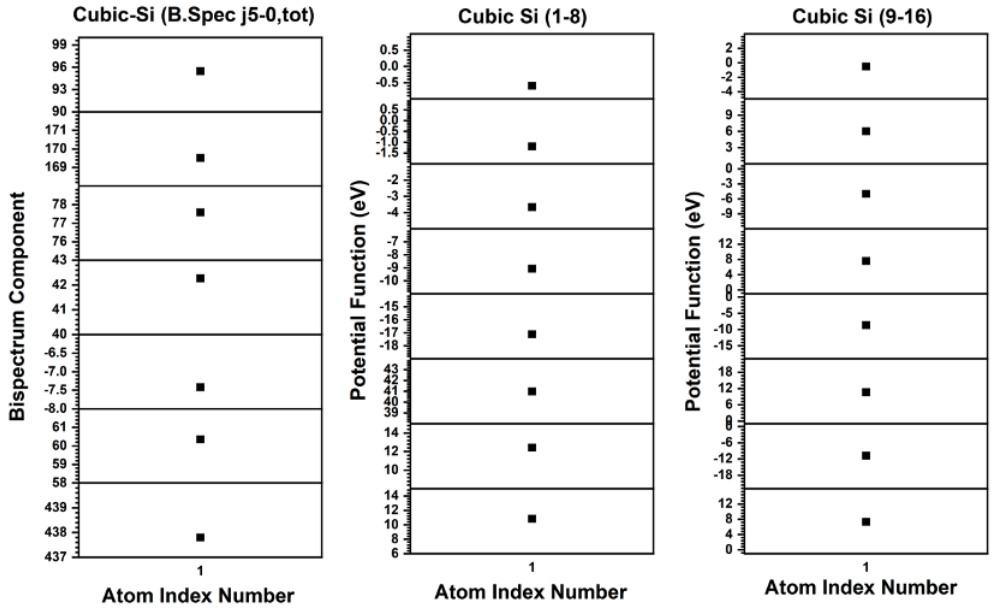


Figure 16: Crystalline Si (cubic) - Bispectrum components calculation (left) and SFC calculation of the potential in OLCAO (middle and right).

4.2.3 Passive Defect Silicon

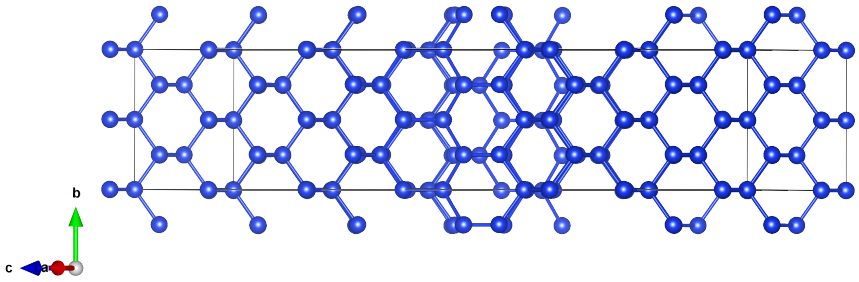


Figure 17: A model of silicon with a passive defect $N_A = 180$ atoms

a	b	c	α	β	γ
12.59210	7.59330	36.65250	90°	89.6564°	90°

Unit-cell volume = 3504.4875 Å^3, (a, b, c) unit is Å, space group: P 1.

Table 2: Passive defect Si structure

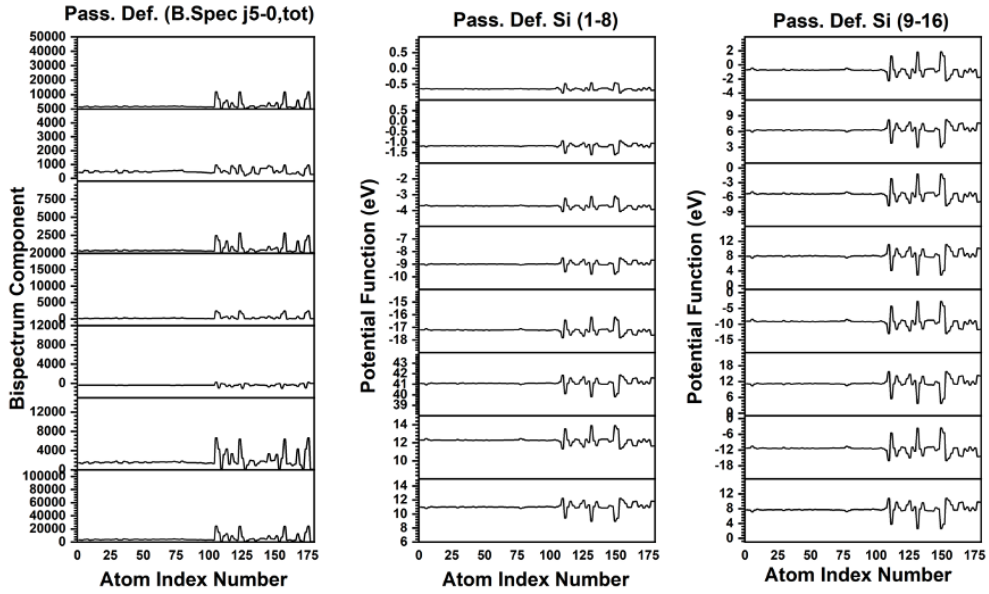


Figure 18: Passive defect Si model: Bispectrum components calculation (left) and SFC calculation of the potential in OLCAO (middle and right).

4.2.4 I4 Self-interstitial Silicon

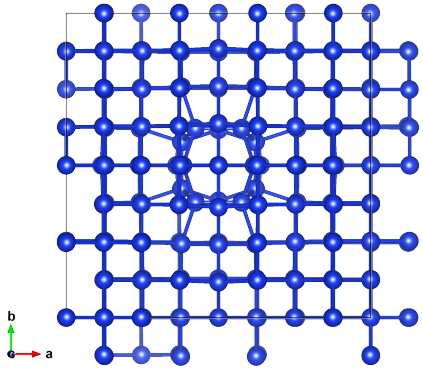


Figure 19: Crystalline silicon with an I4 set of self interstitial silicon $N_A = 196$ atoms

a	b	c	α	β	γ
15.20740	15.20740	16.49250	90°	90°	90°

Unit-cell volume = 3814.1385 Å^3, (a, b, c) unit is Å.

Table 3: I4 Interstitial Si Structure

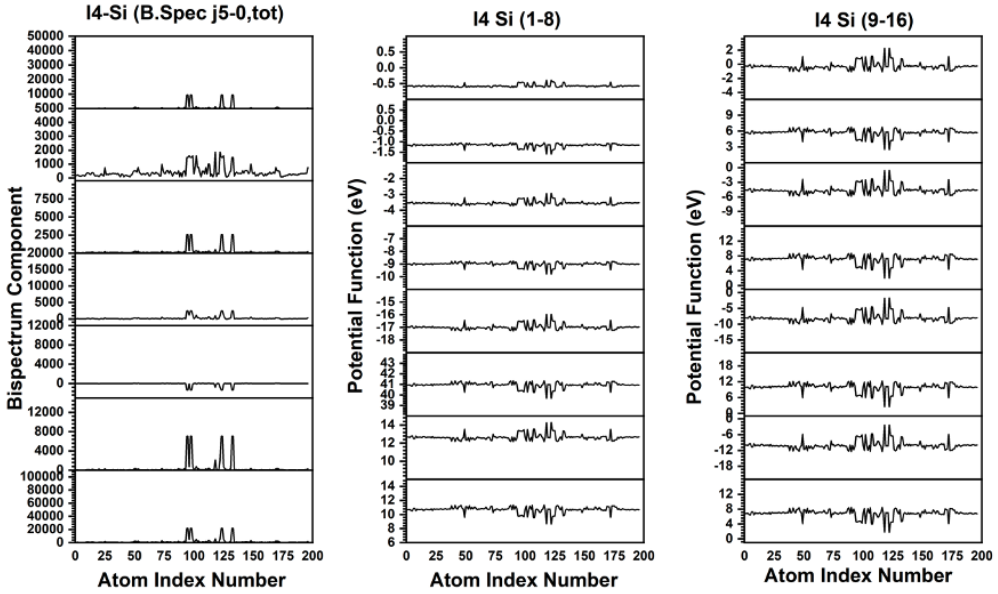


Figure 20: I4 Interstitial Si model - Bispectrum components calculation (left) and SFC calculation of the potential in OLCAO (middle and right).

4.3 Validation Key Functions

4.3.1 Clebsch-Gordan Coefficients

We conducted a thorough validation of the key functions employed in our calculations. The accuracy of the Clebsch-Gordan coefficients calculation was

verified by comparing the results with a Clebsch-Gordan Table and a third-party library. We determine the value for $C^{j=5/2,m=+3/2}_{j_1=2,m_1=1,j_2=1/2,m_2=1/2}$ to be $\sqrt{4/5}$ (see Figure 21). This value is obtained by considering the addition of two spins, $j_1 = 2$ and $j_2 = 1/2$, resulting in a total spin range of $3/2 \leq j \leq 5/2$. The results obtained from the computer program described in Appendix B.2 yielded identical results to those obtained using other methods, confirming the correct implementation of the program.

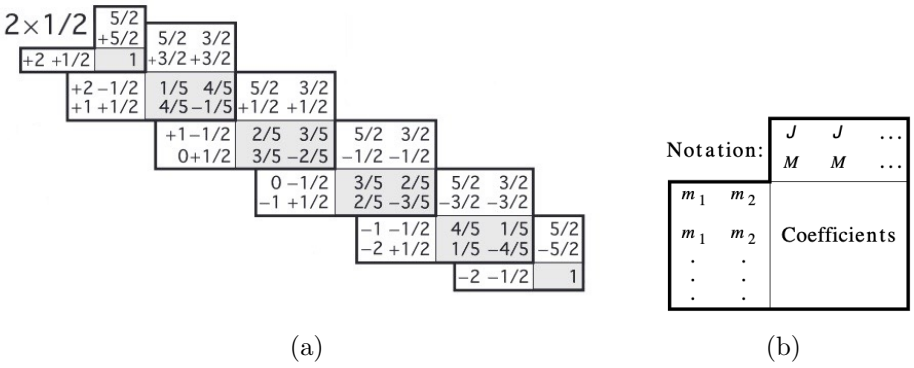


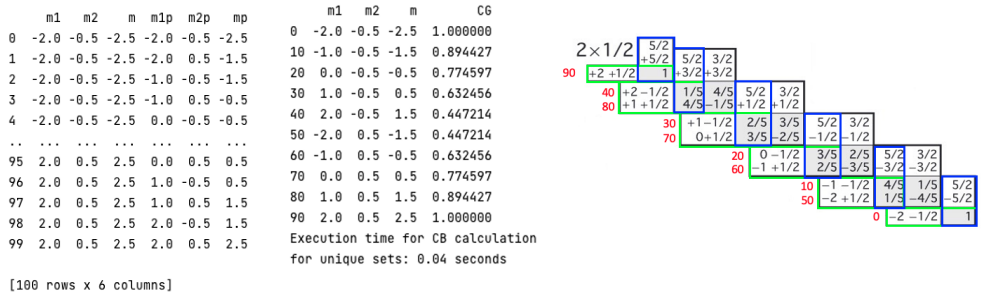
Figure 21: Tables of the Clebsch-Gordan coefficients, Cohen et al.

Note: A square-root sign is to be understood over every coefficient, with any minus sign taken outside the radical.

Method	$C^{jm}_{j_1 m_1 j_2 m_2}$
Calc	0.8944271909999159+0j
SymPy	0.894427190999916
Table	$\sqrt{4/5} \approx 0.894$

Table 4: Comparing Clebsch-Gordan Coefficient calculation methods

We also performed tests on the `generate_m_values` function (see Appendix A.4) by comparing the sets of unique values (m_1, m_2, m) generated from a specific input (j, j_1, j_2) with their corresponding Clebsch-Gordan coefficients calculated using the `clebsch_gordan` function (see Appendix A.4), and compared them with the Clebsch-Gordan coefficient table provided by Cohen et al.



	m1	m2	m	m1p	m2p	mp
0	-2.0	-0.5	-2.5	-2.0	-0.5	-2.5
1	-2.0	-0.5	-2.5	-2.0	0.5	-1.5
2	-2.0	-0.5	-2.5	-1.0	-0.5	-1.5
3	-2.0	-0.5	-2.5	-1.0	0.5	-0.5
4	-2.0	-0.5	-2.5	0.0	-0.5	-0.5
..	...	...	...	...	...	...
95	2.0	0.5	2.5	0.0	0.5	0.5
96	2.0	0.5	2.5	1.0	-0.5	0.5
97	2.0	0.5	2.5	1.0	0.5	1.5
98	2.0	0.5	2.5	2.0	-0.5	1.5
99	2.0	0.5	2.5	2.0	0.5	2.5

[100 rows x 6 columns]

	m1	m2	m	CG
0	-2.0	-0.5	-2.5	1.000000
10	-1.0	-0.5	-1.5	0.894427
20	0.0	-0.5	-0.5	0.774597
30	1.0	-0.5	0.5	0.632456
40	2.0	-0.5	1.5	0.447214
50	-2.0	0.5	-1.5	0.447214
60	-1.0	0.5	-0.5	0.632456
70	0.0	0.5	0.5	0.774597
80	1.0	0.5	1.5	0.894427
90	2.0	0.5	2.5	1.000000

Execution time for CB calculation for unique sets: 0.04 seconds

(a) (b)

Figure 22: Clebsch-Gordan coefficients: complete generated set and unique matching sets

(a) Full generated set with input ($j = 5/2$, $j_1 = 2$, $j_2 = 1/2$). (b) Unique set with Clebsch-Gordan coefficient calculation and (c) matching set to the Clebsch-Gordan table (right).

Figure 22 illustrates that the generated sets with Clebsch-Gordan coefficients match the results in the Clebsch-Gordan table.

4.3.2 Rotation Matrix via Wigner-D Functions

Similarly, we validated the rotation matrix via Wigner-D functions by comparing the computed values with those obtained from established tables (see Table 5). The results demonstrated a high level of agreement, further substantiating the accuracy of our implementation. Editting
Table 4.5 and 4.25 in Ref. [20] show that

$$\begin{aligned} d^{3/2}_{m=3/2,m'=-3/2}(\theta) &= -\sin^3\frac{\theta}{2} \\ U^{j=3/2}_{m=3/2,m'=-3/2}(\theta_0,\theta,\phi) &= i(\sin\frac{\theta_0}{2}\,\sin\,\theta\;e^{-i\phi})^3 \end{aligned} \tag{4.2}$$

Method	$d^{3/2}_{mm'}(\theta)$	$U^{3/2}_{mm'}(\theta_0,\theta,\phi)$
Calc	≈ -0.125	≈ 0.0884
Table	≈ -0.125	≈ 0.0884

Table 5: Comparing calculation methods for $d^{j}_{mm'}(\theta)$ and rotation matrix $U^{j}_{mm'}(\theta_0,\theta,\phi)$

j=3/2, m= 3/2, m'=-3/2, $\theta_0 = \pi/4$, $\theta = \pi/3$, $\phi = \pi/4$

Our computer programming results, produce the same values as the tables presented in Ref. [20].

CHAPTER 5

CONCLUSION AND FUTURE DRIECTIONS

5.1 Conclusion

Results regarding the optimization of the run time for calculating the bispectrum component is discussed, including a comparison with key function program code that uses third-party libraries such as SymPy. A computer program is developed to automatically generate bispectrum components for a single-element system in a periodic unit cell and for various silicon models. We investigated the symmetric properties of the bispectrum components (see section 4.1), which align with the proof established in [15].

It is important to emphasize the significance of the bispectrum as three-point correlation [14] function that is a resulting coefficients have real values, and are similar to how the structure factor is computed from Fourier components. It can be thought of as a set of "cubic rotational invariant" that characterize the internal structure of materials, might be effectively revealing the arrangement of atoms within crystals. This understanding aligns well with the observation that peaks in the bispectrum plots appear in the same regions as peaks in the potential function (refer to Figures 18 and 20). These peaks are a result of lattice distortions, which shows the abnormal of atoms

arrangement and structure in local neighbors.

Additional analysis is necessary to comprehensively grasp the outcomes for distinct Si models and the correlation between bispectrum components set $\{B_i\}$ for each local environment and the coefficients set $\{A_i\}$ in the potential funtions computed from SFC process in OLCAO (mentioned in section 4.2). Moreover, it is essential to undertake further refinement and rigorous testing of the program prior to its broader application to systems involving multiple elements.

One challenge in this research is defining a suitable cut-off radius for evaluation of the bispectrum component to avoid neglecting the interaction between a targeted atom and its neighbors. The cut-off radius is weighted as a function of the elements involved to accommodate different types of bonding (e.g., ionic, covalent, metallic). Additionally, for properly defining and training a neural network (see below), it is vital that we provide a clear correlation between the physical (geometric) features of the bispectrum components and the electronic features that may simultaneously be present to avoid too much redundancy in the input data. This lack of understanding can limit the development of methods to predict the electronic structure properties based on the bispectrum components, underscoring the need for further research in this area.

5.2 Proposed Neural Network

In OLCAO, the total electronic potential function of a crystal is expressed as a sum of atom-centered potential functions. Each atom-centered potential function is represented as a sum of Gaussian functions. However, it is vital to recognize that although the potential function is an assembly of site-centered functions it cannot be said that the potential function from a given site is the potential function "of" the atom at that site. Rather, the potential function at a given site is determined by the influence of all nearby atoms. Therefore, it is intuitive to seek a ML model that follows a similar structure. In this case, it is important to find a way for the input data structure to incorporate that feature of the potential function, which consists of a mixture of influences derived from the neighboring atoms. Each component of this mixture represents a cluster or subpopulation within the local region. To capture this structure, we propose a neural network framework based on Mixture Density Network (MDN) [24] for the training process. It is a neural network architecture that models a probability distribution as a mixture of several Gaussian distributions. MDNs are commonly used in probabilistic modeling tasks where the output is a continuous value with uncertain or multimodal behavior. The network predicts parameters for each Gaussian component, such as means, variances, and mixing coefficients, allowing it to capture complex and non-linear distributions effectively.

This approach involves encoding the local, medium-range, and long-range (global) influences for each atom. In many cases, electron interactions are considered 'short-sighted,' meaning that they are mainly affected by nearby atoms only. However, our proposed method might overcome this limitation and effectively addresses novel long-range electronic structure properties such as those found in metallic or certain magnetic materials.

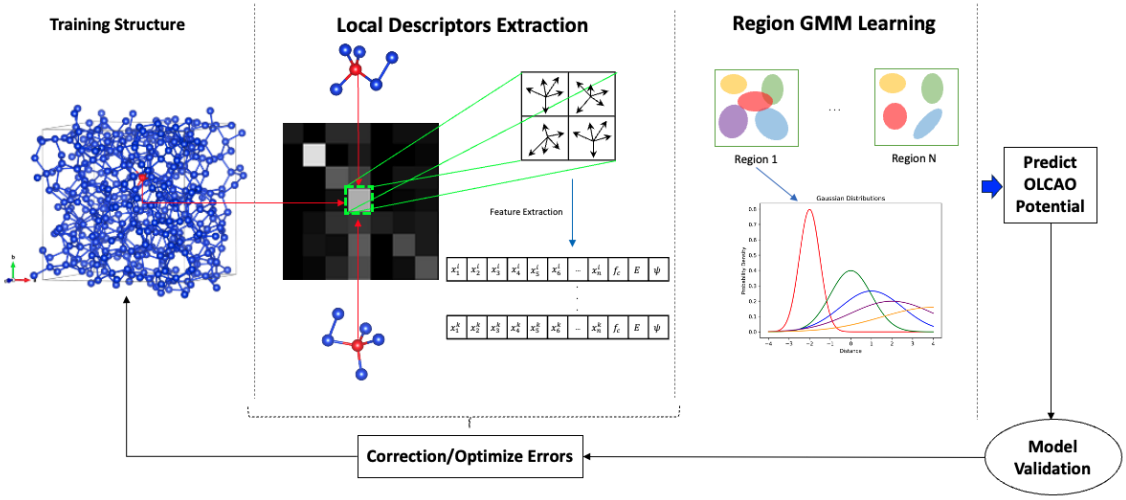


Figure 23: Conceptual Framework for a proposed neural network

5.3 Future Directions

5.3.1 Incorporating Real-Life Complexities

Most material models typically assume the idealized theoretical structure of materials, disregarding real-life complexities that can have a significant impact on their properties. These complexities, including impurities, mineral

inclusions, and structural defects, can affect the accuracy and applicability of material models in practical scenarios. Similar to computer vision, which relies on real-life data such as images and text to train models, it is crucial to incorporate these real-life complexities to enhance the accuracy and applicability of material models. This approach helps mitigate biases in training data and enables a more comprehensive understanding of the studied descriptors.

One promising research direction involves the pursuit of an atomic-level comprehension of the structure and behavior of nanomaterials. A notable example of this is the investigation into crystalline diblock copolypeptoids. In this study, researchers utilize a combination of sequence-controlled synthesis, cryogenic transmission electron microscopy (cryo-TEM), and molecular dynamics simulation [25]. Cryo-TEM is a highly effective imaging technique that enables the direct visualization of nanoscale structures at low temperatures. By employing this technique, we can gain valuable insights into the atomic details in position space that are otherwise inaccessible through conventional scattering techniques.

However, it's important to note that cryo-TEM primarily provides information about the two-dimensional structure of the nanosheets, rather than the complete three-dimensional atomic arrangements. To fully comprehend atomic structures, additional techniques must be employed in conjunction

with cryo-TEM.

5.3.2 Enhancing Cross-Disciplinary Applications: Descriptors in Computer Vision and Electronic Structure Predictions

Bridging descriptors in computer vision and electronic structure predictions aims to establish connections between the representations used in these two domains, enabling knowledge transfer and cross-disciplinary applications.

In computer vision, descriptors derived from pixels serve as mathematical representations or features that depict the visual characteristics of an image or a specific region within it. These descriptors encompass essential aspects like shape, intensity distribution, or gradient information. Typically, descriptors are computed by analyzing a neighborhood of pixels surrounding a specific point of interest or by extracting features from larger regions of the image. They play a pivotal role in extracting significant information from visual data, such as images or videos.

On the other hand, electronic structure predictions involve studying the properties and behavior of atoms and molecules using quantum mechanics. Descriptors in this domain represent the atomic or molecular structure numerically, enabling processing by machine learning or other computational methods. Examples of electronic structure descriptors include Coulomb matrices, Bag of Bonds, or local environment descriptors, bispectrum.

Bridging these two domains necessitates finding common representations or feature extraction methods applicable to both computer vision and electronic structure predictions. One potential approach involves leveraging transfer learning, adapting pre-trained models or representations from computer vision for electronic structure predictions, and vice versa. This can be achieved by fine-tuning pre-trained models on specific datasets or extracting features from intermediate layers of the models.

For example, convolutional neural networks (CNNs), widely used in computer vision tasks, can be repurposed for extracting features from atomic or molecular structures by utilizing their convolutional layers. These features can then serve as descriptors for electronic structure predictions. Similarly, techniques like graph neural networks (GNNs), successful in electronic structure predictions, can be adapted for image analysis tasks in computer vision by representing images as graphs.

Another approach to bridging descriptors involves exploring the relationship between image features used in computer vision and the electronic properties of materials. By investigating correlations or patterns between image descriptors and material descriptors, it may be possible to develop new descriptors that simultaneously capture visual information and electronic structure information. This opens up possibilities such as predicting material properties from images or enhancing computer vision tasks using electronic

structure information.

For instance, local feature descriptors, commonly used in computer vision for image feature detection and matching, offer a similar concept. The Scale Invariant Feature Transform (SIFT) algorithm, for example, identifies and describes distinctive features in images that are invariant to scale, rotation, and affine transformations. These descriptors enable feature matching across different images.

While the application of techniques like SIFT/ PCA-SIFT [26] to predict electronic structure holds potential, it requires establishing meaningful correlations between the image features extracted by SIFT/ PCA-SIFT and the electronic structure properties. This involves converting electronic structure data into visual representations or generating images that capture relevant information regarding the electronic structure and local region of the studied system. However, it's essential to recognize that predicting electronic structure is a complex problem requiring quantum mechanical calculations and analysis. The feasibility of applying SIFT/ PCA-SIFT or similar techniques depends on the ability to establish meaningful correlations, necessitating further research and development.

Bridging descriptors in computer vision and electronic structure predictions is an active area of research, aiming to leverage the strengths of both domains and enable cross-disciplinary applications. By finding common rep-

resentations or developing new descriptors that capture information from both computer vision and electronic structure predictions, researchers can enhance the understanding and predictive capabilities of both fields.

Listings

APPENDIX A

PROGRAM CODES

Key functions for computing bispectrum components

Listing A.1: INPUT_param

```
"""
This module is to get neighbor atoms and generate all
   ↪ possible parameters for bispectrum calculation
"""
from Bio.PDB.MMCIF2Dict import MMCIF2Dict
import pandas as pd
import numpy as np
import json

def neighbor_params(center_atom_id:int, r_mu, R_cut, input_file
   ↪ :str, output_dir:str, file_type:str):
    """
    This function is used to get input value from cif file and
       ↪  define neighbor atoms parameters
    """
    if file_type == "cif":
        # DATA PREPARATION
        dico = MMCIF2Dict(input_file)
        df_cif = pd.DataFrame.from_dict(dico, orient='index')
        x = df_cif.iloc[-3]
        y = df_cif.iloc[-2]
        z = df_cif.iloc[-1]
        atom_type = df_cif.iloc[-4]
        x_array = np.array(x[0], dtype=float)
        y_array = np.array(y[0], dtype=float)
        z_array = np.array(z[0], dtype=float)
        atom_type_array = np.array(atom_type[0], dtype=str)
        df = pd.DataFrame({"atom_type": atom_type_array, "X":
           ↪ x_array, "Y": y_array, "Z": z_array})
        # Estimate list of potentially atoms in the center
           ↪ cell
```

```
id = center_atom_id
# id
x_i = df['X'].iloc[id]
y_i = df['Y'].iloc[id]
z_i = df['Z'].iloc[id]
# print(x_i,y_i,z_i)
X_array = df['X'].to_numpy()
Y_array = df['Y'].to_numpy()
Z_array = df['Z'].to_numpy()
X_k_array = X_array - x_i
Y_k_array = Y_array - y_i
Z_k_array = Z_array - z_i
r_ik = np.sqrt(np.square(X_k_array) + np.square(
    ↪ Y_k_array) + np.square(Z_k_array))
df['X_k'], df['Y_k'], df['Z_k'], df['r_ik'] = X_k_array
    ↪ , Y_k_array, Z_k_array, r_ik
# INPUT values
cell_length = df.iloc[4] # index row start from 0
    ↪ _cell_length_a at row 5 index [4]
#r_mu = 0.0779 # scale atomic radius w.r.t cell length
#R_cut = 0.25 # scaled value w.r.t cell length (for Si
    ↪ -Si case)
condition = (df['r_ik'] + r_mu <= R_cut) & (df['r_ik']
    ↪ != 0)
df_ik = df.loc[condition].copy(deep=True)
# ANGEL CONVERSION
# theta_0
r_ik_array = df_ik['r_ik'].to_numpy() # r_ik from
    ↪ selected neighbors
r_0_array = np.full((r_ik_array.shape), R_cut)
theta_0_array = np.pi * (np.divide(r_ik_array,
    ↪ r_0_array))
# theta
Z_k_abs_array = np.abs(df_ik['Z_k'].to_numpy())
theta_array = np.arccos(np.divide(Z_k_abs_array,
    ↪ r_ik_array))
# phi
X_k_array = df_ik['X_k'].to_numpy()
```

```
    Y_k_array = df_ik['Y_k'].to_numpy()
    phi_array = np.arctan(np.divide(Y_k_array, X_k_array))
    # convert angle to positive value between [0,2pi]
    phi_array_convert = np.mod(phi_array, 2 * np.pi)
    for angle in phi_array_convert:
        if (angle >= 2 * np.pi) and (angle < 0):
            raise ValueError('phi angle in between 0 and 2pi
               ↪ ')
    # replace NaN with 0: (code will have error for
       ↪ invalid value center atom values 0/0)_
    df_ik['theta_0'] = theta_0_array
    df_ik['theta_0'] = df_ik['theta_0'].replace(np.nan, 0)
    df_ik['theta'] = theta_array
    df_ik['theta'] = df_ik['theta'].replace(np.nan, 0)
    df_ik['phi'] = phi_array_convert
    df_ik['phi'] = df_ik['phi'].replace(np.nan, 0)
    # array for weight coefficient w.r.t to atom type
    w_ik_arr = np.full((r_ik_array.shape), 1)
    # delta function delta=1 if i and k has the same
       ↪ element type, if not delta =0
    delta = np.full((r_ik_array.shape), 0)
    delta_arr = np.where(df_ik['atom_type'] == df_ik['
       ↪ atom_type'].iloc[0], 1, delta)
    df_ik['w_ik'] = w_ik_arr
    df_ik['delta'] = delta_arr
# save the DataFrame as a JSON file
    neighbor_list_path = output_dir + '-atom-' + str(
       ↪ center_atom_id) + '-neighbor-list.json'
    df_ik.to_json(neighbor_list_path, orient='records')
    """
    This function is to read in the neighbor list file
    """
    with open(neighbor_list_path, 'r') as f:
        json_data = json.load(f)

        # create a list of dictionaries from the JSON data
        data_list = [dict(row) for row in json_data]
```

```
            # extract data from the list of dictionaries
            r_ik_array = np.array([float(row['r_ik']) for row in
               ↪  data_list])
            theta_0_array = np.array([float(row['theta_0']) for
               ↪ row in data_list])
            theta_array = np.array([float(row['theta']) for row
               ↪ in data_list])
            phi_array = np.array([float(row['phi']) for row in
               ↪ data_list])
            w_ik_array = np.array([float(row['w_ik']) for row in
               ↪  data_list])
            delta_array = np.array([float(row['delta']) for row
               ↪ in data_list])
            r_cut_array = np.full((r_ik_array.shape), R_cut)
            # return a dictionary with the extracted data
    else:
        raise ValueError(f"Unsupported file type: {file_type}")
    return {
            'r_ik': r_ik_array,
            'theta_0': theta_0_array,
            'theta': theta_array,
            'phi': phi_array,
            'w_ik': w_ik_array,
            'delta': delta_array,
            'r_cut': r_cut_array
        }
def generate_m_values(j, j1, j2):
    """
    This function generates (m1, m2, m, m1p, m2p, mp) from
       ↪ input set (j1,j2,j)
    and only keeps sets that satisfy the conditions.
    Return: a one and only unique set (m1, m2, m, m1p, m2p, mp
       ↪ )
    """
    J=j1 + j2 + j
    #Condition 1
    if not ((abs(j1 - j2) <= j <= j1 + j2) or isinstance(J,int)
       ↪  or (J >=0)):
```

```
    raise ValueError("Condition (1) is not satisfied.")
#Condition 2
if not ((isinstance(val,int) or val >=0 for val in [j + j1
    ↪ - j2,j - j1 + j2,j1 + j2 - j])):
    raise ValueError("(+12) and (1+2) and (1+2) and is non-
        ↪ negative integer")
#Condition 3
if j1 > J or j2 > J or j > J:
    raise ValueError("1,2, not exceed a positive integer =1
        ↪ +2+")
# Generate m values
m1_vals = np.linspace(-j1, j1, int(2 * j1 + 1))
m2_vals = np.linspace(-j2, j2, int(2 * j2 + 1))
m_vals = np.linspace(-j, j, int(2 * j + 1))
mp_vals = m_vals.copy()
m1p_vals = m1_vals.copy()
m2p_vals = m2_vals.copy()
m1, m2, m, m1p, m2p, mp = np.meshgrid(m1_vals, m2_vals,
    ↪ m_vals, m1p_vals, m2p_vals, mp_vals)
s = np.stack((m1.ravel(), m2.ravel(), m.ravel(), m1p.ravel
    ↪ (), m2p.ravel(), mp.ravel()), axis=1)
full_list = s
keep_list = []
for i in range(len(s)):
    m1_val, m2_val, m_val, m1p_val, m2p_val, mp_val = s[i]
    #Condition 4-8: Clebsch-Gordan calc for set (1,2,,1,2,
        ↪ ), (1,2,,1p,2p,p)
    c4 = [isinstance(val, (int, float)) and val % 0.5 == 0
        ↪ for val in [m1_val, m2_val, m_val, m1p_val,
        ↪ m2p_val, mp_val]]
    c5 = [isinstance(vals, (int,float)) and vals >= 0 for
        ↪ vals in [j1 + m1_val, j1 - m1_val, j2 + m2_val,
        ↪ j2 - m2_val, j + m_val, j - m_val]]
    c5_p = [isinstance(vals, (int, float)) and vals >= 0
        ↪ for vals in [j1 + m1p_val, j1 - m1p_val, j2 +
        ↪ m2p_val, j2 - m2p_val, j + mp_val, j - mp_val]]
    c6 = [m1_val+m2_val == m_val and m1p_val+m2p_val ==
        ↪ mp_val]
```

```
            c7 = [abs(val) <= limit for val, limit in [(m1_val, j1)
               ↪ , (m2_val, j2), (m_val, j), (m1p_val, j1), (
               ↪ m2p_val, j2), (mp_val, j)]]
            #c8 = [j2 + j + m1_val >= 0 and j1 - j2 - m_val >= 0
               ↪ and isinstance(j2 + j + m1_val, int) and
               ↪ isinstance(j1 - j2 - m_val, int)]
            #c8_p = [j2 + j + m1p_val >= 0 and j1 - j2 - mp_val >=
               ↪  0 and isinstance(j2 + j + m1p_val, int) and
               ↪ isinstance(j1 - j2 - mp_val, int)]
            #Condition 9: Wigner-D calc
            #c9 = [isinstance(vals, (int, float)) and vals >= 0
               ↪ for vals in [mp_val-m_val, m1p_val-m1_val,
               ↪ m2p_val-m2_val]]
            if (all(c4) and all(c5) and all(c5_p) and all(c6) and
               ↪ all(c7)):
                keep_list.append(s[i])
            else:
                pass
        return keep_list, full_list


def test_neighbor(center_atom, neighbor_atoms, R_cut):
    """
    This function tests the parameters calculation for the
       ↪ neighbor atoms are within the cutoff radius of
    the center atom.
    """
    if neighbor_atoms is None:
        neighbor_atoms = [center_atom]

    if len(neighbor_atoms) == 1:
        x_k = np.array([neighbor_atoms[0][0] - center_atom[0]])
        y_k = np.array([neighbor_atoms[0][1] - center_atom[1]])
        z_k = np.array([neighbor_atoms[0][2] - center_atom[2]])
    else:
        x_k = np.array([neighbor[0] - center_atom[0] for
           ↪ neighbor in neighbor_atoms])
```

```
        y_k = np.array([neighbor[1] - center_atom[1] for
            ↪ neighbor in neighbor_atoms])
        z_k = np.array([neighbor[2] - center_atom[2] for
            ↪ neighbor in neighbor_atoms])
    # Calculate r_ik
    r_ik_array = np.sqrt(x_k ** 2 + y_k ** 2 + z_k ** 2)
    r_0_array = np.full(r_ik_array.shape, R_cut)
    # Calculate angles
    theta_0_array = np.pi * (r_ik_array / r_0_array)
    z_k_abs_array = np.abs(z_k)
    theta_array = np.arccos(z_k_abs_array / r_ik_array)
    phi_array = np.arctan2(y_k, x_k)
    phi_array_convert = np.mod(phi_array, 2 * np.pi)
    # Convert NaN values to zero
    theta_0_array = np.nan_to_num(theta_0_array, nan=0.0)
    theta_array = np.nan_to_num(theta_array, nan=0.0)
    phi_array = np.nan_to_num(phi_array, nan=0.0)
    phi_array_convert = np.nan_to_num(phi_array_convert, nan
        ↪ =0.0)
    # Create dictionary data
    params = {
        'w_ik': np.full(r_ik_array.shape, 1),
        'delta': np.full(r_ik_array.shape, 1),
        'r_ik': r_ik_array,
        'r_cut': np.full(r_ik_array.shape, R_cut),
        'theta_0': theta_0_array,
        'theta': theta_array,
        'phi': phi_array_convert
    }
    return params
```

Listing A.2: ClebschGordan

```
import numpy as np
import cmath

fact_cache = {}

def fact(n):
```

```
    if n < 0 or not np.isclose(n, int(n)):
        raise ValueError("Invalid input parameter: n must be a
            ↪ non-negative integer.")
    if n in fact_cache:
        return fact_cache[n]
    result = np.math.factorial(int(n))
    fact_cache[n] = result
    return result
class Clebsch_Gordan:
    """
    Definition:
    A Clebsch-Gordan coefficients are vector addition
        ↪ coefficients. They play an important role in
        ↪ decomposition of
    reducible representations of rotation. Let j1 and j2 with
        ↪ projections on m1 and m2 on the quantization axis.
    The coefficients represent the probability amplitude that
        ↪ j1 and j2 are coupled into a resultant angular
        ↪ momentum
    j with projection m.
    Args:
        j1 (scalar): angular momentum
        j2 (scalar): angular momentum
        j (scalar): angular momentum
        m1 (scalar): eigenvalue of angular momentum
        m2 (scalar): eigenvalue of angular momentum
        m (scalar): eigenvalue of angular momentum
    Returns: numbers
    ==========================Reference
        ↪ ==================================
    [5] Chapter 8 D.A. Varshalovich, A.N. Moskalev, V.K
        ↪ Khersonskii,
        Quantum Theory of Angular Momentum (1988)
    [12] Chapter 3 Biedenharn, L., & Louck, J.D. ,
        Encyclopedia of Mathematics and its Applications
            ↪ (1981)
    """
    def __init__(self, j1, j2, j, m1, m2, m):
```

```
self.j1 = j1
self.j2 = j2
self.j = j
self.m1 = m1
self.m2 = m2
self.m = m
self.J = j1 + j2 + j
#Conditions
J = j1 + j2 + j
# Condition 1
if not ((abs(j1 - j2) <= j <= j1 + j2) or isinstance(J,
   ↪  int) or (J >= 0)):
    raise ValueError("|12| 1+2 and 1+2+ are non-negative
       ↪  integer and 1+2=")
# Condition 2
if not ((isinstance(val, int) or val >= 0 for val in [j
   ↪  + j1 - j2, j - j1 + j2, j1 + j2 - j])):
    raise ValueError("(+12) and (1+2) and (1+2) and is
       ↪ non-negative integer")
# Condition 3
if j1 > J or j2 > J or j > J:
    raise ValueError("1,2, not exceed a positive integer
       ↪  =1+2+")
# Condition 4
if not (isinstance(val, (int, float)) and val % 0.5 ==
   ↪ 0 for val in [m1, m2, m]):
    raise ValueError("1,2, must be integer or half-
       ↪ integer (positive or negative) numbers")
# Condition 5
if not (isinstance(vals,(int, float)) and vals >= 0 for
   ↪  vals in [j1 + m1, j1 - m1, j2 + m2, j2 - m2, j +
   ↪  m, j - m]):
    raise ValueError("1+1,11, 2+2,22 +, are non-negative
       ↪  integer")
# Condition 6
#if not m1 + m2 == m:
    #raise ValueError("1+2= and 1+2=")
# Condition 7
```

```
    if not (abs(val) <= limit for val, limit in [(m1, j1),
       ↪ (m2, j2), (m, j)]):
       raise ValueError("|1|1, |2|2, ||")
    # Condition 8
    #if not (j2 + j + m1 > 0 and j1 - j2 - m > 0 and
       ↪ isinstance(j2 + j + m1, int) and isinstance(j1 -
       ↪  j2 - m, int)):
       #raise ValueError("2++10 and 120 and 2++1 and 12
          ↪ are non-negative integer")
def cg(self):
    if self.m1 + self.m2 != self.m:
       return 0.0 # delta function fails
    prefactor = cmath.sqrt((2 * self.j + 1) * fact(self.j +
       ↪  self.j1 - self.j2) * fact(self.j - self.j1+self.
       ↪ j2) \
                        * fact(self.j1 + self.j2 - self.j)
                           ↪ / fact(self.j + self.j1 +
                           ↪ self.j2 + 1))
    coefficient = cmath.sqrt(fact(self.j + self.m) * fact(
       ↪ self.j - self.m) / (fact(self.j1 + self.m1) \
                        * fact(self.j1 - self.m1) * fact(
                           ↪ self.j2 + self.m2) * fact(
                           ↪ self.j2 - self.m2)))
    sum = 0.0
    smin= max(0, int(self.m1-self.j1),int(self.j2-self.j1+
       ↪ self.m))
    smax= min(int(self.j2+self.j+self.m1),int(self.j-self.
       ↪ j1+self.j2),\
             int(self.j+self.m))

    for s in range(smin,smax+1):
       den = fact(s) * fact(self.j - self.j1 + self.j2 - s)
          ↪  * fact(self.j + self.m - s) \
            * fact(self.j1 - self.j2 - self.m + s)
       num = ((-1) ** (self.j2 + self.m2 + s))* fact(self.
          ↪ j2 + self.j + self.m1 - s) * fact(self.j1 -
          ↪ self.m1 + s)
       sum += num / den
```

```
        cg = prefactor * coefficient * sum
        return cg

def H_coeff(j1,j2,j,m1,m2,m,m1p,m2p,mp):
    '''
    This function calculate coupling coefficient H via
       ↪ computing
    the Clebsch-Gordan coefficient for cg(j1,m1,j2,m2,j,m)
    and cg(j1,m1p,j2,m2p,j,mp)
    Parameters:
        j1: angular momentum 1
        j2: angular momentum 2
        j: total angular momentum (j1+j2)
        m1: eigenvalue of angular momentum j1
        m2: eigenvalue of angular momentum j2
        m: eigenvalue of angular momentum j
        m1p: eigenvalue of j1 along rotated axis
        m2p: eigenvalue of j2 along rotated axis
        mp: eigenvalue of j along rotated axis
    Returns: Coupling coefficient H(j1,j2,j,m1,m2,m.m1p,m2p,mp
       ↪ )
    ======================Reference========================
    [1] Thompson, Swiler, Trott, Foiles, Tucker,
        Spectral neighbor analysis method for automated
           ↪ generation of quantum-accurate interatomic
           ↪ potentials (2015)
    [4] Meremianin,
        Multipole expansions in four-dimensional
           ↪ hyperspherical harmonics (2006
    [5] Chapter 8 D.A. Varshalovich, A.N. Moskalev, V.K
       ↪ Khersonskii,
        Quantum Theory of Angular Momentum (1988)
    '''
    CG = Clebsch_Gordan(j1,j2,j,m1,m2,m)
    cg = CG.cg()
    CGp = Clebsch_Gordan(j1,j2,j,m1p,m2p,mp)
    cg_p = CGp.cg()
    H = (cg)*(cg_p)
```

```
    return H
```

Listing A.3: WignerD

```
"""
Function to calculate the Wigner D function, Rotational
   ↪ Matrix U, and expansion coefficients density function u
"""
import numpy as np
fact_cache = {}

def fact(n):
    if n < 0 or not np.isclose(n, int(n)):
        raise ValueError("Invalid input parameter: n must be a
           ↪ non-negative integer.")
    if n in fact_cache:
        return fact_cache[n]
    result = np.math.factorial(int(n))
    fact_cache[n] = result
    return result
class Wigner_D:
    """
    Args:
        j (scalar): angular momentum
        m (scalar): eigenvalue of angular momentum
        mp (scalar): eigenvalue of j along rotated axis
        theta_0 (scalar): first angle of rotation [0, pi]
        theta (scalar): second angle of rotation [0, pi]
        phi (scalar): third angle of rotation [0, 2*pi]
    Returns: complex number, Wigner D function
    ==========================Reference
       ↪ ================================
    [5] Chapter 4.3-(p.76,eq.1) D.A. Varshalovich, A.N.
       ↪ Moskalev, V.K Khersonskii,
        Quantum Theory of Angular Momentum (1988)
    """
    def __init__(self, j, m, mp, theta_0, theta, phi):
        #Conditions for j, m, mp, theta_0, theta, phi
```

```
    #+, , +,  are non-negative integers
    if not (isinstance(vals, int) and vals >= 0 for vals in
       ↪  [j + m, j - m, j + mp, j - mp]):
        raise ValueError("Invalid input parameters: j+m, j-m
           ↪ , j+m', and j-m' must be non-negative integers
           ↪ ")
    #//, //
    if not ([abs(val) <= limit for val, limit in [(m, j), (
       ↪ mp, j)]]):
      raise ValueError("Invalid input parameters: |m| and |
         ↪ mp| must be less than or equal to j")
    if not ((j >= 0) or ((j % 1 == 0.0) or (j % 1 == 0.5)))
       ↪ :
      raise ValueError("Invalid input parameters: j must be
         ↪ a non-negative integer or half-integer")
    if theta_0 < 0 or theta_0 > 2*np.pi or abs(theta) > np.
       ↪ pi or abs(phi) > 2 * np.pi:
      raise ValueError("Invalid input parameters: theta_0,
         ↪ theta, and phi must be within [0, pi] and [0, 2
         ↪ pi], respectively.")
    self.j = j
    self.m = m
    self.mp = mp
    self.theta_0 = theta_0
    self.theta = theta
    self.phi = phi
def compute_dsmall(self):
    """
    This method is used to calculate the Wigner d small-
       ↪ real function involving trigonometric functions
    ==========================Reference
       ↪ ==================================
    [5] Chapter 4.3.1-(p.76,eq.4) D.A. Varshalovich, A.N.
       ↪ Moskalev, V.K Khersonskii,
    Returns: Wigner d - real function
    """
    kmax = max(0, self.m - self.mp)
    kmin = min(self.j + self.m, self.j - self.mp)
```

```
        term1 = np.sqrt(fact(self.j + self.m) * fact(self.j -
            ↪ self.m) * fact(self.j + self.mp) * fact(self.j -
            ↪ self.mp))
        sum = 0
        for k in range(int(kmax), int(kmin) + 1):
            numerator = (-1) ** k * (np.cos(self.theta / 2)) **
                ↪ (2 * self.j - 2 * k + self.m - self.mp) * \
                        (np.sin(self.theta / 2)) ** (2 * k - self
                            ↪ .m + self.mp)
            denominator = fact(k) * fact(self.j + self.m - k) *
                ↪ fact(self.j - self.mp - k) * fact(self.mp -
                ↪ self.m + k)
            sum += numerator / denominator
        return sum*term1
    def wigner_D(self):
        #term1 = np.exp(-1j * self.m * self.theta_0)
        term1 = np.cos(self.m * self.theta_0) - 1j*(np.sin(self
            ↪ .m * self.theta_0))
        term2 = self.compute_dsmall()
        #term3 = np.exp(-1j * self.mp * self.phi)
        term3 = np.cos(self.mp * self.phi) -1j*(np.sin(self.mp
            ↪ * self.phi))

        result = term1 * term2 * term3
        return result
def U_rot(j, m, mp, theta_0, theta, phi):
    '''
    Parameters:
      j: integer/half integer number
         angular momentum
      m: integer/half integer number
         Eigenvalue of angular momentum along rotated axis
      mp: eigenvalue of j along rotated axis
      mpp:
      theta_0: fist angle of rotation [0,pi]
      theta: second angle of rotation [0,pi]
      phi: third angle of rotation [0,2pi]
          rotational od a coordinate system through an angle
```

```
            ↪ theta_0
        about an axis n(theta,phi)
      Returns: Rotational matrix U
      ==========================Reference
         ↪ ================================
      [5] Chapter 4.5.2 (a) Eq. (3) D.A. Varshalovich, A.N.
         ↪ Moskalev, V.K Khersonskii,
             Quantum Theory of Angular Momentum (1988)
    '''
    mpp_vals = np.linspace(-j, j, int(2*j+1))
    U = 0
    for mpp in mpp_vals:
        WD_1 = Wigner_D(j, m, mpp, phi, theta,-phi)
        term1 = WD_1.wigner_D()
        term2 = np.cos(mpp * theta_0) - 1j*(np.sin(mpp *
           ↪ theta_0))
        WD_2 = Wigner_D(j, mpp, mp, phi, -theta, -phi)
        term3 = WD_2.wigner_D()
        Um_mp = term1 * term2 * term3
        U += Um_mp
    return U
def u(j, m, mp, params):
    '''
    Parameter:
        j (scalar): angular momentum
        m (scalar): eigenvalue of angular momentum
        mp (scalar): eigenvalue of j along rotated axis
        params (dict): a dictionary containing the following
           ↪ keys, its values:
            - w_ik (array): the coefficients that are
               ↪ dimensionless weights that are chosen to
               ↪ distinguish atoms
              of different types, while the central atom is
                 ↪ arbitrarily assigned a unit weight,
                 ↪ dimensin (1,k)
            - delta (array): the Dirac delta function,
               ↪ indicates only neighbor atom of element the
               ↪ same as center atom
```

```
        contribute to partial density, dimension (1,k)
      - r_ik (array): distance from center atom to
        ↪ neighbor atom, dimension (1,k), k is number
        ↪ of neighbor atoms
        in cutoff radius, array exclude center atom as
          ↪ well
      - r_cut (array): cutoff radius
      - theta_0: array for theta_0 angel (fist angle of
        ↪ rotation [0,pi])
        of neighbor atoms in reference frame of center
          ↪ atom, dimension (k+1,)
      - theta: array for theta angel ( second angle of
        ↪ rotation [0,pi])
        of neighbor atoms in reference frame of center
          ↪ atom, dimension (k+1,)
      - phi: array for phi angel (third angle of rotation
        ↪  [0,2pi])
        of neighbor atoms in reference frame of center
          ↪ atom, dimension (k+1,)
Returns: expansion coefficients density function u_jm_mp
'''
w_ik_array = params['w_ik']
delta_array = params['delta']
r_ik_array = params['r_ik']
r_cut_array = params['r_cut']
theta_0_array = params['theta_0']
theta_array = params['theta']
phi_array = params['phi']
# Calculate cutoff_function
mask = r_ik_array >= r_cut_array
# Set elements of f_cut_arr to 0 where the mask is True
f_cut_arr = np.where(mask, 0, (1 / 2) * (np.cos(np.pi * (np
   ↪ .divide(r_ik_array, r_cut_array))) + 1))
# Calculate rotational matrix U for all k=n neighbor atoms
U_ik_array = np.array([U_rot(j, m, mp, theta_0, theta, phi)
   ↪  for theta_0, theta, phi in zip(theta_0_array,
   ↪ theta_array, phi_array)], dtype='complex')
# Compute u_jmmp
```

```
    u_jmmp = np.dot((f_cut_arr * U_ik_array), (w_ik_array *
       ↪ delta_array))
    return u_jmmp
```

Listing A.4: SO4

```
import numpy as np
import json
import pandas as pd
from Bio.PDB.MMCIF2Dict import MMCIF2Dict
from typing import Dict

fact_cache = {}
def fact(n):
    if n < 0 or not np.isclose(n, int(n)):
        raise ValueError("Invalid input parameter: n must be a
           ↪ non-negative integer.")
    if n in fact_cache:
        return fact_cache[n]
    result = np.math.factorial(int(n))
    fact_cache[n] = result
    return result
class Bispectrum:
    """
    Calculate bispectrum- S(0)4 components
    """
    def __init__(self, j, j1, j2 , params: Dict[str, np.ndarray
       ↪ ]):
        '''
            j: j index
            j1: j1 index
            j2: j2 index
            input_data: input data dictionary with extracted
               ↪ values:
            r_ik (array): dictance from center atom to n
               ↪ neighbor atom, dim = [n,]
            theta_0 (array): first angle of rotation [0, pi] ,
               ↪ dim = [n,]
            theta (array): second angle of rotation [0, pi],
```

```
            ↪ dim = [n,]
        phi (array): third angle of rotation [0, 2pi], dim
            ↪ = [n,]
        w_ik (array): weight coefficient, dim = [n,]
        delta (array): delta function, dim = [n,]
        r_cut (array): cutoff distance, dim = [n,]
    '''
    self.j = j
    self.j1 = j1
    self.j2 = j2
    self.params = params
    w_ik_array = self.params['w_ik']
    delta_array = self.params['delta']
    r_ik_array = self.params['r_ik']
    r_cut_array = self.params['r_cut']
    theta_0_array = self.params['theta_0']
    theta_array = self.params['theta']
    phi_array = self.params['phi']
    #Condition check
    if not (abs(j1 - j2) <= j <= j1 + j2 and j1 + j2 - j %
        ↪ 1 != 0.5):
        raise ValueError("Invalid input parameters: j1, j2,
            ↪ j must satisfy the triangle inequality.\ "
                         "j1+j2-j must not be a half-integer")
    J = (j1 + j2 + j)
    if J < (int(j1 + j2 + j)) and J < 0:
        raise ValueError("Invalid input parameters: j1, j2,
            ↪ j must not exceed a positive integer J")

@staticmethod
def generate_m_values(j, j1, j2):
    """
    This function generates (m1, m2, m, m1p, m2p, mp) from
        ↪  input set (j1,j2,j)
    and only keeps sets that satisfy the conditions.
    Return: a one and only unique set (m1, m2, m, m1p, m2p
        ↪ , mp)
    """
```

```
J = j1 + j2 + j
# Condition 1
if not ((abs(j1 - j2) <= j <= j1 + j2) or isinstance(J,
   ↪  int) or (J >= 0)):
    raise ValueError("Condition (1) is not satisfied.")
# Condition 2
if not ((isinstance(val, int) or val >= 0 for val in [j
   ↪  + j1 - j2, j - j1 + j2, j1 + j2 - j])):
    raise ValueError("(+12) and (1+2) and (1+2) and is
       ↪ non-negative integer")
# Condition 3
if j1 > J or j2 > J or j > J:
    raise ValueError("1,2, not exceed a positive integer
       ↪  =1+2+")
# Generate m values
m1_vals = np.linspace(-j1, j1, int(2 * j1 + 1))
m2_vals = np.linspace(-j2, j2, int(2 * j2 + 1))
m_vals = np.linspace(-j, j, int(2 * j + 1))
mp_vals = m_vals.copy()
m1p_vals = m1_vals.copy()
m2p_vals = m2_vals.copy()
m1, m2, m, m1p, m2p, mp = np.meshgrid(m1_vals, m2_vals,
   ↪  m_vals, m1p_vals, m2p_vals, mp_vals)
s = np.stack((m1.ravel(), m2.ravel(), m.ravel(), m1p.
   ↪ ravel(), m2p.ravel(), mp.ravel()), axis=1)
keep_list = []
full_list = []
for i in range(len(s)):
    full_list.append(s[i])
    m1_val, m2_val, m_val, m1p_val, m2p_val, mp_val = s[
       ↪ i]
    # Condition 4-8: Clebsch-Gordan calc for set (1,2,,
       ↪ 1,2,), (1,2,,1p,2p,p)
    c4 = [isinstance(val, (int, float)) and val % 0.5 ==
       ↪  0 for val in
          [m1_val, m2_val, m_val, m1p_val, m2p_val,
             ↪ mp_val]]
    c5 = [isinstance(vals, (int, float)) and vals >= 0
```

```
            ↪ for vals in
              [j1 + m1_val, j1 - m1_val, j2 + m2_val, j2 -
                 ↪ m2_val, j + m_val, j - m_val]]
        c5_p = [isinstance(vals, (int, float)) and vals >= 0
            ↪  for vals in
               [j1 + m1p_val, j1 - m1p_val, j2 + m2p_val, j2
                  ↪  - m2p_val, j + mp_val, j - mp_val]]
        c6 = [m1_val + m2_val == m_val and m1p_val + m2p_val
            ↪  == mp_val]
        c7 = [abs(val) <= limit for val, limit in
              [(m1_val, j1), (m2_val, j2), (m_val, j), (
                 ↪ m1p_val, j1), (m2p_val, j2), (mp_val, j)
                 ↪ ]]
        #c8 = [j2 + j + m1_val >= 0 and j1 - j2 - m_val >=
            ↪ 0 and isinstance(j2 + j + m1_val, int) and
            ↪ isinstance(
            #j1 - j2 - m_val, int)]
        #c8_p = [
            #j2 + j + m1p_val >= 0 and j1 - j2 - mp_val >=
               ↪ 0 and isinstance(j2 + j + m1p_val, int)
               ↪ and isinstance(
               #j1 - j2 - mp_val, int)]
        # Condition 9: Wigner-D calc
        #c9 = [isinstance(vals, (int, float)) and vals >= 0
            ↪  for vals in [mp_val - m_val, m1p_val -
            ↪ m1_val, m2p_val - m2_val]]
        if all(c4) and all(c5) and all(c5_p) and all(c6) and
            ↪  all(c7):
            keep_list.append(s[i])
        else:
            pass
    return keep_list, full_list
#Clebsch-Gordan Coefficient
@staticmethod
def clebsch_gordan(j1, j2, j, m1, m2, m):
    """
    Definition:
        A Clebsch-Gordan coefficients are vector addition
```

```
        ↪ coefficients. They play an important role in
        ↪ decomposition of
    reducible representations of rotation. Let j1 and
        ↪ j2 with projections on m1 and m2 on the
        ↪ quantization axis.
    The coefficients represent the probability
        ↪ amplitude that j1 and j2 are coupled into a
        ↪ resultant angular momentum
    j with projection m.
Args:
    j, j1, j2 (scalar): angular momentum
    m, m1, m2 (scalar): eigenvalue of angular momentum
        ↪ j, j1, j2 respectively
    mp, mp1, mp2 (scalar): eigenvalue of j, j1, j2
        ↪ along rotated axis respectively
Returns: Clebsch-Gordan coefficients, real number
==========Reference
    ↪ ==================================
[5] Chapter 8 D.A. Varshalovich, A.N. Moskalev, V.K
    ↪ Khersonskii,
        Quantum Theory of Angular Momentum (1988)
[12] Chapter 3 Biedenharn, L., & Louck, J.D. ,
        Encyclopedia of Mathematics and its
            ↪ Applications (1981)
"""
if m1 + m2 != m:
    return 0.0 # delta function fails
prefactor = np.sqrt((2 * j + 1) * fact(j + j1 - j2) *
    ↪ fact(j - j1 + j2) * fact(j1 + j2 - j) / fact(j +
    ↪ j1 + j2 + 1))
coefficient = np.sqrt(fact(j + m) * fact(j - m) / (fact
    ↪ (j1 + m1) * fact(j1 - m1) * fact(j2 + m2) * fact(
    ↪ j2 - m2)))
sum = 0.0
smin = max(0, int(m1 - j1), int(j2 - j1 + m))
smax = min(int(j2 + j + m1), int(j - j1 + j2), int(j +m
    ↪ ))
for s in range(smin, smax + 1):
```

```
        den = fact(s) * fact(j - j1 + j2 - s) * fact(j + m -
            ↪  s) * fact(j1 - j2 - m + s)
        num = ((-1) ** (j2 + m2 + s)) * fact(j2 + j + m1 - s
            ↪ ) * fact(j1 - m1 + s)
        sum += num / den
    cg = prefactor * coefficient * sum
    return cg
#Coupling Coefficient
@classmethod
def H(cls, j1, j2, j, m1, m2, m, m1p, m2p, mp):
    CG = cls.clebsch_gordan(j1, j2, j, m1, m2, m)
    CGp = cls.clebsch_gordan(j1, j2, j, m1p, m2p, mp)
    H_coeff = CG * CGp
    return H_coeff
@staticmethod
def compute_dsmall(j, m, mp, theta):
    """
    This method is used to calculate the Wigner d small-
        ↪ real function involving trigonometric functions
    Returns: Wigner d - real function
    ========================Reference
        ↪ ================================
    [5] Chapter 4.3.1-(p.76,eq.4) D.A. Varshalovich, A.N.
        ↪ Moskalev, V.K Khersonskii,
    """
    kmax = max(0,m - mp)
    kmin = min(j + m, j - mp)
    term1 = np.sqrt(fact(j + m) * fact(j - m) * fact(j + mp
        ↪ ) * fact(j - mp))
    sum = 0
    for k in range(int(kmax), int(kmin) + 1):
        numerator = (-1) ** k * (np.cos(theta / 2)) ** (2 *
            ↪ j - 2 * k + m - mp) * \
                  (np.sin(theta / 2)) ** (2 * k - m + mp)
        denominator = fact(k) * fact(j + m - k) * fact(j -
            ↪ mp - k) * fact(mp - m + k)
        sum += numerator / denominator
    return sum*term1
```

```
@classmethod
def wigner_D(cls, j, m, mp, theta_0, theta, phi):
    """
    This method is used to calculate the Wigner D matrix
    Args:
        theta_0 (scalar): first angle of rotation [0, pi]
        theta (scalar): second angle of rotation [0, pi]
        phi (scalar): third angle of rotation [0, 2*pi]
    Returns: complex number, Wigner D function
    ==========================Reference
       ↪ ================================
    [5] Chapter 4.3-(p.76,eq.1) D.A. Varshalovich, A.N.
       ↪ Moskalev, V.K Khersonskii,
    """
    term1 = np.cos(m *theta_0) - 1j*(np.sin(m * theta_0))
    term2 = cls.compute_dsmall(j, m, mp, theta)
    term3 = np.cos(mp * phi) -1j*(np.sin(mp * phi))
    result = term1 * term2 * term3
    return result
@classmethod
def U_rot(cls, j, m, mp, theta_0, theta, phi):
    """
    This method is used to calculate the rotation matrix U
    Note: set (phi,-theta,-phi), (phi, theta, -phi) need
       ↪ to convert to positive angel set to satisfy the
       ↪ Wigner D function angle range
         D(alpha,beta,gamma), 0<=alpha<=2pi, 0<=beta<=pi,
            ↪ 0<=gamma<=2pi
    Returns: complex number, Rotational matrix U function
    ==========================Reference
       ↪ ================================
    [5] Chapter 4 D.A. Varshalovich, A.N. Moskalev, V.K
       ↪ Khersonskii,
            Quantum Theory of Angular Momentum (1988)
    """
    mpp_vals = np.linspace(-j, j, int(2 * j + 1))
    U = 0
    for mpp in mpp_vals:
```

```
            term1 = cls.wigner_D(j, m, mpp, phi, theta, -phi)
            term2 = np.cos(mpp * theta_0) - 1j * (np.sin(mpp *
                ↪ theta_0))
            term3 = cls.wigner_D(j, mpp, mp, phi, -theta, -phi)
            Um_mp = term1 * term2 * term3
            U += Um_mp
        return U
    @classmethod
    def u_small(cls, j, m, mp, params):
        """
        Args:
            j (scalar): angular momentum
            m (scalar): eigenvalue of angular momentum
            mp (scalar): eigenvalue of j along rotated axis
            params (dict): a dictionary containing the
                ↪ following keys, its values:
                - w_ik (array): the coefficients that are
                    ↪ dimensionless weights that are chosen to
                    ↪ distinguish atoms
                  of different types, while the central atom is
                      ↪ arbitrarily assigned a unit weight,
                      ↪ dimensin (1,k)
                - delta (array): the Dirac delta function,
                    ↪ indicates only neighbor atom of element
                    ↪ the same as center atom
                  contribute to partial density, dimension (1,k)
                - r_ik (array): distance from center atom to
                    ↪ neighbor atom, dimension (1,k), k is
                    ↪ number of neighbor atoms
                  in cutoff radius, array exclude center atom as
                      ↪  well (r_ik <= r_cut)
                - r_cut (array): cutoff radius
                - theta_0: array for theta_0 angel (fist angle
                    ↪ of rotation [0,pi])
                  of neighbor atoms in reference frame of center
                      ↪  atom, dimension (k+1,)
                - theta: array for theta angel ( second angle
                    ↪ of rotation [0,pi])
```

```
            of neighbor atoms in reference frame of center
               ↪ atom, dimension (k+1,)
            - phi: array for phi angel (third angle of
               ↪ rotation [0,2pi])
            of neighbor atoms in reference frame of center
               ↪ atom, dimension (k+1,)
    Returns: expansion coefficients density function
       ↪ u_jm_mp
    """
    w_ik_array = params['w_ik']
    delta_array = params['delta']
    r_ik_array = params['r_ik']
    r_cut_array = params['r_cut']
    theta_0_array = params['theta_0']
    theta_array = params['theta']
    phi_array = params['phi']

    # Calculate cutoff_function
    mask = r_ik_array >= r_cut_array
    # Set elements of f_cut_arr to 0 where the mask is
       ↪ True
    f_cut_arr = np.where(mask, 0, (1 / 2) * (np.cos(np.pi *
       ↪ (np.divide(r_ik_array, r_cut_array))) + 1))
    # Calculate rotational matrix U for all k=n neighbor
       ↪ atoms
    U_ik_array = np.array([cls.U_rot(j, m, mp, theta_0,
       ↪ theta, phi) for theta_0, theta, phi in
                           zip(theta_0_array, theta_array,
                              ↪ phi_array)], dtype='complex'
                              ↪ )
    # Compute u_jmmp
    u_jmmp = np.dot((f_cut_arr * U_ik_array), (w_ik_array *
       ↪ delta_array))
    return u_jmmp
@classmethod
def evaluate(cls, j, j1, j2, params):
    w_ik_array = params['w_ik']
    delta_array = params['delta']
```

```
r_ik_array = params['r_ik']
r_cut_array = params['r_cut']
theta_0_array = params['theta_0']
theta_array = params['theta']
phi_array = params['phi']
m_list, full_list = cls.generate_m_values(j,j1,j2)
B_total = 0
for i in m_list:
    m1, m2, m, m1p, m2p, mp = i[0], i[1], i[2], i[3], i
        ↪ [4], i[5]
    H_coeff = cls.H(j1, j2, j, m1, m2, m, m1p, m2p, mp)
    u_jmmp = cls.u_small(j, m, mp, params)
    u1_j1m1m1p = cls.u_small(j1, m1, m1p, params)
    u2_j2m2m2p = cls.u_small(j2, m2, m2p, params)
    B = np.conj(u_jmmp) * (H_coeff * u1_j1m1m1p *
        ↪ u2_j2m2m2p)
    B_total += B
return B_total
```

APPENDIX B

TESTING CODES

Testing key functions and case studies for neighbor atom configurations.

Listing B.1: `test_INPUT_param`

```
import numpy as np
from itertools import product
import pandas as pd
from bispectrum.methods.calc.INPUT_param import neighbor_params
center_atom_id = 17
r_mu =0.0779
R_cut= 0.25
input_file = "/Users/duonghoang/Documents/GitHub/
   ↪ bispectrum_component/data/avgBL-Model.cif"
output_dir = "/Users/duonghoang/Documents/GitHub/
   ↪ bispectrum_component/data"
file_type = 'cif'
input_data = neighbor_params(center_atom_id, r_mu, R_cut,
   ↪ input_file, output_dir, file_type='cif')
print (input_data)
# Convert dictionary to DataFrame
df = pd.DataFrame(input_data)
#Display DataFrame
print(df)
```

Listing B.2: `test_ClebschGordan_calc`

```
"""
Compare the Clebsch Gordan Coefficient calculation from class
   ↪  function (calc) vs SymPy
Compare the coupling coefficient calculation from class
   ↪ function (calc) vs SymPy
"""
from sympy.physics.quantum.cg import CG
from sympy import *
```

```
from bispectrum.methods.calc.ClebschGordan import
   ↪ Clebsch_Gordan, H_coeff
import timeit
from sympy.physics.quantum.cg import CG
#Function using Sympy
j1 = 2
m1 = 1
j2 = 1/2
m2 = 1/2
j = 5/2
m = 3/2
t0=timeit.default_timer()
cg = CG(j1,m1,j2,m2,j,m)
cg = cg.doit()
t1=timeit.default_timer()
print(N(cg))
print("Execution time for CG function from Sympy:", t1-t0, "
   ↪ seconds")

#Our function
t2=timeit.default_timer()
CG_calc = Clebsch_Gordan(j1,j2,j,m1,m2,m)
cg_calc = CG_calc.cg()
print (cg_calc)
t3=timeit.default_timer()
print("Execution time for CG function from Clebsch_Gordan:", t3
   ↪ -t2, "seconds")
print("Execution time for CG calculation using class method is"
   ↪ , round((t3-t2)/(t1-t0)), \
     "times faster than Sympy function")
```

Listing B.3: test_WignerD_calc

```
"""
Compare the Wigner D matrix calculation from class function
"""

import numpy as np
from bispectrum.methods.calc.WignerD import *
```

```
#from sympy import *
#from sympy.physics.quantum.spin import Rotation
import timeit
j, m, mp, theta_0, theta,phi= 3/2, 3/2, -3/2, np.pi/4, np.pi/3,
   ↪  np.pi/4

t0=timeit.default_timer()
#Wigner_D function from calc.WignerD
WD = Wigner_D(j, m, mp, theta_0, theta,phi)
wd = WD.wigner_D()
print ("Wigner_D calculation from our function", wd)
t1=timeit.default_timer()
print("Execution time for Wigner_D function from calc.WignerD:"
   ↪ , t1 - t0, "seconds")

#test D(phi,theta, -phi) D(theta,-theta, -phi)
WD_1 = Wigner_D(j, m, mp, phi, theta,-phi)
wd_1 = WD_1.wigner_D()
print ("Wigner_D term 1", wd_1)

WD_1p = Wigner_D(j, m, mp, phi, -theta,-phi)
wd_1p = WD_1p.wigner_D()
print ("Wigner_D term 3", wd_1p)

#test d_small
WD_1= Wigner_D(j, m, mp, theta_0, theta,phi)
d_small = WD_1.compute_dsmall()
print ("d_small from calc", d_small)
d_small_table = -(np.sin(theta/2))**3
print ("d_small from table", d_small)

#Case1 j=3/2, m=3/2, mp=-3/2
j, m, mp, theta_0, theta,phi= 3/2, 3/2, -3/2, np.pi/3, np.pi/2,
   ↪  np.pi/4

#Test rotaion matrix U_rot
U_calc_1 = U_rot(j, m, mp, theta_0, theta,phi)
U_table_1= 1j*((np.sin(theta_0/2)*np.sin(theta)*(np.exp(-1j*phi
```

```
   ↪ )))**3)
print("U_table_1", U_table_1)
print("U_calc_1", U_calc_1)
```

Listing B.4: test_SO4_calc

```
from bispectrum.methods.calc.INPUT_param import neighbor_params
from bispectrum.methods.calc.SO4 import *
import pandas as pd
from sympy import *
import timeit
from sympy.physics.quantum.cg import CG
import numpy as np
import timeit

t0=timeit.default_timer()
path = '/Users/duonghoang/Documents/GitHub/bispectrum_component
   ↪ //data/avgBL-Model.cif'
center_atom_id = 17
r_mu = 0.0779
R_cut = 0.25
input_file_path = path
output_directory = '/Users/duonghoang/Documents/GitHub/
   ↪ bispectrum_component/data'
#read data from file
data = neighbor_params(center_atom_id, r_mu, R_cut,
   ↪ input_file_path, output_directory, file_type='cif')
#Test bispectrum
j,j1,j2= 5/2,2,1/2
B = Bispectrum(j, j1, j2,data)
Bispectrum_SO4 = B.evaluate(j, j1, j2,data)
t1=timeit.default_timer()
print("Bispectrum coefficient for atom ID17", np.round(
   ↪ Bispectrum_SO4,2))
print("Execution time for bispectrum calculation", round(t1-t0
   ↪ ,2), "seconds")


t2=timeit.default_timer()
```

```
#test generate set+Clebsch-Gordan coefficients
keep_set, full_set = B.generate_m_values(j,j1,j2)
keep_set_arr = np.array(keep_set)
print ("Test generate set", keep_set_arr.shape)
print (pd.DataFrame(keep_set))
df = pd.DataFrame(keep_set, columns=["m1", "m2", "m", "m1p", "
   ↪ m2p", "mp"])
print(df)
unique_combinations = df[["m1", "m2", "m"]].drop_duplicates()
CG_unique =[]
for _, row in unique_combinations.iterrows():
    m, m1, m2 = row["m"], row["m1"], row["m2"]
    CB = B.clebsch_gordan(j1,j2,j,m1,m2,m)
    CG_unique.append(CB)
unique_combinations['CG'] = CG_unique
print(unique_combinations)
t3=timeit.default_timer()
print("Execution time for CB calculation")
print("for unique sets:", round(t3-t2,2), "seconds")


#Interchange j,j1,j2
j,j1,j2= 5/2,2,1/2
B1 = Bispectrum(j, j1, j2,data)
B_1 = B1.evaluate(j, j1, j2,data)
print("B1", np.round(B_1,2))
print(np.divide(B_1,2*j+1))

B2 = Bispectrum(j1, j2, j,data)
B_2 = B2.evaluate(j1, j2, j,data)
print("B2", np.round(B_2,2))
print(np.divide(B_2,2*j1+1))

B3 = Bispectrum(j2, j, j1,data)
B_3 = B3.evaluate(j2, j, j1,data)
print("B3", np.round(B_3,2))
print(np.divide(B_3,2*j2+1))
```

```
B4 = Bispectrum(j2, j1, j,data)
B_4 = B4.evaluate(j2, j1, j,data)
print("B4", np.round(B_4,2))
print(np.divide(B_4,2*j2+1))
```

Listing B.5: study1_SO4_calc.py

```
"""
This file is used to study the cases example of SO(4)
   ↪ bispectrum calculation.
"""
import numpy as np
import matplotlib.pyplot as plt
from bispectrum.methods.calc.SO4 import Bispectrum
from bispectrum.methods.calc.INPUT_param import test_neighbor
from bispectrum.methods.plot.neighbors import plot_atoms

#Parameters stay constant for all cases
center_atom = (0.5, 0.5, 0.5)
cell_length = 1.0
R_cut = 0.2
j,j1,j2= 5/2, 3/2, 1

#Case 1-Special case: 1 center atom in an empty cell

data_1 = {
       'w_ik': np.array([1]),
       'delta': np.array([0]),
       'r_ik': np.array([0]),
       'r_cut': np.array([0.2]),
       'theta_0': np.array([0]),
       'theta': np.array([0]),
       'phi': np.array([0])
   }
B_1= Bispectrum(j=5/2, j1=3/2, j2=2/2, params=data_1)
B_data_1 = B_1.evaluate(j=5/2, j1=3/2, j2=2/2, params=data_1)
print(B_data_1)
case_1_plot=save_path = "/Users/duonghoang/Documents/GitHub/
   ↪ bispectrum_component/plots/plot_case_1.png"
```

```
plot_atoms(center_atom, None, R_cut, cell_length, B_data_1,
    ↪ save_path=case_1_plot)

#Case 2-[Same cell at (1)] with 1 neighbor atom at (0.6, 0.5,
    ↪  0.5)
center_atom = [0.5, 0.5, 0.5]
neighbor_atoms_2 = [(0.6, 0.5, 0.5)]
data_2 = test_neighbor(center_atom, neighbor_atoms_2, R_cut)
B_2= Bispectrum(j=5/2, j1=3/2, j2=2/2, params=data_2)
B_data_2 = B_2.evaluate(j=5/2, j1=3/2, j2=2/2, params=data_2)
print(B_data_2)
case_2_plot=save_path = "/Users/duonghoang/Documents/GitHub/
    ↪ bispectrum_component/plots/plot_case_2.png"
plot_atoms(center_atom, neighbor_atoms_2, R_cut, cell_length, B
    ↪ = B_data_2, save_path=case_2_plot)

#Case 3-[Same cell at (1)] with 1 neighbor atoms at (0.5,
    ↪ 0.4, 0.5)
neighbor_atoms_3 = [(0.5,0.4,0.5)]
data_3 = test_neighbor(center_atom, neighbor_atoms_3, R_cut)
B_3= Bispectrum(j=5/2, j1=3/2, j2=2/2, params=data_3)
B_data_3 = B_3.evaluate(j=5/2, j1=3/2, j2=2/2, params=data_3)
print(B_data_3)
case_3_plot=save_path = "/Users/duonghoang/Documents/GitHub/
    ↪ bispectrum_component/plots/plot_case_3.png"
plot_atoms(center_atom, neighbor_atoms_3, R_cut, cell_length, B
    ↪ = B_data_3, save_path=case_3_plot)

#Case 4-[Same cell at (1)] with 1 neighbor atom at (0.4, 0.5,
    ↪  0.5)
neighbor_atoms_4 = [(0.4,0.5,0.5)]
data_4 = test_neighbor(center_atom, neighbor_atoms_4, R_cut)
B_4= Bispectrum(j=5/2, j1=3/2, j2=2/2, params=data_4)
B_data_4 = B_4.evaluate(j=5/2, j1=3/2, j2=2/2, params=data_4)
print(B_data_4)
case_4_plot=save_path = "/Users/duonghoang/Documents/GitHub/
    ↪ bispectrum_component/plots/plot_case_4.png"
plot_atoms(center_atom, neighbor_atoms_4, R_cut, cell_length, B
```

```
   ↪ = B_data_4, save_path=case_4_plot)

#Case 5-[Same cell at (1)] with 1 neighbor atom at (0.5, 0.6,
   ↪  0.5)
neighbor_atoms_5 = [(0.5,0.6,0.5)]
data_5 = test_neighbor(center_atom, neighbor_atoms_5, R_cut)
B_5= Bispectrum(j=5/2, j1=3/2, j2=2/2, params=data_5)
B_data_5 = B_5.evaluate(j=5/2, j1=3/2, j2=2/2, params=data_5)
print(B_data_5)
case_5_plot= "/Users/duonghoang/Documents/GitHub/
   ↪ bispectrum_component/plots/plot_case_5.png"
plot_atoms(center_atom, neighbor_atoms_5, R_cut, cell_length, B
   ↪ = B_data_5, save_path=case_5_plot)

#Case 6-[Same cell at (1)] 2 neighbor atoms combine case 2+3
neighbor_atoms_23 = [[0.6, 0.5, 0.5],[0.5,0.4,0.5]]
data_23 = test_neighbor(center_atom, neighbor_atoms_23, R_cut)
B_23= Bispectrum(j=5/2, j1=3/2, j2=2/2, params=data_23)
B_data_23 = B_23.evaluate(j=5/2, j1=3/2, j2=2/2, params=data_23
   ↪ )
print(B_data_23)
case_23_plot= "/Users/duonghoang/Documents/GitHub/
   ↪ bispectrum_component/plots/plot_case_23.png"
plot_atoms(center_atom, neighbor_atoms_23, R_cut, cell_length,
   ↪ B= B_data_23, save_path=case_23_plot)

#Case 7-[Same cell at (1)] 2 neighbor atoms combine case 4+5
neighbor_atoms_45 = [[0.4,0.5,0.5],[0.5,0.6,0.5]]
data_45 = test_neighbor(center_atom, neighbor_atoms_23, R_cut)
B_45= Bispectrum(j=5/2, j1=3/2, j2=2/2, params=data_45)
B_data_45 = B_5.evaluate(j=5/2, j1=3/2, j2=2/2, params=data_45)
print(B_data_45)
case_45_plot= "/Users/duonghoang/Documents/GitHub/
   ↪ bispectrum_component/plots/plot_case_45.png"
plot_atoms(center_atom, neighbor_atoms_45, R_cut, cell_length,
   ↪ B= B_data_45, save_path=case_45_plot)
```

```
#Case 8-[Same cell at (1)] 3 neighbor atoms combine case
    ↪ 2+3+4
neighbor_atoms_234 = [[0.6, 0.5, 0.5],[0.5,0.4,0.5],
    ↪ [0.4,0.5,0.5]]
data_234 = test_neighbor(center_atom, neighbor_atoms_234, R_cut
    ↪ )
B_234= Bispectrum(j=5/2, j1=3/2, j2=2/2, params=data_234)
B_data_234 = B_234.evaluate(j=5/2, j1=3/2, j2=2/2, params=
    ↪ data_234)
print(B_data_234)
case_234_plot= "/Users/duonghoang/Documents/GitHub/
    ↪ bispectrum_component/plots/plot_case_234.png"
plot_atoms(center_atom, neighbor_atoms_234, R_cut, cell_length,
    ↪  B= B_data_234, save_path=case_234_plot)

#Case 9-[Same cell at (1)] 3 neighbor atoms combine case
    ↪ 2+3+5
neighbor_atoms_235 = [[0.6, 0.5, 0.5],[0.5,0.4,0.5], [0.5, 0.6,
    ↪  0.5]]
data_235 = test_neighbor(center_atom, neighbor_atoms_235, R_cut
    ↪ )
B_235= Bispectrum(j=5/2, j1=3/2, j2=2/2, params=data_235)
B_data_235 = B_235.evaluate(j=5/2, j1=3/2, j2=2/2, params=
    ↪ data_235)
print(B_data_235)
case_235_plot= "/Users/duonghoang/Documents/GitHub/
    ↪ bispectrum_component/plots/plot_case_235.png"
plot_atoms(center_atom, neighbor_atoms_235, R_cut, cell_length,
    ↪  B= B_data_235, save_path=case_235_plot)


#Case 10-[Same cell at (1)] 3 neighbor atoms combine case
    ↪ 3+4+5
neighbor_atoms_345 = [[0.5,0.4,0.5],[0.4, 0.5, 0.5], [0.5, 0.6,
    ↪  0.5]]
data_345 = test_neighbor(center_atom, neighbor_atoms_345, R_cut
    ↪ )
B_345= Bispectrum(j=5/2, j1=3/2, j2=2/2, params=data_345)
```

```
B_data_345 = B_345.evaluate(j=5/2, j1=3/2, j2=2/2, params=
   ↪ data_345)
print(B_data_345)
case_345_plot= "/Users/duonghoang/Documents/GitHub/
   ↪ bispectrum_component/plots/plot_case_345.png"
plot_atoms(center_atom, neighbor_atoms_345, R_cut, cell_length,
   ↪  B= B_data_345, save_path=case_345_plot)
```

Listing B.6: data_frame_Si_atom_17_neighbors.py

```
import numpy as np
import pandas as pd
import sympy as sp
from sympy.physics.quantum.cg import CG
from sympy.physics.wigner import wigner_d
from sympy.physics.quantum.spin import Rotation
from sympy import *
from Bio.PDB.MMCIF2Dict import MMCIF2Dict
from bispectrum.methods.plot.neighbors import plot_atoms
from bispectrum.methods.calc.INPUT_param import test_neighbor
import matplotlib.pyplot as plt
import itertools
path = "/Users/duonghoang/Documents/GitHub/bispectrum_component
   ↪ /data/avgBL-Model.cif"
dico = MMCIF2Dict(path)
df_cif = pd.DataFrame.from_dict(dico, orient='index')
x = df_cif.iloc[-3]
y = df_cif.iloc[-2]
z = df_cif.iloc[-1]
atom_type = df_cif.iloc[-4]
x_array = np.array(x[0],dtype=float)
y_array = np.array(y[0],dtype=float)
z_array = np.array(z[0],dtype=float)
atom_type_array = np.array(atom_type[0], dtype=str)
df = pd.DataFrame({"atom_type":atom_type_array,"X" : x_array, "
   ↪ Y":y_array, "Z": z_array})

#Estimate list of potentially atoms in the center cell
df_atoms = df[(df['X'].between(0.5,0.7,inclusive='both'))
```

```
                        & (df['Y'].between(0.5,0.7,inclusive='
                            ↪ both'))
                        & (df['Z'].between(0.5,0.7, inclusive='
                            ↪ both'))]
#print (df_atoms)


# id
x_i = df['X'].iloc[17]
y_i = df['Y'].iloc[17]
z_i = df['Z'].iloc[17]
#print(x_i,y_i,z_i)
X_array = df['X'].to_numpy()
Y_array = df['Y'].to_numpy()
Z_array = df['Z'].to_numpy()
X_k_array = X_array - x_i
Y_k_array = Y_array - y_i
Z_k_array = Z_array - z_i
r_ik= np.sqrt(np.square(X_k_array)+np.square(Y_k_array)+np.
    ↪ square(Z_k_array))
df['X_k'],df['Y_k'], df['Z_k'],df['r_ik']= X_k_array, Y_k_array
    ↪ , Z_k_array, r_ik

#Check to see if chosen center atom coordinate sets to
    ↪ (0,0,0)
print(df.iloc[17])

#INPUT values
atomic_radius = 1.46 #silicon atomic radius, unit: angstrom
cell_length = df.iloc[4] #index row start from 0
    ↪ _cell_length_a at row 5 index [4]

r_mu = 0.0779 #scale atomic radius w.r.t cell length
R_cut = 0.25 #scaled value w.r.t cell length (for Si-Si case)
df_ik = df[(df['r_ik'] + r_mu)<= (R_cut)].copy(deep=true)
print(df_ik[['X_k', 'Y_k', 'Z_k', 'r_ik']])
print (df_ik[['X', 'Y', 'Z']])
print(df_ik)
```

APPENDIX C

PLOTTING CODES

Displaying and visualizing

Listing C.1: neighbors.py

```
"""
This file is used to plot the atoms in the cell and the
   ↪ neighbor atoms in study_1_SO4_calc.py
"""
import matplotlib.pyplot as plt
import numpy as np

def plot_atoms(center_atom, neighbor_atoms, R_cut, cell_length,
   ↪  B, save_path=None):
    # Plot
    fig, ax = plt.subplots()
    ax.set_xlim([0, cell_length])
    ax.set_ylim([0, cell_length])
    ax.set_aspect('equal')

    ax.scatter(center_atom[0], center_atom[1], color='red',
       ↪ label='Center Atom')

    if neighbor_atoms is not None:
        for neighbor_atom in neighbor_atoms:
            ax.scatter(neighbor_atom[0], neighbor_atom[1], color
               ↪ ='blue', label='Neighbor Atom')

    circle = plt.Circle((center_atom[0], center_atom[1]), R_cut
       ↪ , fill=False, color='green')
    ax.add_artist(circle)

    ax.annotate(r'$R_{\mathrm{cut}}$', xy=(center_atom[0] +
       ↪ R_cut, center_atom[1]), xytext=(center_atom[0] +
       ↪ R_cut, center_atom[1] + 0.1),
```

```
                arrowprops=dict(facecolor='blue', arrowstyle='
                    ↪ ->'))
    ax.set_title(r'$B_{{j_1,j_2,j}}={}$'.format(np.round(B, 2))
        ↪ )
    if save_path is not None:
        plt.savefig(save_path, dpi=300, bbox_inches='tight')
    plt.show()
```

Listing C.2: GMM_curve_example

```
'''
This script is used to plot the Gaussian distribution curves
'''
import numpy as np
import matplotlib.pyplot as plt

# Generate x values
x = np.linspace(-4, 4, 1000)

# Compute the standard Gaussian distribution values
y1 = 1 / np.sqrt(2 * np.pi) * np.exp(-0.5 * x**2)

# Compute additional Gaussian distributions with different
    ↪ parameters
mu_values = [-2, 0, 1, 2, 4]
sigma_values = [0.5, 1, 1.5, 2, 2.5]
colors = ['red', 'green', 'blue', 'purple', 'orange']

plt.figure()
# Plot the curves
for i in range(5):
    y = 1 / (sigma_values[i] * np.sqrt(2 * np.pi)) * np.exp
        ↪ (-0.5 * ((x - mu_values[i]) / sigma_values[i])**2)
    plt.plot(x, y, color=colors[i], label=f'mu={mu_values[i]},
        ↪ sigma={sigma_values[i]}')

plt.title('Gaussian Distributions')
plt.xlabel('Distance')
plt.ylabel('Probability Density')
```

```
plt.savefig("/Users/duonghoang/Documents/GitHub/
    ↪ bispectrum_component/plots/GMM_example", dpi=300,
    ↪ bbox_inches='tight')
# Display the plot
plt.show()
```

REFERENCE LIST

[1] Constantin A, Johnson RS. Large Gyres as a Shallow-Water Asymptotic Solution of Euler's Equation in Spherical Coordinates;473(2200):20170063. Available from: https://royalsocietypublishing.org/doi/10.1098/rspa.2017.0063.

[2] Oliver WD, Welander PB. Materials in superconducting quantum bits;38(10):816-25. Available from: http://link.springer.com/10.1557/mrs.2013.229.

[3] Tanaka K, Cho Y. Actual information storage with a recording density of 4 Tbit/in.2 in a ferroelectric recording medium;97(9):092901. Available from: https://pubs.aip.org/apl/article/97/9/092901/339716/Actual-information-storage-with-a-recording.

[4] National Science and Technology Council. Materials Genome Initiative for Global Competitiveness; 2011. Internet. Available from: http://www.whitehouse.gov/sites/default/files/microsites/ostp/materials_genome_initiative-final.pdf.

[5] Kohn W. Nobel Lecture: Electronic structure of matter—wave functions and density functionals;71(5):1253-66. Available from: https://link.aps.org/doi/10.1103/RevModPhys.71.1253.

[6] Pople JA. Nobel Lecture: Quantum chemical models;71(5):1267-74. Available from: https://link.aps.org/doi/10.1103/RevModPhys.71.1267.

[7] Schrödinger E. An Undulatory Theory of the Mechanics of Atoms and Molecules;28(6):1049-70. Available from: https://link.aps.org/doi/10.1103/PhysRev.28.1049.

[8] Kulik HJ, Hammerschmidt T, Schmidt J, Botti S, Marques MAL, Boley M, et al. Roadmap on Machine Learning in Electronic Structure;4(2):023004. Available from: https://iopscience.iop.org/article/10.1088/2516-1075/ac572f.

[9] Duan C, Nandy A, Terrones GG, Kastner DW, Kulik HJ. Active Learning Exploration of Transition-Metal Complexes to Discover Method-Insensitive and Synthetically Accessible Chromophores;3(2):391-401. Available from: https://pubs.acs.org/doi/10.1021/jacsau.2c00547.

[10] Lu F, Cheng L, DiRisio RJ, Finney JM, Boyer MA, Moonkaen P, et al. Fast Near *Ab Initio* Potential Energy Surfaces Using Machine Learning;126(25):4013-24. Available from: https://pubs.acs.org/doi/10.1021/acs.jpca.2c02243.

[11] Ching WY, Rulis P. Electronic Structure Methods for Complex Materials: The Orthogonalized Linear Combination of Atomic Orbitals. Oxford University Press;. Available from: http://www.oxfordscholarship.com/view/10.1093/acprof:oso/9780199575800.001.0001/acprof-9780199575800.

[12] Ching WY, Lin CC, Huber DL. Electronic Energy Structure of Amorphous Silicon;14(2):620-31. Available from: https://link.aps.org/doi/10.1103/PhysRevB.14.620.

[13] Ching WY, Lin CC. Orthogonalized Linear Combinations of Atomic Orbitals: Application to the Calculation of Energy Bands of Si III;12(12):5536-44. Available from: https://link.aps.org/doi/10.1103/PhysRevB.12.5536.

[14] Bartók AP, Payne MC, Kondor R, Csányi G. Gaussian Approximation Potentials: The Accuracy of Quantum Mechanics, Without the Electrons;104(13):136403. Available from: http://arxiv.org/abs/0910.1019.

[15] Thompson AP, Swiler LP, Trott CR, Foiles SM, Tucker GJ. A Spectral Analysis Method for Automated Generation of Quantum-Accurate Interatomic Potentials;285:316-30. Available from: http://arxiv.org/abs/1409.3880.

[16] Cusentino MA, Wood MA, Thompson AP. Explicit Multi-element Extension of the Spectral Neighbor Analysis Potential for Chemically Complex Systems;124(26):5456-64. Available from: http://arxiv.org/abs/2003.11570.

[17] Fiedler L, Modine NA, Schmerler S, Vogel DJ, Popoola GA, Thompson AP, et al.. Predicting Electronic Structures at Any Length Scale with Machine Learning. arXiv;. Available from: http://arxiv.org/abs/2210.11343.

[18] Yanxon H, Zagaceta D, Tang B, Matteson D, Zhu Q. PyXtal FF: A Python Library for Automated Force Field Generation;2(2):027001. Available from: http://arxiv.org/abs/2007.13012.

[19] Frye C, Efthimiou CJ. Spherical Harmonics in p Dimensions. arXiv;. Available from: http://arxiv.org/abs/1205.3548.

[20] Varshalovich DA, Moskalev AN, Khersonskii VK. Quantum Theory of Angular Momentum. WORLD SCIENTIFIC;. Available from: https://www.worldscientific.com/worldscibooks/10.1142/0270.

[21] Biedenharn LC, Louck JD, Carruthers PA. Angular Momentum in Quantum Physics: Theory and Application. 1st ed. Cambridge University Press;. Available from: https://www.cambridge.org/core/product/identifier/9780511759888/type/book.

[22] Meremianin AV. Multipole Expansions in Four-Dimensional Hyperspherical Harmonics;39(12):3099-112. Available from: http://arxiv.org/abs/math-ph/0510080.

[23] Pain JC. Sum Rules for Clebsch-Gordan Coefficients from Group Theory and Runge-Lenz-Pauli Vector. Publisher: arXiv Version Number: 1. Available from: https://arxiv.org/abs/2205.06151.

[24] Bishop C. Mixture Density Networks. Aston University; 1994.

[25] Xuan S, Jiang X, Spencer RK, Li NK, Prendergast D, Balsara NP, et al. Atomic-Level Engineering and Imaging of Polypeptoid Crystal Lattices;116(45):22491-9. Available from: https://pnas.org/doi/full/10.1073/pnas.1909992116.

[26] Ke Y, Sukthankar R. PCA-SIFT: A More Distinctive Representation for Local Image Descriptors. In: Proceedings of the 2004 IEEE Computer Society Conference on Computer Vision and Pattern Recognition, 2004. CVPR 2004.. vol. 2. IEEE;. p. 506-13. Available from: http://ieeexplore.ieee.org/document/1315206/.

VITA

Duong Thuy Hoang was born on March 26nd, 1997 in Quang Tri, Vietnam. They was in the public education system in Vietnam and graduated from Quang Tri town High School in 2015. In 2020 they received a Bachelor of Science in Physics from the University of Missouri-Kansas City.