RASĀYAN *J. Chem.*
Vol. 18 | No. 2 |678-696| April - June | 2025
ISSN: 0974-1496 | e-ISSN: 0976-0083 | CODEN: RJCABP
http://www.rasayanjournal.com
http://www.rasayanjournal.co.in

COMPUTATIONAL VISUALIZATION OF 3D BRAVAIS LATTICES USING PYTHON

Jyoti Singh[1], Sarita Passey[1], Anjali[2] and Hemant Verma[3,✉]
[1]Department of Chemistry, Zakir Husain Delhi College, University of Delhi, India.
[2]Institute of Informatics and Communication, University of Delhi, India.
[3]Department of Chemistry, Hindu College, University of Delhi-110007, India.
✉Corresponding author: hemantverma@hinducollege.ac.in

ABSTRACT

The Bravais lattice is a fundamental concept in crystallography that provides a mathematical framework to describe the periodic arrangement of points in a crystal. The significance of Bravais lattices lies in representing the fundamental symmetry and shape of crystals, making accurate visualization crucial for chemists and material scientists. Traditional visualization methods often rely on specialized software that may lack flexibility or customization options. This paper presents a novel approach for visualizing Bravais lattices in both 2D and 3D, using Python, an increasingly popular programming language in scientific research. This method covers all five 2D lattices and fourteen 3D crystal structures, leveraging Python's robust libraries such as the Matplotlib and NumPy to generate precise and customizable lattice representations. Incorporating Python into crystallographic visualization demonstrates its flexibility, accessibility and versatility showcasing its ability to integrate seamlessly with other scientific computing jobs. This approach exemplifies the fusion of Python programming with chemistry enhancing detailed analysis and educational capabilities. By using the Python code, this paper aims to equip chemists to perform the above-mentioned operations theoretically from their desks, eliminating the need for tedious manual drawing of crystal lattices, and providing access to practical computation tools relevant in today's research environment.

Keywords: Unit cell, seven crystal, Bravais lattice, Matplotlib and NumPy libraries, Python.
RASĀYAN *J. Chem.*, Vol. 18, No.2, 2025

INTRODUCTION

The Bravais lattice defines the three-dimensional arrangement of ions, atoms, or molecules in a crystal, classifying them into 14 distinct types across seven crystal systems[1]. All known crystalline substances fall into one of these 14 lattice systems. Bravais lattices holds tremendous significance in the field of crystallography[2,3] as well as material science[4] providing a standardized framework to visualize and classify the three-dimensional arrangements of atoms in crystals. This classification into 14 distinct types aids in understanding crystal symmetry and geometry, which are crucial for predicting material properties[5] such as thermal, electrical conductivity and mechanical strength. The way atoms are arranged in the Bravais lattice influences the physical properties of the materials. Accurate drawings of the crystal structures clarify arrangements of the underlying atoms, aiding in property prediction. Numerous research groups have attempted to predict the properties of materials and crystal structures using a variety of software packages and machine learning techniques.[6-11] The geometric Bravais lattices lay the Foundation for many theoretical models in material science and solid-state physics. Precise drawings of these models assist in simulations and calculations. Designing new materials[12,13] with required worthwhile properties also becomes possible with the understanding of Bravais lattices. Hence, by modifying these lattices, materials with the desirable specific properties can be created. Accurately drawing Bravais lattice is a fundamental skill for students, educators, and researchers, as it facilitates the comprehension of many intricate and challenging concepts in education, material science and crystallography. It therefore serves as a great educational tool. Although it may be possible to draw the Bravais lattices by hand, but usually software offers better precision, especially for complex and high symmetry cases. Additionally, software tools simplify the visualization of these lattices in 3D, a task that can be laborious and challenging to achieve with handmade drawings. Currently, several software programs are used for drawing the Bravais lattices[14] as well as visualizing molecular and crystal lattice structures in 3D and to investigate the arrangement of atoms in different materials. The popularly used tools such as VESTA[15], CrystalMaker[16], Diamond, Avogadro, Mercury, Jmol,

http://doi.org/10.31788/RJC.2025.1829028

Blender, XCrySDen[17] aid in analysing atomic arrangements for various research purposes. Python[18] has emerged as a popular programming language for scripting or task automation, valued for its ease of learning, readability, and extensive libraries. Key libraries include NumPy for numerical computation, Pandas for data manipulation, Matplotlib and Seaborn for visualization, SciPy for scientific computing, ChemPy for solving chemistry problems and ChemPlot for visualizing chemical space. ChemPy[19] contains well-known physical chemistry formulas and equations as well as analytical solutions. These libraries streamline complex research task, saving time and effort. Python's adaptability makes it suitable for numerous research domains, such as computational physics, bioinformatics, machine learning, and data science. Its flexibility allows collaboration across disciplines, facilitating multidisciplinary research while its fast development cycle and interactive shell enable quick experimentation and prototyping. Additionally, Python also has a vibrant, sizable community and is open-source, allowing researchers to share their work, benefit from community-contributed packages, and seek assistance. Its well-documented libraries and readable code enhance transparency and reproducibility, making research more accessible to the broader scientific community. With the advent of Python in the present times, particularly in academics and research, it has become an excellent choice for drawing the Bravais lattices owing to its numerous advantages. Its flexibility supports a wide range of applications, from quick scripts to complex simulations, making it suitable for both 2D and 3D visualizations. Python allows extensive customization of plots, enabling users to tailor visualizations to their specific needs. Libraries like Plotly and Mayavi enable interactive visualizations, allowing users to explore and manipulate the lattice in real-time. However, despite the availability of interactive tools, real-time manipulation of Bravais lattices with user-friendly interfaces is still an area for enhancement. Improving these capabilities can benefit both education and research applications. Python can also automate repetitive tasks, making it easy to generate multiple lattice configurations. It can run on multiple operating systems, including Windows, macOS, and Linux. Python can be easily integrated with many crystallographic tools such as VESTA, CrystalMaker, or CIF files[20]. Although, connecting computational lattice visualization and experimental data, remains a key research challenge. In light of all this, the Python codes presented in this article generate Bravais lattices with complete precision, producing structures with specific symmetry. Furthermore, these Python codes can be integrated into a web-page or application to create an interactive, real-time, and automated framework for customizable Bravais lattice visualization.

EXPERIMENTAL

The experimental work is divided into three sections A, B and C with each section containing the Python Program specific to its aim.

Section-A

Visualizing the 2-Dimensional Bravais Lattice

Here the Python code is generated for all the 5 possible Bravais lattices in 2-dimensional space by using the Matplotlib and NumPy libraries of python software. All 5 lattices, Square, Rectangular, Rhombic, Oblique and Hexagonal were obtained as output of Python code with complete precision as shown in Fig.-1.

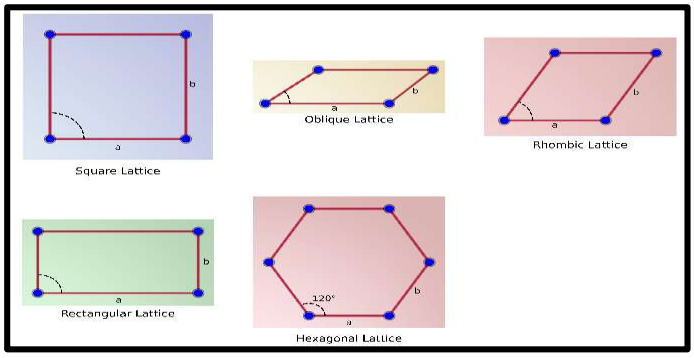

Fig.-1: Five Bravais Lattices in 2D

Pythos Code for 2-Dimentional Bravais Lattice

```
import matplotlib.pyplot as plt
import numpy as np
```

```
from matplotlib.patches import Polygon
from matplotlib.colors import LinearSegmentedColormap

def draw_lattice(ax, vertices, edges, labels, angles, title, background_color, title_pad=6, extent=[-1, 2, -1, 2]):
    # Create a custom gradient background
    x = np.linspace(0, 1, 256)
    y = np.linspace(0, 1, 256)
    X, Y = np.meshgrid(x, y)
    Z = np.sin(X**2 + Y**2)
    ax.imshow(Z, extent=extent, origin='lower', cmap=background_color, alpha=0.3)

    # Calculate the bounding box of the shape
    vertices_array = np.array(vertices)
    min_x, max_x = vertices_array[:, 0].min(), vertices_array[:, 0].max()
    min_y, max_y = vertices_array[:, 1].min(), vertices_array[:, 1].max()
    # Adjust limits to fit the shape inside the background
    padding = 0.2
    ax.set_xlim(min_x - padding, max_x + padding)
    ax.set_ylim(min_y - padding, max_y + padding)
    # Edges with a shadow effect for 3D appearance
    for edge in edges:
        ax.plot(*zip(*edge), color='black', linestyle="-", linewidth=4, alpha=0.3)  # Shadow
        ax.plot(*zip(*edge), color='crimson', linestyle="-", linewidth=2)
    # Vertices with a shadow effect for 3D appearance
    for vertex in vertices:
        ax.plot(*vertex, color='black', marker="o", markersize=14, alpha=0.3)  # Shadow
        ax.plot(*vertex, color='blue', marker="o", markersize=10)
    # Labels
    for label, pos in labels.items():
        ax.text(*pos, label, fontsize=12, ha='center')
    # Angles
    for angle in angles:
        arc = np.linspace(angle['start_angle'], angle['end_angle'], 100)
        arc_x = angle['center'][0] + angle['radius'] * np.cos(arc)
        arc_y = angle['center'][1] + angle['radius'] * np.sin(arc)
        ax.plot(arc_x, arc_y, 'k--')

    # Title below the lattice
    ax.text(0.5, -0.05, title, fontsize=14, ha='center', va='top', transform=ax.transAxes)
    ax.set_aspect('equal')
    ax.axis('off')

# Custom colormaps for gradient backgrounds
background_colors = [LinearSegmentedColormap.from_list("blue_gradient", ["lightblue", "darkblue"]),
                     LinearSegmentedColormap.from_list("green_gradient", ["lightgreen", "darkgreen"]),
                     LinearSegmentedColormap.from_list("red_gradient", ["lightcoral", "darkred"]),
                     LinearSegmentedColormap.from_list("yellow_gradient", ["lightgoldenrodyellow", "darkgoldenrod"]),
                     LinearSegmentedColormap.from_list("pink_gradient", ["lightpink", "darkred"])]
    # Square lattice
vertices_square = [(0, 0), (1, 0), (1, 1), (0, 1)]
edges_square = [[vertices_square[i], vertices_square[(i+1)%4]] for i in range(4)]
labels_square = {'a': (0.5, -0.105), 'b': (1.05, 0.5)}
angles_square = [{'center': (0, 0), 'start_angle': 0, 'end_angle': np.pi/2, 'radius': 0.25}]

# Rectangular lattice
vertices_rect = [(-0.5, 0), (1.5, 0), (1.5, 1), (-0.5, 1)]
edges_rect = [[vertices_rect[i], vertices_rect[(i+1)%4]] for i in range(4)]
labels_rect = {'a': (0.512, -0.15), 'b': (1.60, 0.47)}
angles_rect = [{'center': (-0.5, 0), 'start_angle': 0, 'end_angle': np.pi/2, 'radius': 0.3}]

# Rhombic lattice
alpha_rhombic = np.pi / 3  # 60 degrees
vertices_rhombic = [(0, 0), (1, 0), (1 + np.cos(alpha_rhombic), np.sin(alpha_rhombic)), (np.cos(alpha_rhombic), np.sin(alpha_rhombic))]
edges_rhombic = [[vertices_rhombic[i], vertices_rhombic[(i+1)%4]] for i in range(4)]
labels_rhombic = {'a': (0.5, -0.09), 'b': (1.3, 0.327)}
angles_rhombic = [{'center': (0, 0), 'start_angle': 0, 'end_angle': alpha_rhombic, 'radius': 0.28}]

# Oblique lattice
alpha_oblique = np.pi / 4  # 45 degrees
vertices_oblique = [(-0.5, 0), (1.5, 0), (1.5 + np.cos(alpha_oblique), np.sin(alpha_oblique)), (np.cos(alpha_oblique)-0.35, np.sin(alpha_oblique))]
edges_oblique = [[vertices_oblique[i], vertices_oblique[(i+1)%4]] for i in range(4)]
labels_oblique = {'a': (0.62, -0.15), 'b': (1.921, 0.21)}
angles_oblique = [{'center': (-0.5, 0), 'start_angle': 0, 'end_angle': alpha_oblique, 'radius': 0.4}]
# Hexagonal lattice
vertices_hex = [(0, 0), (1, 0), (1.5, np.sqrt(3)/2), (1, np.sqrt(3)), (0, np.sqrt(3)), (-0.5, np.sqrt(3)/2)]
edges_hex = [[vertices_hex[i], vertices_hex[(i+1)%6]] for i in range(3)] + [[vertices_hex[i], vertices_hex[(i+1)%6]] for i in range(3, 6)]
labels_hex = {'a': (0.5, -0.15), 'b': (1.35, 0.36), '120°': (0.190, 0.237)}
angles_hex = [{'center': (0, 0), 'start_angle': 0, 'end_angle': 2.0943951, 'radius': 0.2}]

fig, axs = plt.subplots(2, 3, figsize=(14, 9))
axs = axs.flatten()
fig.suptitle('Bravais Lattice in 2D', fontsize=20, y=0.98)
```

```
draw_lattice(axs[0], vertices_square, edges_square, labels_square, angles_square, 'Square Lattice', background_colors[0])
draw_lattice(axs[1], vertices_oblique, edges_oblique, labels_oblique, angles_oblique, 'Oblique Lattice', background_colors[3], extent=[-1.5, 2.5, -1.5, 2.5])
draw_lattice(axs[2], vertices_rhombic, edges_rhombic, labels_rhombic, angles_rhombic, 'Rhombic Lattice', background_colors[2])
draw_lattice(axs[3], vertices_rect, edges_rect, labels_rect, angles_rect, 'Rectangular Lattice', background_colors[1])
draw_lattice(axs[4], vertices_hex, edges_hex, labels_hex, angles_hex, 'Hexagonal Lattice', background_colors[4])
# Hide the last subplot (empty)
fig.delaxes(axs[5])
plt.tight_layout(rect=[0, 1, 1, 0.96])
plt.show()
```

Section-B

Visualizing of Primitive, Body-centered, Face-centered and base-centered Lattice

Python code is generated for the four possible variations in lattice point groups for the simple cubic lattice: primitive, body-centered, face-centered and base-centered point groups. For each lattice, information about vertices, interfacial angles, edges etc. is stored in Python dictionary and the Matplotlib.pyplot and Numpy libraries of python are used to obtain these four cubic lattices as shown in Fig.-2.

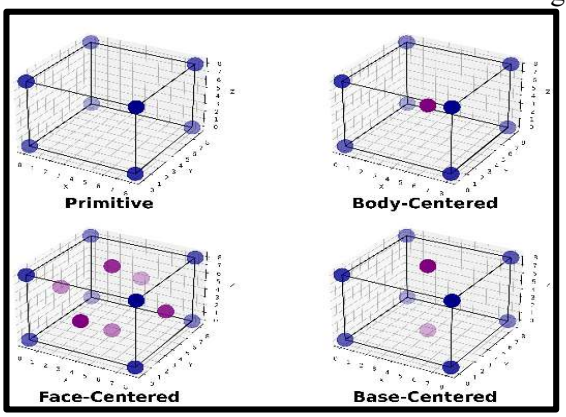

Fig.-2: Cubic Unit Cell Crystal Lattice in 3D

Python Code for Primitive, Body-Centered, Face-Centered and Body-Centered Lattice

```
#Primitive, Body-Centered, Face-Centered and Base-Centered Lattice
import numpy as np
import matplotlib.pyplot as plt

def plot_cell(ax, vertices, corner_vertices_color):
    edges = [ [vertices[0], vertices[4]], [vertices[0], vertices[1]], [vertices[1], vertices[2]], [vertices[1], vertices[5]], [vertices[2], vertices[3]], [vertices[2], vertices[6]],
[vertices[3], vertices[0]], [vertices[3], vertices[7]], [vertices[4], vertices[5]], [vertices[5], vertices[6]], [vertices[6], vertices[7]], [vertices[7], vertices[4]]
    ]

    corner_vertices = vertices[:8]
    other_vertices = vertices[8:]

    for edge in edges:
        xa, ya, za = zip(*edge)
        ax.plot(xa, ya, za, color='black', linestyle='-')

    ax.scatter(corner_vertices[:, 0], corner_vertices[:, 1], corner_vertices[:, 2], color = corner_vertices_color, s = 600)

    ax.scatter(other  vertices[:, 0], other  vertices[:, 1], other  vertices[:, 2], color = 'purple', s = 600)

    ax.set_xlabel('X')
    ax.set_ylabel('Y')
    ax.set_zlabel('Z')

# Simple Cubic
def plot_primitive(ax, a, b, c):
    vertices = np.array([ [0, 0, 0], [a, 0, 0], [a, b , 0], [0, b, 0], [0,0, c], [a, 0, c], [a, b, c], [0, b,c] ])

    plot_cell(ax, vertices, 'darkblue')

# Body-Centered Cubic
def plot  BC(ax, a, b, c):
    vertices = np.array([ [0, 0, 0], [a,0, 0], [a, b , 0], [0, b, 0], [0,0, c], [a, 0, c], [a, b, c], [0, b,c], [a/2, b/2, c/2] ])

    plot_cell(ax, vertices, 'darkblue')

# Face-Centered Cubic
def plot_FC(ax, a, b, c):
```

```
    vertices = np.array([ [0, 0, 0], [a,0, 0], [a, b , 0], [0, b, 0], [0,0, c], [a, 0, c], [a, b, c], [0, b,c], [a/2, 0, c/2],  [a, b/2, c/2],  [a/2, b, c/2],  [0, b/2, c/2],  [a/2, b/2, 0],  [a/2, b/2, c]])
    plot_cell(ax, vertices, 'darkblue')

# Base-Centered Cubic
def plot_base_centered(ax, a, b, c):
    vertices = np.array([ [0, 0, 0], [a,0, 0], [a, b , 0], [0, b, 0], [0,0, c], [a, 0, c], [a, b, c], [0, b,c], [a/2, b/2, 0], [a/2, b/2, c] ])

    plot_cell(ax, vertices, 'darkblue')

def main():
    try:
        a = 8
        b = 8
        c = 8
        alpha = 90
        beta = 90
        gamma = 90
        fig = plt.figure(figsize=(18, 14))
        titles = []
        plot_count = 4
        titles = ["Simple Cubic", "Body-Centered Cubic", "Face-Centered Cubic", "Base-Centered Cubic"]
        plot  functions = [plot  primitive, plot  BC, plot  FC, plot  base  centered]

        for i in range(plot_count):
            ax = fig.add_subplot(2, 2, i+1, projection='3d')
            plot_functions[i](ax, a, b, c)
            ax.text2D(0.5, -0.2, titles[i], transform = ax.transAxes, ha = 'center', fontsize = 18)

        plt.tight_layout()
        plt.subplots_adjust(bottom = 0.1, top = 0.98)
        plt.show()

    except ValueError as e:
        print(e)

if __name__ == "__main__":
    main()
```

Section-C

Visualizing the 3-Dimentional Bravais Lattices for the 7 Crystal System

Finally, the Python code for generating the 3-dimensional Bravais lattice for the seven-crystal system: Triclinic, Monoclinic, orthorhombic, trigonal, cubic, tetragonal, and hexagonal, has been written in subsequent sections. Using these crystal systems, all the 14 basic Bravais lattices can be obtained.

The Python code for each case is designed to store all information about interfacial angle, vertices, and edges in the Python dictionaries. The Matplotlib, and NumPy libraries have been used to obtain the 14 Bravais lattices with complete accuracy. While running programs for any of the crystal lattices, the program prompts for the numeric value of a, b, c, α, β and γ. This allows the program to generate numerous crystal systems for better understanding of each crystal system.

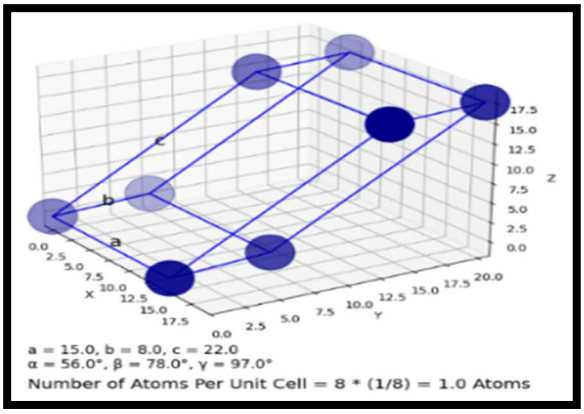

Fig.-3: 3D Crystal Lattice of Triclinic

Figure-3 shows single Bravais lattice for Triclinic crystal system, Fig.-4 displays the two Bravais lattices for the Monoclinic system: Primitive and Base-centered. Figure-5 illustrates different orientations of the single Bravais lattice for the Trigonal or Rhombohedral system. Figure-6 presents the four Bravais lattices

for the Orthorhombic system: Primitive, Body- centered, Base-centered and the Face-centered. Figure-7 shows the Primitive, Body-centered and the Face-centered Bravais lattices for the Cubic system. Figure-8 presents the Primitive and the Body-centered Bravais lattices for the Tetragonal system and Fig.- 9 shows the single Bravais Lattice for the Hexagonal system.

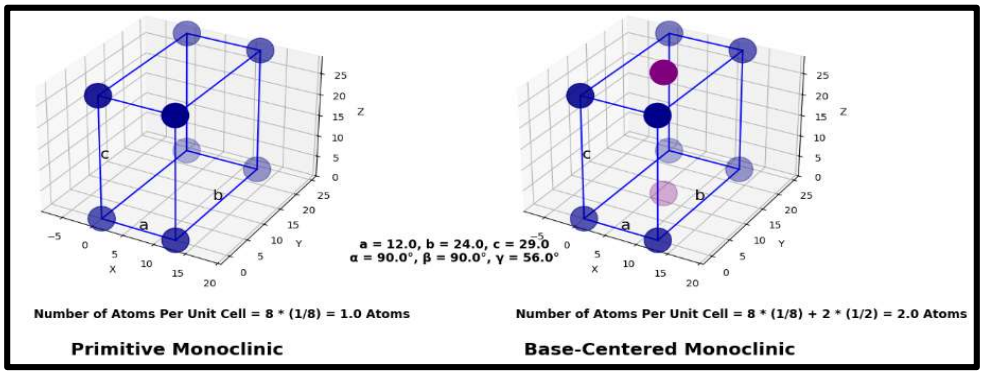

Fig.- 4: 3D Crystal Lattice of Monoclinic

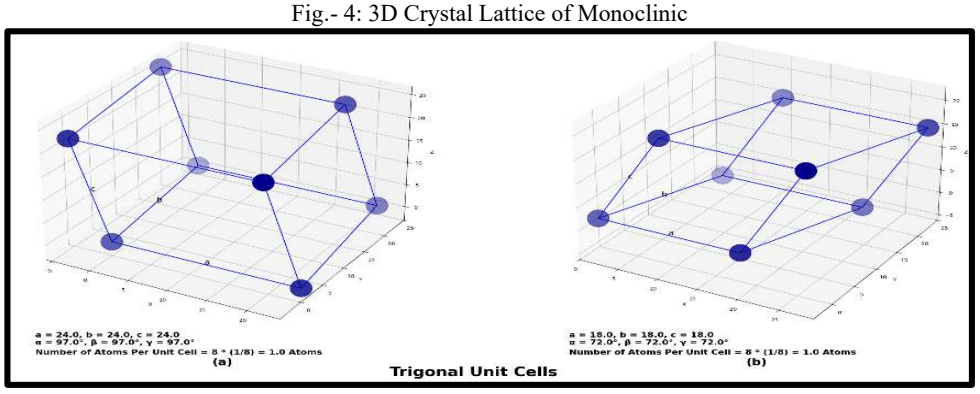

Fig.-5: 3D Crystal Lattice of Trigonal

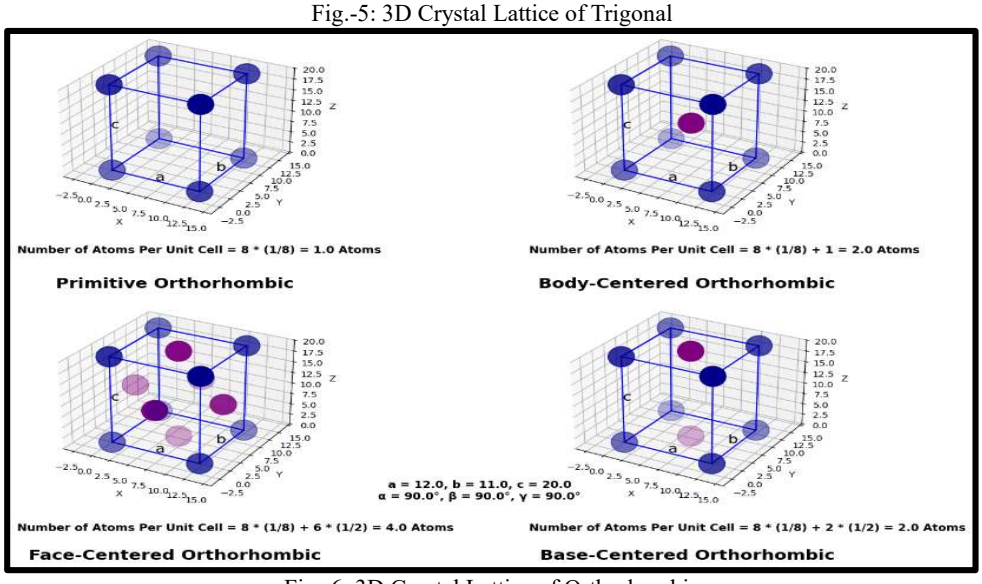

Fig.-6: 3D Crystal Lattice of Orthorhombic

Python Code for Triclinic 3-D Crystal Lattice

```
#Triclinic
import numpy as np
import matplotlib.pyplot as plt

def angle_deg_to_rad(degrees):
    return degrees * np.pi / 180

def Calculate_Vertices(a, b, c, alpha, beta, gamma):
```

```
    alpha, beta, gamma = angle_deg_to_rad(alpha), angle_deg_to_rad(beta), angle_deg_to_rad(gamma)

    v1 = [a, 0, 0]
    v2 = [b *np.cos(gamma), b *np.sin(gamma), 0]
    v3 = [c *np.cos(beta), c *(np.cos(alpha) - np.cos(beta) *np.cos(gamma)) / np.sin(gamma), c *np.sqrt(max(0, 1 + 2 *np.cos(alpha) *np.cos(beta) *np.cos(gamma) - np.cos(alpha)**2 - np.cos(beta)**2 - np.cos(gamma)**2)) / np.sin(gamma)]

    origin = [0, 0, 0]
    vertices = np.array([origin, v1, v2, v3, np.add(v1, v2), np.add(v1, v3), np.add(v2, v3), np.add(np.add(v1, v2), v3)])

    return vertices

def plot_cell(ax, vertices, a, b, c, alpha, beta, gamma):
    edges = [ [vertices[0], vertices[2]], [vertices[0], vertices[1]], [vertices[0], vertices[3]], [vertices[1], vertices[5]], [vertices[1], vertices[4]], [vertices[2], vertices[6]], [vertices[2], vertices[4]], [vertices[3], vertices[6]], [vertices[3], vertices[5]], [vertices[4], vertices[7]], [vertices[5], vertices[7]], [vertices[6], vertices[7]] ]

    corner_vertices = vertices[:8]

    for edge in edges:
        xa, ya, za = zip(*edge)
        ax.plot(xa, ya, za, color='blue', linestyle='-')

    ax.scatter(corner  vertices[:,0], corner  vertices[:, 1], corner  vertices[:, 2], color= 'darkblue', s=1400)

    # Annotations for a, b, c
    ax.text(a/2, 0, 0, 'a', color='black', fontsize=16)
    ax.text(vertices[2][0]/2, vertices[2][1]/2, 0, 'b', color='black', fontsize=16)
    ax.text(vertices[3][0]/2, vertices[3][1]/2, vertices[3][2]/2, 'c', color='black', fontsize=16)

    ax.set_xlabel('X')
    ax.set_ylabel('Y')
    ax.set_zlabel('Z')

    #Calculate the Number of Atoms per Unit Cell
    num_of_vertices = len(vertices)
    num_of_atoms_per_unit_cell = num_of_vertices * (1/num_of_vertices)

    #Display num of atoms per unit cell
    ax.text2D(0.05, -0.08, f"Number of Atoms Per Unit Cell = {num_of_vertices} * (1/{num_of_vertices}) = {num_of_atoms_per_unit_cell} Atoms", transform = ax.transAxes, fontsize = 14, verticalalignment = 'top')

    #Set Equal Aspect Ratio
    maximum_range = np.ptp(vertices, axis = 0).max() /2.0
    mid_val = vertices.mean(axis = 0)
    ax.set_xlim(mid_val[0]- maximum_range, mid_val[0] + maximum_range)
    ax.set_ylim(mid_val[1]- maximum_range, mid_val[1] + maximum_range)
    ax.set_ylim(mid_val[2]- maximum_range, mid_val[2] + maximum_range)

    #Set equal aspect  ratio
    max_range = np.array([vertices[:, 0].max() - vertices[:, 0].min(), vertices[:, 1].max() - vertices[:, 1].min(), vertices[:, 2].max() - vertices[:, 2].min()]).max() /2.0
    x  mid = (vertices[:, 0].max() + vertices[:, 0].min()) * 0.5
    y_mid = (vertices[:, 1].max() + vertices[:, 1].min()) * 0.5
    z_mid = (vertices[:, 2].max() + vertices[:, 2].min()) * 0.5
    ax.set_xlim(x_mid - max_range, x_mid + max_range)
    ax.set_ylim(y_mid - max_range, y_mid + max_range)
    ax.set_zlim(z_mid - max_range, z_mid + max_range)

    # Show parameters on plot
    parameters = f"a = {a}, b = {b}, c = {c}\nα = {alpha}°, β = {beta}°, γ = {gamma}°"
    ax.text2D(0.05, -0.06, parameters, transform = ax.transAxes, fontsize=12, verticalalignment = 'bottom')

def determine_crystal(a, b, c, alpha, beta, gamma):
    if not np.isclose(alpha, 90) and not np.isclose(beta, 90) and not np.isclose(gamma, 90) and not np.isclose(a, b) and not np.isclose(b, c) and not np.isclose(c, a) and not np.isclose(alpha, beta) and not np.isclose(beta, gamma):
        return "Triclinic"
    return "Unknown"

def main():
    try:
        print("Conditions for a Triclinic system: a ≠ b ≠ c and α ≠ β ≠ γ ≠ 90°")
        print("Enter the values of first Triclinic system: ")
        a = float(input("Enter the value of side a: "))
        b = float(input("Enetr the value of side b: "))
        c = float(input("Enter the value of side c: "))
        alpha = float(input("Enter the value of angle alpha (in degree): "))
        beta = float(input("Enter the value of angle beta (in degree): "))
        gamma = float(input("Enter the value of angle gamma (in degree): "))

        print("Enter the values of second Triclinic system: ")
        a1 = float(input("Enter the value of side a: "))
        b1 = float(input("Enter the value of side b: "))
        c1 = float(input("Enter the value of side c: "))
```

```
        alpha1 = float(input("Enter the value of angle alpha (in degree): "))
        beta1 = float(input("Enter the value of angle beta (in degree): "))
        gamma1 = float(input("Enter the value of angle gamma (in degree): "))

        if a <= 0 or b <= 0 or c <= 0 or not (0 < alpha < 180) or not (0 < beta < 180) or not (0 < gamma < 180):
            raise ValueError("Invalid input. Side lengths must be positive and angles must be between 0 and 180 degrees.")

        if a <= 0 or b <= 0 or c <= 0 or not (0 < alpha < 180) or not (0 < beta < 180) or not (0 < gamma < 180):
            raise ValueError("Invalid Input, side lengths must be positive and angles must be between 0 and 180 degree.")

        crystal_system = determine_crystal(a, b, c, alpha, beta, gamma)
        crystal_system1 = determine_crystal(a1, b1, c1, alpha1, beta1, gamma1)

        if crystal_system == "Triclinic" and crystal_system1 == "Triclinic":
            fig, axes = plt.subplots(1, 2, subplot_kw = {'projection': '3d'}, figsize = (30, 10))

            vertices = Calculate_Vertices(a, b, c, alpha, beta, gamma)
            vertices1 = Calculate_Vertices(a1, b1, c1, alpha1, beta1, gamma1)
            plot_cell(axes[0], vertices, a, b, c, alpha, beta, gamma)
            plot_cell(axes[1], vertices1, a1, b1, c1, alpha1, beta1, gamma1)

            titles = ["(a)", "(b)"]
            for i in range(2):
                axes[i].text2D(0.5, -0.19, titles[i], transform=axes[i].transAxes, ha='center', fontsize=16)

            # Add a common title
            fig.text(0.5, 0.05, "Triclinic Unit Cells", ha='center', fontsize=20, weight='bold', verticalalignment='bottom')

            plt.subplots_adjust(top = 1.1)
            plt.show()
        else:
            print("The provided parameters do not form a Triclinic unit cell.")
            print("Conditions for a Triclinic system: a ≠ b ≠ c and α ≠ β ≠ γ ≠ 90°")

    except ValueError as e:
        print(e)

if __name__ == "__main__":
    main()
```

Python Code for Monoclinic 3-D Crystal Lattice

```
#Monoclinic
import numpy as np
import matplotlib.pyplot as plt

def plot_cell(ax, vertices, corner_vertices, color, atom_count, a, b, c, alpha, beta, gamma):
    edges = [ [vertices[0], vertices[4]], [vertices[0], vertices[1]], [vertices[1], vertices[2]], [vertices[1], vertices[5]], [vertices[2], vertices[3]], [vertices[2], vertices[6]],
[vertices[3], vertices[0]], [vertices[3], vertices[7]], [vertices[4], vertices[5]], [vertices[5], vertices[6]], [vertices[6], vertices[7]], [vertices[7], vertices[4]]
    ]

    corner_vertices = vertices[:8]
    other_vertices = vertices[8:]

    for edge in edges:
        xa, ya, za = zip(*edge)
        ax.plot(xa, ya, za, color='blue', linestyle='-')

    ax.scatter(corner_vertices[:, 0], corner_vertices[:, 1], corner_vertices[:, 2], color = corner_vertices_color, s = 700)

    ax.scatter(other_vertices[:, 0], other_vertices[:, 1], other_vertices[:, 2], color = 'purple', s = 700)

    #Annotations for a, b, c
    ax.text((vertices[0][0] + vertices[1][0])/2 , (vertices[0][1] + vertices[1][1])/2 , (vertices[0][2] + vertices[1][2])/2, 'a', color = 'black', fontsize = 16 )
    ax.text((vertices[0][0] + vertices[2][0])/1.1 , (vertices[0][1] + vertices[2][1])/2 + 0.5 , (vertices[0][2] + vertices[2][2])/2, 'b', color = 'black', fontsize = 16 )
    ax.text((vertices[0][0] + vertices[4][0])/2 , (vertices[0][1] + vertices[4][1])/2 , (vertices[0][2] + vertices[4][2])/2, 'c', color = 'black', fontsize = 16 )

    ax.set_xlabel('X')
    ax.set_ylabel('Y')
    ax.set_zlabel('Z')

    #Calculate and display the Number of Atoms per Unit Cell
    num_of_corner_vertices = len(corner_vertices)
    num_of_other_vertices = len(other_vertices)
    if num_of_other_vertices == 0:
        num_of_atoms_per_unit_cell = num_of_corner_vertices * (1 / num_of_corner_vertices)
        ax.text2D(-0.1, -0.09, f"Number of Atoms Per Unit Cell = {num_of_corner_vertices} * (1/{num_of_corner_vertices}) = {num_of_atoms_per_unit_cell} Atoms",
transform=ax.transAxes, fontsize=12, verticalalignment='top')
    else:
        num_of_atoms_per_unit_cell = num_of_corner_vertices * (1 / num_of_corner_vertices) + num_of_other_vertices * (1 / 2)
        ax.text2D(-0.1, -0.12, f"Number of Atoms Per Unit Cell = {num_of_corner_vertices} * (1/{num_of_corner_vertices}) + {num_of_other_vertices} * (1/2) =
{num_of_atoms_per_unit_cell} Atoms", transform=ax.transAxes, fontsize=12, verticalalignment='top')
```

```
    #Set equal aspect  ratio
    maximum_range = np.array([vertices[:, 0].max() - vertices[:, 0].min(), vertices[:, 1].max() - vertices[:, 1].min(), vertices[:, 2].max() - vertices[:, 2].min()]).max()
/2.0
    x_mid = (vertices[:, 0].max() + vertices[:, 0].min()) * 0.5
    y_mid = (vertices[:, 1].max() + vertices[:, 1].min()) * 0.5
    z_mid = (vertices[:, 2].max() + vertices[:, 2].min()) * 0.5
    ax.set_xlim(x_mid - maximum_range, x_mid + maximum_range)
    ax.set_ylim(y_mid - maximum_range, y_mid + maximum_range)
    ax.set  zlim(z  mid - maximum  range, z  mid + maximum  range)

def plot_primitive(ax, a, b, c, alpha, beta, gamma):
    vertices = np.array([ [0, 0, 0], [a, 0, 0], [a, b , 0], [0, b, 0], [0,0, c], [a, 0, c], [a, b, c], [0, b,c] ])
    plot_cell(ax, vertices, 'darkblue', 8, a, b, c, alpha, beta, gamma)

def plot_base_centered(ax, a, b, c, alpha, beta, gamma):
    vertices = np.array([ [0, 0, 0], [a,0, 0], [a, b , 0], [0, b, 0], [0,0, c], [a, 0, c], [a, b, c], [0, b,c], [a/2, b/2, 0], [a/2, b/2, c] ])

    plot_cell(ax, vertices, 'darkblue', 10, a, b, c, alpha ,beta, gamma)

def determine_crystal(a, b, c, alpha, beta, gamma):
    if ( (np.isclose(alpha, 90) and np.isclose(beta, 90)) )or ( (np.isclose(beta, 90) and np.isclose(gamma, 90)) )or ( (np.isclose(alpha, 90) and np.isclose(gamma, 90)) )
and not np.isclose(a, b) and not np.isclose(b, c):
        return "Monoclinic"
    return "Unknown"

def main():
    try:
        print("Conditions for a Monoclinic system: a ≠ b ≠ c and α = β = 90° ≠ γ")
        a = float(input("Enter the value of side a: "))
        b = float(input("Enter the value of side b: "))
        c = float(input("Enter the value of side c: "))
        alpha = float(input("Enter the value of angle alpha (in degree): "))
        beta = float(input("Enter the value of angle beta (in degree): "))
        gamma = float(input("Enter the value of angle gamma (in degree): "))

        if a <= 0 or b <= 0 or c <= 0 or not (0 < alpha < 180) or not (0 < beta < 180) or not (0 < gamma < 180):
            raise ValueError("Invalid Input, side lengths must be positive and angles must be between 0 and 180 degree.")

        crystal_system = determine_crystal(a, b, c, alpha, beta, gamma)

        fig = plt.figure(figsize=(15, 12))

        # Show parameters on plot
        parameters = f"a = {a}, b = {b}, c = {c}\nα = {alpha}°, β = {beta}°, γ = {gamma}°"
        fig.text(0.47, 0.6, parameters, ha='center', fontsize=12, verticalalignment='bottom')

        titles = []
        plot_count = 0

        if crystal  system == "Monoclinic":
            titles = ["Primitive Monoclinic", "Base-Centered Monoclinic"]
            plot_functions = [plot_primitive, plot_base_centered]
            plot_count = 2
        else:
            print("The provided parameters do not form a Monoclinic Unit Cell.")
            print("Conditions for a Monoclinic system: a ≠ b ≠ c and α = β = γ = 90°")
            return

        for i in range(plot_count):
            ax = fig.add_subplot(2, 2, i+1, projection = '3d')
            plot_functions[i](ax, a, b, c, alpha, beta, gamma)
            ax.text2D(0.5, -0.3, titles[i], transform = ax.transAxes, ha = 'center', fontsize = 18, weight = 'bold')

        plt.tight_layout()
        plt.show()

    except ValueError as e:
        print(e)

if __name__ == "__main__":
    main()
```

Python Code for Orthorhombic 3-D Crystal Lattice

```
#Orthorhomic
import numpy as np
import matplotlib.pyplot as plt

def plot_cell(ax, vertices, corner_vertices_color, atom_count, a, b, c, alpha, beta, gamma):
    edges = [ [vertices[0], vertices[4]], [vertices[0], vertices[1]], [vertices[1], vertices[2]], [vertices[1], vertices[5]], [vertices[2], vertices[3]], [vertices[2], vertices[6]],
[vertices[3], vertices[0]], [vertices[3], vertices[7]], [vertices[4], vertices[5]], [vertices[5], vertices[6]], [vertices[6], vertices[7]], [vertices[7], vertices[4]]
```

```
    ]

    corner_vertices = vertices[:8]
    other  vertices = vertices[8:]

    for edge in edges:
        xa, ya, za = zip(*edge)
        ax.plot(xa, ya, za, color='blue', linestyle='-')

    ax.scatter(corner  vertices[:, 0], corner  vertices[:, 1], corner  vertices[:, 2], color = corner  vertices  color, s = 700)

    ax.scatter(other_vertices[:, 0], other_vertices[:, 1], other_vertices[:, 2], color = 'purple', s = 700)

    #Annotations for a, b, c
    ax.text((vertices[0][0] + vertices[1][0])/2 , (vertices[0][1] + vertices[1][1])/2 , (vertices[0][2] + vertices[1][2])/2, 'a', color = 'black', fontsize = 16 )
    ax.text((vertices[0][0] + vertices[2][0])/1.1 , (vertices[0][1] + vertices[2][1])/2 + 0.5 , (vertices[0][2] + vertices[2][2])/2, 'b', color = 'black', fontsize = 16 )
    ax.text((vertices[0][0] + vertices[4][0])/2 , (vertices[0][1] + vertices[4][1])/2 , (vertices[0][2] + vertices[4][2])/2, 'c', color = 'black', fontsize = 16 )

    ax.set_xlabel('X')
    ax.set_ylabel('Y')
    ax.set_zlabel('Z')

    #Calculate and display the Number of Atoms per Unit Cell
    num_of_corner_vertices = len(corner_vertices)
    num_of_other_vertices = len(other_vertices)

    if num_of_other_vertices == 0:
      num_of_atoms_per_unit_cell = num_of_corner_vertices * (1 / num_of_corner_vertices)
      ax.text2D(-0.1, -0.09, f"Number of Atoms Per Unit Cell = {num_of_corner_vertices} * (1/{num_of_corner_vertices}) = {num_of_atoms_per_unit_cell} Atoms",
transform=ax.transAxes, fontsize=12, verticalalignment='top')
    elif num_of_other_vertices == 1:
      num_of_atoms_per_unit_cell = num_of_corner_vertices * (1 / num_of_corner_vertices) + num_of_other_vertices
      ax.text2D(-0.1, -0.09, f"Number of Atoms Per Unit Cell = {num_of_corner_vertices} * (1/{num_of_corner_vertices}) + 1 = {num_of_atoms_per_unit_cell}
Atoms", transform=ax.transAxes, fontsize=12, verticalalignment='top')
    else:
      num_of_atoms_per_unit_cell = num_of_corner_vertices * (1 / num_of_corner_vertices) + num_of_other_vertices * (1 / 2)
      ax.text2D(-0.1, -0.12, f"Number of Atoms Per Unit Cell = {num_of_corner_vertices} * (1/{num_of_corner_vertices}) + {num_of_other_vertices} * (1/2) =
{num  of  atoms  per  unit  cell} Atoms", transform=ax.transAxes, fontsize=12, verticalalignment='top')

    #Set equal aspect  ratio
    maximum_range = np.array([vertices[:, 0].max() - vertices[:, 0].min(), vertices[:, 1].max() - vertices[:, 1].min(), vertices[:, 2].max() - vertices[:, 2].min()]).max()
/2.0
    x_mid = (vertices[:, 0].max() + vertices[:, 0].min()) * 0.5
    y_mid = (vertices[:, 1].max() + vertices[:, 1].min()) * 0.5
    z_mid = (vertices[:, 2].max() + vertices[:, 2].min()) * 0.5
    ax.set_xlim(x_mid - maximum_range, x_mid + maximum_range)
    ax.set_ylim(y_mid - maximum_range, y_mid + maximum_range)
    ax.set_zlim(z_mid - maximum_range, z_mid + maximum_range)

def plot_primitive(ax, a, b, c, alpha, beta, gamma):
    vertices = np.array([ [0, 0,0], [a, 0, 0], [a, b , 0], [0, b, 0], [0,0, c], [a, 0, c], [a, b, c], [0, b,c] ])

    plot_cell(ax, vertices, 'darkblue', 8, a, b, c, alpha, beta, gamma)

def plot_BC(ax, a, b, c, alpha, beta, gamma):
    vertices = np.array([ [0, 0, 0], [a,0, 0], [a, b , 0], [0, b, 0], [0,0, c], [a, 0, c], [a, b, c], [0, b,c], [a/2, b/2, c/2] ])

    plot_cell(ax, vertices, 'darkblue', 9, a, b, c, alpha, beta, gamma)

def plot_FC(ax, a, b, c, alpha, beta, gamma):
    vertices = np.array([ [0, 0, 0], [a,0, 0], [a, b , 0], [0, b, 0], [0,0, c], [a, 0, c], [a, b, c], [0, b,c], [a/2, 0, c/2],  [a, b/2, c/2],  [a/2, b, c/2],  [0, b/2, c/2],  [a/2, b/2, 0],  [a/2,
b/2, c]])

    plot  cell(ax, vertices, 'darkblue', 14, a, b, c, alpha, beta, gamma)

def plot_base_centered(ax, a, b, c, alpha, beta, gamma):
     vertices = np.array([ [0, 0, 0], [a,0, 0], [a, b , 0], [0, b, 0], [0,0, c], [a, 0,c], [a, b, c], [0, b,c], [a/2, b/2, 0], [a/2, b/2, c] ])

     plot_cell(ax, vertices, 'darkblue', 10, a, b, c, alpha, beta, gamma)

def determine_crystal(a, b, c, alpha, beta, gamma):
    if np.isclose(alpha, 90) and np.isclose(beta, 90) and np.isclose(gamma, 90) and not np.isclose(a,b) and not np.isclose(b,c):
        return "Orthorhombic"
    return "Unknown"


def main():
    try:
        print("Conditions for a Orthorhombic system: a ≠ b ≠ c and α = β = γ = 90°")
        a = float(input("Enter the value of side a: "))
        b = float(input("Enter the value of side b: "))
        c = float(input("Enter the value of side c: "))
```

```
        alpha = float(input("Enter the value of angle alpha (in degree): "))
        beta = float(input("Enter the value of angle beta (in degree): "))
        gamma = float(input("Enter the value of angle gamma (in degree): "))

        if a <= 0 or b <= 0 or c <= 0 or not (0 < alpha < 180) or not (0 < beta < 180) or not (0 < gamma < 180):
            raise ValueError("Invalid Input, side lengths must be positive and angles must be between 0 and 180 degree.")

        crystal_system = determine_crystal(a, b, c, alpha, beta, gamma)

        fig = plt.figure(figsize=(15, 10))
        titles = []
        plot_count = 0

        if crystal_system == "Orthorhombic":
            titles = ["Primitive Orthorhombic", "Body-Centered Orthorhombic", "Face-Centered Orthorhombic", "Base-Centered Orthorhombic"]
            plot_functions = [plot_primitive, plot_BC, plot_FC, plot_base_centered]
            plot_count = 4
        else:
            print("The provided parameters do not form a Orthorhombic unit cell.")
            print("Conditions for a Orthorhombic system: a ≠ b ≠ c and α = β = γ = 90°")
            return

        for i in range(plot  count):
            ax = fig.add_subplot(2, 2, i+1, projection = '3d')
            plot_functions[i](ax, a, b, c, alpha, beta, gamma)
            ax.text2D(0.5, -0.3, titles[i], transform = ax.transAxes, ha = 'center', fontsize = 18, weight = 'bold')

        # Show parameters on plot
        parameters = f"a = {a}, b = {b}, c = {c}\nα = {alpha}°, β = {beta}°, γ = {gamma}°"
        fig.text(0.5, 0.12, parameters, ha='center', fontsize=12, verticalalignment='bottom')

        plt.tight_layout()
        plt.show()

    except ValueError as e:
        print(e)

if    name    == "    main    ":
    main()
```

Python Code for Trigonal / Rhombohedral 3-D Crystal Lattice

```
#Trigonal/Rhombohedral
import numpy as np
import matplotlib.pyplot as plt

def angle_deg_to_rad(degrees):
    return degrees * np.pi / 180

def Calculate_Vertices(a, b, c, alpha, beta, gamma):
    alpha, beta, gamma = angle_deg_to_rad(alpha), angle_deg_to_rad(beta), angle_deg_to_rad(gamma)

    v1 = [a, 0, 0]
    v2 = [b *np.cos(gamma), b *np.sin(gamma), 0]
    v3 = [c *np.cos(beta), c *(np.cos(alpha) - np.cos(beta) *np.cos(gamma)) / np.sin(gamma), c *np.sqrt(max(0, 1 + 2 *np.cos(alpha) *np.cos(beta) *np.cos(gamma) -
np.cos(alpha)**2 - np.cos(beta)**2 - np.cos(gamma)**2)) / np.sin(gamma)]

    origin = [0, 0, 0]
    vertices = np.array([origin, v1, v2, v3, np.add(v1, v2), np.add(v1, v3), np.add(v2, v3), np.add(np.add(v1, v2), v3)])

    return vertices

def plot_cell(ax, vertices, a, b, c, alpha, beta, gamma):
    edges = [ [vertices[0], vertices[2]], [vertices[0], vertices[1]], [vertices[0], vertices[3]], [vertices[1], vertices[5]], [vertices[1], vertices[4]], [vertices[2], vertices[6]],
[vertices[2], vertices[4]], [vertices[3], vertices[6]], [vertices[3], vertices[5]], [vertices[4], vertices[7]], [vertices[5], vertices[7]], [vertices[6], vertices[7]] ]

    corner_vertices = vertices[:8]

    for edge in edges:
        xa, ya, za = zip(*edge)
        ax.plot(xa, ya, za, color='blue', linestyle='-')

    ax.scatter(corner_vertices[:,0], corner_vertices[:, 1], corner_vertices[:, 2], color= 'darkblue', s=1400)

    # Annotations for a, b, c
    ax.text(a/2, 0, 0, 'a', color='black', fontsize=16)
    ax.text(vertices[2][0]/2, vertices[2][1]/2, 0, 'b', color='black', fontsize=16)
    ax.text(vertices[3][0]/2, vertices[3][1]/2, vertices[3][2]/2, 'c', color='black', fontsize=16)

    ax.set_xlabel('X')
    ax.set_ylabel('Y')
    ax.set_zlabel('Z')
```

```
    #Calculate the Number of Atoms per Unit Cell
    num_of_vertices = len(vertices)
    num of atoms per unit cell = num of vertices * (1/num of vertices)

    #Display num of atoms per unit cell
    ax.text2D(0.05, -0.08, f"Number of Atoms Per Unit Cell = {num_of_vertices} * (1/{num_of_vertices}) = {num_of_atoms_per_unit_cell} Atoms", transform =
ax.transAxes, fontsize = 12, verticalalignment = 'top')

    #Set equal aspect ratio
    maximum_range = np.array([vertices[:, 0].max() - vertices[:, 0].min(), vertices[:, 1].max() - vertices[:, 1].min(), vertices[:, 2].max() - vertices[:, 2].min()]).max()
/2.0
    x_mid = (vertices[:, 0].max() + vertices[:, 0].min()) * 0.5
    y_mid = (vertices[:, 1].max() + vertices[:, 1].min()) * 0.5
    z_mid = (vertices[:, 2].max() + vertices[:, 2].min()) * 0.5
    ax.set_xlim(x_mid - maximum_range, x_mid + maximum_range)
    ax.set_ylim(y_mid - maximum_range, y_mid + maximum_range)
    ax.set_zlim(z_mid - maximum_range, z_mid + maximum_range)

    # Show parameters on plot
    parameters = f"a = {a}, b = {b}, c = {c}\nα = {alpha}°, β = {beta}°, γ = {gamma}°"
    ax.text2D(0.05, -0.06, parameters, transform = ax.transAxes, fontsize=12, verticalalignment = 'bottom')

def determine_crystal(a, b, c, alpha, beta, gamma):
    if np.isclose(a, b) and np.isclose(b, c) and np.isclose(alpha, beta) and np.isclose(beta, gamma) and not np.isclose(alpha, 90):
        return "Trigonal"
    return "Unknown"


def main():
    try:
        print("Conditions for a Trigonal system: a = b = c and α = β = γ ≠ 90°")
        print("Enter the values of first Trigonal system: ")
        a = float(input("Enter the value of side a: "))
        b = float(input("Enter the value of side b: "))
        c = float(input("Enter the value of side c: "))
        alpha = float(input("Enter the value of angle alpha (in degree): "))
        beta = float(input("Enter the value of angle beta (in degree): "))
        gamma = float(input("Enter the value of angle gamma (in degree): "))

        print("Enter the values of second Trigonal system: ")
        a1 = float(input("Enter the value of side a: "))
        b1 = float(input("Enter the value of side b: "))
        c1 = float(input("Enter the value of side c: "))
        alpha1 = float(input("Enter the value of angle alpha (in degree): "))
        beta1 = float(input("Enter the value of angle beta (in degree): "))
        gamma1 = float(input("Enter the value of angle gamma (in degree): "))

        if a <= 0 or b <= 0 or c <= 0 or not (0 < alpha < 180) or not (0 < beta < 180) or not (0 < gamma < 180):
            raise ValueError("Invalid Input, side lengths must be positive and angles must be between 0 and 180 degree.")

        if a1 <= 0 or b1 <= 0 or c1 <= 0 or not (0 < alpha1 < 180) or not (0 < beta1 < 180) or not (0 < gamma1 < 180):
            raise ValueError("Invalid input. Side lengths must be positive and angles must be between 0 and 180 degrees.")

        crystal_system = determine_crystal(a, b, c, alpha, beta, gamma)
        crystal_system1 = determine_crystal(a1, b1, c1, alpha1, beta1, gamma1)

        if crystal_system == "Trigonal" and crystal_system1 == "Trigonal":
            fig, axes = plt.subplots(1, 2, subplot_kw = {'projection': '3d'}, figsize = (30, 10))

            vertices = Calculate_Vertices(a, b, c, alpha, beta, gamma)
            vertices1 = Calculate_Vertices(a1, b1, c1, alpha1, beta1, gamma1)
            plot_cell(axes[0], vertices, a, b, c, alpha, beta, gamma)
            plot  cell(axes[1], vertices1, a1, b1, c1, alpha1, beta1, gamma1)

            titles = ["(a)", "(b)"]
            for i in range(2):
                axes[i].text2D(0.5, -0.19, titles[i], transform=axes[i].transAxes, ha='center', fontsize=16)

            # Add a common title
            fig.text(0.5, 0.05, "Trigonal Unit Cells", ha='center', fontsize=20, weight='bold', verticalalignment='bottom')

            plt.subplots_adjust(top = 1.1)
            plt.show()
        else:
            print("The provided parameters do not form a Trigonal unit cell.")
            print("Conditions for a Trigonal system: a = b = c and α = β = γ ≠ 90°")

    except ValueError as e:
        print(e)

if __name__ == "__main__":
```

```
main()
```

Python Code for Cubic 3-D Crystal Lattice

```
#Cubic
import numpy as np
import matplotlib.pyplot as plt

def plot_cell(ax, vertices, corner_vertices_color, atom_count, a, b, c, alpha, beta, gamma):
    edges = [ [vertices[0], vertices[4]], [vertices[0], vertices[1]], [vertices[1], vertices[2]], [vertices[1], vertices[5]], [vertices[2], vertices[3]], [vertices[2], vertices[6]],
[vertices[3], vertices[0]], [vertices[3], vertices[7]], [vertices[4], vertices[5]], [vertices[5], vertices[6]], [vertices[6], vertices[7]], [vertices[7], vertices[4]]
    ]

    corner_vertices = vertices[:8]
    other_vertices = vertices[8:]

    for edge in edges:
        xa, ya, za = zip(*edge)
        ax.plot(xa, ya, za, color='blue', linestyle='-')

    ax.scatter(corner_vertices[:, 0], corner_vertices[:, 1], corner_vertices[:, 2], color = corner_vertices_color, s = 700)

    ax.scatter(other_vertices[:, 0], other_vertices[:, 1], other_vertices[:, 2], color = 'purple', s = 700)

    #Annotations for a, b, c
    ax.text(a/2, 0, 0 , 'a', color = 'black', fontsize = 16)
    ax.text(vertices[2][0]/2, vertices[2][1]/2, 0, 'b', color='black', fontsize = 16)
    ax.text(vertices[3][0]/2, vertices[3][1]/2, vertices[3][2]/2, 'c', color = 'black', fontsize = 16)

    ax.set_xlabel('X')
    ax.set_ylabel('Y')
    ax.set_zlabel('Z')

    #Calculate and display the Number of Atoms per Unit Cell
    num_of_corner_vertices = len(corner_vertices)
    num_of_other_vertices = len(other_vertices)
    if num_of_other_vertices == 0:
      num_of_atoms_per_unit_cell = num_of_corner_vertices * (1 / num_of_corner_vertices)
      ax.text2D(-0.1, -0.09, f"Number of Atoms Per Unit Cell = {num_of_corner_vertices} * (1/{num_of_corner_vertices}) = {num_of_atoms_per_unit_cell} Atoms",
transform=ax.transAxes, fontsize=12, verticalalignment='top')
    elif num_of_other_vertices == 1:
      num_of_atoms_per_unit_cell = num_of_corner_vertices * (1 / num_of_corner_vertices) + num_of_other_vertices
      ax.text2D(-0.1, -0.09, f"Number of Atoms Per Unit Cell = {num_of_corner_vertices} * (1/{num_of_corner_vertices}) + 1 = {num_of_atoms_per_unit_cell}
Atoms", transform=ax.transAxes, fontsize=12, verticalalignment='top')
    else:
      num_of_atoms_per_unit_cell = num_of_corner_vertices * (1 / num_of_corner_vertices) + num_of_other_vertices * (1 / 2)
      ax.text2D(-0.1, -0.12, f"Number of Atoms Per Unit Cell = {num_of_corner_vertices} * (1/{num_of_corner_vertices}) + {num_of_other_vertices} * (1/2) =
{num_of_atoms_per_unit_cell} Atoms", transform=ax.transAxes, fontsize=12, verticalalignment='top')


    # Set equal aspect ratio and plot limits
    min_x, max_x = vertices[:, 0].min(), vertices[:, 0].max()
    min_y, max_y = vertices[:, 1].min(), vertices[:, 1].max()
    min_z, max_z = vertices[:, 2].min(), vertices[:, 2].max()

    maximum_range = max(max_x - min_x, max_y - min_y, max_z - min_z)

    ax.set_xlim(min_x - 0.1 * maximum_range, max_x + 0.1 * maximum_range)
    ax.set_ylim(min_y - 0.1 * maximum_range, max_y + 0.1 * maximum_range)
    ax.set_zlim(min_z - 0.1 * maximum_range, max_z + 0.1 * maximum_range)

def plot_primitive(ax, a, b, c, alpha, beta, gamma):
    vertices = np.array([ [0, 0, 0], [a, 0, 0], [a, b , 0], [0, b, 0], [0,0, c], [a, 0, c], [a, b, c], [0, b,c] ])
    plot_cell(ax, vertices, 'darkblue', 8, a, b, c, alpha ,beta, gamma)

def plot_BC(ax, a, b, c, alpha, beta, gamma):
    vertices = np.array([ [0, 0, 0], [a,0, 0], [a, b ,0], [0, b, 0], [0,0, c], [a, 0, c], [a, b, c], [0, b,c], [a/2, b/2, c/2] ])

    plot_cell(ax, vertices, 'darkblue', 9, a, b, c, alpha ,beta, gamma)

def plot_FC(ax, a, b, c, alpha, beta, gamma):
    vertices = np.array([ [0, 0, 0], [a,0, 0], [a, b , 0], [0, b, 0], [0,0, c], [a, 0, c], [a, b, c], [0, b,c], [a/2, 0, c/2], [a, b/2, c/2], [a/2, b, c/2], [0, b/2, c/2], [a/2, b/2, 0], [a/2,
b/2, c]])

    plot_cell(ax, vertices, 'darkblue', 14, a, b, c, alpha ,beta, gamma)

def determine_crystal(a, b, c, alpha, beta, gamma):
    if np.isclose(a, b) and np.isclose(b, c) and np.isclose(alpha, 90) and np.isclose(alpha, beta) and np.isclose(beta, gamma):
            return "Cubic"
    return "Unknown"
```

```
def main():
    try:
        print("Conditions for a Cubic System: a = b = c and α = β = γ = 90°")
        a = float(input("Enter the value of side a: "))
        b = float(input("Enter the value of side b: "))
        c = float(input("Enter the value of side c: "))
        alpha = float(input("Enter the value of angle alpha (in degree): "))
        beta = float(input("Enter the value of angle beta (in degree): "))
        gamma = float(input("Enter the value of angle gamma (in degree): "))

        if a <= 0 or b <= 0 or c <= 0 or not (0 < alpha < 180) or not (0 < beta < 180) or not (0 < gamma < 180):
            raise ValueError("Invalid Input, side lengths must be positive and angles must be between 0 and 180 degree.")

        crystal_system = determine_crystal(a, b, c, alpha, beta, gamma)

        fig = plt.figure(figsize=(15, 12))

        # Show parameters on plot
        parameters = f"a = {a}, b = {b}, c = {c}\nα = {alpha}°, β = {beta}°, γ = {gamma}°"
        fig.text(0.7, 0.12, parameters, ha='center', fontsize=12, verticalalignment='bottom')

        titles = []
        plot_count = 0

        if crystal_system == "Cubic":
            titles = ["Primitive Cubic", "Body-Centered Cubic (BCC)", "Face-Centered Cubic (FCC)" ]
            plot_functions = [plot_primitive, plot_BC, plot_FC]
            plot_count = 3
        else:
            print("The provided parameters do not form a Cubic Unit Cell.")
            print("Conditions for a Cubic system: a = b = c and α = β = γ = 90°")
            return

        for i in range(plot_count):
            ax = fig.add_subplot(2, 2, i+1, projection = '3d')
            plot  functions[i](ax, a, b, c, alpha, beta, gamma)
            ax.text2D(0.5, -0.3, titles[i], transform = ax.transAxes, ha = 'center', fontsize = 18, weight = 'bold')

        plt.tight_layout()
        plt.show()

    except ValueError as e:
        print(e)

if __name__ == "__main__":
    main()
```

Python Code For Tetragonal 3-D Crystal Lattice

```
#Tetragonal
import numpy as np
import matplotlib.pyplot as plt

def plot  cell(ax, vertices, corner  vertices  color, atom  count, a, b, c, alpha, beta, gamma):
    edges = [ [vertices[0], vertices[4]], [vertices[0], vertices[1]], [vertices[1], vertices[2]], [vertices[1], vertices[5]], [vertices[2], vertices[3]], [vertices[2], vertices[6]],
[vertices[3], vertices[0]], [vertices[3], vertices[7]], [vertices[4], vertices[5]], [vertices[5], vertices[6]], [vertices[6], vertices[7]], [vertices[7], vertices[4]]
    ]

    corner_vertices = vertices[:8]
    other_vertices = vertices[8:]

    for edge in edges:
        xa, ya, za = zip(*edge)
        ax.plot(xa, ya, za, color='blue', linestyle='-')

    ax.scatter(corner_vertices[:, 0], corner_vertices[:, 1], corner_vertices[:, 2], color = corner_vertices_color, s = 700)

    ax.scatter(other_vertices[:, 0], other_vertices[:, 1], other_vertices[:, 2], color = 'purple', s = 700)

    #Annotations for a, b, c
    ax.text((vertices[0][0] + vertices[1][0])/2 , (vertices[0][1] + vertices[1][1])/2 , (vertices[0][2] + vertices[1][2])/2, 'a', color = 'black', fontsize = 16 )
    ax.text((vertices[0][0] + vertices[2][0])/1.1 , (vertices[0][1] + vertices[2][1])/2 + 0.5 , (vertices[0][2] + vertices[2][2])/2, 'b', color = 'black', fontsize = 16 )
    ax.text((vertices[0][0] + vertices[4][0])/2 , (vertices[0][1] + vertices[4][1])/2 , (vertices[0][2] + vertices[4][2])/2, 'c', color = 'black', fontsize = 16 )

    ax.set_xlabel('X')
    ax.set_ylabel('Y')
    ax.set_zlabel('Z')

    #Calculate and display the Number of Atoms per Unit Cell
    num_of_corner_vertices = len(corner_vertices)
```

```
    num_of_other_vertices = len(other_vertices)

    if num_of_other_vertices == 0:
        num_of_atoms_per_unit_cell = num_of_corner_vertices * (1 / num_of_corner_vertices)
        ax.text2D(0.05, -0.09, f"Number of Atoms Per Unit Cell = {num_of_corner_vertices} * (1/{num_of_corner_vertices}) = {num_of_atoms_per_unit_cell} Atoms",
transform=ax.transAxes, fontsize=12, verticalalignment='top')
    elif num_of_other_vertices == 1:
        num_of_atoms_per_unit_cell = num_of_corner_vertices * (1 / num_of_corner_vertices) + num_of_other_vertices
        ax.text2D(0.05, -0.09, f"Number of Atoms Per Unit Cell = {num_of_corner_vertices} * (1/{num_of_corner_vertices}) + 1 = {num_of_atoms_per_unit_cell}
Atoms", transform=ax.transAxes, fontsize=12, verticalalignment='top')
    else:
        num_of_atoms_per_unit_cell = num_of_corner_vertices * (1 / num_of_corner_vertices) + num_of_other_vertices * (1 / 2)
        ax.text2D(0.05, 0.95, f"Number of Atoms Per Unit Cell = {num_of_corner_vertices} * (1/{num_of_corner_vertices}) + {num_of_other_vertices} * (1/2) =
{num_of_atoms_per_unit_cell} Atoms", transform=ax.transAxes, fontsize=12, verticalalignment='top')

     #Set equal aspect  ratio
    maximum_range = np.array([vertices[:, 0].max() - vertices[:, 0].min(), vertices[:, 1].max() - vertices[:, 1].min(), vertices[:, 2].max() - vertices[:, 2].min()]).max()
/2.0
    x_mid = (vertices[:, 0].max() + vertices[:, 0].min()) * 0.5
    y_mid = (vertices[:, 1].max() + vertices[:, 1].min()) * 0.5
    z_mid = (vertices[:, 2].max() + vertices[:, 2].min()) * 0.5
    ax.set_xlim(x_mid - maximum_range, x_mid + maximum_range)
    ax.set_ylim(y_mid - maximum_range, y_mid + maximum_range)
    ax.set_zlim(z_mid - maximum_range, z_mid + maximum_range)

def plot_primitive(ax, a, b, c, alpha, beta, gamma):
    vertices = np.array([ [0, 0, 0], [a, 0, 0], [a, b , 0], [0, b, 0], [0,0, c], [a, 0, c], [a, b, c], [0, b,c] ])
    plot_cell(ax, vertices, 'darkblue', 8, a, b, c, alpha, beta, gamma)

def plot_BC(ax, a, b, c, alpha, beta, gamma):
    vertices = np.array([ [0, 0, 0], [a,0, 0], [a, b , 0], [0, b, 0], [0,0, c], [a, 0, c], [a, b, c], [0, b,c], [a/2, b/2, c/2] ])
    plot_cell(ax, vertices, 'darkblue', 9, a, b, c, alpha, beta, gamma)

def determine_crystal(a, b, c, alpha, beta, gamma):
     if (np.isclose(a, b) or np.isclose(b, c) or np.isclose(a, c)) and np.isclose(alpha, 90) and np.isclose(alpha, beta) and np.isclose(alpha, gamma):
      return "Tetragonal"
     else:
      return "Unknown"

def main():
    try:
        print("Conditions for a Tetragonal system: a = b ≠ c and α = β = γ = 90°")
        a = float(input("Enter the value of side a: "))
        b = float(input("Enter the value of side b: "))
        c = float(input("Enter the value of side c: "))
        alpha = float(input("Enter the value of angle alpha (in degree): "))
        beta = float(input("Enter the value of angle beta (in degree): "))
        gamma = float(input("Enter the value of angle gamma (in degree): "))

        if a <= 0 or b <= 0 or c <= 0 or not (0 < alpha < 180) or not (0 < beta < 180) or not (0 < gamma < 180):
            raise ValueError("Invalid Input, side lengths must be positive and angles must be between 0 and 180 degree.")

        crystal_system = determine_crystal(a, b, c, alpha, beta, gamma)

        fig = plt.figure(figsize=(15, 10))
        # Show parameters on plot
        parameters = f"a = {a}, b = {b}, c = {c}\nα = {alpha}°, β = {beta}°, γ = {gamma}°"
        fig.text(0.47, 0.6, parameters, fontsize=12, verticalalignment='center', horizontalalignment='center')

        titles = []
        plot_count = 0

        if crystal_system == "Tetragonal":
            titles = ["Primitive Tetragonal", "Body-Centered Tetragonal"]
            plot_functions = [plot_primitive, plot_BC]
            plot_count = 2
        else:
            print("The provided parameters do not form a Tetragonal unit cell.")
            print("Conditions for a Tetragonal system: a = b ≠ c and α = β = γ = 90°")
            return

        for i in range(plot_count):
            ax = fig.add_subplot(2, 2, i+1, projection = '3d')
            plot_functions[i](ax, a, b, c, alpha, beta, gamma)
            ax.text2D(0.5, -0.3, titles[i], transform = ax.transAxes, ha = 'center', fontsize = 18, weight = 'bold')

        plt.tight_layout()
        plt.show()

    except ValueError as e:
        print(e)
```

```
if __name__ == "__main__":
    main()
```

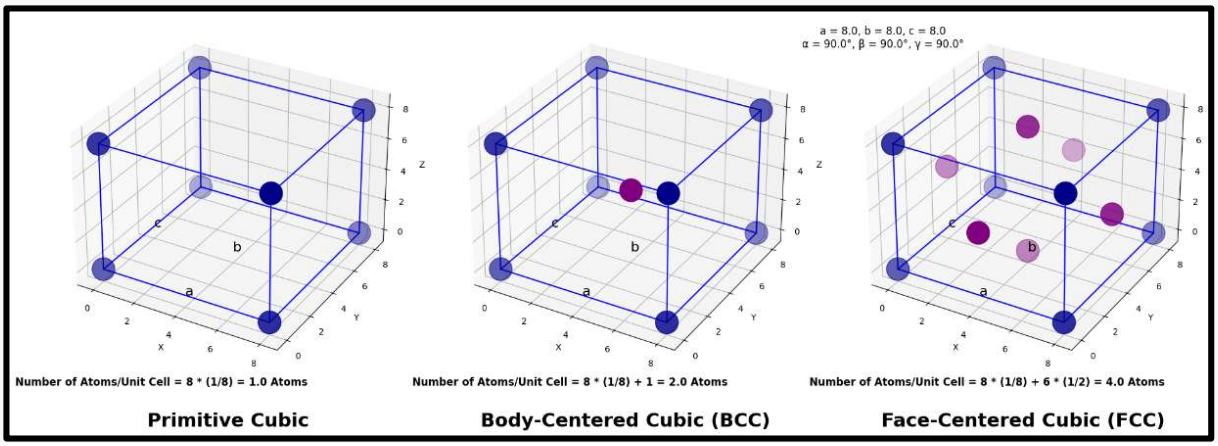

Fig.-7: 3D Crystal Lattice of Cube

Python Code for Hexagonal 3-D Crystal Lattice

```
#Hexagonal
import numpy as np
import matplotlib.pyplot as plt

def plot_hexagonal(ax, a, c, b, alpha, beta, gamma):
    # Define the vertices for a hexagonal unit cell
    vertices = np.array([
        [a, 0, 0],
        [a/2, a*np.sqrt(3)/2, 0],
        [-a/2, a*np.sqrt(3)/2, 0],
        [-a, 0, 0],
        [-a/2, -a*np.sqrt(3)/2, 0],
        [a/2, -a*np.sqrt(3)/2, 0],
        [a, 0, c],
        [a/2, a*np.sqrt(3)/2, c],
        [-a/2, a*np.sqrt(3)/2, c],
        [-a, 0, c],
        [-a/2, -a*np.sqrt(3)/2, c],
        [a/2, -a*np.sqrt(3)/2, c],
        [0, 0, 0],
        [0, 0, c]
    ])

    # Define the edges of hexagonal unit cell
    edges = [
        [vertices[5], vertices[0]], [vertices[10], vertices[11]], [vertices[11], vertices[6]], [vertices[0], vertices[6]],
        [vertices[4], vertices[10]], [vertices[4], vertices[5]], [vertices[5], vertices[11]], [vertices[10], vertices[13]],
        [vertices[13], vertices[6]]
    ]

    dotted_edges = [
        [vertices[0], vertices[1]], [vertices[1], vertices[2]], [vertices[2], vertices[3]], [vertices[3], vertices[4]],
        [vertices[6], vertices[7]], [vertices[7], vertices[8]], [vertices[8], vertices[9]], [vertices[9], vertices[10]],
        [vertices[1], vertices[7]], [vertices[2], vertices[8]], [vertices[3], vertices[9]]
    ]

    for edge in edges:
        xa, ya, za = zip(*edge)
        ax.plot(xa, ya, za, color='blue', linestyle='-')

    for edge in dotted_edges:
        xa, ya, za = zip(*edge)
        ax.plot(xa, ya, za, color='blue', linestyle='--')

    #Annotations for a, b, c
    ax.text(a/2, -b/3, -c/6, 'a', color='black', fontsize=16)
    ax.text(a/4, a*np.sqrt(3)/4, c/7, 'b', color='black', fontsize=16)
    ax.text(a, 0, c/2, 'c', color='black', fontsize=16)

    ax.scatter(vertices[:, 0], vertices[:, 1], vertices[:, 2], color = 'darkblue', s = 800)

    center_vertex = np.mean(vertices, axis = 0)

    height_of_triangle = a / 4
    new_vertices = np.array([
```

```
        center_vertex + np.array([height_of_triangle, 0, 0]),
        center_vertex + np.array([-height_of_triangle/2, height_of_triangle*np.sqrt(3)/2, 0]),
        center_vertex + np.array([-height_of_triangle/2, -height_of_triangle*np.sqrt(3)/2, 0])
    ])

    vertices = np.vstack([vertices, new_vertices ])

    # Plotting the new vertices
    ax.scatter(new_vertices[:, 0], new_vertices[:, 1], new_vertices[:, 2], color = 'purple', s = 800)

    ax.set_xlabel('X')
    ax.set_ylabel('Y')
    ax.set_zlabel('Z')

    #Calculate and display the Number of Atoms per Unit Cell
    num_of_vertices = 12
    num_of_atoms_per_unit_cell = num_of_vertices * (1 / 6) + 2* (1/2) + 3
    ax.text2D(0.05, -0.01, f"Number of Atoms Per Unit Cell = {num_of_vertices} * (1/{6}) + 2* (1/2) + 3 = {num_of_atoms_per_unit_cell} Atoms", transform =
ax.transAxes, fontsize=12, verticalalignment = 'top')

    #Set equal aspect  ratio
    maximum_range = np.array([vertices[:, 0].max() - vertices[:, 0].min(), vertices[:, 1].max() - vertices[:, 1].min(), vertices[:, 2].max() - vertices[:, 2].min()]).max()
/2.0
    x_mid = (vertices[:, 0].max() + vertices[:, 0].min()) * 0.5
    y_mid = (vertices[:, 1].max() + vertices[:, 1].min()) * 0.5
    z_mid = (vertices[:, 2].max() + vertices[:, 2].min()) * 0.5
    ax.set_xlim(x_mid - maximum_range, x_mid + maximum_range)
    ax.set_ylim(y_mid - maximum_range, y_mid + maximum_range)
    ax.set_zlim(z_mid - maximum_range, z_mid + maximum_range)

     # Show parameters
    parameters = f"a = {a}, b = {b}, c = {c} \nα = {alpha}°, β = {beta}°, γ = {gamma}°"
    ax.text2D(0.05, 0.01, parameters, transform=ax.transAxes, fontsize=12, verticalalignment='bottom')

def determine_crystal(a, b, c, alpha, beta, gamma):
    if np.isclose(a, b) or np.isclose(b, c) or np.isclose(a, c) and (np.isclose(alpha, 90) and np.isclose(beta, 90) and np.isclose(gamma, 120)) or (np.isclose(alpha, 90) and
np.isclose(gamma, 90) and np.isclose(beta, 120)) or (np.isclose(gamma, 90) and np.isclose(beta, 90) and np.isclose(alpha, 120)):
            return "Hexagonal"
    return "Unknown"

def main():
    try:
        print("Conditions for a Hexagonal System: a = b ≠ c and α = β = 90°, γ = 120°")
        a = float(input("Enter the value of side a: "))
        b = float(input("Enter the value of side b: "))
        c = float(input("Enter the value of side c: "))
        alpha = float(input("Enter the value of angle alpha (in degree): "))
        beta = float(input("Enter the value of angle beta (in degree): "))
        gamma = float(input("Enter the value of angle gamma (in degree): "))

        if a <= 0 or b <= 0 or c <= 0 or not (0 < alpha < 180) or not (0 < beta < 180) or not (0 < gamma < 180):
            raise ValueError("Invalid Input, side lengths must be positive and angles must be between 0 and 180 degree.")

        crystal_system = determine_crystal(a, b, c, alpha, beta, gamma)

        fig = plt.figure(figsize=(8, 6))

        if crystal_system == "Hexagonal":
            ax = fig.add_subplot(111, projection = '3d')
            plot_hexagonal(ax, a, c,b, alpha, beta, gamma)
            ax.text2D(0.5, -0.09, crystal_system, transform=ax.transAxes, ha='center', fontsize=18, weight='bold')
            plt.subplots_adjust(top=1)
            plt.show()
            return
        else:
            print("The provided parameters do not form a Hexagonal unit cell.")
            print("Conditions for a Hexagonal System: a = b ≠ c and α = β = 90°, γ = 120°")

    except ValueError as e:
        print(e)

if __name__ == "__main__":
    main()
```

CONCLUSION

This paper demonstrates the effectiveness of using Python as a versatile and powerful tool for drawing Bravais lattices in both two and three dimensions. By leveraging Python libraries viz Matplotlib and Pyplot, the authors present an approach that is not only accurate and efficient but also user-friendly for a broad

spectrum of users. Transitioning from hand drawing to computational visualization offers several benefits, including improved accuracy, ease of use, and the ability to manage complex structures effortlessly.

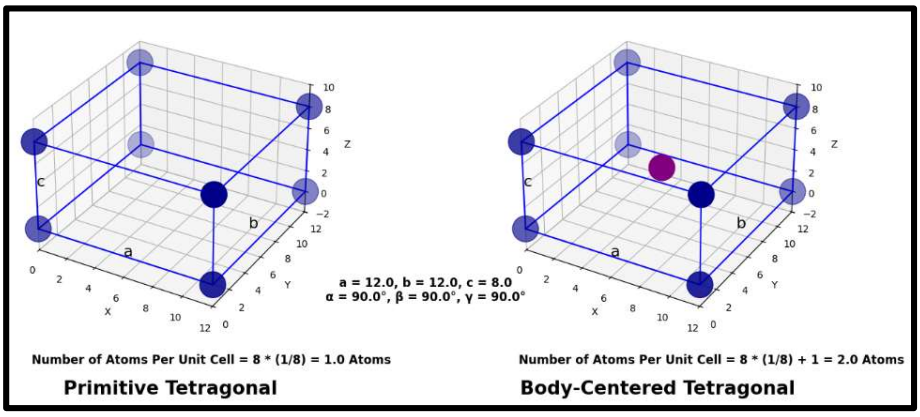

Fig.-8: 3D Crystal Lattice of Tetragonal

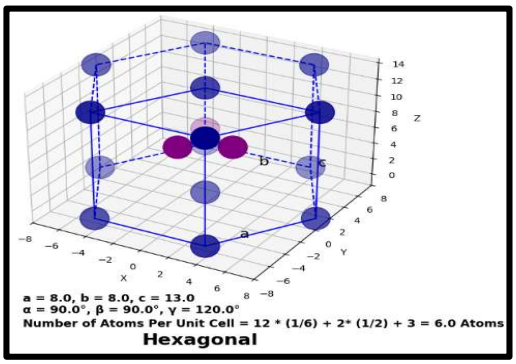

Fig.-9: 3D Crystal Lattice of Hexagonal

With the introduction of Python as a computational / theoretical tool at the undergraduate level, this paper would enhance students' learning experiences by providing practical programming skills in crystallography alongside fundamental scientific knowledge. By simplifying the process of visualizing and understanding the crystal structures, this integration promotes a deeper understanding of the concepts. Alongside this, Python's adaptability and user-friendly nature make it an excellent alternative to conventional methods of creating Bravais lattices. By incorporating these tools in educational settings, advanced computational resources can be made more widely accessible to students and researchers, equipping them with the skills needed to succeed in the fields of material science and chemistry. The authors also aim to apply symmetry elements to these lattices to generate 32-point groups and 230 space groups. This paper highlights Python's potential to revolutionize the teaching and analysis of crystallographic structures, paving the way for enhanced scientific visualization via computational chemistry.

ACKNOWLEDGEMENTS

Dr. Hemant Verma would like to thank Hindu College, University of Delhi for all its support. Prof. Sarita Passey and Dr. Jyoti Singh would like to thank Zakir Husain Delhi College for all their support.

CONFLICT OF INTERESTS

The authors declare that there is no conflict of interest.

AUTHOR CONTRIBUTIONS

All the authors contributed significantly to the manuscript, participating in writing/ reviewing/ editing and approved the final draft for publication. The research profile of the authors can be verified from their ORCHID ids, given below:

Sarita Passey http://orchid.org/0000-0001-9856-9163

Hemant Verma http://orchid.org/0009-0008-8713-3450
Jyoti Singh http://orchid.org/0009-0001-2640-6549
Anjali https://orcid.org/0009-0001-2118-2291

REFERENCES

1. Cantor, Brian, 'Bravais Lattices: Crystals', *The Equations of Materials* (Oxford, 2020; online edn, Oxford Academic, 17 Sept, 2020), https://doi.org/10.1093/oso/9780198851875.003.0001, accessed 25 July 2024
2. C. Giacovazzo, H. Monaco, G. Artioli, D. Viterbo, M. Milanesio, G. Gilli, P. Gilli, G. Zanotti, G. Ferraris, M. Catti, Fundamentals of Crystallography, (2011), Oxford University Press, https://doi.org/10.1093/acprof:oso/9780199573653.001.000
3. B. D. Cullity and S. R. Stock, Elements of X-Rays Diffraction (3rd ed.), Pearson, Third Edition. Prentice-Hall p. 31,90, (2001).
4. W. D. Callister Jr., D. G. Rethwisch, Materials Science and Engineering: An Introduction (10th edition), Hoboken, NJ: Wiley, p. 31,90 (2018).
5. R.C. Powell, 2010, Tensor Properties of Crystals. In: Symmetry, Group Theory, and the Physical Properties of Crystals. Lecture Notes in Physics, Vol 824. Springer, New York, NY. https://doi.org/10.1007/978-1-4419-7598-0_3
6. K. Valli Priyadharshini, A. Vijay, K. Swaminathan, T. Avudaiappan, V. Banupriya, *Materials Today: Proceedings*, **69(3)**, 710(2022), https://doi.org/10.1016/j.matpr.2022.07.134
7. H. Liang, V. Stanev, A. G. Kusne and I. Takeuchi, *Physical Review Materials*, **4**, 123802(2020) https://doi.org/10.1103/PhysRevMaterials.4.123802
8. Y. Li, R. Dong, W. Yang and J. Hu, *Computational Materials Science*, **198**, 110686(2021), https://doi.org/10.1016/j.commatsci.2021.110686
9. Y. Li, W. Yang, R. Dong and J. Hu, *ACS Omega*, **6(17)**, 11585(2021), https://doi.org/10.1021/acsomega.1c00781
10. S. Jarin, Y. Yuan, M. Zhang, M. Hu, M. Rana, M. Wang and S. Knibbe, *Crystals*, **12(11)**, 1570(2022), https://doi.org/10.3390/cryst12111570
11. J. Balasingham, V. Zamaraev and V. Kurlin, *Integrating Materials and Manufacturing Innovation*, **13**, 555(2024), https://doi.org/10.1007/s40192-024-00351-9
12. F. Libonati, S. Graziosi, F. Ballo, M. Mognato and G. Sala, *ACS Biomaterials Science and Engineering*, **9(7)**, 3935(2023), https://doi.org/10.1021/acsbiomaterials.0c01708
13. D. Yavas, Q. Liu, Z. Zhang, and D. Wu, *Materials and Design*, **217**, 110613(2022), https://doi.org/10.1016/j.matdes.2022.110613
14. S. Kundu, K. Chakraborty, and A. Das, *Condensed Matter* (2024), https://doi.org/10.48550/arXiv.2407.08808
15. K. Momma, F. Izumi, *Journal of Applied Crystallography*, **44**, 1272(2011), https://doi.org/10.1107/S0021889811038970
16. D. C. Palmer, *Zeitschrift für Kristallographie - Crystalline Materials*, **230(9-10)**, (2015), https://doi.org/10.1515/zkri-2015-1869
17. A. Kokalj, XCrySDen, *Journal of Molecular Graphics and Modelling*, **17(3-4)**, 176(1999), https://doi.org/10.1016/S1093-3263(99)00028-5
18. W. McKinney, Python for Data Analysis, 2nd ed., ISBN-13: 978-1491957660, Publisher: O'Reilly, (2017)
19. B. Dahlgren, *Journal of Open Source Software*, **3(24)**, 565(2018), https://doi.org/10.21105/joss.00565
20. J. Gabirondo-Lopez, I. Gabirondo-Lopez, E. S. Tasci, G. Madariaga, *Journal of Applied Crystallography* **57**, 1640(2024), https://doi.org/10.1107/S1600576724007908

[RJC-9028/2024]