1.5em 0pt

Atomistic Simulation in Materials Modeling

Instructor: Qiang Zhu

August 6, 2025

Contents

Preface

This book was originally created based on a graduate course that I taught at the University of North Carolina at Charlotte beginning in Fall 2024. In that course, I aim to give engineering students an introduction to modern atomistic computer simulation techniques. It primarily focuses on two approaches: molecular dynamics (MD) and electronic structure calculation based on density functional theory (DFT). Assuming that most engineering students do not have the necessary background in solid state physics, computational chemistry and statistical mechanics, I expect that students, after taking this course, can understand the physics behind each computational model and know how to apply these techniques to study real materials and their accompanying processes and phenomena.

The major difference compared to other available books or lecture notes is the hands-on, practical approach to learning through coding and simulation. For each technique covered, I have crafted a series of carefully designed examples - from simulating liquid argon with MD to calculating the electronic structure of H_2 molecules, silicon crystal with DFT - that students implement themselves using relatively simple Python code. Each chapter is accompanied by an interactive Python Jupyter Notebook that runs either on Google Colab or locally, allowing students to experiment with the code, modify parameters, and observe the results in real-time. These hands-on exercises typically take under 20 minutes to complete, making them ideal for reinforcing theoretical concepts through practical implementation. This approach ensures students not only understand the underlying theory but gain direct experience implementing these methods, preparing them to tackle real-world materials science challenges using modern high-performance computing codes.

How to Use This Book?
This textbook is organized into a progressive learning path. For beginners in atomistic simulation, I recommend first reviewing the essential mathematical concepts (primarily calculus and linear algebra) and their Python implementation in Appendix A.

The core content begins with **molecular dynamics simulation** (Chapters 1-7), covering fundamental concepts and advancing to enhanced sampling techniques like metadynamics. It begins by introducing Molecular Dynamics simulation with the foundational NVE ensemble (constant number of particles, volume, and energy) in Chapter 1, using the example of liquid Argon described by the Lennard-Jones potential. Essential aspects of the MD workflow are explained, including initialization of positions and velocities, force calculation, and integration using the Velocity Verlet algorithm. Chapter 2 advances the simulation capabilities by introducing thermostats for controlling temperature and simulating systems in the NVT ensemble (constant N, V, and T), discussing methods like the Anderson and Langevin thermostats. Pressure control is added in Chapter 3 with the introduction of barostats for the NPT ensemble (constant N, P, and T), covering techniques such as the Berendsen and Parrinello-Rahman barostats which adjust the simulation box. Chapter 4 then provides an introduction to using the powerful and widely-used LAMMPS package for running more complex simulations. With simulations in hand, Chapter 5 focuses on structural characterization through analysis techniques like visualization and computing the Radial Distribution Function (RDF) and Vibrational Density of States (VDOS) to understand atomic arrangements and dynamics. Chapter 6 delves into calculating transport properties, such as diffusion (using Mean Squared Displacement - MSD) and thermal conductivity, often utilizing equilibrium fluctuations and linear response theory. Finally, Chapter 7 introduces enhanced sampling techniques, specifically Metadynamics, designed to overcome the limitations of standard MD in sampling rare events and exploring complex free energy landscapes by adding bias potentials based on chosen collective variables. Well-Tempered Metadynamics is presented as an improvement for better convergence and free energy estimation.

For **electronic structure calculations**, students should first master spherical harmonics and Ewald summation (Appendices B and C) before proceeding to the density functional theory chapters (8-12). In chapter 8, it introduces the Schrödinger equation and transitions to DFT as a solution for multi-electron systems, explaining the fundamental Hohenberg-Kohn theorems and the practical Kohn-Sham equations solved via an iterative self-consistent field (SCF) procedure that updates the electron density based on an effective potential. Chapter 9 applies the Kohn-Sham formalism to the H_2 molecule using a numerical grid approach, guiding the reader through developing code to iteratively solve the equations within an SCF loop and highlighting the computational expense of diagonalizing large matrices. Next, the localized basis sets like Gaussian-Type Orbitals (GTOs) are introduced in Chapter 10 to compute integrals for constructing the Hamiltonian matrix and perform the SCF calculation more efficiently. Our journey continues with Chapter 11, where we explore the electronic structure calculation in periodic systems, specifically the silicon crystal with concepts like Bloch's Theorem and basis sets such as Tight Binding and Plane Waves, introduces the Brillouin Zone, and demonstrates band structure calculations using the Empirical Pseudopotential Method. Chapter 12 combines DFT with the plane-wave basis set and pseudopotentials for accurate periodic system simulations, discussing pseudopotentials, Brillouin Zone sampling, Hamiltonian construction, the plane-wave SCF procedure, and the calculation of nuclear-nuclear interaction using Ewald summation.

The book concludes with advanced topics crucial for materials characterization: phonon calculations for understanding lattice dynamics (Chapter 13) and local structural environment analysis for describing atomic arrangements (Chapter 14). Chapter 14 requires familiarity with Wigner-D matrices and Clebsch-Gordan coefficients, covered in Appendix D.

For students interested in exploring this rapidly evolving field further, Appendix E provides a curated collection of additional references and resources.

Acknowledgments
I would like to thank my students, colleagues, and mentors who have contributed to my knowledge and understanding of the field, and who have helped refine my teaching. This work would not have been possible without their support and insights. Additionally, I am grateful to the creators of open-source platforms and tools, particularly Python and Jupyter, which make computational science accessible and empower the next generation of researchers. Finally, I would like to thank my family (Rachel Chen, Gary Zhu and

Jerry Zhu) for their unwavering support and encouragement.

While preparing the materials, I was strongly influenced by two excellent textbooks:

- *Understanding Molecular Simulation: From Algorithms to Applications*, by Daan Frenkel and Berend Smit, 3rd Edition
- *Electronic Structure*, by Richard Martin, 2nd Edition

I studied both books back in my graduate student days. Undoubtedly, both books have influenced many researchers in my generation. I hope that this lecture note may serve as a resource that enables students and young researchers to develop a deeper understanding of molecular simulation and electronic structure methods and apply them confidently to their own studies.

Qiang Zhu
Waxhaw, NC, 2025

About the Author
Qiang Zhu is an Associate Professor in the Department of Mechanical Engineering and Energy Science at the University of North Carolina at Charlotte. Prior to joining UNC Charlotte, he was an Associate Professor in the Department of Physics and Astronomy at the University of Nevada Las Vegas between 2016 and 2023. Qiang obtained his Bachelor degree in Materials Science and Engineering from Beijing University of Aeronautics and Astronautics in China between in 2007, and the Ph.D. in Mineral Physics under Prof. Artem Oganov's supervision at Stony Brook University in 2013. He is also the recipients of 2021 Early Career Awards from both the National Science Foundation and the U.S. Department of Energy.

1. Simulating the NVE Ensemble

1.1. Early History of Computer Simulation

The Los Alamos MANIAC (Mathematical Analyzer, Numerical Integrator, and Computer) became operational in 1952, marking a significant milestone in the history of computing. Nicholas Metropolis, one of its most notable early users, pioneered the development of the Monte Carlo method—a statistical technique that leverages random sampling to solve complex mathematical problems. This method soon became foundational in numerous fields, from physics and chemistry to finance, due to its effectiveness in handling problems with high-dimensional spaces and probabilistic elements.

The advent of computers also opened new possibilities for exploring fundamental scientific problems, particularly in materials science. Many material systems consist of vast numbers of atoms or molecules; understanding their properties requires innovative approaches. Traditionally, researchers relied on analytical methods, such as thermodynamics and statistical mechanics, which were developed to study classical systems like ideal gases, the Ising model, ferromagnetic phase transitions, and alloys. Although these methods provided valuable insights, they often lacked atomic-level detail. To model these systems directly, scientists used hands-on methods like Buffon's needle experiment to estimate values such as π, Bernal's ball-bearing model [1] to study dense packing in liquids and glasses, and Kitaigorodskii's structure seeker [2] to explore molecular arrangements. While insightful, these direct approaches were often labor-intensive and time-consuming.

The invention of computers transformed this landscape, making it feasible to simulate atomic and molecular systems with unprecedented precision and scale. Using computational power, researchers could not only automate these laborious processes but also model the time evolution of atomic and molecular systems, which is crucial for understanding dynamic properties. This breakthrough laid the foundation for modern simulation techniques, enabling scientists to analyze material behavior at the atomic level, predict novel materials, and gain insights beyond the reach of traditional experimental methods.

1.2. Molecular Dynamics Simulation

Molecular Dynamics (MD) Simulation is a technique to directly study the atomic evolution of atoms or molecules in a material system based on Newtonian Dynamics. Shortly after the invention of computers, researchers started to develop different kinds of MD simulation techniques to study different systems. Some notable examples include:

1. Phase transition of hard spheres by Alder and Wainwright (1956) [3],

2. Dynamics of radiation damage by Gibson et al. (1959) [4]

3. Correlations in Liquid Argon by Rahman (1964) [5]

Thanks to the continuous development of MD simulation techniques, we can now surpass experimental limitations to study materials at the atomic or molecular scale. Certain material properties—such as atomic motions, vibrational modes, and interatomic forces—are challenging to measure directly due to constraints in resolution and accessibility. MD simulations allow researchers to observe these atomic-level details with remarkable precision, offering insights that complement and extend beyond experimental findings.

More importantly, MD simulations provide a unique window into the fundamental processes governing material behavior by enabling scientists to observe how atoms and molecules move, interact, and respond to various conditions. These simulations facilitate the study of microscopic events such as diffusion, fracture, melting, and crystallization, providing detailed data on the evolution of these processes over time. Additionally, MD simulations deepen our understanding of how macroscopic properties—like mechanical strength, thermal conductivity, and chemical reactivity—emerge from atomic-scale interactions. By analyzing simulated trajectories, scientists can uncover the principles that govern material stability, predict novel material properties, and improve the design of materials for targeted applications.

In the following chapters, we will delve into the foundational concepts of MD simulation techniques and explore how these principles are numerically implemented in computer code.

1.3. A First MD Simulation under the NVE Ensemble

Following Rahman's seminal work in 1964 [5], we aim to study liquid Argon—a relatively simple yet physically meaningful system. This example will guide us through the entire process of setting up and running an MD simulation from scratch, providing a hands-on introduction to key concepts and techniques. In this simplest setting, we will simulate an isolated system without thermal interaction with any other systems. This is referred to as the **microcanonical (NVE)** ensemble in statistical physics. In this ensemble, the number of particles N, volume V, and total energy E are conserved throughout the simulation. The system evolves according to Newton's equations of motion, which govern the dynamics of particles based on their positions and velocities. The goal is to observe how the system behaves over time, allowing us to extract meaningful physical properties from the simulation data.

1.3.1 The MD Workflow

Fig. 1.1 represents a typical loop in an MD simulation, where the system evolves over time by repeatedly updating the positions, velocities, and forces of particles until a specified termination condition is met. The workflow follows these steps:

1. Initialization $\{\mathbf{R}, \mathbf{V}\}$: It sets up the initial conditions for the simulation. Initial positions $\mathbf{R}$ and velocities $\mathbf{V}$ of all particles are defined.

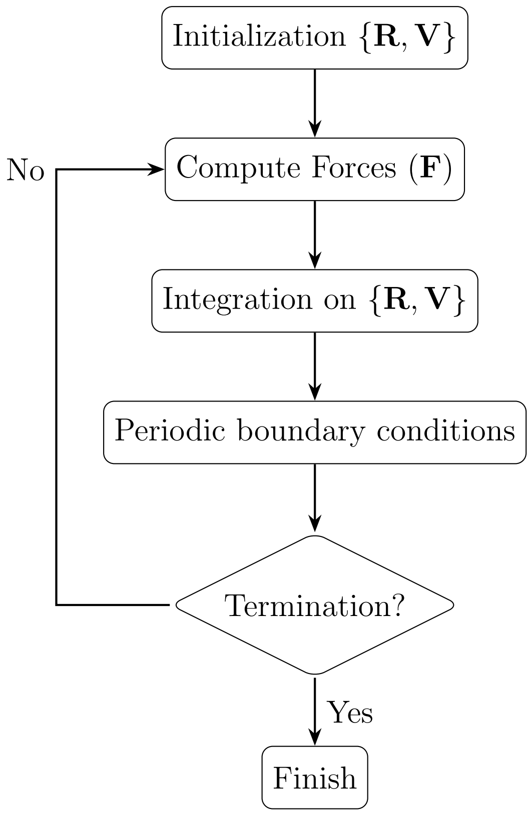

Initialization $\{\mathbf{R}, \mathbf{V}\}$
No
Compute Forces $(\mathbf{F})$
Integration on $\{\mathbf{R}, \mathbf{V}\}$
Periodic boundary conditions
Termination?
Yes
Finish

Figure 1.1: The workflow of a NVE MD simulation.

2. Compute Forces $(\mathbf{F})$: The forces $\mathbf{F}$ acting on each particle are calculated based on their positions $\mathbf{R}$ and an interaction model.

3. Integration on $\{\mathbf{R}, \mathbf{V}\}$: Using the computed forces, the integration step updates the positions $\mathbf{R}$ and velocities $\mathbf{V}$ of the particles over a small time step.

4. Periodic Boundary Conditions (if necessary): Periodic boundary conditions are applied to simulate an infinite system by making particles that exit one side of the simulation box re-enter from the opposite side.

5. Termination Condition: This decision block checks whether a maximum number of iterations is reached. If no, the workflow loops back to the Compute Forces step; otherwise, the process advances to the Finish step, ending the simulation.

This workflow illustrates the iterative nature of MD simulations, where the system's state is continuously updated to model its time evolution. Each iteration moves the system forward by a small time increment, ultimately generating a trajectory that can be analyzed to understand the physical properties and behaviors of the simulated material.

1.3.2 Initialization

The initialization phase involves setting up both the positions and velocities of the particles in the system. This step is crucial, as it establishes the initial conditions from which the simulation will evolve.

- **Atomic Positions:** The placement of particles depends on the phase of the material being simulated. If the system is a crystalline solid, particles should be placed on a compatible lattice, such as a face-centered cubic (FCC) or body-centered cubic (BCC) lattice, depending on the material's structure. This initial arrangement ensures that the particles reflect the regular, repeating structure of a crystal. In contrast, for a liquid or amorphous (non-crystalline) material, a random distribution of particles may be used, but care must be taken to avoid overlapping particles. For liquids, starting from a dense random packing or using a pre-equilibrated configuration can be helpful.

- **Velocities:** Initial velocities of particles should follow the well-known **Maxwell-Boltzmann distribution**, which describes the probability distribution of particle speeds in a system at thermal equilibrium. The distribution is given by:

$$p(v) = 4\pi \left(\frac{m}{2\pi k_B T} \right)^{3/2} v^2 \exp\left(-\frac{mv^2}{2k_B T} \right), \tag{1.1}$$

  where v is the speed of a particle, m is the particle mass, k_B is the Boltzmann constant, and T is the temperature.

  To approximate this distribution, we can sample each velocity component from a normal (Gaussian) distribution with a mean of zero and a standard deviation related to the desired temperature and particle mass. Specifically, the standard deviation σ_v for each component is:

$$\sigma_v = \sqrt{\frac{k_B T}{m}}. \tag{1.2}$$

  This approach ensures that the velocities follow the Maxwell-Boltzmann distribution in magnitude, and the mean velocity is adjusted to zero to satisfy conservation of momentum in the system.

The following code describes how to generate initial velocities and compare it with the theoretical Maxwell-Boltzmann distribution.

```python
import numpy as np

def generate_velocities(N, T, M):
    """
    Generate initial velocities from Maxwell-Boltzmann distribution.

    Parameters:
    -----------
    N (int): The number of particles in the system.
    T (float): The temperature of the system in Kelvin.
    M (float): The mass of each particle in kg.

    Returns:
    --------
    V : A 2D array of velocities (N, 3)
    """

    # Standard deviation of the velocity distribution
    sigma_v = np.sqrt(k_B * T / M)

    # Generate velocities from a normal distribution
    V = np.random.normal(0, sigma_v, (N, 3))

    return V

# Example usage
N = 10000           # Number of particles
T = 300             # Temperature in Kelvin
M = 1.67e-27        # Mass of a particle in kg
k_B = 1.380649e-23  # J/K

velocities, speeds = generate_velocities(N, T, M)

# Plot the histogram of speeds
plt.hist(speeds, bins=50, density=True, alpha=0.6, color='g')

# Overlay the Maxwell-Boltzmann theoretical distribution
v = np.linspace(0, np.max(speeds), 200)
distribution = (4 * np.pi * v**2) * (M / (2 * np.pi * k_B * T))**(3/2)
    * np.exp(-M * v**2 / (2 * k_B * T))
plt.plot(v, distribution, linewidth=2, color='r')
plt.xlabel('Speed (m/s)')
plt.ylabel('Probability Density')
plt.title('Maxwell-Boltzmann Speed Distribution')
plt.tight_layout()
```

This Python function **generate_velocities**, initializes the velocities of particles in a system based on a specified temperature and particle mass, following the Maxwell-Boltzmann distribution. Using the Boltzmann constant k_B, the function calculates the standard deviation σ_v of the velocity distribution as $\sqrt{k_B T/m}$. Then it generates random velocities for each particle by drawing from a normal (Gaussian) distribution with mean 0 and standard deviation σ_v. The output is an array of velocities, representing the velocity of each particle. Fig. 1.2 shows the resulting distribution of the generated velocities and its overlap with the theoretical distribution.

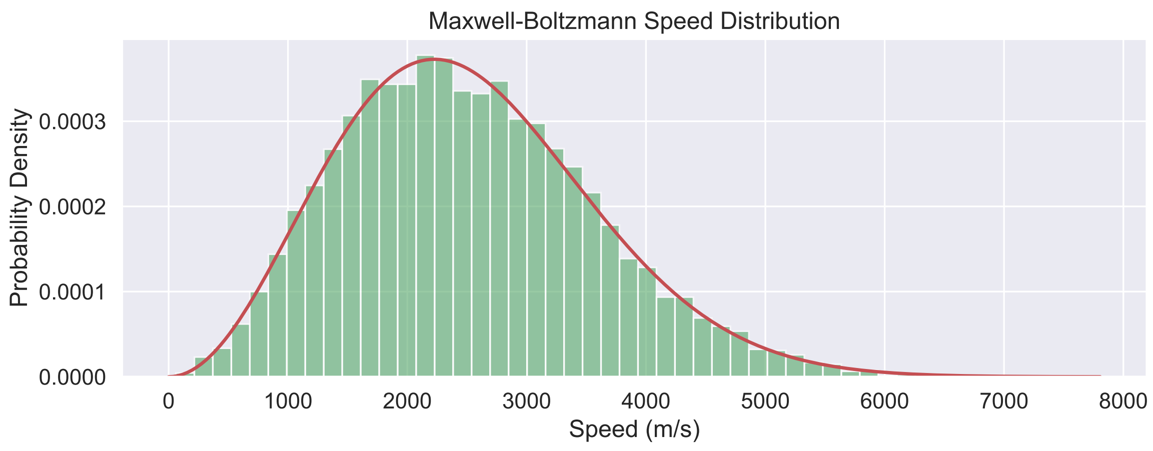


Figure 1.2: The comparison of velocity distribution between theory and modeling.

1.3.3 Interatomic Interaction: The Lennard-Jones Potential

Once the system is initialized, we need to evaluate the energy of the system and calculate forces to determine the motion of the particles. In MD simulations, the total energy of the system is described by interatomic interactions, which are defined by a specific **force field**. The force field is a mathematical model that specifies how particles in the system interact, and it dictates the forces experienced by each particle based on their relative positions.

Among the many force field options available, the **Lennard-Jones Potential** is one of the simplest and most commonly used models, especially for systems involving noble gases or other non-bonded interactions. It assumes that particles interact via pairwise interactions, meaning that the potential energy between any two particles depends only on the distance between them as follows.

$$V(r) = 4\epsilon \left[\left(\frac{\sigma}{r}\right)^{12} - \left(\frac{\sigma}{r}\right)^{6} \right], \tag{1.3}$$

where:

- r is the distance between two interacting particles,
- ϵ controls the depth of the potential well, which represents the strength of the attractive interaction, and
- σ is the distance at which the interparticle potential is zero, roughly corresponding to the size of the particles.

As shown in Fig. 1.3, the Lennard-Jones potential consists of two parts:

- $1/r^{12}$ term accounts for short-range repulsion,
- $-1/r^6$ term accounts for long-range attraction (London dispersion force),

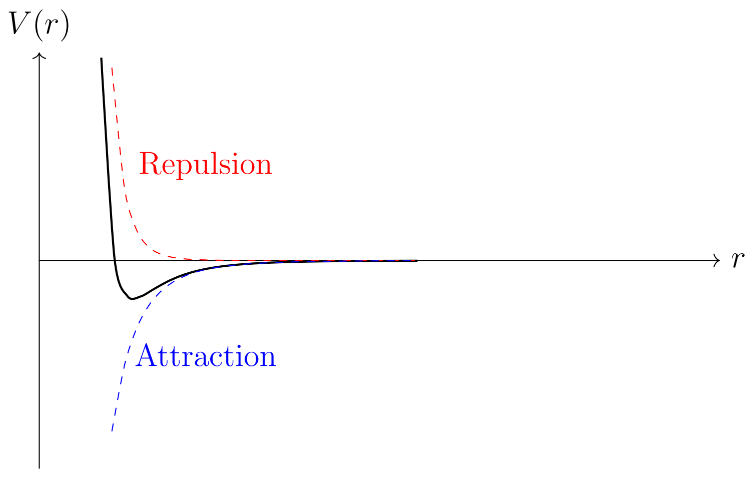


Figure 1.3: Lennard-Jones potential illustrating the attractive and repulsive regions. The potential minimum occurs at approximately 1.12σ.

In general, the force $\mathbf{F}(r)$ between two particles is the negative gradient of the potential:

$$\mathbf{F}(\mathbf{r}) = -\frac{\partial E(\mathbf{r})}{\partial \mathbf{r}}$$

For a Lennard-Jones potential, the force $\mathbf{F}(r)$ is:

$$\mathbf{F}(r) = \frac{\partial E(r)}{\partial \mathbf{r}} = 4\epsilon \left[-12 \left(\frac{\sigma}{r}\right)^{12} \frac{1}{\mathbf{r}} + 6 \left(\frac{\sigma}{r}\right)^{6} \frac{1}{\mathbf{r}} \right] \tag{1.4}$$

The Python code for calculating energy and forces is shown as follows.

```python
import numpy as np
from numba import njit

@njit
def compute_lj_energy_forces(positions, epsilon, sigma):
    """
    Calculate the total energy and forces by the LJ potential.

    Parameters:
    ------------
    positions (N, 3): the positions of each particle,
    epsilon (float): LJ potential parameter,
    sigma (float): LJ potential parameter.

    Returns:
    ---------
    energy (float): The total LJ potential energy
    forces (N, 3): An array of forces
    """

    N = len(positions)
    energy = 0.0
    forces = np.zeros_like(positions)

    for i in range(N):
        for j in range(i + 1, N):
            r_ij = positions[j] - positions[i]
            r = np.linalg.norm(r_ij)
            r6 = (sigma / r) ** 6
            r12 = r6 ** 2

            # Lennard-Jones potential
            e_ij = 4 * epsilon * (r12 - r6)
            energy += e_ij

            # Force calculation
            f_ij = 24 * epsilon * (2 * r12 - r6) / r**2 * r_ij
            forces[i] += f_ij
            forces[j] -= f_ij

    return energy, forces
```

The code defines a function, **compute_lj_energy_forces**, which calculates the total energy and forces in a system using the Lennard-Jones potential. The function iterates over each unique pair of particles, calculating the distance between them and compute both the potential energy and the force for that pair. The computed potential energy is accumulated into a total energy variable, while the forces on each particle are updated based on the calculated force, ensuring that each pair interaction is equal and opposite to satisfy Newton's third law. It should be noted that when the **periodic boundary condition** is imposed, each atom would have an infinite number of neighbors. One needs to rewrite the code to include more neighbors in the periodic images. A practical solution is to include only the neighbors within a cutoff value.

To speed up this calculation, the Numba library is used. Numba's `@njit` decorator enables Just-In-Time (JIT) compilation, which translates the Python code into optimized machine code at runtime. This optimization can significantly improve performance, especially for computationally intensive loops like those in this function. By adding `@njit`, the code avoids Python's interpretation overhead and can achieve speeds close to that of compiled languages such as C or FORTRAN.

1.3.4 Integration

Once the forces on each particle become available, we can then estimate the acceleration and update $\mathbf{R}$ and $\mathbf{V}$ of the particles over a small time increment dt. The most natural way to do this is to use the Euler's method, which is a simple numerical integration technique. The Euler's method updates the position and velocity of a particle as follows:

$$\mathbf{r}(t+dt) = \mathbf{r}(t) + \mathbf{v}(t)\,dt, \tag{1.5}$$

$$\mathbf{v}(t+dt) = \mathbf{v}(t) + \mathbf{a}(t)\,dt, \tag{1.6}$$

where:

- $\mathbf{r}(t)$ and $\mathbf{v}(t)$ are the position and velocity of a particle at time t,
- $\mathbf{a}(t)$ is the acceleration (or force per unit mass) at time t, and
- dt is the time step for the integration.

However, Euler's method is not very accurate and can lead to instability in the simulation, especially for long time steps. To improve the accuracy and stability of the integration, we can use more sophisticated methods that take into account the forces acting on the particles at different time steps. One commonly used method for this purpose is the Velocity Verlet algorithm. It performs the updates as follows:

$$\mathbf{r}(t+dt) = \mathbf{r}(t) + \mathbf{v}(t)\,dt + 0.5\,\mathbf{a}(t)\,dt^2, \tag{1.7}$$

$$\mathbf{v}(t+dt) = \mathbf{v}(t) + 0.5\,[\mathbf{a}(t) + \mathbf{a}(t+dt)]\,dt, \tag{1.8}$$

The first equation updates the position $\mathbf{r}(t+dt)$ based on the current position $\mathbf{r}(t)$, current velocity $\mathbf{v}(t)$, and the acceleration $\mathbf{a}(t)$ at the current time step. This equation accounts for the particle's velocity as well as any changes in position due to acceleration, providing a second-order accurate estimate.

The second equation updates the velocity $\mathbf{v}(t+dt)$ by taking the average of the acceleration at the current time step, $\mathbf{a}(t)$, and the acceleration at the next time step, $\mathbf{a}(t+dt)$. This **average acceleration** approach improves the accuracy of the velocity update, making the Velocity Verlet algorithm both time-reversible and energy-conserving.

According to Taylor expansion, the Velocity Verlet algorithm is accurate to $O(dt^3)$ for position updates and $O(dt^2)$ for velocity updates. This level of accuracy allows the integration to maintain stability over long simulation times, even with relatively large time steps.

1.4. Code Implementation and Testing

1.4.1 Problem Setup

In the following, we will develop our first MD code for simulating the liquid Argon system, which can be effectively described by the Lennard-Jones potential with $\epsilon = 0.0103$ eV (equivalent to $120k_B$) and $\sigma \approx 0.34$ nm. To keep the simulation manageable on a single-core computer, we consider a system of 864 atoms under periodic boundary conditions. According to experimental data, Argon has a melting point of 83.8 K, below which it adopts a face-centered cubic (fcc) crystal structure. For this simulation, we follow the setup used in Rahman's 1964 paper [5], studying Argon's liquid behavior at 94.4 K. Based on experimentally determined density, we estimate the required cubic box size to enclose the 864 atoms as 10.229σ.

Although it is challenging to directly generate atomic positions representing liquid Argon, we can initialize the system in an fcc arrangement and allow the MD simulation to naturally reach the liquid state at the desired temperature. In a single fcc unit cell, it contains four atoms at fractional positions (0, 0, 0), (0, 1/2, 1/2), (1/2, 0, 1/2), and (1/2, 1/2, 0). By replicating this unit cell six times along each dimension, we create a $6 \times 6 \times 6$ supercell consisting of 864 atoms.

Next, the initial velocities are assigned according to a Maxwell-Boltzmann distribution, after which the iterative force calculations and velocity Verlet integration are performed, as described in the previous section.

1.4.2 The Python Code for NVE MD Simulation

The code snippets below implement such a setup and the detailed simulation steps.

```python
import numpy as np
from numba import njit
from time import time

def initialize_position(L):
    """
    Initialize positions in a 6*6*6 of the FCC unit cell

    Args:
        L (float): unit length of the cubic box

    Returns:
        R (float) : (4*6*6*6, 3) array
    """
    # FCC unit cell fractional positions
    r = np.array([
        [0.0, 0.0, 0.0],
        [0.5, 0.5, 0.0],
        [0.5, 0.0, 0.5],
        [0.0, 0.5, 0.5]
    ])
    a = L/6

    r *= a
    R = []
    for i in range(6):
        for j in range(6):
            for k in range(6):
                R.extend(r + np.array([i, j, k])*a)
    return np.array(R)

def initialize_velocity(N):
    """
    Initialize velocities using Maxwell-Boltzmann distribution

    Args:
        N (int): Number of atoms

    Returns:
        V (float) : (N, 3) array
    """
    # Standard deviation of the velocity distribution
    sigma_v = np.sqrt(TEMPERATURE * KB / MASS)
    V = np.random.normal(0, sigma_v, (N, 3))

    # Center the velocities
    V -= np.mean(V, axis=0)

    return V

@njit
def LJ_energy_forces(R, L):
    """
    Compute the energy and forces from the given system

    Args:
        R (float) : (N, 3) array
        L (float): unit length of the cubic box

    Returns:
        PE (float): total energy
        F (float): atomic forces [N, 3] array
    """

    N = len(R)
    F = np.zeros_like(R)
    PE = 0.0

    for i in range(N - 1):
        for j in range(i + 1, N):
            # Compute R between (i, j)
            rvec = R[i] - R[j]
            rvec -= np.round(rvec / L) * L
            r = np.linalg.norm(rvec)

            # Compute the potential Energy
            R6 = (SIGMA / r) ** 6
            R12 = R6 ** 2
            PE += 4 * EPSILON * (R12 - R6)

            # Compute and update forces
            force = 24 * EPSILON * (2 * R12 - R6) / r ** 2
            forec_vec = force * rvec
            F[i] += forec_vec
```

```python
                F[j] -= forec_vec

    return PE, F

def verlet_integration(R, V, F, L):
    """
    Intergration based on the Verlet Velocity algorithm

    Args:
        R (float) : (N, 3) array
        V (float) : (N, 3) array
        F (float) : (N, 3) array
        L (float) : Cell length

    Returns:
        R (float) : (N, 3) array
        V (float) : (N, 3) array
        F (float) : (N, 3) array
    """

    # Update R
    R += V * TIMESTEP + 0.5 * F / MASS * TIMESTEP ** 2
    R = np.mod(R, L)

    # Compute F_new
    PE, F_new = LJ_energy_forces(R, L)

    # Update V
    V += 0.5 * (F + F_new) / MASS * TIMESTEP

    # Update F
    F = F_new

    return R, V, F, PE

if __name__ == "__main__":

    # Ensure reproduction
    np.random.seed(42)

    # Constants
    KB = 1.380649e-23             # Boltzmann constant in J/K
    mass = 1.66053906660e-27      # unit mass in kg

    # Force Field parameters
    EPSILON = 120*KB              # in J (epsilon = 120 * k_B)
    SIGMA = 3.4e-10               # in meters (3.4 Angstrom)

    # System Parameters
    N = 864                       # number of atoms
    MASS = 39.948 * mass          # mass of Argon atom in kg
    L = 10.229 * SIGMA            # cubic box side length

    # MD Parameters
    TEMPERATURE = 94.4            # in K
    TIMESTEP = 1.0 * 1e-14        # time step in seconds
    num_steps = 200               # Number of steps

    # Initialize the system
    R = initialize_position(L)
    V = initialize_velocity(N)
    E, F = LJ_energy_forces(R, L)

    KEs = []
    PEs = []

    # MD propogation
    t0 = time()
    print(f"Step      PE             KE        E_total    Time")
    for step in range(num_steps):
        R, V, F, PE = verlet_integration(R, V, F, L)
        KE = 0.5 * MASS * np.sum(V ** 2)
        KEs.append(KE)
        PEs.append(PE)
        if step % 10 == 0:
            t = time() - t0
            E = KE + PE
            print(f"{step:4d}{PE:12.4e}{KE:12.4e}{E:12.4e}{t:6.1f}")
```

In this code, the main routine initializes a molecular dynamics (MD) simulation for a system of argon atoms, employing the Lennard-Jones (LJ) potential. It then integrates the equations of motion using the Verlet velocity algorithm. The simulation is orchestrated through a series of functions—**initialize_position**, **initialize_velocity**, **LJ_energy_forces**, and **verlet_integration**—each meticulously detailed in the preceding sections. Notably, the **LJ_energy_forces** function, when applying periodic boundary conditions, considers only the shortest distance between any pair of atoms (ij) for calculating interactions. This approach is justified by the rapid decay of the Lennard-Jones potential and other short-range force fields with distance, rendering interactions negligible for atom pairs separated by larger distances, provided the unit cell is sufficiently large.

Executing the code generates outputs that illustrate the dynamic interplay between potential and kinetic energies as the simulation progresses. Initially, SI units are adopted for their inherent consistency and clarity. However, as the discussion evolves, a transition to a more context-specific and convenient unit system will be made. Despite the individual fluctuations in potential and kinetic energies, the total energy of the system should ideally remain constant. This reflects the fundamental principle of energy conservation within an isolated NVE system. Deviations from this constant total energy may signal underlying issues such as numerical inaccuracies, an inadequately small integration time step, or improper implementation of boundary conditions. Therefore, ensuring the stability and accuracy of simulation parameters is paramount for achieving the expected energy conservation.

```
Step      PE             KE        E_total    Time
   0 -9.7496e-18  1.6423e-18 -8.1073e-18   0.1
  10 -9.5248e-18  1.4171e-18 -8.1077e-18   1.0
  20 -8.9089e-18  8.0209e-19 -8.1068e-18   1.8
  30 -8.8762e-18  7.6884e-19 -8.1074e-18   2.7
  40 -8.9330e-18  8.2564e-19 -8.1074e-18   3.5
  50 -8.9173e-18  8.0998e-19 -8.1073e-18   4.4
  60 -8.9381e-18  8.3079e-19 -8.1073e-18   5.3
  70 -8.9757e-18  8.6843e-19 -8.1073e-18   6.1
  80 -9.0010e-18  8.9366e-19 -8.1073e-18   7.0
  90 -8.9847e-18  8.7744e-19 -8.1073e-18   7.9
 100 -8.9837e-18  8.7644e-19 -8.1073e-18   8.7
 110 -8.9892e-18  8.8187e-19 -8.1073e-18   9.6
 120 -9.0023e-18  8.9496e-19 -8.1073e-18  10.5
 130 -9.0034e-18  8.9613e-19 -8.1073e-18  11.3
 140 -8.9840e-18  8.7673e-19 -8.1073e-18  12.2
 150 -8.9604e-18  8.5317e-19 -8.1073e-18  13.1
 160 -8.9895e-18  8.8216e-19 -8.1073e-18  13.9
 170 -8.9890e-18  8.8165e-19 -8.1073e-18  14.8
 180 -8.9766e-18  8.6932e-19 -8.1073e-18  15.7
 190 -9.0085e-18  9.0121e-19 -8.1073e-18  16.5
```

One can visualize the evolution of energies in Fig. 1.4. Clearly, we observe an initial significant change in kinetic and potential energies, reflecting substantial atomic movement. Once the system reaches a local equilibrium, atoms only oscillate around their average positions, leading to a more stable energy state. The potential energy (PE) and kinetic energy (KE) exhibit fluctuations, but the total energy remains constant, indicating that the system is in a steady state. This behavior is characteristic of an NVE ensemble, where the total energy is conserved over time.

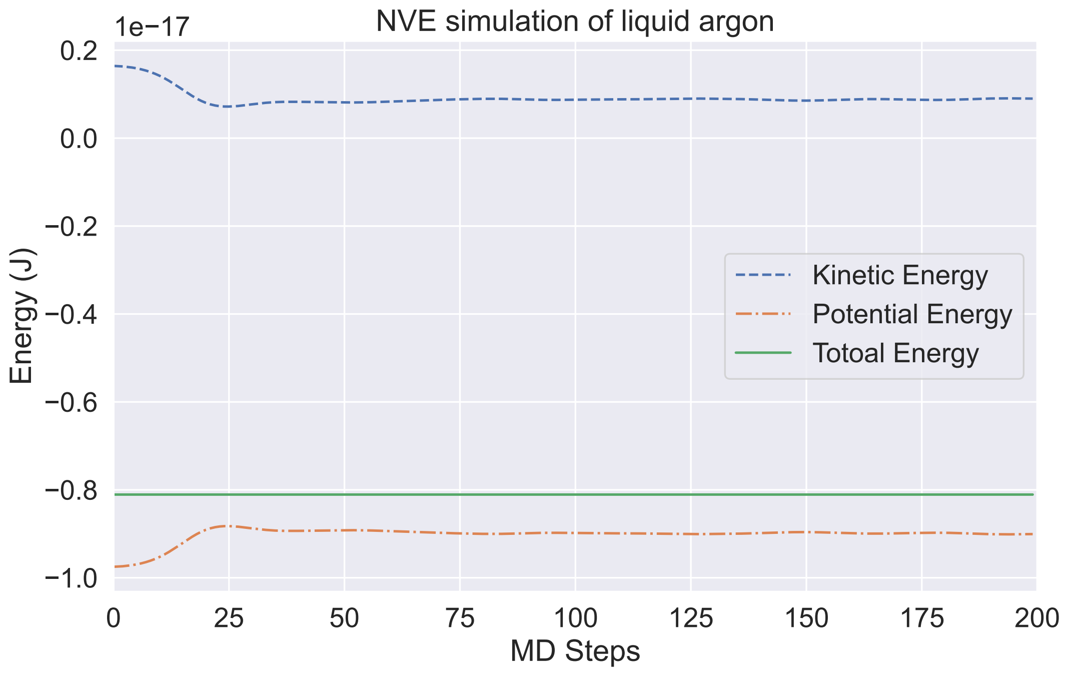
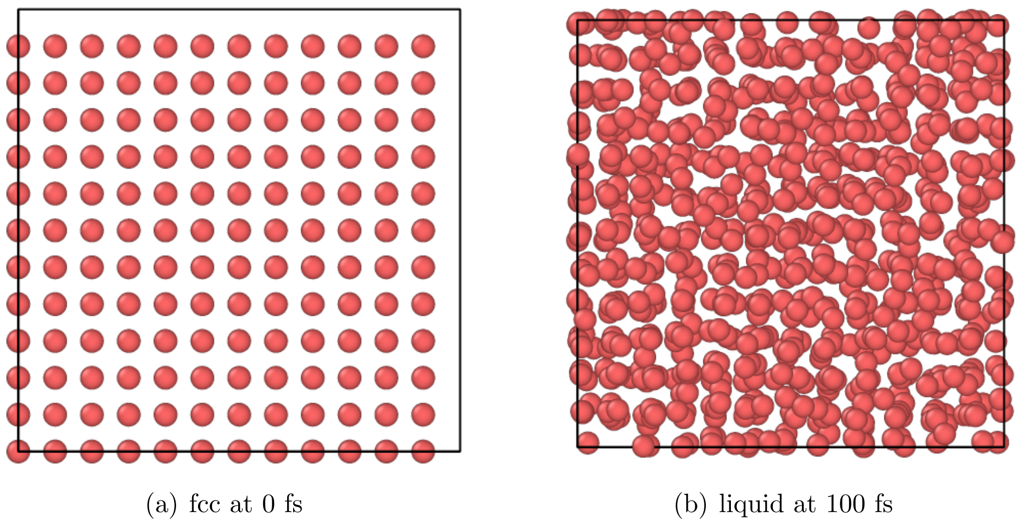


Figure 1.4: The simulated evolution of energies for liquid Argon at 94.4 K.

Additionally, Fig. 1.5 displays two MD snapshots taken at the initial timestep and at 100 fs, illustrating a transition process from solid to liquid. In the following chapters, we will explore more advanced simulation and characterization techniques (e.g., radial distribution function, vibrational density of states) to gain deeper insights into these transition mechanisms.



(a) fcc at 0 fs (b) liquid at 100 fs

Figure 1.5: The NVE MD snapshots from our simulation.

1.5. Summary

In this chapter, we introduced MD simulations within the NVE ensemble, covering the historical context, fundamental principles, and essential steps in implementation, including system initialization, force calculations, integration, and analysis for isolated systems, both with and without periodic boundary conditions. This foundation enables simulation and study of atomic systems, uncovering insights that may be difficult to achieve through experiments alone.

To become familiar with the MD framework, it is beneficial to repeat the simulation with varied parameters or to set up other lightweight model simulations (e.g., other noble gases). By examining the results and questioning whether the output energy values make sense, you gain valuable insights into the numerical aspects of MD simulations. These practices prepare you for larger-scale simulations that involve more atoms, longer simulation times, and more complex force fields. For such scenarios, more mature packages like LAMMPS, GROMACS and NAMD, are recommended, as they operate based on the same foundational principles introduced here.

2. Thermostat in the NVT Ensemble

2.1. Motivation

So far, we have learned how to run an NVE MD simulation for a periodic system from both programming and application points of view. In such simulations, the total energy E should be constant with the propagation of time. This is called the **microcanonical ensemble** in statistical physics. However, this setup is not really the truth for many practical simulations. It is more likely that the system would interact with the surrounding environment and exchange heat over the boundaries. Therefore, we will explore how to perform a MD simulation that allows the heat exchanges in this chapter.

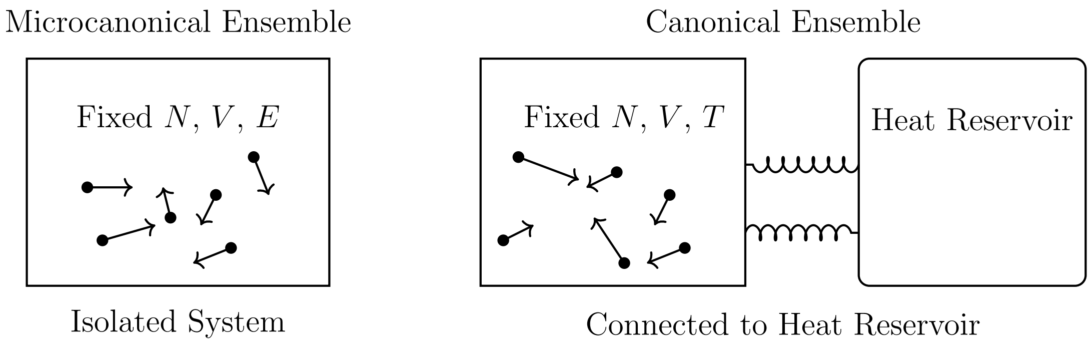


Figure 2.1: The schematic comparison of NVE and NVT ensembles.

2.2. Extension to NVT by Allowing Heat Exchange

In a real scenario, we often want to study the system under a constant temperature condition, instead of constant energy. This method is particularly useful for simulating systems in the **Canonical Ensemble** (constant number of particles, volume, and temperature, often denoted as NVT).

As shown in Fig. 2.1, to maintain the system at a desired temperature, we couple it to an external **heat bath**. There are several thermostat techniques for this purpose. In real life, temperature is a measure of how fast the particles are moving on average. But in a computer simulation, you need a way to control this *temperature* to make sure it stays at the desired level throughout the simulation. A thermostat in molecular dynamics is a tool that helps you keep the temperature of your simulated system steady, just like how a thermostat in your house keeps the room temperature stable.

2.3. Intuitive Thermostats

To introduce the thermostat to a MD system, the trick is to modify the integrator. Currently, there exist several flavors of thermostat techniques. Perhaps the easiest way to rescale velocities to force the total kinetic energy to be equal to $3Nk_BT/2$ at every few steps. However, this kind of rescaling can definitely perturb the MD trajectory strongly and thus not recommended. In this section, we introduce two widely used thermostats for maintaining a constant temperature: the Anderson thermostat and the Langevin thermostat. These thermostats enable simulations under canonical (NVT) ensembles by controlling the system's temperature, which is crucial for studying temperature-dependent properties of materials.

2.3.1 The Anderson Thermostat

The main idea of Anderson thermostat [6] is inspired by the observation of physical collisions between particles in the system and particles in the surrounding environment (see Fig. 2.2). After collisions, the particles in the system would change the velocities. These **collisions** ensure that the system exchanges energy with the environment, maintaining thermal equilibrium.

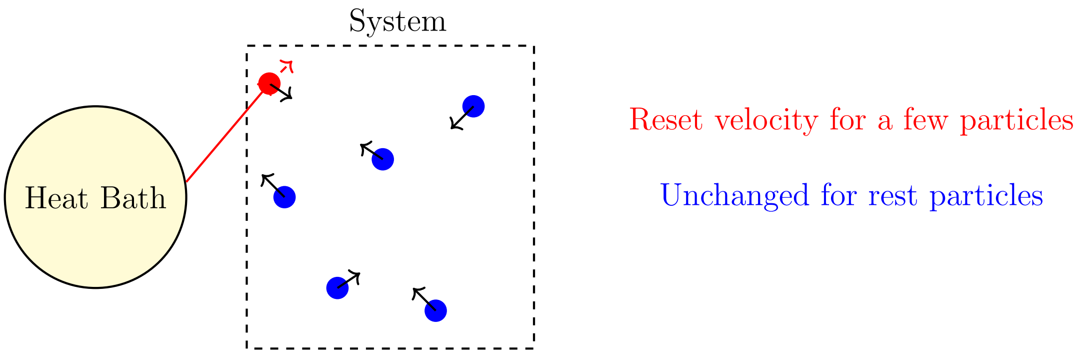

System
Heat Bath
Reset velocity for a few particles
Unchanged for rest particles

Figure 2.2: Illustration of the Andersen Thermostat. A heat bath randomly selects a few particles (in red) and resets their velocities according to the Maxwell-Boltzmann distribution. Other particles continue with their current velocities.

Thus, we could periodically pick some particles and randomize the velocities of some particles in the system. These randomizations mimic the effect of an external environment interacting with the particles, ensuring that the system's temperature remains constant. Hence, we allow two types of MD integration rules in the actual code.

1. **Random particle selection and velocity assignment**. With a certain probability ν, the velocity of each particle is reassigned by sampling from a Maxwell-Boltzmann distribution corresponding to the desired temperature T.

2. **Ordinary update**. If the particle's velocity is not reassigned, it evolves according to the usual equations of motion (e.g., using the Verlet integration method).

In this technique, **Collision Frequency** ν determines how often the particle velocities are randomized (following a Poisson Distribution). A higher ν means more frequent collisions (interaction) with the heat bath, leading to stronger coupling to the temperature bath. We should choose a ν so that velocity reassignment happens at an appropriate rate to maintain the desired temperature without overly disrupting the natural dynamics of the system.

The Anderson thermostat is relatively simple to implement, requiring only the addition of random velocity reassignment at each time step. To include it to MD, one can refer to the following Algorithm 1.

Algorithm 1 MD based on Anderson thermostat

```
Initialization
Set up initial positions and velocities for the particles
calculate initial forces.
Main MD Loop:
for each time step do
    Update positions using the current velocities and forces.
    Calculate new forces based on updated positions.
    Update velocities using the average of old and new forces.
    Apply the Anderson thermostat by reassigning velocities with a probability.
end for
Output: Collect and store data for analysis
```

However, it may not reflect the real dynamics. Since velocities are randomly reassigned, the resulting particle trajectories may not correspond to those in a real physical system where energy exchange occurs through physical interactions. This is particularly true for a periodic system without the clear definition of boundary. In addition, one needs to play with the ν values.

2.3.2 The Langevin Thermostat

The Langevin thermostat maintains the temperature of a system while also modeling the effects of friction and random forces, similar to those that might be encountered in a viscous fluid. As shown in Fig. 2.3, the basic idea is to modify the equations of motion by adding two additional terms to the standard Newtonian dynamics:

$$m_i \frac{d\mathbf{v}_i}{dt} = \mathbf{F}_i - \gamma m_i \mathbf{v}_i + \mathbf{R}_i(t) \tag{2.1}$$

- Frictional Force $\gamma m_i \mathbf{v}_i$: the damping effect of the environment, which tends to slow down the particles.

- Random Force $\mathbf{R}_i(t)$: the random collisions with particles from the heat bath, which cause the particles to move randomly, maintaining the system's temperature. These kicks help maintain the temperature of the system by continuously injecting energy into the system.

A typical value of γ used in many MD simulations is around $\gamma = 0.1$ ps^{-1}. This value provides a good balance between maintaining temperature control and preserving realistic dynamics. The system is weakly coupled to the heat bath, ensuring that it can sample the canonical ensemble without heavily damping the natural motion of the particles.

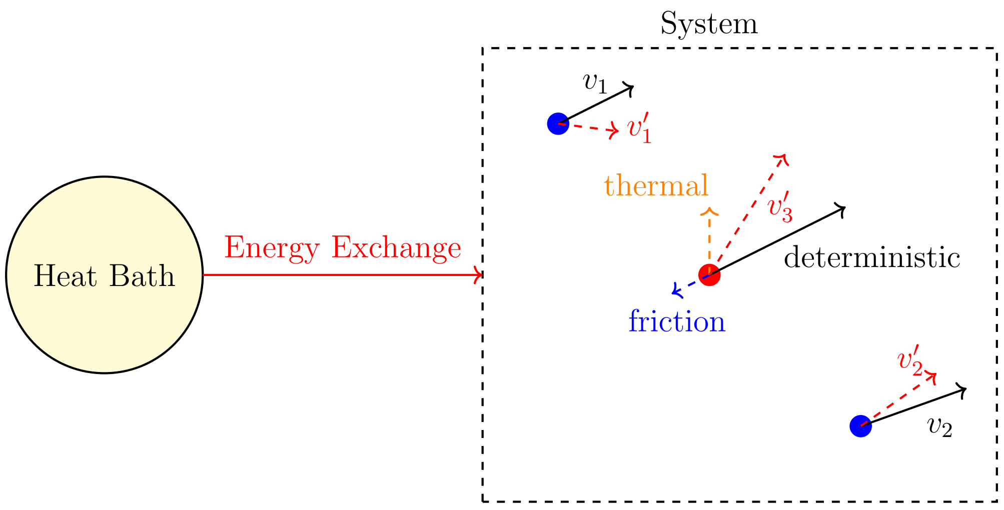


Figure 2.3: Illustration of the Langevin Thermostat. Each particle experiences deterministic forces, frictional damping, and random thermal noise due to coupling with a heat bath.

The Langevin thermostat is implemented through a modified Velocity Verlet integration, incorporating the additional friction and random forces into the velocity update step:

$$\mathbf{v}_i(t+\Delta t) = \mathbf{v}_i(t) + \frac{\mathbf{F}_i(t)}{m_i}\Delta t - \gamma \mathbf{v}_i(t)\Delta t + \mathbf{R}_i\sqrt{\Delta t} \tag{2.2}$$

```python
import numpy as np

def langevin_thermostat(V, F, gamma):
    # Update R, V, F using Verlet integration
    R += V * TIMESTEP + 0.5 * F * TIMESTEP **2 / MASS

    # Update velocities with deterministic part
    V += 0.5 * F * TIMESTEP / MASS

    # Apply friction and random force (stochastic part)
    V += -gamma * V * TIMESTEP
    sigma = np.sqrt(2 * gamma * kB * T / MASS)
    V += np.random.normal(0, sigma, V.shape) * np.sqrt(dt)

    # Update forces and velocities
    F_new = calculate_forces(positions)
    V += 0.5 * (F + F_new) / MASS * TIMESTEP
    F = F_new
    return R, V, F

# Initialization
T = 300
R = Initialize_positions()
V = Initialize_velocities()
F = calculate_forces(R)

# Main MD loop
for step in range(num_steps):
    R, V, F = langevin_thermostat(R, V, F, L, gamma)
```

2.3.3 Comparison

Both the Anderson and Langevin thermostats provide effective temperature control in MD simulations, but differ in approach and application.

The Anderson thermostat periodically resets the particle velocities, which is simple to implement but disrupts the natural trajectory of the particles. In contrast, the Langevin thermostat introduces a more realistic interaction with a thermal environment by combining frictional damping and random forces, allowing for smooth, continuous dynamics. Hence, the Langevin thermostat maintains realistic trajectories, making it ideal for preserving the natural dynamics of the system. However, it is computationally more expensive due to the additional stochastic terms. whereas the Anderson thermostat is computationally efficient and effective for rapid temperature equilibration but may be less suited for systems where dynamic fidelity is critical.

In summary, the Anderson thermostat excels in simplicity and speed but compromises on dynamic realism, while the Langevin thermostat strikes a balance between realistic dynamics and effective temperature control at the cost of higher computational complexity.

2.4. Statistical Physics Perspective

While both the Anderson and Langevin thermostats enable instantaneous velocity updates to simulate heat exchange with a fictitious reservoir, they lack rigorous validation to confirm that these updates accurately replicate interactions with a real heat bath. Additionally, both methods introduce random components into the simulation, making the results non-deterministic. This stochasticity raises the question of whether these approaches truly adhere to the underlying principles of physical systems.

To address these concerns, it is crucial to adopt a more formal approach to validate the system's evolution within the simulation. A robust starting point is to examine the statistical properties of the simulated system within the context of the NVE and NVT ensembles from a probability perspective. By analyzing these properties, we can assess whether the system's behavior aligns with the expected distributions and dynamics dictated by fundamental physical laws. This validation ensures that the simulation reliably represents real-world systems and provides meaningful insights.

In an NVE ensemble, the system is isolated from its surroundings, so the total energy E, which includes both the kinetic and potential energy, remains constant. With no external interaction, the energy conservation principle dictates that E should remain invariant throughout the simulation.

In contrast, an NVT ensemble allows the system to exchange heat with an external reservoir. As a result, the total energy of the system is not fixed; it fluctuates over time due to heat exchanges with the reservoir. Assuming at time t_1, the total energy is measured as E_1, and at time t_2, it is measured as E_2. How do we calculate the probability of observing E_1 and E_2 in subsequent time frames? For convenience, we denote these probabilities as $p(E_1)$ and $p(E_2)$. Statistical mechanics provides a framework to derive these probabilities, offering a rigorous foundation for understanding the evolution of the system under heat exchange conditions in the NVT ensemble.

Here we track each particle (i) in terms of its position ($\mathbf{r}_i$) and momentum ($\mathbf{p}_i$), giving us a total of six numbers for each particle. Each unique snapshot of $\{\mathbf{p}_i, \mathbf{r}_i\}$ is referred to as a **microstate**. It is possible for a group of microstates to share the same total energy.

This group of microstates is called a **macrostate**, and the number of microstates in a macrostate is referred to as its **multiplicity** (Ω).

A Toy example of microstate/macrostate/multiplicity.

Imagine that you are flipping three coins multiple times and counting the statistics of heads and tails. There are 8 possible combinations: HHH, HHT, HTH, THH, HTT, TTH, THT, TTT. While we are not concerned with the specific sequences, we are interested in the total counts. This means we have 8 microstates in total but only 3 types of macrostates:

- 1 occurrence of 3 heads (HHH), thus $\Omega(3H) = 1$,
- 3 occurrences of 2 heads (HHT, HTH, THH), thus $\Omega(2H) = 3$,
- 3 occurrences of 1 head (HTT, TTH, THT), thus $\Omega(1H) = 3$,
- 1 occurrence of 0 heads (TTT), thus $\Omega(0H) = 1$,

Returning to our original problem, the ratio of $p(E_1)/p(E_2)$ depends on the multiplicities $\Omega(E_1)/\Omega(E_2)$.

$$\frac{P(E_2)}{P(E_1)} = \frac{\Omega(E_2)}{\Omega(E_1)} \approx \frac{\Omega_R(E_2)}{\Omega_R(E_1)} \tag{2.3}$$

Why do we look at the reservoir?

Here we consider an isolated system, consisting of a reservoir and a system, and the reservoir is much bigger than the system. So the actual $\Omega = \Omega_R\Omega_S$, can be approximated by Ω_R. One can intuitively think the majority of multiplicity should be done by the reservoir.

There is a famous entropy equation, $S = k_B \ln \Omega$,

$$\frac{P(E_2)}{P(E_1)} = \frac{e^{S_R(E_2)/k_B}}{e^{S_R(E_1)/k_B}} = e^{[S_R(E_2)-S_R(E_1)]/k_B} \tag{2.4}$$

The relation between S and U under NVT.

From the **microscopic** view, the system reaches an equilibrium when it has the largest entropy. So the change of entropy can be counted as a function of (U, V, N)

$$dS = \frac{1}{T}\left[dU + PdV - \mu dN\right] = \frac{dU}{T} \qquad \textbf{NVT ensemble}$$

From the **macroscopic** view, you can also think it from the 1st raw.
Since the change of energy (dU) can be either from Work (pdV, omitted at a constant V) or a heat transfer (TdS), so $dU = TdS$ at the NVT ensemble.

For simplicity, we can neglect PdV and μdN terms (since N and V do not change). Thus,

$$dS_R = \frac{1}{T}\left[U_R(E_2) - U_R(E_1)\right] \tag{2.5}$$

Note that $U(E_2) + U_R E_2$ should be conserved. Therefore, we have:

$$\frac{P(E_2)}{P(E_1)} = \frac{e^{-E_2/k_BT}}{e^{-E_1/k_BT}} = e^{-(E_2-E_1)/k_BT} \tag{2.6}$$

In conclusion, the probability of each state in a canonical ensemble (NVT) is proportional to an exponential term:

$$P(s) \propto e^{-E(s)/k_BT} \quad \Rightarrow \quad P(s) = \frac{1}{Z}e^{-E(s)/k_BT} \tag{2.7}$$

where Z is the normalization constant, also known as the **Partition function**. If we sum over all possible states (e.g., $s_1, s_2, \cdots$), the total $\sum_{i=1}^{N} p(s_i) = 1$. Therefore,

$$Z = \sum e^{-E(s)/k_BT} \tag{2.8}$$

Thus, in the NVT ensemble, E can take different values, and the probability follows the relation of e^{-E/k_BT}, also called Boltzmann distribution.

2.5. Partition Function of the Extended System

If we want to avoid the use of brute-force velocity reassignment, a gentler approach is to control the temperature by coupling the system to an additional degree of freedom, which acts as a **thermal reservoir** that exchanges energy with the system. Nose introduced a method where the system's Hamiltonian is extended by adding an artificial variable s that represents the thermal reservoir [7]. Fig. 2.4 demonstrates this approach, where s serves as a virtual variable to rescale the velocity at each MD update. When s=1, there is no impact on the system. When s is greater than 1, it means the velocity would become larger, mimicking the heat absorption process from the reservoir to the system. On the other hand, a small s value mimic the heat exchange from the system to the reservoir.

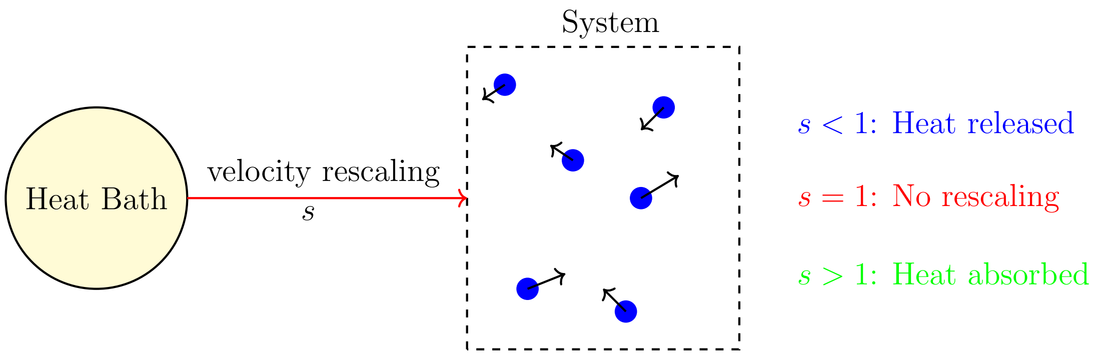


Figure 2.4: Illustration of Nosé's extended Hamiltonian method using a thermal reservoir variable s. In each timestep, s is be applied to rescale the velocity for all particles in the system.

Based the above design principle, it is straightforward to write down the new system's Hamiltonian as follows,

$$H = \sum_{i=1}^{N} \frac{\mathbf{p}_i^2}{2ms^2} + U(\mathbf{r}_1, \mathbf{r}_2, \ldots, \mathbf{r}_N) + \frac{p_s^2}{2Q} + gk_BT\ln(s) \tag{2.9}$$

The first two terms represent the system's kinetic energy (after the velocity rescaling), and potential energy as usual. To describe the whole system, we also include two more terms to express the kinetic and potential energy for the fictitious variable s. For the kinetic energy part, we need to define a mass term Q, and thus the energy becomes $p_s^2/2Q$ as usual. For the potential energy part, the choice would be somewhat arbitrary. Hypothetically, we want to make sure this term is zero when s=1 (no heat exchange), becomes positive when $s > 1$ (giving heat to the system), and becomes negative when $s < 1$ (receiving heat from the system). Obviously, the term should take a function like $C\ln(s)$ where C is a constant. For the sake of convenience, we use the form of $gk_BT\ln(s)$ in the following discussions.

In order to ensure the NVT ensemble, we need to compute the partition function Z and make sure that it is proportional to e^{-E/k_BT}. Hence, we proceed to compute the partition function Z for the given H

$$\begin{aligned}
Z &= \frac{1}{N!}\int dp_s ds d\mathbf{p}^N d\mathbf{r}^N \delta(H-E) \qquad (2.10)\\
&= \frac{1}{N!}\int dp_s ds d\mathbf{p}'^{3N} d\mathbf{r}^{3N} s^{3N} \delta\left(\sum_{i=1}^{N}\frac{\mathbf{p}_i'^2}{2m} + U(\mathbf{r}_1, \mathbf{r}_2, \ldots, \mathbf{r}_N) + \frac{p_s^2}{2Q} + gk_BT\ln(s) - E\right)\\
&= \frac{1}{N!}\int dp_s ds d\mathbf{p}'^{3N} d\mathbf{r}^{3N} s^{3N} \delta\left(\sum_{i=1}^{N} H^0 + \frac{p_s^2}{2Q} + gk_BT\ln(s) - E\right)
\end{aligned}$$

Note that here we defined

$$\begin{aligned}
\mathbf{p}' &= \mathbf{p}/s\\
H^0 &= \frac{\mathbf{p}_i'^2}{2m} + U(\mathbf{r}_1, \mathbf{r}_2, \ldots, \mathbf{r}_N)
\end{aligned}$$

Now, let

$$f(s) = \sum_{i=1}^{N} H^0 + \frac{p_s^2}{2Q} + gk_BT\ln(s)$$

$$s_0 = \exp\left[-\frac{H^0 + p_s^2/2Q}{gk_BT}\right] \qquad \leftarrow \quad f(s_0) = 0$$

$$f'(s_0) = \frac{gk_BT}{s}$$

$$\delta(f(s)) = \delta(s - s_0)/f'(s_0)$$

Plugin them back to Z (eq. 2.10)

$$\begin{aligned}
Z &= \frac{1}{N!}\int dp_s ds d\mathbf{p}'^{3N} d\mathbf{r}^{3N} s^{3N} \frac{s}{gk_BT}\delta\left(s - \exp\left[-\frac{H^0 + p_s^2/2Q - E}{gk_BT}\right]\right) \\
&= \frac{1}{N!}\frac{1}{gk_BT}\int dp_s ds d\mathbf{p}'^{3N} d\mathbf{r}^{3N} s^{3N+1}\delta\left(s - \exp\left[-\frac{H^0 + p_s^2/2Q - E}{gk_BT}\right]\right) \\
&= \frac{1}{N!}\frac{1}{gk_BT}\int dp_s d\mathbf{p}'^{3N} d\mathbf{r}^{3N} ds s^{3N+1}\delta\left(s - \exp\left[-\frac{H^0 + p_s^2/2Q - E}{gk_BT}\right]\right) \\
&= \frac{1}{N!}\frac{1}{gk_BT}\int dp_s d\mathbf{p}'^{3N} d\mathbf{r}^{3N} \exp\left[\left(-\frac{H^0 + p_s^2/2Q - E}{gk_BT}\right)(3N+1)\right] \\
&= \frac{1}{N!}\frac{1}{gk_BT}\int dp_s \exp\left[\left(-\frac{p_s^2/2Q - E}{gk_BT}\right)(3N+1)\right]\int d\mathbf{p}'^{3N} d\mathbf{r}^{3N} \exp\left[-\frac{H^0(\mathbf{p}',\mathbf{r}')}{gk_BT}(3N+1)\right] \\
&= \frac{1}{N!}\frac{1}{gk_BT}\int dp_s \exp\left[\left(-\frac{p_s^2/2Q - E}{gk_BT}\right)(3N+1)\right]\int d\mathbf{p}'^{3N} d\mathbf{r}^{3N} \exp\left[-\frac{3N+1}{g}\frac{H^0(\mathbf{p}',\mathbf{r}')}{k_BT}\right] \\
&= C\int d\mathbf{p}'^{3N} d\mathbf{r}^{3N} \exp\left(-\frac{3N+1}{g}\frac{H^0(\mathbf{p}',\mathbf{r}')}{k_BT}\right) \\
&= C\int d\mathbf{p}'^{3N} d\mathbf{r}^{3N} \exp\left(\frac{-H^0}{k_BT}\right) \quad (\text{when} \quad g = 3N+1)
\end{aligned}$$

The p_s part gives a constant dependent on the parameters E, T, Q and g. If we choose $g = 3N + 1$, the partition function Z of the extended system is equivalent to that of the physical system in the canonical ensemble Z_c except for a constant factor, $Z = CZ_c$. Hence, this approach maintains the system at a desired temperature, allowing it to sample from the canonical ensemble, making the Nose-Hoover thermostat an effective tool for molecular dynamics simulations that require temperature control.

2.6. The Nose-Hoover Thermostat

The Nose's extended system approach was further reformulated by Hoover to simplify the numerical process [8]. In the Nose-Hoover thermosta, the equations of motion include a friction term (ξ) that dynamically adjusts the particle velocities to maintain the target temperature. Thus, the velocity is updated via the following term

$$\frac{d\mathbf{v}_i}{dt} = \frac{\mathbf{F}_i}{m_i} - \xi\mathbf{v}_i \tag{2.11}$$

where $\mathbf{v}_i$ is the velocity of particle i , $\mathbf{F}_i$ is the force acting on particle i , m_i is the mass of the particle, and ξ is the friction coefficient or thermostat variable.

Then ξ is updated as follows

$$\frac{d\xi}{dt} = \frac{1}{Q}\left(\sum_i \frac{m_i\mathbf{v}_i^2}{3Nk_BT} - 1\right) \tag{2.12}$$

where Q is the **thermal inertia** parameter (which controls how strongly the system is coupled to the thermostat), N is the number of particles, and k_B is the Boltzmann constant.

The Nose-Hoover thermostat can be realized in the following Python Code.

```python
import numpy as np

# Thermostat variables
Q = 100.0
xi = 0.0

# Initialization
R = Initialize_positions()
V = Initialize_velocities()
F = calculate_forces(R)

# Main MD loop
for step in range(num_steps):

    # Verlet-like integration
    R += V * TIMESTEP + 0.5 * F * TIMESTEP**2 / MASS
    F_new = calculate_forces(R)

    # Update velocities
    V += 0.5 * (F + F_new) * TIMESTEP / MASS
    V *= (1 - 0.5 * xi * TIMESTEP) / (1 + 0.5 * xi * TIMESTEP)

    # Update the Nose-Hoover thermostat variable
    kE= 0.5 * np.sum(mass * V**2)
    xi += dt * (2 * kE / (3 * N * k_B * T) - 1) / Q
```

2.7. Code Implementation

Following the numerical instructions, three thermostat functions (**anderson_thermostat**, **langevin_thermostat**, **nose_hoover_thermostat** can be created as follows. They essentially modify or replace the **verlet_integration** function.

```python
import numpy as np

def anderson_thermostat(V, nu=0.5):
    """
    Anderson thermostat
    """
    sigma = np.sqrt(KB * TEMPERATURE / MASS)
    # Randomly assign new velocities
    lists = []
    for i in range(len(V)):
        if np.random.rand() < nu:
            V[i] = np.random.normal(0, sigma, 3)
            lists.append(i)

    return V

def langevin_thermostat(R, V, F, L, gamma):
    """
    Langevin thermostat
    """
    # Update R
    R += V * TIMESTEP + 0.5 * F/MASS * TIMESTEP ** 2
    R = R % L

    # Update velocities with deterministic part
    V += 0.5 * F * TIMESTEP / MASS

    # Apply friction and random force (stochastic part)
    V -= gamma * V * TIMESTEP
    sigma = np.sqrt(2 * gamma * KB * TEMPERATURE / MASS)
    V += np.random.randn(len(V), 3) * np.sqrt(TIMESTEP) * sigma

    # Update forces
    PE, F_new = LJ_energy_forces(R, L)
    V += 0.5 * (F + F_new) / MASS * TIMESTEP
    F = F_new

    return R, V, F

def nose_hoover_thermostat(R, V, F, xi, L, Q):
    """
    Nose-Hoover thermostat
    """
    # Update R
    R += V * TIMESTEP + 0.5 * F/MASS * TIMESTEP ** 2
    R = R % L

    # Update forces
    PE, F_new = LJ_energy_forces(R, L)
    V += 0.5 * (F + F_new) / MASS * TIMESTEP
    V *= (1 - 0.5 * xi * TIMESTEP) / (1 + 0.5 * xi * TIMESTEP)

    # Update xi
    KE = 0.5 * np.sum(V**2) * MASS
    xi += TIMESTEP * (2 * KE / (3 * len(R) * KB * TEMPERATURE) - 1) / (
   Q*MASS)

    # Update forces
    F = F_new
    return R, V, F, xi
```

In addition, I created another function **MD** to allow the specification of MD simulation under different choices of integration routines. The function simulates the dynamics of particles in a system over a given number of steps, applying specified thermodynamic controls to maintain the desired ensemble.

```python

def MD(thermostat=None, nu=0.1, gamma=1e+13, Q=1.0, num_steps=500):
    """
    Run MD simulation

    Args:
        thermostat (str): "Langevin", "Anderson", "Nose-Hoover" or None
        nu (float): Anderson thermostat parameter
        gamma (float): Langevin thermostat parameter
        Q (float): Nose-Hoover thermostat parameter
        num_steps (int): Number of steps to simulate
    """
    # Initialize system
    R = initialize_position(L)
    V = initialize_velocity(N)
    E, F = LJ_energy_forces(R, L)
    KEs = []
    PEs = []
    TEs = []
    xi = 0.0  # Used by Nose-Hoover

    # MD propogation
    for step in range(num_steps):
        if thermostat == "Nose-Hoover":
            R, V, F, xi = Nose_Hoover_thermostat(R, V, F, xi, L, Q)
        elif thermostat == "Langevin":
            R, V, F = langevin_thermostat(R, V, F, L, gamma)
        else:
            R, V, F = verlet_integration(R, V, F, L)
            if thermostat == "Anderson":
                V = anderson_thermostat(V, nu)

        # Compute PE, KE, and TE
        PE, _ = LJ_energy_forces(R, L)
        KE = 0.5 * np.sum(V**2) * MASS
        KEs.append(KE)
        PEs.append(PE)
        TEs.append(PE+KE)
        if step % 10 == 0:
            s = np.exp(-xi * TIMESTEP)
            print(f"Step {step:6d}, PE: {PE:.5e} KE: {KE:.5e} E: {PE+KE
   :.5e} s: {s:.4f}")
    return KEs, PEs, TEs


if __name__ == "__main__":
    results = []
    for thermostat in ["Anderson", "Langevin", "Nose-Hoover", None]:
        print(f"Simulation with {thermostat} thermostat")
        KEs, PEs, TEs = MD(thermostat=thermostat, num_steps=1500)
        results.append((thermostat, KEs, PEs, TEs))

    for result in results:
        thermostat, KEs, PEs, TEs = result
        if thermostat is None: thermostat = "NVE"
        plt.plot(KEs, label=thermostat, alpha=0.8)
    plt.legend()
    plt.xlabel('Timestep')
    plt.xlim([0, 1500])
    plt.ylabel('Kinetic Energy (J)')
    plt.savefig('lec_02_nvt_nve.png')
```

In the main block, the MD simulation is run for each thermostat type (Anderson, Langevin, Nose-Hoover, and NVE). The results, including kinetic, potential, and total energy values for each thermostat, are stored in the list of `results`.

Last, the code plots the kinetic energy versus time step for each thermostat method. The corresponding plot is shown in Fig. 2.5. Compared to the NVE simulation, we can clearly see that all three NVT thermostat simulation converge their kinetic energy to about the same value (corresponding to the target temperature 94.4 K).

Comparing different thermostat models, each one has unique strengths: Anderson is computationally efficient, Langevin offers a closer physical representation of thermal interactions. In both Anderson and Langevin thermostats, we can see that the kinetic

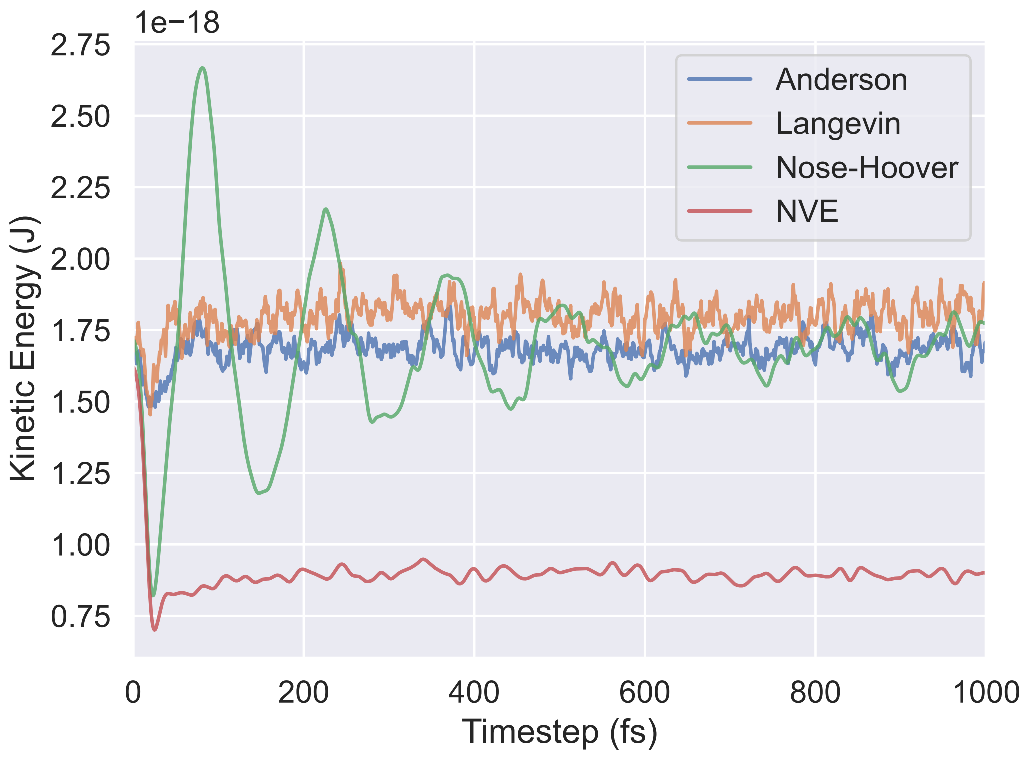


Figure 2.5: The comparison of different thermostats for liquid Argon at 94.4 K.

energy quickly approach to the desired values and then fluctuate around it. On the other hand, the Nose-Hoover undergoes several rounds of stronger fluctuation and then approach to the desired values. The Nose-Hoover approach ensures more accurate ensemble sampling, making it especially suitable for studies requiring precise temperature control over time.

2.8. Summary

In this chapter, we introduced the concept of thermostats within the NVT ensemble, essential for simulating systems where temperature control is required. We began with the motivation behind using thermostats and explored two intuitive methods: the Anderson and Langevin thermostats. The Anderson thermostat, based on stochastic velocity reassignment, is computationally straightforward but may disrupt realistic particle dynamics. The Langevin thermostat, in contrast, incorporates friction and random forces, simulating more natural energy exchange with a thermal bath and providing continuous temperature control.

We then expanded on the need for a more rigorous temperature-control mechanism, introducing the Nosé-Hoover thermostat. This method incorporates an additional degree of freedom, maintaining temperature without arbitrary velocity reassignment by coupling the system to an effective heat reservoir.

Finally, we implemented these thermostats in Python, comparing their performance in controlling the kinetic energy of a simulated liquid argon system. The results, visualized in Fig. 2.5, illustrate how each thermostat maintains a steady kinetic energy level, validating their effectiveness. This exploration provides a foundation for using thermostats in molecular dynamics simulations, enabling accurate temperature control in studies of temperature-dependent material properties.

3. Barostat in the NPT ensemble

3.1. Motivation

In the previous lecture, we introduced temperature control, enabling simulations within the *canonical ensemble* (NVT), where the system maintains a constant temperature by exchanging heat with an external reservoir. This setup allows for more realistic modeling by simulating temperature effects. However, the NVT ensemble still has some limitations. For instance, if you want to model a periodic system at various temperatures, it is natural to consider how thermal expansion affects the system's volume.

Within the NVT ensemble, one has to manually adjust the volume to match the expected thermal expansion for each temperature, a process that can be tedious and requires trial-and-error. Ideally, we want a solution that allows the system to adjust its volume automatically during the MD simulation, responding dynamically to temperature changes.

To address this challenge, we introduce the concept of a barostat (see Fig. 3.1), which functions similarly to a thermostat. While a thermostat maintains a constant temperature, a barostat adjusts the simulation box dimensions and particle positions to ensure that the system stays at the desired pressure. This approach is essential for simulating ensembles like NPT, where both temperature and pressure fluctuations are controlled, making it possible to study realistic material behavior under various temperature and pressure conditions. In the following sections, we will explore two widely used barostat techniques.

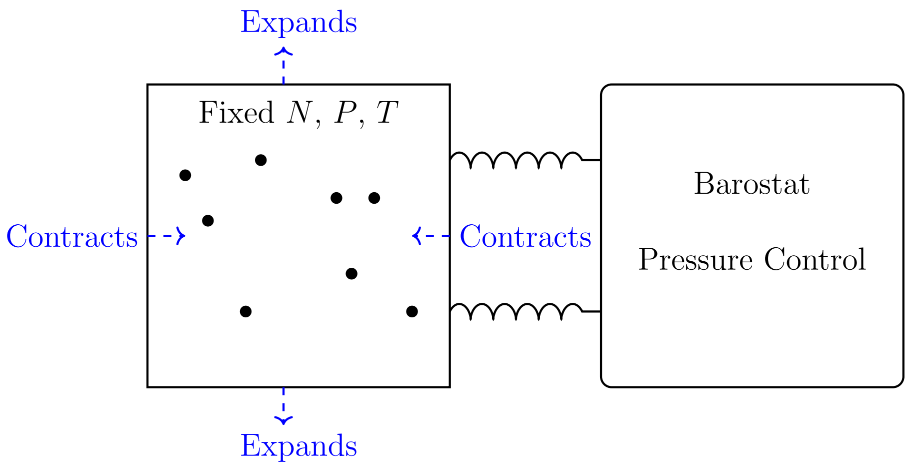


Figure 3.1: The schematic NPT ensemble.

3.2. The Berendsen Barostat

This is a simple barostat that rescales the simulation box gradually toward the target pressure. To implement a barostat, the key idea is to adjust the simulation box size in response to the difference between the current pressure and the target pressure. This is done by scaling the box dimensions and particle positions, and updating the system's volume accordingly.

1. **Compute the instantaneous pressure.** The system pressure P in an MD simulation can be calculated using the virial equation. It includes contributions from the kinetic energy (related to ideal gas) and the virial of the system (related to particle interactions):

$$\begin{aligned} P &= \frac{Nk_BT}{V} + \frac{1}{3V}\sum_{i<j} \mathbf{r}_{ij}\cdot\mathbf{F}_{ij} \\ &= \frac{1}{3V}\left[\sum_i mv_i^2 + \sum_{i<j}\mathbf{r}_{ij}\cdot\mathbf{F}_{ij}\right], \end{aligned} \tag{3.1}$$

where:

- N is the number of particles.
- k_B is the Boltzmann constant.
- T is the temperature.
- V is the volume of the simulation box.
- $\mathbf{r}_{ij}$ is the position vector between particles i and j.
- $\mathbf{F}_{ij}$ is the force acting on particle i due to particle j.
- m_i is the mass of particle i.

2. **Compute pressure difference.** At each time step, calculate the difference between the current and target pressures ($P - P_{\text{target}}$).

3. **Adjust the simulation box volume.** For isotropic pressure control (same scaling in all directions), the new volume is updated by:

$$V_{\text{new}} = V_{\text{old}}\left(1 + \frac{\Delta P}{\tau_P}\cdot dt\right) \tag{3.2}$$

Where:

- τ_P is a time constant controlling the pressure coupling strength.
- dt is the time step.
- V_{old} is the current volume.

4. **Rescale the positions and velocities.** The positions of all particles are scaled accordingly to maintain their relative distances within the simulation box. For isotropic scaling, each position **r** is rescaled:

$$r_{\text{new}} = r_{\text{old}} \cdot \left(\frac{V_{\text{new}}}{V_{\text{old}}}\right)^{1/3} \tag{3.3}$$

The Python code should look like the following:

```python
import numpy as np

def compute_virial_pressure(R, V, volume):
    """
    Compute the pressure using the virial equation.

    Parameters:
    R (np.array): N * 3 array of particle positions.
    V (np.array): N * 3 array of particle velocities.
    volume (float): Volume of the simulation box.

    Returns:
    float: Computed pressure of the system.
    """
    # Number of particles
    N = len(positions)

    # Kinetic contribution to the pressure
    P_kinetic = MASS * np.sum(V**2)

    # Virial contribution to the pressure
    P_virial = 0.0
    for i in range(N):
        for j in range(i + 1, N):
            r_ij = R[i] - R[j]       # Displacement vector
            F_ij = forces[i]         # Force on particle i due to j
            P_virial += r_ij @ F_ij # Dot product r_ij*F_ij

    # The sum of kinetic and virial contributions
    P = (P_kinetic + P_virial) / (3 * volume)

    return P

def Berendsen_barostat(R, V, F, volume, P_target, tau_P):
    """
    The Berendsen barostat used to adjust volume and positions.

    Args:
        R (np.ndarray): particle positions (N, 3)
        V (np.ndarray): particle velocities (N, 3).
        F (np.ndarray): forces acting on each particle (N, 3).
        volume (float): Current volume of the simulation box.
        P_target (float): Target pressure for the system.
        tau_P (float): Time constant for pressure coupling

    Returns:
        R (np.ndarray): Adjusted particle positions after rescaling.
        V (np.ndarray): Adjusted particle velocities after rescaling.
        volume (float): Updated volume of the simulation box.
    """

    # Calculate current pressure
    P = compute_virial_pressure(R, V, volume)

    # Calculate the scaling factor
    dP = P - P_target
    scale_factor = 1.0 + (dP / tau_P) * TIMESTEP
    volume *= scale_factor
    rescale_factor = scale_factor ** (1.0 / 3.0)

    # rescale positions and velocities
    R *= rescale_factor
    V *= rescale_factor

    return R, V, volume
```

This is a relatively simple method, where the system's volume is gradually rescaled to match the target pressure. It does not rigorously conserve the ensemble, but it is computationally efficient and often used for equilibration runs for the simulation of isotropic systems like liquid.

3.3. The Parrinello-Rahman Barostat

The Parrinello-Rahman barostat is a more advanced method for controlling pressure in MD simulations, particularly useful when the system undergoes anisotropic volume changes [9]. Unlike Berendsen barostat that scales the simulation box isotropically, the Parrinello-Rahman barostat allows both the shape and size of the simulation box to change. This is especially important in simulations of materials under strain, phase transitions, or when dealing with anisotropic systems like crystals.

It involves the following steps.

1. **Matrix Representation of box.** To enable this barostat, we first represent simulation box by a matrix $\mathbf{h}$ that defines the three lattice vectors of the simulation box. This matrix allows for changes in both the box size and shape.

$$\mathbf{h} = \begin{pmatrix} \mathbf{a}_x & \mathbf{b}_x & \mathbf{c}_x \\ \mathbf{a}_y & \mathbf{b}_y & \mathbf{c}_y \\ \mathbf{a}_z & \mathbf{b}_z & \mathbf{c}_z \end{pmatrix}$$

   Here, $\mathbf{a}$, $\mathbf{b}$, and $\mathbf{c}$ are the lattice vectors.

2. **Pressure Tensor**: The barostat works with the full pressure tensor $\mathbf{P}$, which describes how pressure acts differently along different directions. The pressure tensor $\mathbf{P}$ can be computed from the system's kinetic energy and virial:

$$\mathbf{P} = \frac{1}{V}\left(\sum_i m_i \mathbf{v}_i \otimes \mathbf{v}_i + \sum_i \mathbf{r}_i \otimes \mathbf{F}_i\right) \tag{3.4}$$

   This equation can be considered as a tensorial version of eq. 3.1.

3. **Strain Rate Tensor W**. The time evolution of the box matrix $\mathbf{h}$ is governed by the equation:

$$\dot{\mathbf{h}} = \mathbf{h} \cdot \mathbf{W} \tag{3.5}$$

where $\mathbf{W}$ is the strain rate tensor, which determines how the box evolves over time.

$$\frac{d\mathbf{W}}{dt} = \frac{1}{Q} \left(\mathbf{P} - P_{\text{target}} \mathbf{I} \right) \tag{3.6}$$

Here, Q is the fictitious barostat mass, and $P_{\text{target}}\mathbf{I}$ is the target pressure tensor.

4. **Update Particle Positions and Box**. Once $\mathbf{h}$ is updated, the particle positions need to be rescaled by the new box matrix. The rescaled positions are calculated as:

$$\mathbf{r}_i = \mathbf{h} \cdot \mathbf{s}_i \tag{3.7}$$

In short, this approach introduces a few additional variables:

1. $\mathbf{h}$: The simulation box matrix, which evolves over time and controls both the size and shape of the box.

2. Q: The fictitious mass associated with the barostat, controlling the rate of volume and shape changes.

3. $\mathbf{W}$: The strain rate tensor, which governs how the box matrix changes over time.

These variables allow the Parrinello-Rahman barostat to apply pressure anisotropically, enabling the box to deform naturally while maintaining the target pressure in the system.

Below is a Python code to achieve the Parrinello-Rahman barostat.

```python
import numpy as np

def compute_pressure_tensor(R, V, volume):
    """
    Compute the internal pressure tensor using the virial equation.

    Parameters:
    R (np.array): N * 3 array of particle positions.
    V (np.array): N * 3 array of particle velocities.
    volume (float): Volume of the simulation box.

    Returns:
    np.array: 3 * 3 pressure tensor.
    """

    # Number of particles
    N = len(positions)

    # kinetic energy contribution to the pressure
    P_kinetic = MASS * np.sum(V**2)

    # virial contribution to the pressure tensor
    P_virial = np.zeros((3, 3))
    for i in range(N):
        for j in range(i + 1, N):
            r_ij = R[i] - R[j]  # Displacement vector
            F_ij = forces[i]  # Force on particle i (be careful)
            P_virial += np.outer(r_ij, F_ij)

    # The sum of kinetic and virial contributions
    P_total = (P_kE * np.eye(3) + P_virial) / volume
    return P_total

def parrinello_rahman_barostat(H, R, V, P_target, Q):
    """
    Update the box and positions using Parrinello-Rahman barostat.
    """

    # Compute current pressure tensor
    volume = np.linalg.det(H)  # Current volume
    P = compute_pressure_tensor(R, V, volume)

    # Compute strain rate tensor (dW/dt)
    W_dot = (P - P_target) / Q

    # Update the box matrix h
    h_new = h + h @ W_dot * dt

    # Rescale the positions to fractional coordinates
    R = R @ np.linalg.inv(h) @ h_new

    return R, h_new
```

3.4. NPT Code Implementation

In this section, we simulate the liquid Argon with an intentionally overestimated volume and then run both NVT and NPT simulations to study the impact of barostat techniques. One should find that the code quite follows what we have described in the previous chapters, except the addition of **Berendsen_barostat** function to allow the control of volume during the MD simulation.

```
import numpy as np
from numba import njit
import matplotlib.pyplot as plt

def initialize_position(L):
    """
    Initialize positions in a 6*6*6 of the FCC unit cell

    Args:
        L (float): unit length of the cubic box

    Returns:
        R (float) : (4*6*6*6, 3) array
    """
    # FCC unit cell fractional positions
    r = np.array([
        [0.0, 0.0, 0.0],
        [0.5, 0.5, 0.0],
        [0.5, 0.0, 0.5],
        [0.0, 0.5, 0.5]
    ])
    a = L/6

    r *= a
    R = []
    for i in range(6):
        for j in range(6):
            for k in range(6):
                R.extend(r + np.array([i, j, k])*a)
    return np.array(R)

def initialize_velocity(N):
    """
    Initialize velocities using Maxwell-Boltzmann distribution

    Args:
        N (int): Number of atoms

    Returns:
        V (float) : (N, 3) array
    """
    # Standard deviation of the velocity distribution
    sigma = np.sqrt(TEMPERATURE * KB / MASS)
    V = np.random.normal(0, sigma, (N, 3))

    # Center the velocities
    V -= np.mean(V, axis=0)
    return V

@njit
def LJ_energy_forces_stress(R, L):
    """
    Compute the energy and forces from the given system

    Args:
        R (float) : (N, 3) array
        L (float): unit length of the cubic box

    Returns:
        PE (float): total energy
        F (float): atomic forces [N, 3] array
        P (float): P_virial
    """
    N = len(R)                  # Number of atoms as a scalor
    F = np.zeros_like(R)        # forces [N, 3] as a 2D array
    PE = 0.0                    # total potential energy as a scalor
    P_virial = 0.0              # if stress is on

    for i in range(N-1):
        for j in range(i + 1, N):
            # Compute R between (i, j)
            r_vec = R[i] - R[j]
            r_vec -= np.round(r_vec / L) * L  # Peridoic condition
            r = np.linalg.norm(r_vec)

            # Compute the potential Energy
            R6 = (SIGMA / r) ** 6
            R12 = R6 ** 2
            PE += 4 * EPSILON * (R12 - R6)

            # Compute and update forces
            force = 24 * EPSILON * (2 * R12 - R6) / r ** 2
            force_vec = r_vec * force
            F[i] += force_vec
            F[j] -= force_vec

            # Compute the virial stress
            P_virial += np.dot(r_vec, force_vec)

    return PE, F, P_virial

def Nose_Hoover_thermostat(R, V, F, xi, L, Q):
    """
    Nose-Hoover thermostat
    """
    # Update R
    R += V * TIMESTEP + 0.5 * F/MASS * TIMESTEP ** 2
    R = R % L

    # Update forces
    PE, F_new, P_virial = LJ_energy_forces(R, L)
    V += 0.5 * (F + F_new) / MASS * TIMESTEP
    V *= (1 - 0.5 * xi * TIMESTEP) / (1 + 0.5 * xi * TIMESTEP)

    # Update xi
    KE = 0.5 * np.sum(V**2) * MASS
    xi += TIMESTEP * (2 * KE / (3 * len(R) * KB * TEMPERATURE) - 1) / (
   Q*MASS)

    # Update forces
    F = F_new
    return R, V, F, xi, PE, P_virial

def Berendsen_barostat(R, V, L, P_virial, tau_P):
    """
    Adjust volume and positions to maintain constant pressure.
    Compute the scalor pressure using the virial equation.

    Args:
        R (np.array): N * 3 array of particle positions.
        V (np.array): N * 3 array of particle velocities.
        L (float): volume of the simulation box

    Returns:
        Updated R, V, L
    """
    volume = L ** 3
    P = compute_pressure(R, V, volume, P_virial)
    dP = P - PRESSURE
    scale_factor = 1.0 + (dP / tau_P) * TIMESTEP

    # Rescale positions and velocities
```

```python
    rescale_factor = scale_factor ** (1.0 / 3.0)
    R *= rescale_factor
    V *= rescale_factor
    L *= rescale_factor
    return R, V, L, P

def MD(L0, barostat=None, Q=1.0, tau_P=0.1, num_steps=500):
    """
    Run MD simulation

    Args:
        barostat (str): "Berendsen", "" or None
        Q (float): Nose-Hoover thermostat parameter
        num_steps (int): Number of steps to simulate
    """
    # Data to monitor
    KEs, PEs, TEs, Pressures, Volumes = [], [], [], [], []

    # Initialize system
    xi = 0.0  # Used by Nose-Hoover
    L = L0
    R = initialize_position(L)
    V = initialize_velocity(N)
    E, F, P_virial = LJ_energy_forces_stress(R, L)


    # MD propogation
    for step in range(num_steps):
        # Thermostat
        R, V, F, xi, PE, P_virial = Nose_Hoover_thermostat(R, V, F, xi,
    L, Q)

        if barostat == "Berendsen":
            R, V, L, P = Berendsen_barostat(R, V, L, P_virial, tau_P)
        else:
            # Compute pressure
            P_KE = MASS * np.sum(V**2)
            P = (P_KE + P_virial) / (3 * L**3)

        # Compute PE, KE, and TE
        KE = 0.5 * np.sum(V**2) * MASS
        vol = L**3

        KEs.append(KE)
        PEs.append(PE)
        TEs.append(PE+KE)
        Volumes.append(vol)
        Pressures.append(P)

        if step % 10 == 0:
            E = KE + PE
            print(f"{step:4d}{PE:12.4e}{KE:12.4e}{E:12.4e}{vol:.5e}")

    return KEs, PEs, TEs, Volumes,
    Pressures

if __name__ == "__main__":

    # Parameters
    KB = 1.3806452 * 1e-23       # Boltzmann constant in J/K
    TEMPERATURE = 94.4           # in K
    PRESSURE = 101325            # in pascal
    EPSILON = 120.0 * KB         # in J (epsilon = 120 * k_B)
    SIGMA = 3.4 * 1e-10          # in meters (3.4 Angstrom)
    MASS = 39.95*1.6747*1e-27    # mass of Argon atom in kg
    L0 = 10.229 * SIGMA          # cubic box side length
    N = 864                      # Number of atoms
    TIMESTEP = 1.0 * 1e-14       # time step in seconds

    results = []
    for params in [("Large Volume", 1.05*L0, None, None),
                   ("Large Volume", 1.05*L0, "Berendsen", 0.001)]:

        (tag, L, barostat, tau_P) = params
        print(f"Simulation with {barostat} barostat {tag}")
        KEs, PEs, TEs, Vs, Ps = MD(L, barostat=barostat, tau_P=tau_P,
   num_steps=2000)
        results.append((tag, barostat, KEs, PEs, TEs, Vs, Ps))

    fig, axs = plt.subplots(2, len(results), figsize=(16, 6))
    for i, result in enumerate(results):
        (tag, barostat, KEs, PEs, TEs, Vs, Ps) = result
        if barostat is None:
            barostat = "NVT"
        else:
            barostat += " NPT"
        axs[0, i].set_title(tag + '-' + barostat)
        axs[0, i].plot(KEs, label="KE")
        axs[0, i].plot(PEs, label="PE")
        axs[0, i].plot(TEs, label='Total')
        axs[0, i].set_ylim([-1.0e-17, 0.3e-17])
        axs[0, i].set_ylabel("Energy")

        ax_vol = axs[1, i]
        ax_pre = ax_vol.twinx()
        ax_vol.plot(Vs, color="b")
        ax_pre.plot(Ps, color="r")
        ax_vol.set_ylabel("Volume", color="b")
        ax_vol.tick_params(axis="y", labelcolor="b")
        ax_pre.set_ylabel("Pressure", color="r")
        ax_pre.tick_params(axis="y", labelcolor="r")
        ax_vol.set_ylim([3.6e-26, 5.1e-26])
        ax_pre.set_ylim([-3.0e+8, 1.2e+8])

        axs[0, i].set_xlim([0, 2000])
        axs[1, i].set_xlim([0, 2000])
        axs[0, i].grid(False)
        ax_vol.grid(False)
        ax_pre.grid(False)

    plt.savefig("lec_03-npt-nvt.pdf")
```

In the end, it generates a 2×2 subplots as shown in Fig. 3.2, in which each column describes the evolution of energies in the upper panel and the evolution of volumes and pressure values in the lower panel for regular NVT and Berendesen NPT simulations, respectively. As compared to the NVT setup, the consideration of Berendsen barostat

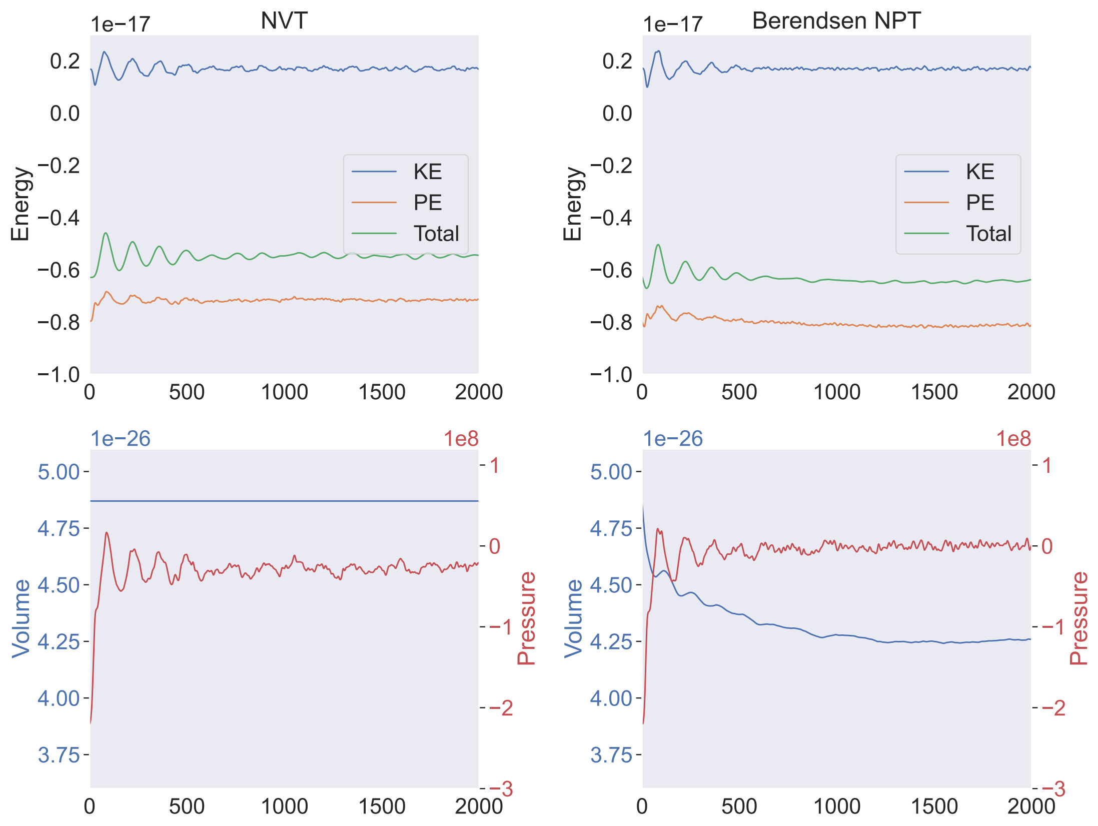


Figure 3.2: The comparison of NVT and NPT simulation of liquid argon.

can effectively adjust the volume by itself according to the target pressure condition, thus providing more realistic modelling of the system.

3.5. Summary

In this chapter, we introduced how to implement pressure control (barostat) in the context of MD simulations. Similar to temperature control, the key lies in assigning an appropriate updating rule for the simulation cell. For isotropic systems such as liquids or gases, the Berendsen barostat provides an efficient method for pressure regulation. On the other hand, the Parrinello-Rahman barostat is well-suited for anisotropic systems, such as solids, where the lattice can deform in response to stress.

Finally, we implemented the Berendsen barostat in Python to simulate liquid argon. By comparing the results with established methods, the implementation demonstrates accuracy and efficiency in pressure control. This exploration provides a solid foundation for incorporating barostats into MD simulations, complementing thermostats to enable accurate modeling of pressure- and temperature-dependent material properties.

4. MD Simulation with LAMMPS

4.1. Introduction to LAMMPS

`LAMMPS` (Large-scale Atomic/Molecular Massively Parallel Simulator) is an open-source software tool designed for performing classical MD simulations. It can be used to model an array of particle interactions, ranging from simple atomic systems to complex materials and biomolecular systems. As one of the most popular materials simulation packages, `LAMMPS` is specifically optimized for large-scale simulations, which involve millions to billions of particles, making it suitable for high-performance computing (HPC) environments. Its versatility allows for simulations of a variety of systems such as metals, polymers, proteins, and membranes. Some typical applications include:

- **Crystal Defects and Deformation:** (e.g., dislocation motion, grain boundary evolution, and fracture in materials).
- **Phase Transitions:** Simulating phase changes in metals and alloys, such as melting, solidification, or the formation of microstructures.
- **Transport properties of materials** via Green-Kubo or direct methods.
- **Mechanical Properties** such as stress-strain relationships, elasticity, and plasticity at the atomic scale.
- **Drug Interactions:** Simulating how drug molecules interact with proteins or other biological targets.

4.2. Why is `LAMMPS` Efficient?

`LAMMPS` is designed to perform large-scale simulations by taking advantage of parallel computing architectures, including multi-core processors and HPC clusters. In particular, `LAMMPS` uses the domain decomposition technique to divide the simulation space into subdomains. Each processor or core is assigned to a subdomain, and they work together by communicating boundary conditions and interacting forces. `LAMMPS` also uses Message Passing Interface (MPI) to handle communication between processors, ensuring minimal overhead and efficient data transfer during the simulation. Thanks to these considerations, `LAMMPS` has demonstrated excellent scalability across thousands of processors, which makes it suitable for simulating systems with millions to billions of particles over long time scales.

The use of neighbor list reduces computational cost by avoiding direct distance calculations between all possible pairs of particles (scaling as $O(N^2)$). Then it is updated periodically based on a user-specified frequency. The distance moved by particles since the last update (must remain within the skin distance).

In parallel simulations, neighbor lists are constructed locally on each processor. This ensures that each processor maintains neighbor lists for particles it owns and for ghost atoms (particles in neighboring subdomains). Communication between processors ensures that ghost atom data is up to date.

4.3. Input and Output Files

After you compile the `LAMMPS` code into an executable (often called `lmp_serial` or `lmp_mpi` by default), the following command can be used to invoke a calculation on your local computer terminal.

```
path_to_lmp_mpi < lmp.in > lmp.out
```

This command involves the preparation of input and output that will be discussed as follows.

4.3.1 LAMMPS Input Files

`LAMMPS` simulations are controlled by input script (often called `lmp.in`), which consists of a series of commands written in a simple text format. These scripts define the simulation parameters, system setup, and specific instructions for running the simulation. A typical `LAMMPS` input script is organized into several key sections:

Initialization Section to include the units/boundary conditions/atom styles.

- `units real`: units for physical quantities (e.g., distance in angstroms).
- `boundary p p p`: periodic boundary conditions in all three dimensions.
- `atom_style atomic`: how atoms are represented (e.g., atomic).

Atom Definition Section to includes atomic coordinates/initial velocities.

- `read_data data.file`: reads the atomic configuration from a file.
- `velocity all create 300.0 12345`: random initial velocities at 300 K

Force Field Definition Section to defines the force field parameters

- `pair_style lj/cut 2.5`: LJ potential with a cutoff distance of 2.5.
- `pair_coeff * * 0.1 3.0`: LJ coefficients (epsilon and sigma)

Simulation Parameters Section to define timestep/temperature/pressure.

- `timestep 1.0`: Time step for integration for the given time unit
- `fix 1 all nve`: Applies a NVE ensemble to all atoms.

Output Control Section for frequency and format of the output.

- `thermo 100`: Prints data (e.g., temperature) every 100 steps.
- `dump 1 all atom 1000 dump.atom`: Outputs positions every 1000 steps.

Run Section to invoke the actual iterative simulation.

- `run 10000`: Runs the simulation for 10,000 timesteps.
- `minimize 1.0e-4 1.0e-6 1000 10000`: Energy minimization.

4.3.2 LAMMPS Output Files

`Log files` are an essential output generated by the `LAMMPS` during a simulation run. They record detailed information about the simulation process, including configuration details, computational settings, and simulation results. The log file serves as a useful resource for debugging, analyzing results, and verifying the simulation's correctness.

`Dump files` store detailed trajectory information about the system's atomic coordinates, velocities, and forces. These files are typically used for post-processing to analyze system configurations, create visualizations, or calculate structural properties.

`Restart Files` store the entire state of a simulation, allowing users to pause and later continue a simulation from where it left off. These files contain information about the atom positions, velocities, forces, and other system properties.

4.4. Simulation Process

Below is a minimal script to simulate argon atoms using the Lennard-Jones potential as we discussed in the previous lectures:

```
units real
atom_style atomic
read_data argon.data
pair_style lj/cut 2.5
pair_coeff * * 0.238 3.4                # LJ potential for argon
velocity all create 300.0 12345

fix 1 all nvt temp 300.0 300.0 100.0    # NVT ensemble at 300 K
timestep 1.0                            # Time step of 1.0 fs
run 50000                               # Run 50,000 timesteps
```

4.5. Post-Processing

After a simulation, the results need to be visualized and analyzed. `LAMMPS` produces several types of output files, which contain thermodynamic data, atom positions, velocities, and forces. VMD (Visual Molecular Dynamics) and OVITO (Open Visualization Tool) are popular tools for visualizing molecular dynamics simulations.

4.6. Running LAMMPS on HPC

For most research projects, running `LAMMPS` on a HPC environment is essential for large-scale simulations that require significant computational resources. Most modern supercomputers use job schedulers like SLURM to manage computational tasks.

```
#!/bin/bash
#SBATCH --job-name=lammps_job         # Job name
#SBATCH --nodes=4                     # Number of nodes
#SBATCH --ntasks-per-node=32          # Number of processes per node
#SBATCH --time=24:00:00               # Max time limit (HH:MM:SS)
#SBATCH --partition=compute           # Partition or queue to submit to
#SBATCH --output=job_output.log       # Output log file

module load lammps/3Mar2020           # Load LAMMPS module
mpirun -np 128 lmp_mpi -in file.in    # Run LAMMPS with 128 processes
```

- `SBATCH {options` are used to specify the number of nodes, tasks, job name, and time limit.
- `mpirun -np 128` launches the calculation across 128 processes in parallel, ensuring that the simulation scales across multiple cores.
- `lmp_mpi` is the parallel version of LAMMPS used for multi-node execution.

4.7. MD simulations of Argon via LAMMPS

In general, `LAMMPS` can be compiled and installed on various platforms, including personal computers, HPC clusters, and cloud-based environments. For instructional purposes, we will use the Google Colab Jupyter Notebook environment to demonstrate the setup process.

First, `LAMMPS` can be installed in Google Colab using the following commands:

Installing LAMMPS in Google Colab

```
!sudo apt-get update
!sudo apt-get install -y lammps
```

These commands will install the LAMMPS package along with its required dependencies. After installation, you can verify whether LAMMPS has been successfully installed by running the following command:

Verifying LAMMPS Installation

```
!lmp -h
```

This will print the `LAMMPS` help message, which confirms that the installation was successful and provides an overview of available commands and options. If `LAMMPS` is not installed correctly, the output will indicate the issue, and you may need to troubleshoot or reinstall.

Next one can prepare the LAMMPS input files and execute the command as follows.

```
# Define the LAMMPS BASE input script as strings
BASE = """
# Units
units lj
atom_style atomic

# Box and Atoms
lattice fcc 0.8442
region box block 0 10 0 10 0 10
create_box 1 box
create_atoms 1 box

mass 1 39.95
velocity all create 1.44 87287 loop geom

# Force Field
pair_style lj/cut 2.5
pair_coeff 1 1 1.0 1.0 2.5

# Control of neighbor calculations
neighbor 0.3 bin
neigh_modify every 20 delay 0 check no

# Output
thermo 100
dump 1 all atom 50 dump.lammpstr
"""

# Create the lammps input file
with open("lmp_nve.in", "w") as file:
    file.write(BASE + "\n")
    file.write("fix 1 all nve\n")
    file.write("RUN 1000\n")

# Execute the lammps command
!lmp < lmp_nve.in > lmp_nve.out
```

In the above script, we first generate a base input files to set up the system and force field for a FCC argon. Using the base input, we further added two line to let the system run 1000 steps with the NVE ensemble. One should observe the following output in the log file.

```
LAMMPS (29 Sep 2021 - Update 2)
  using 1 OpenMP thread(s) per MPI task
Lattice spacing in x,y,z = 1.6795962 1.6795962 1.6795962
Created orthogonal box = (0.000 0.000 0.000) to (10.077 10.077 10.077)
  1 by 1 by 1 MPI processor grid
Created 864 atoms



Per MPI rank memory allocation (min/avg/max) = 3.70 | 3.70 | 3.70 Mbytes
Step Temp E_pair E_mol TotEng Press
       0         1.44   -6.7733681            0   -4.6158681   -5.0210763
```

```
     100    0.7520781   -5.7514584            0    -4.624647   0.22224801
     200   0.76319587   -5.7681312            0   -4.6246624    0.1876998
     300   0.75569127   -5.7576689            0    -4.625444   0.24985373
     400   0.71772096   -5.6993903            0   -4.6240549   0.51935749
     500   0.71112191   -5.6904918            0   -4.6250435   0.53347476
Loop time of 0.336132 on 1 procs for 500 steps with 864 atoms

Total # of neighbors = 32356
Ave neighs/atom = 37.449074
Neighbor list builds = 25
Dangerous builds not checked
Total wall time: 0:00:00
```

Finally, one can repeat the NVT and NPT simulations, and analyze the results as we introduced in previous chapters.

4.8. Summary and Further Tasks

In this chapter, we provided a brief introduction to the LAMMPS package, highlighting its efficiency, basic setup, and showcasing a simple example to illustrate its functionality. For further exploration and a deeper understanding of LAMMPS, readers are encouraged to:

- Explore the implementation of the neighbor list in the `LAMMPS` source code to understand how interactions are efficiently managed.
- Review the thermostat and barostat algorithms used in `LAMMPS`, comparing them with previous Python implementations discussed in earlier chapters. Key files in the `LAMMPS` codebase include:
    1. `compute_pressure.cpp`: Computes pressure and related properties.
    2. `fix_press_berendsen.cpp`: Implements the Berendsen barostat.
    3. `fix_nh.cpp`: Implements the Nosé-Hoover thermostat and barostat.
    4. `pair_lj_cut.cpp`: Handles Lennard-Jones potential with a cutoff.
- Identify and consider the types of MD simulations you would like to explore with `LAMMPS`, such as equilibrium or nonequilibrium dynamics, structural transitions, or transport property calculations.

By delving into these aspects, you can gain a more comprehensive understanding of `LAMMPS` and its capabilities, equipping you to tackle a wide range of molecular dynamics simulations.

5. MD Structural Characterization

So far, we have learned some fundamentals about how to write code and run an MD simulation to model the atomistic processes of materials. By running the simulation, we expect to generate a set of time-dependent atomic trajectories by solving Newton's equations of motion for a system of particles. Next, it is important to understand these simulation results. Indeed, it is essential to extract meaningful physical properties from the simulation results. In this lecture, we will cover several fundamental post-analysis techniques to understand the structural behaviors from a MD simulation.

5.1. MD Trajectory Visualization

Visualization is an essential tool for analyzing MD simulations. It enables researchers to visually inspect the system's dynamics, identify abnormalities, and better understand atomic movements. Among the various visualization tools available, OVITO is widely used due to its versatility and user-friendly interface.

When working with MD trajectories, it is crucial to customize the visualization settings to highlight specific features of interest. A universal visualization setting often fails to reveal detailed information, particularly when dealing with complex systems involving many atoms. For example:

1. Adjusting Atomic Sizes and Colors: Enhance the visibility of certain atoms by resizing their spheres or assigning distinct colors based on the atomic properties of interest (e.g., chemical identity, local coordination numbers, atomic energies, .etc). This is particularly useful when focusing on specific atomic species or regions.

2. Deemphasizing Bulk Atoms: Suppress the representation of bulk atoms to emphasize atoms in regions of interest, such as defects, surfaces, or grain boundaries.

3. Drawing Bonds: Visualize bonds between atoms when analyzing molecules or identifying molecular structures.

4. Dynamic Filtering: Apply filters to isolate specific atomic trajectories, highlight displacements, or track changes in local environments.

Customizing settings such as coloring schemes, transparency levels, and rendering styles allows researchers to adapt visualizations to their specific analysis goals. For instance, highlighting atoms near defects or interfaces can reveal critical insights into structural transitions. This process of tailoring visualization is often referred to as creating a **pipeline**, where a series of visualization and processing steps are designed and executed systematically.

Modern visualization software, such as OVITO, further streamlines this process by allowing users to save and reuse pipelines for other simulations. This feature enables automation of the visualization process, reducing the reliance on interactive adjustments and saving considerable time. Additionally, pipelines can be scripted to process large datasets or batch analyze results efficiently, integrating seamlessly into high-throughput workflows.

For detailed guidance on OVITO's advanced capabilities, including property-based filtering, rendering techniques, and pipeline automation, please refer to the documentation and associated publications. By leveraging these tools, researchers can ensure consistent, reproducible, and insightful visualizations tailored to their specific scientific needs.

5.2. Radial Distribution Function

While it is always appealing to see the nice images or movies to understand the structural features and transitions in a subjective manner, we still rely on some metrics to quantify the structural changes during the simulation. With such metrics, we can minimize the human bias, detect and measure changes, even those too subtle for the human eye.

One of the most widely used structural metric is the Radial Distribution Function $g(r)$, which measures the probability of finding a particle at a distance r from a reference particle.

$$g(r) = \frac{V}{N^2} \left\langle \sum_{i=1}^{N} \sum_{j \neq i}^{N} \delta(r - r_{ij}) \right\rangle \cdot \frac{1}{4\pi r^2 \Delta r} \tag{5.1}$$

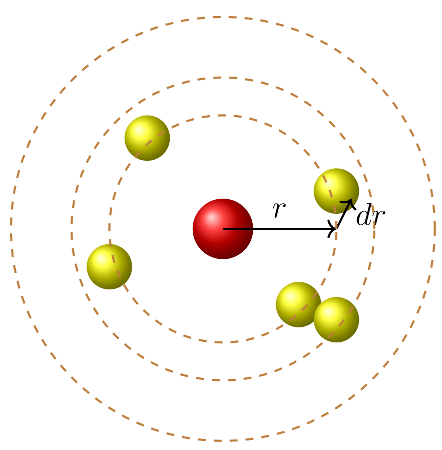
Where:

- V is the volume of the system.
- N is the number of particles.
- r_{ij} is the distance between particles i and j.
- $\delta(r - r_{ij})$ is the Dirac delta function ensuring that only pairs with separation r_{ij} equal to r contribute.
- $4\pi r^2 \Delta r$ is the volume of a spherical shell at distance r with thickness Δr.

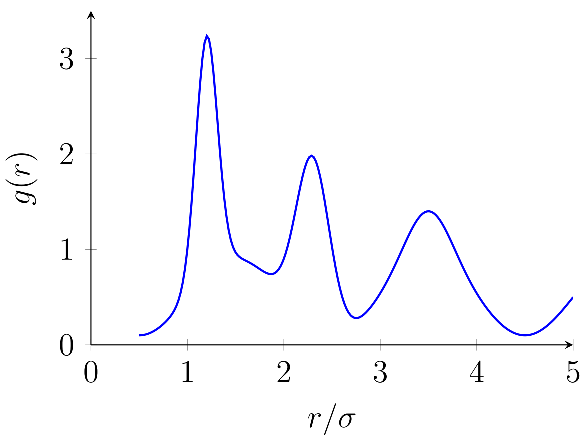
As shown in Fig. 5.1, the computation of RDF involves counting the number of neighboring atoms within a series of concentric shells around a reference particle, followed by normalization. This process essentially depicts the spatial distribution of neighboring particles' density.

5.2.1 Physical Meaning and Applications

In a solid, the RDF typically exhibits a series of sharp peaks corresponding to well-defined first, second and third nearest neighboring interatomic distances, reflecting the long-range periodic order characteristic of crystalline materials. In contrast, in a liquid, the RDF displays a prominent first peak representing the nearest neighbors, followed by dampened oscillations, indicative of short-range order but the absence of long-range periodicity.



(a)



(b)

Figure 5.1: (a) Radial distance and shells, (b) Radial distribution function

The RDF is particularly valuable for identifying short-range and long-range structural order in various phases of matter, such as liquids, solids, and amorphous materials. This makes it a fundamental tool for understanding structural transitions and correlations in molecular simulations, enabling researchers to quantify and analyze the underlying atomic or molecular arrangements with minimal bias.

5.2.2 Computation of RDF

The computation of RDF is very similar to the calculation of energy and forces since both need to identify the neighbors. The whole process is outlined as in Algo. 2.

Algorithm 2 Compute Radial Distribution Function

```
1: Start
2: Compute the distance pairs between particles
3: Group distances into bins based on their values
4: Update each bin's count according to the grouped distances
5: Normalize the counts by dividing each by the number of pairs in the bin
6: Calculate:
       1. Total number of pairs
       2. Shell Volume: $4\pi r^2 \Delta r$
       3. Particle Density: $N/V$
7: End
```

While the LAMMPS or other main-stream codes can conveniently output the RDF raw data via some simple commands, we provide a sample Python code below in case one needs to process RDF manually.

```python
def compute_rdf(positions, num_bins=100, r_max=10.0):
    N = len(positions)  # Number of atoms
    rdf = np.zeros(num_bins)
    dr = r_max / num_bins
    for i in range(N):
        for j in range(i+1, N):
            r = np.linalg.norm(positions[i] - positions[j])
            if r < r_max:
                bin_index = int(r / dr)
                rdf[bin_index] += 2  # Each pair counted twice

    # Normalize RDF
    r = np.linspace(0, r_max, num_bins)
    rdf /= (4 * np.pi * r**2 * dr * N)
    return r, rdf

# Example usage

positions = np.random.rand(100, 3) * 10  # Generate random positions
r, g_r = compute_rdf(positions)

# Plot RDF

plt.plot(r, g_r)
plt.xlabel("r (Angstrom)")
plt.ylabel("g(r)")
plt.title("Radial Distribution Function")
plt.show()
```

RDF is commonly used to characterize the short-range order in liquids and gases. In a RDF, we are interested in the location of Peaks and their spreads, as they indicate common interatomic distances (e.g., 1st and 2nd nearest-neighbor distance).

5.3. Vibration Spectrum

In addition to RDF, another essential characterization is understanding how particles in a system vibrate. Experimentally, such information is obtained using techniques like Infrared (IR) and Raman Spectroscopy or Inelastic Neutron/X-ray Scattering. In molecular dynamics (MD) simulations, an analogous measurement is the **Vibrational Density of States** (VDOS).

VDOS describes the distribution of vibrational frequencies within a system, providing insights into the dynamics of atomic motion. By examining an MD trajectory, we observe that atoms vibrate in various modes, each with different frequencies and amplitudes. How to calculate the VDOS from MD simulations?

5.3.1 Vibration Frequency of a Single Harmonic Oscillator

We can address this challenge with some simple cases. Let's imagine the simplest case of a single harmonic oscillator. The velocity of an atom is related to its position by:

$$v(t) = \frac{d}{dt}r(t) \tag{5.2}$$

For sinusoidal motion, $r(t) = A\cos(\omega t + \phi)$, where A is the amplitude, the velocity becomes:

$$v(t) = -A\omega\sin(\omega t + \phi) \tag{5.3}$$

This shows that the velocity oscillates at the same frequency ω as the position but is phase-shifted. To analyze the vibrational frequency, we compute the power spectrum of the velocity. For a single harmonic oscillator, we can compute the time correlation as follows,

$$C_v(t) = \frac{1}{T}\int_0^T v(0)v(t)\,dt = \langle v(0)v(t)\rangle$$

In the later expression, the **Dirac notation** is used to simplify the equation. We call it **velocity autocorrelation function** (VACF). Substituting $v(t) = -A\omega\sin(\omega t + \phi)$, the VACF becomes:

$$C_v(t) = \langle(-A\omega\sin(\phi))(-A\omega\sin(\omega t + \phi))\rangle = A^2\omega^2\langle\sin(\phi)\sin(\omega t + \phi)\rangle$$

Thus, $C_v(t)$ is a sinusoidal function with a frequency of ω. The Fourier transform of $C_v(t)$ yields the vibrational density of states (VDOS):

$$g(\omega) = \int_{-\infty}^{\infty} C_v(t)e^{-i\omega t}dt$$

For a single harmonic oscillator, this results in a sharp peak at the natural frequency ω.

Imagine that there exists two kinds of harmonic motions in the system, $g(\omega)$ should reveal two frequencies at which the velocities oscillate. And the magnitudes of each peak should tell the intensity of each harmonic motion.

5.3.2 Python Simulation of Harmonic Oscillators

To better understand the concept of VACF and VDOS, we can perform a simple numerical experiment by simulating two model systems based on the O_2 diatomic molecule, as illustrated in Fig. 5.2. In O_2, the equilibrium bond length is approximately 1.16 Å, with a bond spring constant of $k = 1180, \mathrm{N/m}$.

In the first model, we simulate simple harmonic vibration by initializing the positions of the atoms with a slight deviation from the equilibrium bond length. In the second model, we simulate a combination of vibration and rotation by assigning initial velocities that are misaligned with the bond axis. We then use the following MD code to simulate the motions of the molecule and compute the corresponding VACF and Vibrational Density of States (VDOS).

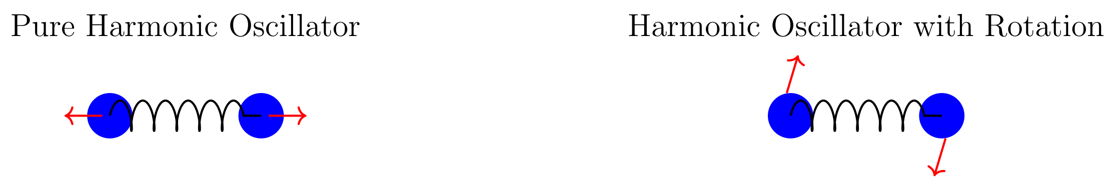


Figure 5.2: Two example harmonic oscillators.

```python
import numpy as np
import matplotlib.pyplot as plt
from scipy.fftpack import fft

def harmonic_force(r1, r2, k, r_eq):
    # Function to compute force due to harmonic potential
    r12 = np.linalg.norm(r2 - r1)
    force_mag = -k * (r12 - r_eq)
    force = -force_mag * (r2 - r1) / r12
    return force

def MD(r1, r2, v1, v2, N_steps):
    velocities = np.zeros([N_steps, 6])
    F12 = harmonic_force(r1, r2, k, r_eq)

    for step in range(N_steps):
        # Verlet update
        r1 += v1 * dt + 0.5 * F12 / mass * dt ** 2
        r2 += v2 * dt - 0.5 * F12 / mass * dt ** 2

        F12_new = harmonic_force(r1, r2, k, r_eq)

        v1 += 0.5 * (F12 + F12_new) * dt / mass
        v2 -= 0.5 * (F12 + F12_new) * dt / mass

        F12 = F12_new
        velocities[step][:3] = v1
        velocities[step][3:6] = v2
    return velocities

# Parameters for the simulation
dt = 1e-15          # in s
mass = 2.55e-26     # in kg
r = 1.16e-10        # in m
k = 1180            # in N/m
data = [("O$_2$ (vibration)", 2000, False),
        ("O$_2$ (vibration + rotation)", 5000, True)]


for i, (name, N_steps, rotate) in enumerate(data):

    # Initial positions
    r1 = np.array([0.0, 0.0, 0.0])
    r2 = np.array([0.0, 0.0, r*1.2])

    # Initialize velocities
    v1 = np.zeros(3)
    v2 = np.zeros(3)
    if rotate:
        v1[0] += 50
        v2[0] -= 50

# MD simulation
Vs = MD(r1, r2, v1, v2, N_steps)

fig, axs = plt.subplots(2, len(data), figsize=(12, 6))

# Plot VACF
VACF = np.array([np.dot(Vs[0], Vs[t]) for t in range(N_steps)])
axs[0, i].plot(np.arange(N_steps) * dt * 1e12, VACF)
axs[0, i].set_title(name)
axs[0, i].set_xlabel("Time (ps)")
axs[0, i].set_ylabel("VACF")

# Plot VDOS
```

```python
VDOS = np.abs(fft(VACF))**2
freqs = np.fft.fftfreq(N_steps, dt) / 1e12
axs[1, i].plot(freqs[:N_steps//2], VDOS[:N_steps//2])
axs[1, i].set_xlabel("Frequency (THz)")
axs[1, i].set_ylabel("log-VDOS")
axs[1, i].set_xlim([0, 60])
axs[1, i].set_yscale("log")
plt.show()
```

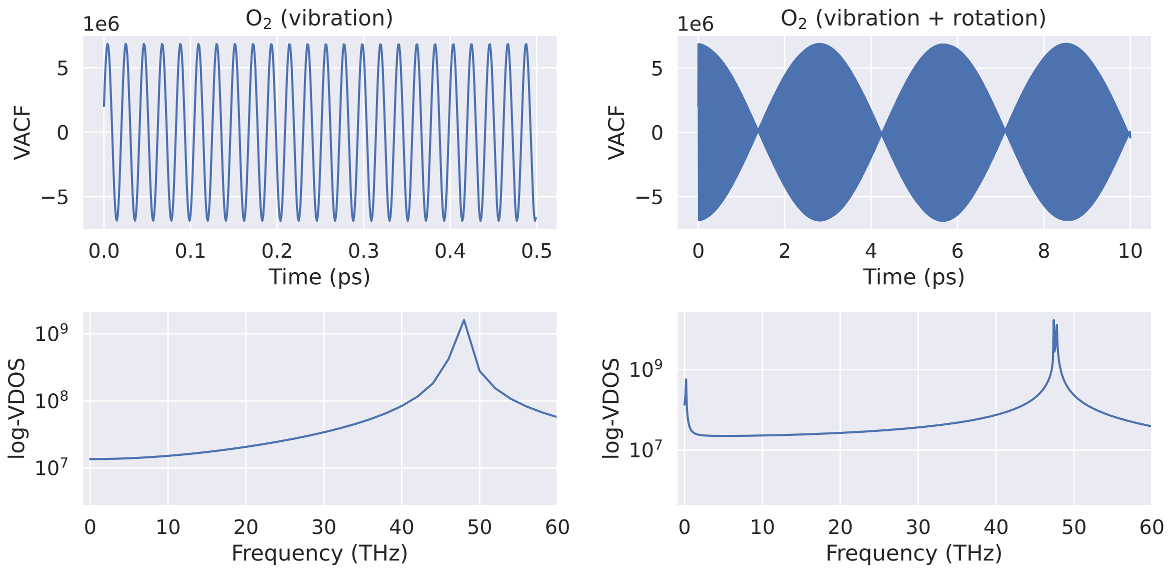


Figure 5.3: The simulated VACF and VDOS for the harmonic oscillators.

Fig. 5.3 shows the simulated VACF and VDOS for the harmonic oscillators. For ideal harmonic oscillators, the VACF exhibits periodic and consistent behavior over time, as expected. Using Fourier analysis, we can conveniently extract the vibrational frequencies for both single and multiple harmonic oscillators.

For pure O_2, the vibrational frequency is centered around 47 THz, which aligns well with predictions from classical mechanics:

$$\frac{1}{2\pi}\sqrt{\frac{k}{m}} = \frac{1}{2 \times 3.14159}\sqrt{\frac{1180}{2 \times 2.66 \times 10^{-26}}} \times 10^{-12} = 47.40 \text{ THz}$$

For the case of hybrid oscillators, the VDOS additionally shows a smaller frequency peak near 0.1 THz, corresponding to the rotational motion of the molecule, as computed below.

The angular frequency for a diatomic rotor.

To compute the angular frequency for a diatomic rotor, such as an O_2 molecule, one can use the rigid rotor model from classical mechanics. The angular frequency ω is related to the molecule's moment of inertia I and the rotational energy level E.

1. The moment of inertia of a diatomic molecule about its center of mass is given by:

$$I = \frac{m_1 m_2}{m_1 + m_2} r^2 = (2.66 \times 10^{-26}\mathrm{kg}/2) \times (1.16 \times 10^{-10}\mathrm{m})^2 = 1.78 \times 10^{-46}\ \mathrm{kg{\cdot}m^2}.$$

2. The rotational energy levels in quantum mechanics are given by:

$$E_J = \frac{\hbar^2 J(J+1)}{2I},$$

Between $J = 1$ and $J = 0$, the energy difference is:

$$\Delta E = \frac{\hbar^2(2)}{2I} = \frac{(1.054 \times 10^{-34})^2 \times 2}{2 \times (1.78 \times 10^{-46})} = 6.25 \times 10^{-23}\,\mathrm{J}.$$

3. Angular frequency is related to energy difference by:

$$\omega = \frac{\Delta E}{\hbar} = \frac{6.25 \times 10^{-23}}{\times 1.054 \times 10^{-34}} = 5.93 \times 10^{11}\ \mathrm{rad/s} = 0.09\ \mathrm{THz}.$$

5.3.3 Many oscillators

In a system of N atoms, each atom has 3 degrees of freedom (one for each Cartesian coordinate: x, y, and z). Therefore, the total number of degrees of freedom is $3N$. To understand the pattern of the collective vibrations, we can decompose it into a sum of normal modes, each vibrating at a distinct frequency. Each **normal mode** corresponds to one of these degrees of freedom in terms of collective motion, and each mode in a system is orthogonal to the others.

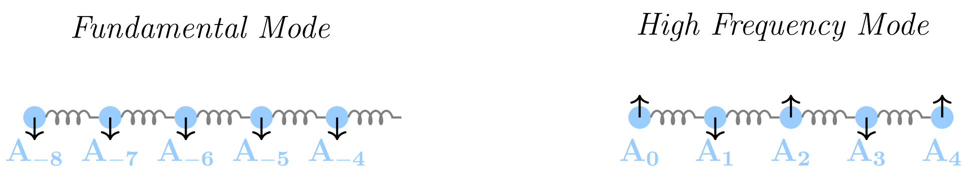

Fundamental Mode
High Frequency Mode
A_{-8} A_{-7} A_{-6} A_{-5} A_{-4}
A_0 A_1 A_2 A_3 A_4

Figure 5.4: Diagram illustrating normal modes in a one-dimensional atomic chain. The left panel represents the fundamental mode, where all atoms move in the same direction with a long wavelength. The right panel represents a higher frequency mode, where adjacent atoms oscillate in opposite directions with a shorter wavelength.

In experiments, you can measure vibrations using IR, Raman, or Neutron scattering. In an MD simulation, we extract vibrational frequencies from each normal mode by analyzing the VACF. The **VACF**, denoted $C(\tau)$, measures how the velocity of a particle at a given time t correlates with its velocity at some later time $t + \tau$. It is useful for understanding particle dynamics and is related to the vibrational properties and transport coefficients (like diffusion).

$$C(\tau) = \frac{1}{N} \sum_{i=1}^{N} \langle \mathbf{v}_i(0) \cdot \mathbf{v}_i(\tau) \rangle \tag{5.4}$$

On the other hand, VDOS provides information about the frequencies at which particles in a system vibrate. The VDOS is computed by taking the Fourier transform of the VACF. In practice, because simulations are finite and VACF data is computed over a limited time interval, we typically use the discrete Fourier transform (DFT) or fast Fourier transform (FFT) to compute the VDOS numerically:

$$D(\omega) = \frac{1}{2\pi} \int_0^\infty C(\tau) \cos(\omega\tau) d\tau \tag{5.5}$$

5.4. LAMMPS Simulation of RDF and VDOS

For realistic systems, LAMMPS can directly compute both RDF and VACF. This enables users to obtain these properties from LAMMPS output and plot them for further analysis. Below we show a script to set up the lammps for both calculations based on the example of liquid argon.

5.4.1 LAMMPS Setup

LAMMPS setup in Colab.

```
!sudo apt-get update
!sudo apt-get install -y lammps
```

```
lammps_script = """
# Initial setup of the simulation
units           metal                    # Use Metal units
dimension       3                        # 3D simulation
boundary        p p p                    # PBC conditions
atom_style      atomic                   # Atomic style

# Declaring necessary parameters
variable T        equal 94.4             # Temperature K
variable epsilon  equal 120*8.61733e-5   # eV
variable sigma    equal 3.4              # Angstrom
variable L        equal 10.229*${sigma}  # Box length
variable a        equal $L/6             # Lattice parameter

# Generating supercell
lattice         fcc $a
region          box block 0 1 0 1 0 1 units lattice
create_box      1 box
create_atoms    1 box
replicate       6 6 6

# Setting up Simulation
```

```
mass            1 39.95
pair_style      lj/cut 12.0
pair_coeff      1 1 ${epsilon} ${sigma}
timestep        1.0
neighbor        0.3 bin
thermo          1000
thermo_style    custom step temp epair etotal ke pe press vol
velocity        all create $T 12345
fix             1 all npt temp $T $T 100 iso 1.0 1.0 1.0

# RDF calculation
compute     myRDF all rdf 100
fix         2 all ave/time 100 1 100 c_myRDF[*] file rdf.txt mode vector
run         10000
unfix       2

# VACF calculation
reset_timestep 0
compute     myVACF all vacf
fix         3 all ave/correlate 10 2000 20000 c_myVACF[4] file vacf.txt
    ave running
run         20000
"""

# Save the script to a file
with open("lmp_nvt_rdf_vacf.in", "w") as file:
    file.write(lammps_script)

!lmp -in lmp_npt_rdf_vacf.in
```

In this script, we set up a MD simulation in LAMMPS to compute the RDF and VACF for a model system of liquid argon. The system model, LJ potential and ensemble choice have been discussed in the previous chapters. Hence we will focus on the illustration of RDF and VACF commands in LAMMPS.

```
compute RDF all rdf 100
fix 2 all ave/time 100 1 100 c_RDF[*] file rdf.txt mode vector
run 10000
```

The above commands bin RDF into 100 intervals. Results are averaged over 100 steps using fix ave/time and written to the file *rdf.txt*. Running 10000 steps will generate 100 RDF files in *rdf.txt*.

```
compute VACF all vacf
fix 3 all ave/correlate 10 2000 20000 c_VACF[4] file vacf.txt ave running
run 20000
```

The above commands calculates VACF for all atoms. Results are saved to vacf.txt. The 2nd line averages the computed VACF over 10 timesteps, with a total of 2000 correlation steps, and then saves the averaged VACF data to the file vacf.txt. And the ave running option ensures that the averaging is cumulative across the simulation.

5.4.2 Plotting RDF, VACF and VDOS

Once the simulation completes, the **rdf.txt** file is organized as follows:

```
# Time-averaged data for fix 2
# TimeStep Number-of-rows
# Row c_myRDF[1] c_myRDF[2] c_myRDF[3]
0 100
1 0.085 0 0
2 0.255 0 0
...
100 16.915 1.05312 458.937
200 100
1 0.085 0 0
2 0.255 0 0
...
...
10000 100
1 0.085 0 0
2 0.255 0 0
99 16.745 1.197 471.653
100 16.915 1.22404 489.382
```

The first three lines starting with # provides some comments. The main body contains blocks of data for different time steps. Each block starts with the time step number and contains four columns: Bin index, Radial distance r, RDF value $g(r)$, and accumulated RDF values.

The **vacf.txt** file is arranged as follows:

```
# Time-correlated data for fix 3
# Timestep Number-of-time-windows
# Index TimeDelta Ncount c_myVACF[4]*c_myVACF[4]
0 2000
1 0 1 32.5261
2 10 0 0.0
3 20 0 0.0
4 30 0 0.0
...
20000 2000
1 0 2001 0.26325
2 10 2000 0.254712
3 20 1999 0.245126
...
1998 19970 0 0.0
1999 19980 0 0.0
2000 19990 0 0.0
20000 2000
1998 19970 4 0.191614
1999 19980 3 0.189268
2000 19990 2 0.185342
```

The first three lines starting with # provides some comments. Each block starts with the time step number and contains four columns: time step, dt, number of count and VACF values.

The following Python code demonstrates how to process and visualize the outputs from **rdf.txt** and **vacf.txt**:

```python
import numpy as np
import matplotlib.pyplot as plt
from scipy.fftpack import fft
import seaborn as sns
sns.set(font_scale=1.5)

def read_rdf(filename):
    with open(filename, "r") as file:
        lines = file.readlines()

    r_values = np.zeros(100)
    rdf = np.zeros([100, 100]) #first 100 RDF values
    count = -1
    for line in lines:
        if line.startswith("#") or line.strip() == "":
            continue  # Skip comments and empty lines

        parts = line.split()

        # Detect start of new time step block
        if len(parts) == 2: # 0 100
            count += 1
            if count >= 100:
                break
        else:
            id = int(parts[0]) - 1
            r = float(parts[1])  # Extract distance
            rdf_value = float(parts[2])  # Extract RDF value
            rdf[id, count] = rdf_value
            if count == 0:
                r_values[id] = r
                #print(rdf[id])

    return r_values, rdf

fig, axs = plt.subplots(3, 1, figsize=(12, 10))
r_values, rdf = read_rdf("rdf.txt")

axs[0].plot(r_values, rdf[:, 0], "-.", lw=0.5, label="0 ps")
axs[0].plot(r_values, rdf[:, 10], "--", lw=1.0, label="1 ps")
axs[0].plot(r_values, rdf[:, -1], "-", label="10 ps")
axs[0].set_xlabel("$r ~(\mathrm{\AA})$")
axs[0].set_ylabel("RDF")
axs[0].set_xlim([0, 12])
axs[0].legend()

vacf = np.loadtxt("vacf.txt", skiprows=2005)
axs[1].plot(vacf[:, 1]/1000, vacf[:, 3])
axs[1].set_xlabel("Time (ps)")
axs[1].set_ylabel("VACF")
#axs[1].set_xlim([0, 20])

N_steps = len(vacf)
dt = 10 * 1e-15                               # fs => s
freq = np.fft.fftfreq(N_steps, dt) / 1e12    # Hz => THz
```

```python
VDOS = np.abs(fft(vacf[:, 3]))**2

axs[2].plot(freq[:N_steps//2], VDOS[:N_steps//2])
axs[2].set_xlabel("Frequency (THz)")
axs[2].set_ylabel("VDOS")
axs[2].set_xlim([0, 2])
plt.tight_layout()
plt.savefig("lec_05_lammps.pdf")
```

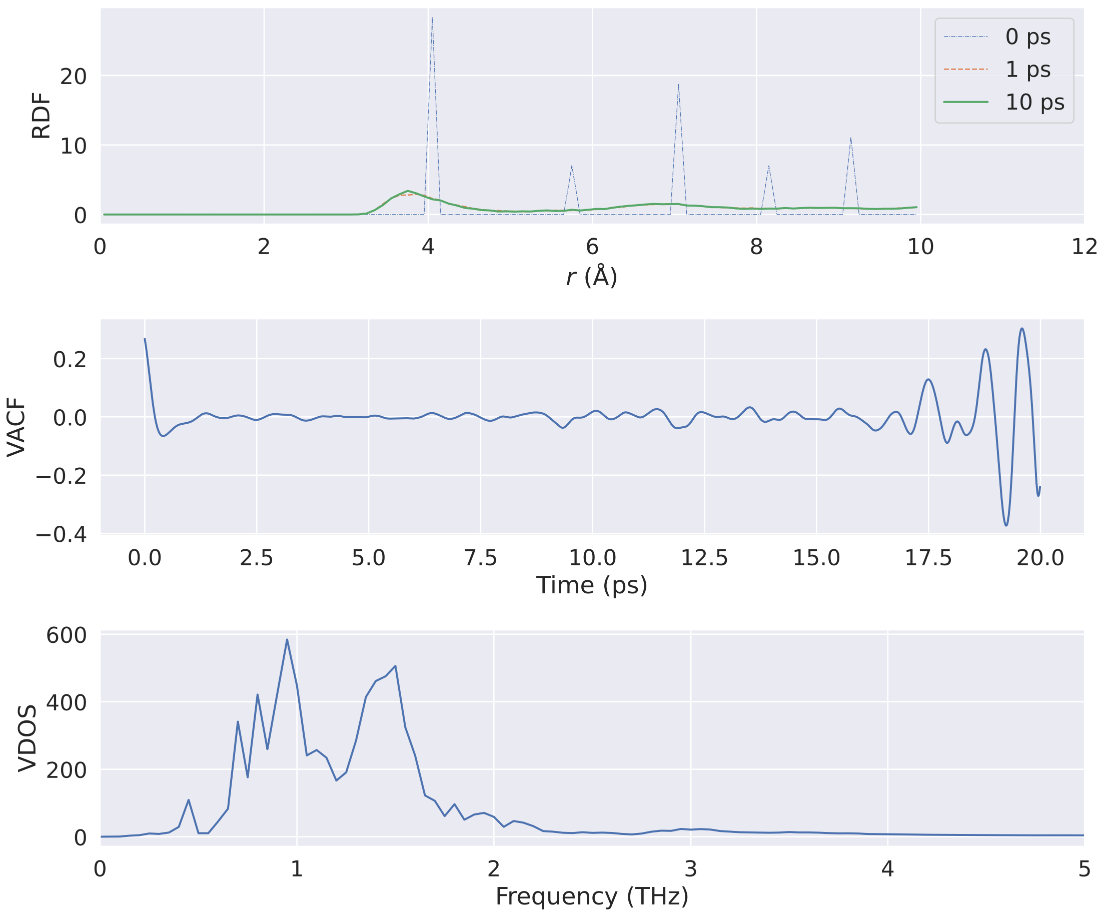


Figure 5.5: The simulated RDF, VACF and VDOS for liquid Argon at 94.4 K.

The plots are shown in Fig. 5.5. These results highlight several key features:

- **RDF**: Peaks in the RDF correspond to preferred atomic separations. At the beginning of the simulation, the RDF displays sharp peaks, characteristic of a solid with long-range crystalline order. However, after approximately 1 ps, these peaks broaden significantly, indicating the formation of a liquid phase. The RDFs at 1 ps and 10 ps are nearly identical, confirming the stabilization of the liquid phase.

- **VACF**: VACF captures temporal correlations in atomic velocities. Oscillatory behavior in VACF indicates vibrational motion in the system.

- **VDOS**: Peaks in the VDOS correspond to specific vibrational modes within the system. Low-frequency modes often represent collective atomic motions, while high-frequency modes correspond to localized vibrations. Analyzing the VDOS provides a direct link to the vibrational dynamics observed in the MD trajectory.

By following the steps outlined above, users can compute RDF and VDOS for any system simulated in LAMMPS, enabling in-depth analysis of structural and dynamic properties. These methods form the foundation for investigating phase transitions, material stability, and vibrational characteristics in complex systems.

5.5. Summary

In this chapter, we discussed methods to analyze the structural behavior from a MD simulation both qualitatively and quantitatively.

First, visualization plays a crucial role in gaining an intuitive understanding of atomic arrangements and dynamic transitions in a system. Using tools like OVITO and customizing visualization settings, researchers can highlight key structural features such as defects, interfaces, and vibrational modes.

Next, we explored the use of quantitative metrics to characterize structural and dynamic properties more rigorously. The RDF provides insights into the spatial distribution of neighboring atoms, helping to distinguish between liquid, solid, and amorphous phases. The VDOS reveals the distribution of vibrational frequencies in the system, offering a link between atomic motions and experimental spectroscopic data.

Together, these methods form a robust framework for analyzing MD simulations, enabling researchers to extract meaningful information about the structural and dynamic behaviors of materials. In the following chapters, we will expand upon these techniques to address more complex systems and phenomena.

6. Transport Processes

Transport properties describe how particles, energy, and momentum move within a system. These properties are critical for understanding materials and systems across various scales, from atomic to macroscopic. In this chapter, we will focus on two key transport properties: diffusion (particle transport) and thermal conductivity (heat transport).

One might intuitively consider simulating such processes directly by observing the system over time to measure particle displacements or energy transfer. However, MD simulations, especially those based on classical potentials, are inherently limited by computational constraints. Even on supercomputers, MD simulations typically span nano- to micro-second timescales. For many real-world transport processes, such durations are insufficient to capture the full dynamics, especially in systems with slow relaxation processes. Therefore, a direct simulation approach may not yield reliable or converged transport properties.

To overcome these limitations, we will take the advantage of an important subject in statistical mechanics, the linear response theory, to design indirect but efficient methods for calculating transport coefficients. This approach provides a robust framework to analyze particle and energy transport in equilibrium systems, significantly reducing the computational cost while maintaining accuracy.

6.1. Diffusion

Let us start with the diffusion problem. According to Fick's 2nd law, the diffusion equation is a partial differential equation that describes how a substance spreads over time. For a one-dimensional system under equilibrium, it is given by:

$$\frac{\partial C(x,t)}{\partial t} = D\frac{\partial^2 C(x,t)}{\partial x^2} \tag{6.1}$$

where:

- $C(x,t)$ is the concentration of particles at position x and time t.
- D is the diffusion constant which characterizes how fast particles diffuse.

The equation states that the rate of change of concentration over time, $\partial C/\partial t$, is proportional to the second spatial derivative of the concentration, $\partial^2 C/\partial x^2$.

The solution to the diffusion equation gives the probability distribution for the particle's position over time. In the case of one-dimensional diffusion starting from a point, the solution is a Gaussian distribution:

$$C(x,t) = \frac{1}{\sqrt{4\pi Dt}} \exp\left(-\frac{x^2}{4Dt}\right) \tag{6.2}$$

The Gaussian distribution suggests that:

1. The peak is at the origin ($x = 0$), assuming particles start there.

2. The spread increases with time, meaning that particles are more likely to be found farther from the origin over time.

To find the mean squared displacement (MSD), we calculate the expected value of x^2 with respect to this distribution. The MSD in one dimension is:

$$\langle x^2(t)\rangle = \int_{-\infty}^{\infty} x^2 C(x,t)\,dx = 2Dt$$

If we change x from 1-dimension to d-dimension, the general form is

$$\langle ||\Delta\mathbf{x}||^2(t)\rangle = 2dDt$$

In derivative form:

$$\frac{\partial\langle ||\Delta\mathbf{x}||^2(t)\rangle}{\partial t} = 2dD \tag{6.3}$$

In MD simulations, we compute the MSD as the time-averaged square of particle displacements from initial positions. Linear regression of the MSD(t) curve gives the diffusion constant.

```python
def compute_msd(R):
    """
    Compute the mean square displacement (MSD) over time.
    R: array of shape (num_timesteps, num_atoms, 3)
    """
    num_timesteps = positions.shape[0]
    num_atoms = positions.shape[1]

    msd = np.zeros(num_timesteps)
    for t in range(1, num_timesteps):
        displacements = R[t] - R[0]
        squared_displacements = np.sum(displacements**2, axis=1)
        msd[t] = np.mean(squared_displacements)

    return msd

# Example usage
positions = np.random.randn(1000, 100, 3)
msd = compute_msd(positions)
time = np.linspace(0, 100, len(msd))

# Fit MSD to time to calculate diffusion constant
D = np.polyfit(time, msd, 1)[0] / 6
plt.plot(time, msd, label="MSD")
plt.xlabel("Time (ps)")
plt.ylabel("MSD (\AA^2)")
plt.title(f"Diffusion Constant: {D:.3e} cm$^2$/s")
plt.show()
```

6.2. The Green-Kubo Relation

6.2.1 Alternative Expression of MSD

The MSD can also be expressed through the velocity of the particle:

$$\Delta x(t) = \int_0^t dt' v_x(t')$$

Thus, the MSD becomes:

$$\begin{aligned}
\langle \Delta x^2(t) \rangle &= \left\langle \left(\int_0^t dt' v_x(t') \right)^2 \right\rangle \\
&= \int_0^t dt' \int_0^t dt'' \langle v_x(t') v_x(t'') \rangle \\
&= 2 \int_0^t dt' \int_0^{t'} dt'' \langle v_x(t') v_x(t'') \rangle
\end{aligned}$$

This formulation allows D to be computed using the VACF:

$$D = \frac{\partial \langle \Delta r^2(t) \rangle}{2d\partial t} = \frac{1}{d} \int_0^\infty \langle \mathbf{v}(0) \cdot \mathbf{v}(t) \rangle dt \tag{6.4}$$

6.2.2 The General Green-Kubo Relation

In fact, such a relation, between diffusion and the velocity can be understood via a physical picture. If the atoms are able to the diffuse quickly, they should quickly forget about the initial velocity. On the other hand, it is the flux of velocity that triggers the diffusion. Such a dual relation may be found in many other physical processes.

The Green-Kubo relation generalizes this idea. According to linear response theory, transport coefficients λ are related to time correlations of microscopic quantities under thermal equilibrium:

$$\lambda = \int_0^\infty \langle J(0) \cdot J(t) \rangle dt \tag{6.5}$$

where $J(t)$ is the current of the relevant quantity (e.g., velocity, heat current, or stress).

One can think of the following scenarios to apply the Green-Kubo Relation.

- **Diffusion constant vs. velocity**: The flux of velocity causes diffusion. If a particle quickly forgets its initial velocity, it leads to faster diffusion as the particle loses memory of its initial velocity.

- **Thermal conductivity vs. heat current**: Applying a temperature gradient causes a heat current to flow, leading to thermal conductivity. The faster the decay of heat current autocorrelation, the greater the thermal conductivity.

- **Viscosity vs. stress tensors**: In a fluid, applying a shear stress leads to a flow, related to viscosity. A high viscosity fluid (like honey) will have a slower decay of the stress autocorrelation function, meaning the fluid remembers shear stresses for a longer time compared to a low viscosity fluid (like water). These relations are derived from linear response theory, which states that the response of a system to a small perturbation is proportional to the equilibrium fluctuations of microscopic quantities (e.g., heat current, velocity, or stress).

For a detailed mathematical proof, we refer to Frenkel and Smidt [10]. In the next section, we will focus on the application of this relation to the calculation of thermal conductivity.

6.2.3 Thermal Conductivity

Thermal conductivity, κ, measures a material's ability to conduct heat. Using the Green-Kubo relation, κ is calculated from the heat current autocorrelation function (HCACF):

$$\kappa = \frac{1}{k_B T^2 V} \int_0^\infty \langle J(0) \cdot J(t) \rangle dt \tag{6.6}$$

where:

- $J(t)$ is the heat current.
- k_B is the Boltzmann constant.
- T is temperature.
- V is the system volume.

The heat current $\mathbf{J}$ is derived from the 1st law of thermodynamics and accounts for both energy and momentum flow within the system. It can be expressed as:

$$\begin{aligned}
\mathbf{J} &= \frac{1}{V}\left[\sum_i e_i \mathbf{v}_i - \sum_i \mathbf{S}_i \mathbf{v}_i\right] \\
&= \frac{1}{V}\left[\sum_i e_i \mathbf{v}_i + \sum_{i<j} (\mathbf{F}_{ij} \cdot \mathbf{v}_j) \mathbf{r}_{ij}\right] \\
&= \frac{1}{V}\left[\sum_i e_i \mathbf{v}_i + \frac{1}{2}\sum_{i<j} \left(\mathbf{F}_{ij} \cdot (\mathbf{v}_i + \mathbf{v}_j)\right) \mathbf{r}_{ij}\right]
\end{aligned}$$

where:

- e_i is the per-atom energy (potential and kinetic).
- S_i is the per-atom stress tensor.
- $\mathbf{v}_i$ is the velocity of atom i.

This formulation includes:

- The transport of energy by individual atoms ($e_i\mathbf{v}_i$ term).
- The energy flux caused by interactions between atoms ($\mathbf{F}_{ij} \cdot \mathbf{v}$ terms).

The HCACF is then evaluated as:

$$\langle J(t) \cdot J(0)\rangle_{\text{eq}} = \frac{1}{N}\sum_{i=1}^{N}\langle J_i(t) \cdot J_i(0)\rangle \tag{6.7}$$

6.2.4 Transport Properties from Equilibrium MD

At this point, it's important to note that the techniques to compute diffusion coefficients and thermal conductivity are indirect. They rely on the statistical properties of the system and are rooted in the linear response theory. This principle connects the equilibrium fluctuations in a system to its response to perturbations, stating that a system's response to a small perturbation is proportional to its equilibrium fluctuations.

These techniques are useful in MD as they allow us to derive important transport properties from fluctuations within the system, bypassing the need for external perturbations.

With the increasing computational power, it is possible to employ **nonequilibrium MD** (NEMD) simulations calculate transport properties directly by applying external perturbations, such as temperature or concentration gradients. However, the practical limitations of NEMD include the need for extended simulation times to reach a steady state and the introduction of boundary effects, which can complicate the analysis.

6.3. LAMMPS Calculation and Analysis

In this section, we demonstrate how to compute the MSD and thermal conductivity of liquid argon using LAMMPS. The provided script sets up the necessary calculations and outputs results for further analysis. Below is the detailed breakdown of the script and its commands.

6.3.1 LAMMPS Setup

After invoking the LAMMPS environment in Colab, one can first run the following block.

```
lammps_script = """
# Initial setup of the simulation
units            metal                   # Use Metal units
dimension        3                       # 3D simulation
boundary         p p p                   # PBC conditions
atom_style       atomic                  # Atomic style

```

```
# Declaring necessary parameters
variable T        equal 94.4            # Temperature K
variable epsilon  equal 120*8.61733e-5  # eV
variable sigma    equal 3.4             # Angstrom
variable L        equal 10.229*${sigma} # Box length
variable a        equal $L/6            # Lattice parameter
variable a        equal $L**3           # Lattice parameter

# Generating supercell
lattice         fcc $a
region          box block 0 1 0 1 0 1 units lattice
create_box      1 box
create_atoms    1 box
replicate       6 6 6

# Setting up Simulation
mass            1 39.95
pair_style      lj/cut 13.0
pair_coeff      1 1 ${epsilon} ${sigma}
timestep        1.0
neighbor        0.3 bin
thermo          1000
thermo_style    custom step temp epair etotal ke pe press vol
velocity        all create $T 12345
fix             1 all nvt temp $T $T 10 drag 0.2

# MSD calculation
compute MSD all msd com yes
fix 2 all ave/time 100 1 100 c_MSD[*] file msd.txt mode vector
run 10000
unfix 2

# kappa calculation
reset_timestep 0
thermo       2000
variable     kB equal 1.3806504e-23    # [J/K] Boltzmann
variable     kCal2J equal 4186.0/6.02214e23
variable     A2m equal 1.0e-10
variable     fs2s equal 1.0e-15
variable     convert equal ${kCal2J}*${kCal2J}/${fs2s}/${A2m}
compute      myKE all ke/atom
compute      myPE all pe/atom
compute      myStress all stress/atom NULL virial
compute      flux all heat/flux myKE myPE myStress
variable     Jx equal c_flux[1]/vol
variable     Jy equal c_flux[2]/vol
variable     Jz equal c_flux[3]/vol
fix          JJ all ave/correlate 10 200 2000 &
             c_flux[1] c_flux[2] c_flux[3] type auto file J0Jt.dat ave
   running
variable     scale equal ${convert}/${kB}/$T/$T/$V*${dt}*10
variable     k11 equal trap(f_JJ[3])*${scale}
variable     k22 equal trap(f_JJ[4])*${scale}
variable     k33 equal trap(f_JJ[5])*${scale}
thermo_style custom step temp v_Jx v_Jy v_Jz v_k11 v_k22 v_k33
run          100000
variable     k equal (v_k11+v_k22+v_k33)/3.0
variable     ndens equal count(all)/vol
```

```
print        "average conductivity: $k[W/mK] @ $T K, ${ndens} /A\^3"
"""

# Save the script to a file
with open("lmp_npt_msd_kappa.in", "w") as file:
    file.write(lammps_script)

!lmp -in lmp_npt_msd_kappa.in
```

In this script, we called two commands to compute MSD and thermal conductivity as follows.

```
compute MSD all msd com yes
fix 2 all ave/time 100 1 100 c_MSD[*] file msd.txt mode vector
run 10000
```

- **compute MSD all msd com yes**: Computes the mean square displacement (MSD) for all atoms relative to the center of mass.
- **fix 2 all ave/time 100 1 100 c_MSD[*] file msd.txt mode vector**: Averages the MSD over 100 timesteps and saves the results in msd.txt.
- **run 10000**: Executes the simulation for 10,000 timesteps, generating 100 blocks of MSD data.

```
variable     kB equal 1.3806504e-23    # [J/K] Boltzmann
variable     kCal2J equal 4186.0/6.02214e23
variable     A2m equal 1.0e-10
variable     fs2s equal 1.0e-15
variable     convert equal ${kCal2J}*${kCal2J}/${fs2s}/${A2m}
compute      myKE all ke/atom
compute      myPE all pe/atom
compute      myStress all stress/atom NULL virial
compute      flux all heat/flux myKE myPE myStress
variable     Jx equal c_flux[1]/vol
variable     Jy equal c_flux[2]/vol
variable     Jz equal c_flux[3]/vol
fix          JJ all ave/correlate 10 200 2000 &
             c_flux[1] c_flux[2] c_flux[3] type &
             auto file J0Jt.dat ave running
variable     scale equal ${convert}/${kB}/$T/$T/$V*$s*${dt}
variable     k11 equal trap(f_JJ[3])*${scale}
variable     k22 equal trap(f_JJ[4])*${scale}
variable     k33 equal trap(f_JJ[5])*${scale}
thermo_style custom step temp v_Jx v_Jy v_Jz v_k11 v_k22 v_k33
run          100000
variable     k equal (v_k11+v_k22+v_k33)/3.0
variable     ndens equal count(all)/vol
print        "average conductivity: $k[W/mK] @ $T K, ${ndens} /A\^3"
```

The above block demonstrates how to calculate the thermal conductivity of a material using the Green-Kubo relation. It uses **compute flux all heat/flux myKE myPE myStress** to compute the total heat flux using the kinetic, potential, and stress contributions. The variables Jx, Jy, and Jz represent the heat flux components in the x, y and z directions, normalized by the system volume. Finally, it computes the κ according to eq. 6.6.

6.3.2 Results Analysis

The thermal conductivity is directly extracted from the LAMMPS log file at the end of the simulation, as shown below:

```
average conductivity:
0.145404507450377 [W/mK] @ 94.4 K,
0.0205389130107768 /A^3
```

While the output provides the averaged thermal conductivity value, it is possible to extract the directional dependence of thermal conductivity by modifying the LAMMPS input script. Directional thermal conductivity analysis is especially critical for solid materials, as they often exhibit anisotropic thermal transport properties due to their crystal structure or defects.

To compute the directional components (e.g., κ_{xx}, κ_{yy}, κ_{zz}), the script should calculate and store the contributions of the heat flux components (J_x, J_y, J_z) to the thermal conductivity. These can be achieved by appropriately modifying the computation and fix commands for the heat flux autocorrelation function. This enhancement can provide deeper insights into the anisotropic thermal behavior of solid systems.

For the MSD and diffusion coefficients, the **msd.txt** file outputs time-averaged data. Every 100 steps (corresponding to 0.1 ps), the MSD values are computed for each atom in the x, y, and z components as well as the total summed displacements. A typical snippet of the **msd.txt** file is as follows:

```
# Time-averaged data for fix 2
# TimeStep Number-of-rows
# Row c_MSD
0 4
1 0
2 0
3 0
4 0
100 4
1 0.0187545
2 0.0190381
3 0.0190832
4 0.0568759
```

Therefore, we use the following code to extract the total msd value for every 0.1 ps, and then perform linear regression to obtain the coefficients according to eq.6.3.

```python
import numpy as np
import matplotlib.pyplot as plt
import seaborn as sns
sns.set(font_scale=1.5)

msd = np.loadtxt("msd.txt", skiprows=3)[4::5, 1]
time = np.arange(len(msd))*0.1

# Compute diffusion coefficient
coeffs = np.polyfit(time[:10], msd[:10], 1)  # Linear fit (degree=1)
slope = coeffs[0]
intercept = coeffs[1]
D = slope / 6  # Diffusion coefficient

# Plot the results
plt.figure(figsize=(8, 4))
plt.plot(time, msd, label="Raw MSD")
plt.plot(time[:10], slope * time[:10] + intercept,
         "r--", label=f"Linear Fit D={D:.3e} cm$^2$/s")
plt.xlabel("Time (ps)")
plt.ylabel("MSD ($\mathrm{\AA^2}$)")
plt.legend()
plt.tight_layout()
plt.savefig("lec06-msd.pdf")
```

The results are shown in Fig. 6.1. A clear linear increasing trend for the MSD values is observed up to 1 ps. Beyond this point, the MSD values begin to fluctuate. This behavior can be attributed to argon at 94.4 K retaining strong solid-like characteristics, where atoms primarily vibrate around their equilibrium positions, resulting in an MSD that oscillates or saturates rather than increasing indefinitely. Consequently, only the first 1 ps of data is used for linear fitting to extract the diffusion coefficient.

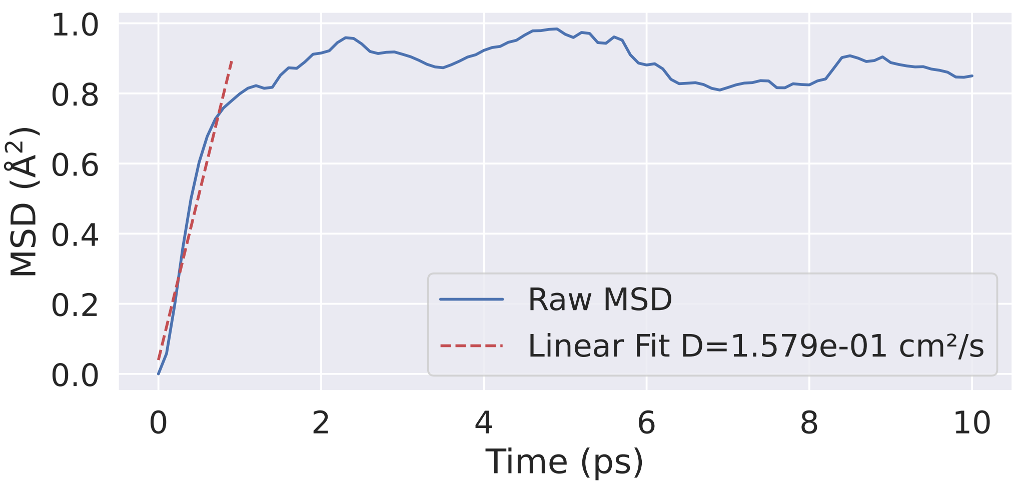


Figure 6.1: The simulated MSD and diffusion coefficient for liquid argon at 94.4 K.

Fig. 6.2 further displays the simulated MSD values at different temperatures. As expected, higher temperatures significantly enhance diffusion, demonstrating the temperature dependence of the diffusion process.

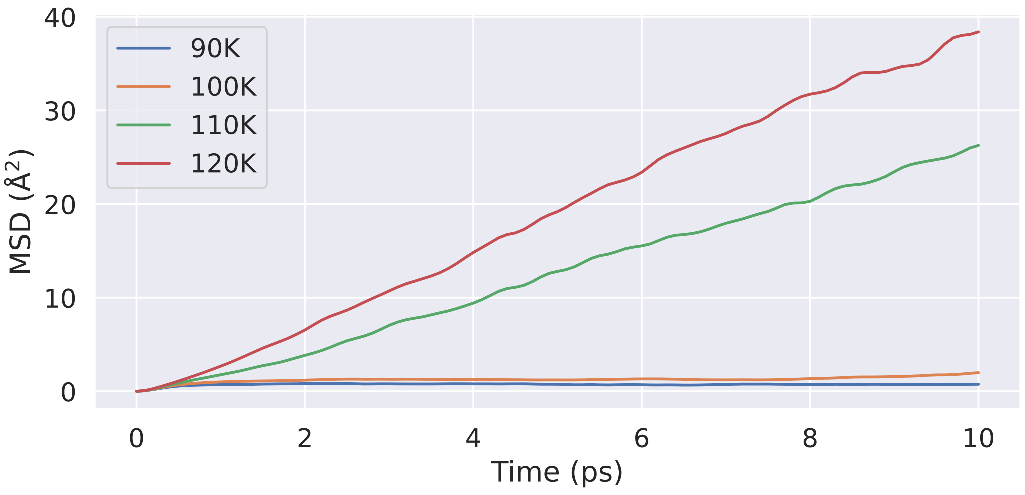


Figure 6.2: The simulated MSD for liquid argon at different temperatures.

6.4. Summary

In this chapter, we explored the computation of transport properties such as diffusion coefficients and thermal conductivity through MD simulations. By leveraging the principles of statistical mechanics, particularly the linear response theory, we demonstrated how equilibrium fluctuations can be utilized to derive transport properties without relying on external perturbations. These relations establish a universal framework for connecting equilibrium fluctuations to macroscopic transport properties, allowing for efficient and accurate computations.

Finally, while the techniques discussed are grounded in equilibrium MD simulations, they can be extended and compared with results from non-equilibrium MD simulations for validation and further insights. This chapter serves as a foundation for understanding and applying transport property computations in diverse systems.

7. Enhanced Sampling with Metadynamics

In a standard MD simulation, the system evolves under Newtonian dynamics, meaning that it can get stuck in local minima for long periods. This limits the exploration of the free energy surface, making it difficult to observe transitions between different states. To address this challenge, one may consider the use of enhanced sampling techniques. In this lecture, we will focus on a particular type of technique called Metadynamics.

7.1. What is Metadynamics?

Metadynamics is an MD-based sampling technique to explore free energy landscapes and overcome energy barriers. It is particularly useful for systems where traditional MD struggles to sample rare events due to high energy barriers or complex transitions. Metadynamics not only accelerates the sampling of rare events but also provides a mechanism for reconstructing the free energy surface. This is achieved by systematically filling the energy wells with biasing potentials, eventually allowing the system to overcome energy barriers and explore alternative configurations.

Compared to traditional MD simulation, Metadynamics adds the following concepts:

1. **Biasing Potential**: The fundamental idea is to add a bias potential to the system that changes over time. By adding Gaussian-shaped potentials periodically to the explored regions, the system is pushed out of energy wells, allowing for better exploration. The accumulation of these Gaussians eventually results in a bias that compensates for the underlying free energy, enabling the system to explore new configurations freely.

2. **Collective Variables (CVs)**: Metadynamics works by adding the bias in the space of carefully chosen collective variables (CVs). CVs are reduced-dimensional representations of the system, such as distances between atoms, angles, or other descriptors capturing essential behavior. The choice of CVs is crucial, as it directly influences the efficiency and success of the metadynamics simulation. Properly chosen CVs can significantly accelerate the exploration of relevant conformational space, while poorly chosen CVs may lead to incomplete sampling.

3. **Free Energy Estimation**: As the bias potential accumulates, it fills the wells of the free energy landscape. The accumulated bias potential can be used to reconstruct the underlying free energy surface, providing valuable insights into the thermodynamic properties of the system. This reconstruction allows researchers to

identify stable states, transition states, and the pathways connecting them, which is critical for understanding the behavior of molecular systems.

7.2. MD and Metadynamics in an 1D Potential Well

To demonstrate the concept of Metadynamics, let's consider a simple model system with a particle moving in a double-well potential as follows:

$$V(x) = x^4 - 3x^2$$

```python
import numpy as np
import seaborn as sns
import matplotlib.pyplot as plt
sns.set(font_scale=1.2)

# Define the double well potential
def double_well_potential(x):
    return x**4 - 3*x**2

# Derivative of the potential (force)
def potential_force(x):
    return -4*x**3 + 6*x

# Time evolution of the particle using Langevin dynamics
def md(steps=20000, dt=1e-2, gamma=0.02, temp=0.01):
    x = 0.5  # initial position
    x_positions = [x]

    for step in range(steps):
        # Langevin dynamics with force from the double well potential
        force = potential_force(x)
        thermal_force = np.sqrt(2 * gamma * temp / dt) * np.random.
    normal()

        # Update position with Langevin equation
        x += force * dt - gamma * x * dt + thermal_force * np.sqrt(dt)
        x_positions.append(x)

    return np.array(x_positions)

# Run the simulation
xs = md()

# Create the figure with two subplots with shared x-axis
fig, (ax1, ax2) = plt.subplots(2, 1, figsize=(8, 6), gridspec_kw={"
    height_ratios": [2, 1]}, sharex=True)

# Double well potential with Langevin dynamics
x_vals = np.linspace(-2, 2, 100)
potential_vals = double_well_potential(x_vals)
ax1.plot(x_vals, potential_vals, "--", lw=1.0, label="Double Well
    Potential", color="k")
sc = ax1.scatter(xs, double_well_potential(xs), c=range(len(xs)), s=2,
    cmap="viridis", alpha=0.5)
ax1.set_ylabel("Potential Energy")
ax1.legend()
```

```python
ax1.set_title("Double Well Potential with Langevin Dynamics",
    fontweight="bold")
ax1.grid(True)
cbar = plt.colorbar(sc, ax=ax1, orientation="horizontal", pad=0.1)
cbar.set_label("Time Step")

# Histogram of x values
ax2.hist(xs, bins=50, color="skyblue", edgecolor="black")
ax2.set_xlabel("Position (x)")
ax2.set_ylabel("Frequency")
ax2.set_title("Histogram of Particle Positions")
ax2.grid(True)

plt.tight_layout()
plt.savefig("1D-MD.png")
```

The particle would just oscillate around the energy well in this simulation, as illustrated below:

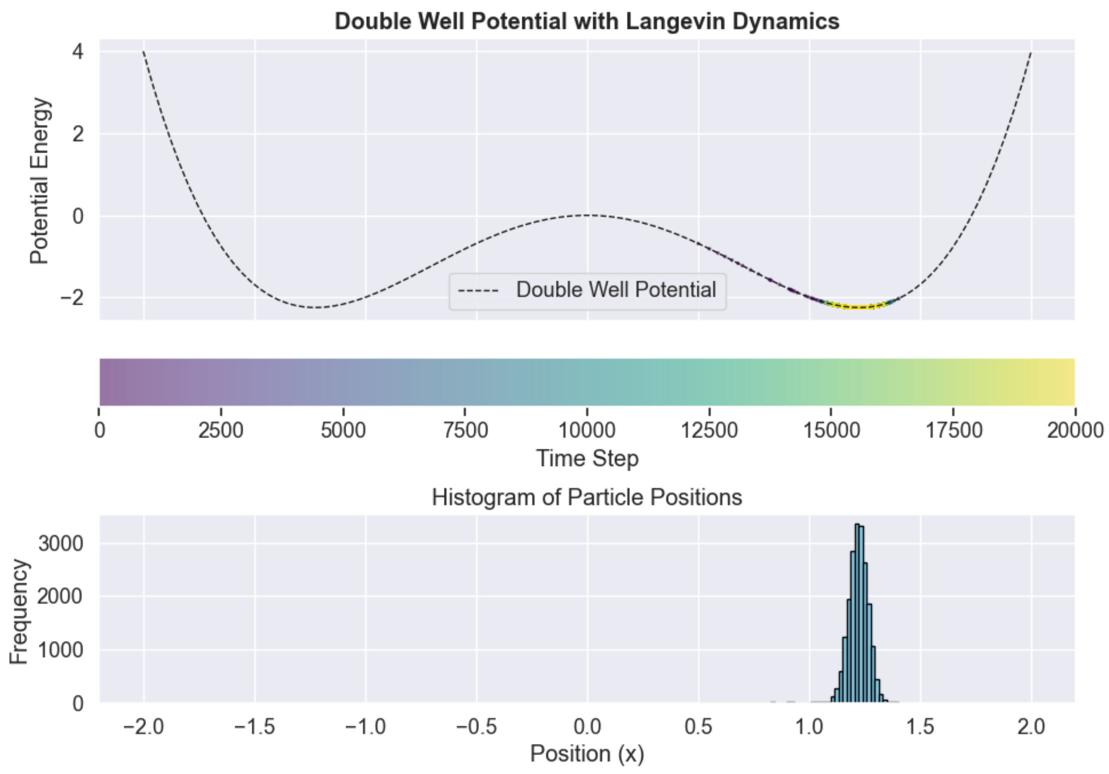


Figure 7.1: MD simulation in a double well potential.

If one is interested in sampling more of the phase space, there must be a way to escape from the local minima. Metadynamics adds a bias to the system to prevent the particle from getting stuck in one well, allowing it to explore other regions of the potential landscape.

$$V_{\text{bias}}(s,t) = V_{\text{system}}(s) + \sum_{t' \leq t} W \exp\left(-\frac{(s - s(t'))^2}{2\sigma^2}\right)$$

where:

- W is the height of the Gaussian, which controls the magnitude of the bias added at each time step,
- σ is the width of the Gaussian, determining how localized the bias is in the CV space,
- $s(t)$ represents the value of the CVs at time t.

The bias potential $V_{\text{bias}}(s, t)$ accumulates Gaussians placed at the positions the system has visited in the CV space, thus gradually filling the wells in the free energy landscape and pushing the system to explore other areas.

The following code demonstrates the implementation of the bias potential in the context of metadynamics.

```python
import numpy as np
import seaborn as sns
import matplotlib.pyplot as plt
sns.set(font_scale=1.2)

# Derivative of the potential (force)
def potential_force(x):
    return -4*x**3 + 6*x

def gaussian_bias(x, centers, width=0.1, height=0.1):
    bias = 0
    for c in centers:
        bias += height * np.exp(-0.5 * (x - c)**2 / width**2)
    return bias

# Derivative of the Gaussian bias (force due to bias)
def bias_force(x, centers, width=0.1, height=0.1):
    force = 0
    for c in centers:
        force += height * (x - c) / (width**2) * np.exp(-0.5 * (x - c)
    **2 / width**2)
    return force

# Time evolution of the particle using Langevin dynamics
def metaD(steps=20000, dt=1e-2, gamma=0.02, temp=0.01):
    x = 0.5  # initial position
    x_positions = [x]
    centers = []  # store the positions where bias is added

    for step in range(steps):
        # Langevin dynamics with force from the double well potential
        force = potential_force(x)
        thermal_force = np.sqrt(2 * gamma * temp / dt) * np.random.
    normal()

        # Apply bias from metadynamics
        force += bias_force(x, centers)

        # Update position with Langevin equation
        x += force * dt - gamma * x * dt + thermal_force * np.sqrt(dt)
        x_positions.append(x)

        # Add Gaussian bias every 100 steps
        if step % 100 == 0:
            centers.append(x)

    return np.array(x_positions)

# Run the simulation
xs = metaD()
```

The full exploration of the potential space is visualized in the following figure:

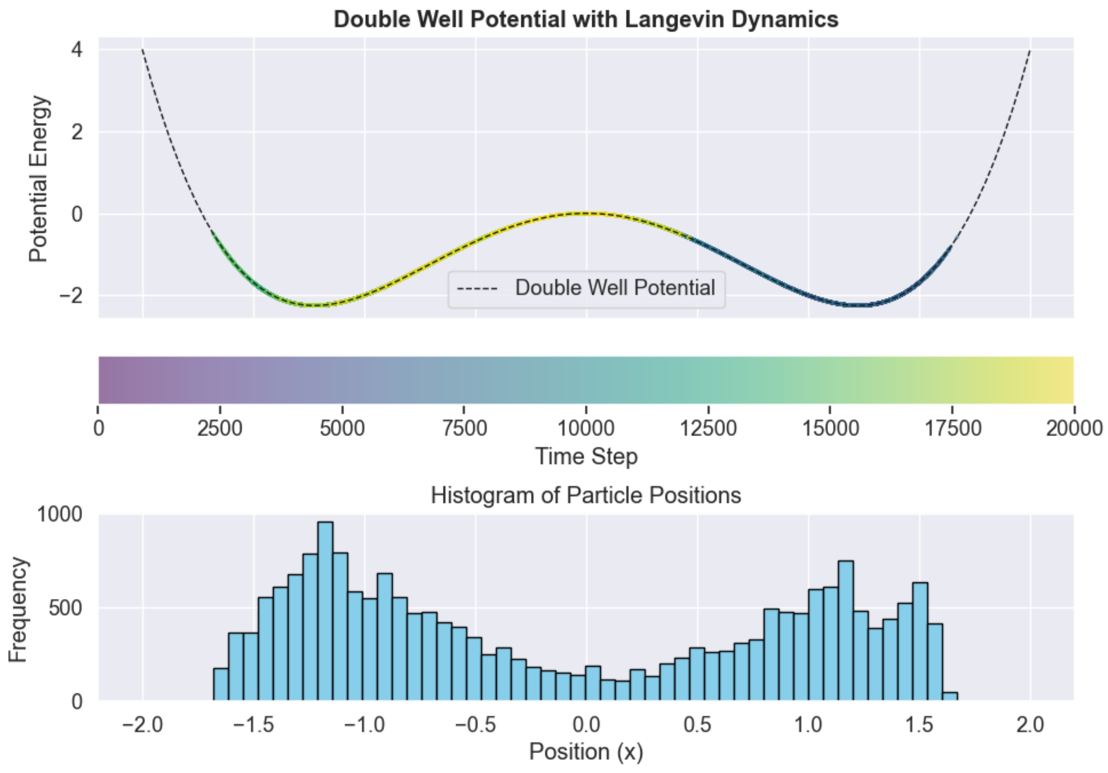


Figure 7.2: Metadynamics simulation in a double well potential.

7.3. How does Metadynamics Work?

Metadynamics simulations can be applied to higher-dimensional systems and more complex potentials, such as chemical reactions, protein folding, and phase transitions. It typically involves the following steps:

1. **Initialization**: Select appropriate CVs that describe the transition of interest. The choice of CVs is critical as they determine how effectively metadynamics can enhance sampling.

2. **Bias Addition**: During the simulation, small Gaussian potentials are periodically added along the CVs to the current position of the system. This gradually discourages the system from revisiting already visited states. The height and width of these Gaussians are important parameters that control the rate of exploration; a careful balance must be struck to ensure efficient sampling without overshooting important regions of the free energy landscape.

3. **Exploration of the Free Energy Surface**: By continuously adding Gaussians, the system is encouraged to move out of energy minima, eventually allowing the exploration of the entire relevant free energy landscape. The bias potential effectively smooths out the energy barriers, enabling the system to transition between states more easily. Over time, the system visits all accessible regions of the free energy surface, and the accumulated bias provides an estimate of the free energy differences between states.

7.4. Choice of CVs

The efficiency of metadynamics heavily depends on the proper choice of CVs. Poor selection can lead to ineffective sampling, as the system may not be driven along the most relevant pathways. The process of selecting suitable CVs often requires trial and error or prior knowledge of the system.

7.5. Well-Tempered Metadynamics

In traditional metadynamics, the constant addition of Gaussian potentials can lead to excessive bias accumulation, which may result in poor sampling or an inaccurate free energy landscape. Well-tempered metadynamics addresses this by gradually reducing the height of the Gaussians added over time, which helps prevent oversampling and ensures that the system does not accumulate too much bias in any particular region. As such, it is expected to improve convergence and enhance the accuracy of the free energy surface estimation.

The key idea behind well-tempered metadynamics is to scale the bias deposition rate according to the amount of bias already present. This scaling is achieved by introducing a parameter called the bias factor γ, which controls how much the bias potential decreases as the simulation progresses. The bias factor is related to a fictitious temperature that effectively dictates how smoothly the bias is added.

The bias potential in well-tempered metadynamics evolves as:

$$V_{\text{bias}}(s,t) = \sum_{t\prime \leq t} W \exp\left(-\frac{V_{\text{bias}}(s,t')}{k_B \Delta T}\right) \exp\left(-\frac{(s(t)-s(t'))^2}{2\sigma^2}\right)$$

Compared to the previous equation, an additional exponential term was applied to scale the Gaussian potential, where

- k_B is the Boltzmann constant. - ΔT is the fictitious temperature, defined as $\Delta T = T_{\text{system}}(\gamma - 1)$, where T_{system} is the real temperature of the system.

In standard metadynamics, the height of the Gaussian is fixed at W. In well-tempered metadynamics, the height becomes $W \exp\left[-V_{\text{bias}}(s,t')/k_B \Delta T\right]$. As V_{bias} increases over time, $\exp\left[-V_{\text{bias}}(s,t')/k_B \Delta T\right]$ gradually decays, preventing excessive bias accumulation in any particular region.

- When $\Delta T \to 0$, $\exp\left[-V_{\text{bias}}(s,t')/k_B \Delta T\right] \to 0$, which means zero Gaussian height, reverting the simulation to standard MD with zero bias.

- When $\Delta T \to \infty$, $\exp\left[-V_{\text{bias}}(s,t')/k_B \Delta T\right] \to 1$, resulting in a constant Gaussian height W, returning to standard metadynamics.

- By choosing a suitable ΔT between 0 and infinity, low V regions will be visited more frequently, allowing better exploration without excessive bias accumulation.

In essence, well-tempered metadynamics allows the atomic coordinates $\mathbf{R}$ to fluctuate around the system temperature T, while the collective variables $\mathbf{s}$ fluctuate around an elevated temperature $T+\Delta T$, promoting barrier-crossing without distorting the partition function.

In applications where free energy differences between states are of interest, one can infer $F(s)$ by counting the histogram via $F(s) = -T \ln N(s,t)$ in a well-tempered metadynamics simulation. The method provides distinct $F(s)$ values, which can yield insights into the free energy landscape.

7.6. Advantages of Well-Tempered Metadynamics

Well-tempered metadynamics has several advantages over traditional metadynamics:

1. By reducing the rate of bias deposition over time, it ensures that the system can focus on the most relevant regions of the free energy surface, leading to a more accurate reconstruction, which is crucial for systems with multiple metastable states or complex free energy landscapes.

2. It provides a natural mechanism for achieving convergence of the free energy surface. As the system becomes more thoroughly explored, the bias added to the system decreases, ultimately reaching a point where it no longer significantly alters the free energy landscape. This gradual reduction in bias allows the system to settle into the correct free energy minima, providing a reliable estimate of the underlying free energy differences between states.

Well-tempered metadynamics can be applied to a wide range of systems, from simple model potentials to complex biomolecular and materials science applications. Its ability to adaptively control the bias potential makes it a versatile tool for studying processes such as protein folding, chemical reactions, and phase transitions. By providing a more controlled and convergent approach to free energy estimation, well-tempered metadynamics has become a preferred method for enhanced sampling in many challenging molecular simulations.

7.7. Further Discussions

Metadynamics is a powerful enhanced sampling technique to overcome the limitations of traditional MD simulation. Its ability to reconstruct free energy surfaces makes it an invaluable tool for studying complex molecular systems, phase transitions, and reaction mechanisms. However, its success depends on careful selection of collective variables and parameters. By systematically adding bias potentials, metadynamics allows for the study of rare events and provides detailed insights into the thermodynamics and kinetics of molecular systems.

- Discuss how the choice of Gaussian parameters can impact ordinary metadynamics simulations.

- Discuss the impact of γ on well-tempered metadynamics simulations.
- Explore the choice of CVs in different kinds of simulations.

8. Introduction to Density Functional Theory

In the previous chapters, we have primarily focused on the study of MD based approaches and their applications to materials modeling. Having studied atomic-scale motion using MD, we now pivot to the quantum mechanical foundation of these interactions—electronic structure calculations. To simulate materials and molecules more comprehensively, we need to understand the electronic properties that underpin their behavior. The transition from classical molecular dynamics to quantum mechanical electronic structure calculations introduces a new set of techniques and concepts critical for simulating materials at an even deeper level.

8.1. Schrödinger Equation

In the 1920s and 1930s, Schrödinger proposed the famous equation to calculate the wavefunction Ψ of a quantum system, encapsulating all its physical properties. The time-independent Schrödinger equation for a system of N interacting electrons is:

$$\hat{H}\Psi(\mathbf{r}_1, \mathbf{r}_2, \ldots, \mathbf{r}_N) = E\Psi(\mathbf{r}_1, \mathbf{r}_2, \ldots, \mathbf{r}_N) \tag{8.1}$$

where:

- $\hat{H}$ is the Hamiltonian operator, including kinetic and potential energies.
- Ψ is the many-electron wavefunction, depending on the positions of all electrons.
- E is the total energy of the system.

The Hamiltonian for such a system is:

$$\hat{H} = \sum_{i=1}^{N} \left(-\frac{\hbar^2}{2m}\nabla_i^2 + V_{\text{external}}(\mathbf{r}_i) \right) + \sum_{i<j} \frac{e^2}{4\pi\epsilon_0 |\mathbf{r}_i - \mathbf{r}_j|} \tag{8.2}$$

where:

- The first term represents the kinetic energy of each electron, often denoted as $\hat{T}$.
- V_{external} represents the external potential.
- The second summation accounts for electron-electron Coulomb interactions.

8.2. The Single-Electron System

Let's first review how to obtain a numerical solution for the simplest case, a single electron under a given external potential:

$$\left(-\frac{\hbar^2}{2m}\nabla^2 + V_{\text{external}}(\mathbf{r})\right)\Psi(\mathbf{r}) = E\Psi(\mathbf{r}) \tag{8.3}$$

A typical solution involves the following steps:

1. Define the discrete grid to describe the wavefunction spanning.
2. Define the external potential (V_{external}) on the given spatial grids.
3. Build up the Hamiltonian matrix (H) on the given spatial grids based on the V_{external} and kinetic energy operator.
4. Solve for the eigenvalues and eigenvectors of H.

8.2.1 Kinetic Energy Operator

The kinetic energy operator ($\hat{T}$) for an electron in one dimension is given by the Laplacian operator, the second derivative of the wavefunction with respect to position:

$$\hat{T} = -\frac{\hbar^2}{2m}\frac{d^2}{dx^2} \tag{8.4}$$

Using the finite difference method, the second derivative of a function $f(x)$ on a discrete grid of points $x_1, x_2, \ldots, x_N$ with spacing dx can be approximated by:

$$\left.\frac{d^2 f}{dx^2}\right|_{x_i} \approx \frac{f(x_{i+1}) - 2f(x_i) + f(x_{i-1})}{dx^2} \tag{8.5}$$

This approximation is represented by a matrix acting on the values of f at all grid points:

$$T = \frac{1}{dx^2}\begin{bmatrix} -2 & 1 & 0 & 0 & 0 \\ 1 & -2 & 1 & 0 & 0 \\ 0 & 1 & -2 & 1 & 0 \\ 0 & 0 & 1 & -2 & 1 \\ 0 & 0 & 0 & 1 & -2 \end{bmatrix} \tag{8.6}$$

8.2.2 Solution of a 1D System

The following code demonstrates obtaining a numerical solution in 1D for an arbitrary potential.

```
# Spatial grid parameters
x_min, x_max = -10.0, 10.0
N = 1000
x = np.linspace(x_min, x_max, N)
dx = x[1] - x[0]

# Define external potential
```

```
V_ext = 0.5 * x**2                    # harmonic oscillator
#V_ext = -1 / np.sqrt((x**2+0.5))     # potential from a positive charge

# Define kinetic energy operator
T = -0.5 * (-2 * np.eye(N) + np.eye(N, k=1) + np.eye(N, k=-1)) / dx**2

# Total Hamiltonian
H = T + np.diag(V_ext)

# Solve eigenvalue problem
energies, wavefunctions = np.linalg.eigh(H)

# Electron density
psi = wavefunctions[:, 0]
psi /= np.sqrt(np.sum(np.abs(psi)**2) * dx)
rho = np.abs(psi)**2

# Plot the effective potential
fig, axs = plt.subplots(1, 2, figsize=(12, 4))

axs[0].plot(x, V_ext, label="$V_{\mathrm{external}}$", linestyle="--")
axs[0].set_xlabel("$x$")
axs[0].set_ylabel("Potential")
for i in range(3):
    axs[0].axhline(energies[i])
axs[0].legend()
axs[0].set_ylim(0, 10)

# Plot the electron density
for i in range(3):
    rho = np.abs(wavefunctions[:, i])**2  # Ground state
    rho /= (rho.sum() * dx)
    axs[1].plot(x, rho, label=f"E$_{i}$={energies[i]:.2f} Ha")
axs[1].plot(x, rho)
axs[1].set_xlabel("$x$")
axs[1].set_ylabel("Electron Density")
axs[1].legend()

plt.tight_layout()
plt.show()
plt.savefig("lec_08-harmonic.pdf")
```

This Python code solves the Schrödinger equation for a quantum particle in a one-dimensional potential using finite-difference methods. The script computes the eigenvalues and eigenfunctions of the Hamiltonian matrix, which represents the system, and visualizes the external potential, energy levels, and electron densities in Fig. 8.1.

The computed kinetic energy and external energy can be validated against known analytical solutions. For a harmonic oscillator, the total energy of the n-th quantum state is given by:

$$E_n = \left(n + \frac{1}{2}\right)\hbar\omega,$$

where $n = 0, 1, 2, \ldots$ represents the quantum number. With $\hbar\omega = 1$ in the arbitrary unit, E_n should be equal to 0.5 Hartree, 1.5 Hartree, 2.5 Hartree, $\cdot$, which is consistent with the results in Fig. 8.1.

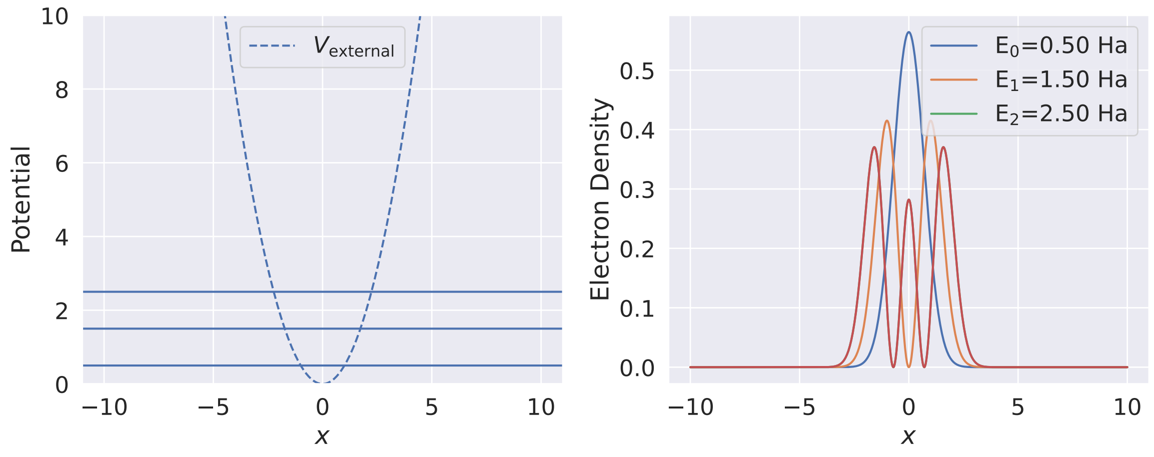


Figure 8.1: The numerical solution of harmonic oscillator.

To deepen understanding, one should experiment with different external potentials in the code and analyze the corresponding solutions. This exercise helps illustrate the physical meaning of each solution, including energy quantization, wavefunctions, and electron densities.

8.2.3 Solution of a 3D System

Next, let us consider a hydrogen atom in 3 dimension. The following Python code numerically solves the Schrödinger equation for a single electron in a three-dimensional (3D) softened Coulomb potential using the same finite-difference as described in the previous section.

```
import numpy as np
from scipy.sparse import kron, eye
from scipy.sparse import diags, csr_matrix
from scipy.sparse.linalg import eigsh  # Sparse eigenvalue solver
import matplotlib.pyplot as plt

def kinetic_energy_operator(N, dx):
    # Define 1D kinetic energy finite difference operator
    main_diag = -2 * np.ones(N)
    side_diag = np.ones(N - 1)
    T_1D = diags([main_diag, side_diag, side_diag], [0, -1, 1], shape=(
    N, N)) / dx**2

    # Build 3D kinetic energy operator using Kronecker products
    I = eye(N, format="csr")  # Identity matrix for each dimension
    T = kron(kron(T_1D, I), I) + kron(kron(I, T_1D), I) T += kron(kron(
    I, I), T_1D)

    return -0.5 * T  # Scale by -0.5 for the kinetic energy operator

# Step 1: Define 3D grid parameters (60*60*60)
N = 60       # Grid size along each dimension
L = 6        # Simulation box size in Bohr
dx = L / N
x = np.linspace(-L/2, L/2, N)
```

```
y = np.linspace(-L/2, L/2, N)
z = np.linspace(-L/2, L/2, N)
X, Y, Z = np.meshgrid(x, y, z, indexing="ij")

R1 = np.array([0, 0, 0])
num_electrons = 1.0

# Step 2: Deifine Softened Coulomb external potential for one proton
# To prevent singularity at nuclei, add a soft parameter
softening = 0.02
V_ext = -1 / np.sqrt((X - R1[0])**2 + (Y - R1[1])**2 + (Z - R1[2])**2 +
    softening**2)

# Step 3: Compute H matrix (stroed in a sparse format)
T = kinetic_energy_operator(N, dx)
H = T + diags(V_ext.flatten(), 0, shape=(N**3, N**3))

# Step 4: Solve the single-electron equation
energies, orbitals = eigsh(H, k=10, which="SA")
print("Energies from the solver\n", energies)

# Normalize wavefunction to ensure it represents one electron
psi = orbitals[:, 0].reshape((N, N, N))  # Ground state orbital
psi /= np.sqrt(np.sum(np.abs(psi)**2) * dx**3)
rho = np.abs(psi)**2  # Electron density

# Recompute energies with normalized wavefunction and density
T_s = np.sum(psi.flatten() * T.dot(psi.flatten())) * dx**3
E_ext = np.sum(rho * V_ext) * dx**3

# Output energies
print(f"Kinetic Energy  (T)       = {T_s:.6f} Hartree")
print(f"External Energy (E_ext).  = {E_ext:.6f} Hartree")
print(f"Total Energy    (E_total)= {T_s + E_ext:.6f} Hartree\n")

# Plot electron densities for the selected solutions
for i in [0, 4, 5]:
    psi = orbitals[:, i].reshape((N, N, N))
    psi /= np.sqrt(np.sum(np.abs(psi)**2) * dx**3)

    rho = np.abs(psi)**2
    rho1 = rho[N//2, :, :]
    print(i, np.sum(rho)*dx**3, np.sum(rho1)*dx**3, rho.max())
    plt.imshow(rho1,
               origin="lower",
               vmin=1e-5,
               vmax=5e-3)
    plt.xlabel("x (Bohr)")
    plt.ylabel("y (Bohr)")
    plt.title(f"Single H atom in level {i}: {energies[i]:.2f} Ha")
    plt.colorbar(label="Electron Density")
    plt.show()
```

To model a hydrogen atom, we first create an external potential on a $60 \times 60 \times 60$ grid. A one-dimensional finite-difference approximation of the kinetic energy operator is then constructed and extended to three dimensions using Kronecker products. The Hamiltonian matrix, $H = T + V_{\text{external}}$, is constructed and solved in sparse format for efficiency. Running this code generates the following output.

```
Energies from the solver
[-0.47800187 -0.0433897  -0.0433897  -0.0433897   0.03139164  0.16645611
  0.16645611  0.16645611  0.23494692  0.23494692]
Kinetic Energy  (T)      = 0.481822 Hartree
External Energy (E_ext). = -0.959824 Hartree
Total Energy    (E_total)= -0.478002 Hartree
```

Note that in the above codes, we also compute the kinetic and external energy to double check if the implemented code works properly.

$$T = -\frac{1}{2}\int \psi^*(\mathbf{r})\nabla^2\psi(\mathbf{r})d^3\mathbf{r}$$

$$E_{\text{external}} = \int \rho(\mathbf{r})V_{\text{external}}(\mathbf{r})d^3\mathbf{r}$$

The numerical expressions are

$$T \approx -\frac{1}{2}\sum_i \psi_i^* \left(\nabla^2\psi\right)_i dx^3$$

$$E_{\text{external}} \approx \sum_i \rho_i V_{\text{external},i} dx^3$$

To check if your implmentation is correct, one should compare the results with the analytical solutions for a single H atom. The energy levels for a hydrogen atom in Hartree units (where 1 Hartree = 27.2 eV) are given by the formula:

$$E_n = -\frac{1}{n^2}\ \text{Hartree}$$

In the ground state ($1s$), hydrogen atom has a kinetic energy of 0.5 Hartree and a potential energy of -1.0 Hartree, resulting in a total energy of -0.5 Hartree. Our computed results are consistent with these theoretical values. However, discrepancies arise for the excited states. For instance, the hydrogen atom should exhibit four degenerate orbitals ($2s$, $2p_x$, $2p_y$, and $2p_z$) with an energy of -0.125 Hartree. In our computation, these degeneracies are not perfectly captured, likely due to the limited resolution of the numerical grid. Refining the grid further is expected to yield closer agreement with theoretical results.

Despite these numerical discrepancies, Fig. 8.2 shows the electron density distributions for the $1s$, $2s$, and $2p$ orbitals, which qualitatively align with our expectations from quantum mechanics. Furthermore, it is worth noting that alternative algorithms and grid optimization techniques can achieve better accuracy. However, we leave these improvements as an exercise for the reader to explore.

8.3. Density Functional Theory for Many-Electrons

For systems with multiple electrons, solving the Schrödinger equation becomes intractable due to electron-electron interactions. Density Functional Theory (DFT), developed by Walter Kohn and Pierre Hohenberg [11], simplifies this problem by focusing on electron density $\rho(\mathbf{r})$.

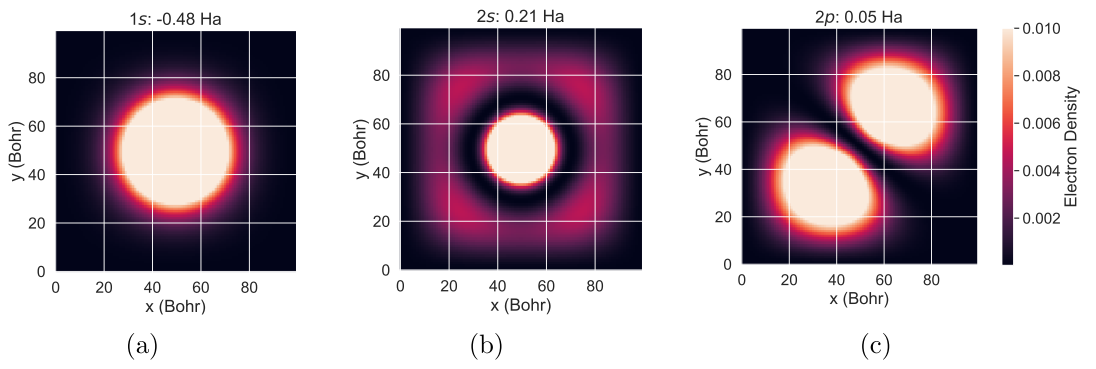


Figure 8.2: Computed hydrogen orbitals: (a) $1s$, (b) $2s$, and (c) $2p$.

8.3.1 The Kohn and Hohenberg Theorems

The Hohenberg-Kohn theorems lay the foundation of DFT:

1. **Uniqueness Theorem**: The ground-state electron density $\rho_0(\mathbf{r})$ uniquely determines the external potential $V_{\text{external}}(\mathbf{r})$, and hence all properties of the system. This implies that the many-body problem is effectively reformulated in terms of $\rho_0(\mathbf{r})$.

2. **Variational Principle**: There exists a universal functional $E[\rho]$ such that the ground-state energy E_0 can be obtained variationally:

$$E_0 = \min_{\rho} [E[\rho]] = \min_{\rho} \left[F[\rho] + \int V_{\text{external}}(\mathbf{r})\rho(\mathbf{r})\, d\mathbf{r} \right]$$

Here, $F[\rho]$ is a universal functional of the electron density, independent of the external potential, and contains the kinetic and electron-electron interaction energies.

The first theorem relies on the assumption that the external potential uniquely determines the Hamiltonian, and hence the ground-state wavefunction. The electron density, being a property of the wavefunction, indirectly determines the external potential. The variational principle follows naturally, as any trial density yields an energy equal to or higher than the true ground-state energy. For more details, please refer to the original work by Kohn and Hohenberg [12].

8.3.2 The Kohn-Sham Equations

While the Uniqueness theorems provide a conceptual framework, they do not offer a practical way to compute $F[\rho]$, which includes the complex electron-electron interactions. In the following work, Kohn and Sham introduced non-interacting electrons to approximate the ground-state density of an interacting system [12]. The total energy functional becomes:

$$E[\rho] = T_s[\rho] + E_{\text{external}}[\rho] + E_{\text{Hartree}}[\rho] + E_{\text{XC}}[\rho] \tag{8.7}$$

where:

- $T_s[\rho]$: Kinetic energy of non-interacting electrons.

- $E_{\text{external}}[\rho]$: Interaction with external potential.
- $E_{\text{Hartree}}[\rho]$: Hartree energy.
- $E_{\text{XC}}[\rho]$: Exchange-correlation energy.

The electron density is determined by solving the **Kohn-Sham equations**:

$$\left[-\frac{\hbar^2}{2m}\nabla^2 + V_{\text{eff}}(\mathbf{r})\right]\phi_i(\mathbf{r}) = \epsilon_i\phi_i(\mathbf{r}) \tag{8.8}$$

where $\phi_i(\mathbf{r})$ are the Kohn-Sham orbitals, ϵ_i are their corresponding energies, and the effective potential $V_{\text{eff}}(\mathbf{r})$ is given by:

$$V_{\text{eff}}(\mathbf{r}) = V_{\text{external}}(\mathbf{r}) + \int \frac{\rho(\mathbf{r}')}{|\mathbf{r}-\mathbf{r}'|}\,d\mathbf{r}' + \frac{\delta E_{\text{XC}}[\rho]}{\delta\rho(\mathbf{r})}$$

8.3.3 Effective Potential in Kohn-Sham Equations

Technically, we can solve the single electron equation as long as the Effective potential $V_{\text{eff}}(\mathbf{r})$ is known. Following the spirit of DFT, we seek to express all potential terms (V_{external}, E_{Hartree} and V_{XC}), base on the electron density.

First, for a system with nuclei located at positions $\mathbf{R}_j$ and with nuclear charges Z_j, the external potential at a point $\mathbf{r}$ in space is given by:

$$V_{\text{external}}(\mathbf{r}) = -\sum_j \frac{Z_j}{|\mathbf{r}-\mathbf{R}_j|}$$

Second, the **Hartree potential** $V_{\text{Hartree}}(\mathbf{r})$ is the electrostatic potential felt by an electron at position $\mathbf{r}$ due to the total electron density:

$$V_{\text{Hartree}}(\mathbf{r}) = \int \frac{\rho(\mathbf{r}')}{|\mathbf{r}-\mathbf{r}'|}d\mathbf{r}' \tag{8.9}$$

In practice, the integral is a long-range term and it needs to be computed using techniques such as Fourier transforms or Poisson solvers to handle the long-range nature of the Coulomb interaction.

The corresponding Hartree energy is given by:

$$E_{\text{Hartree}} = \frac{1}{2}\int\int \frac{\rho(\mathbf{r})\rho(\mathbf{r}')}{|\mathbf{r}-\mathbf{r}'|}d^3\mathbf{r}d^3\mathbf{r}' \tag{8.10}$$

This integral computes the repulsion between all pairs of infinitesimal density elements at positions $\mathbf{r}$ and $\mathbf{r}'$. In a discrete grid, we can approximate this as:

$$E_{\text{Hartree}} \approx \frac{1}{2}\sum_{i,j} \frac{\rho_i\rho_j}{|\mathbf{r}_i-\mathbf{r}_j|}dx^3dx^3 \tag{8.11}$$

Finally, the **Exchange-Correlation Functional** is a tricky term that represents the difference between the true kinetic and electron-electron interaction energies and those of the non-interacting reference system. The exact form is unknown. Hence approximations are necessary.

$$E_{\text{XC}} = \int \epsilon_{\text{XC}}(\rho(\mathbf{r}))\rho(\mathbf{r})d^3\mathbf{r} \approx \sum_i \epsilon_{\text{XC}}(\rho_i)\rho_i dx^3 \tag{8.12}$$

where $\epsilon_{\text{XC}}(\rho)$ is the exchange-correlation energy density per electron. It is typically split into exchange and correlation parts to represent the energy due to exchange and correlation effects.

$$\epsilon_{\text{XC}}(\rho) = \epsilon_{\text{X}}(\rho) + \epsilon_{\text{C}}(\rho) \tag{8.13}$$

The effective potential includes $V_{\text{external}}(\mathbf{r})$, $V_{\text{Hartree}}(\mathbf{r})$, and $V_{\text{XC}}(\mathbf{r})$:

$$V_{\text{Hartree}}(\mathbf{r}) = \int \frac{\rho(\mathbf{r}')}{|\mathbf{r} - \mathbf{r}'|} d\mathbf{r}' \tag{8.14}$$

In the original paper, it was found that when the electron density does not vary significantly (e.g., uniform electron gas, nearly free electrons in a metal). It can be estimated by **local density approximation** LDA. For the exchange energy in the LDA, the commonly used expression (for spin-unpolarized systems) is:

$$\epsilon_{\text{X}}(\rho) = -\frac{3}{4}\left(\frac{3}{\pi}\right)^{1/3} \rho^{1/3} \tag{8.15}$$

$$V_X(\rho) = \frac{4}{3}\epsilon_X(\rho) \tag{8.16}$$

The correlation energy $\epsilon_{\text{C}}(\rho)$ depends on the specific form chosen. The popular yet simple parametrizations for $\epsilon_{\text{C}}(\rho)$ in LDA are the Vosko, Wilk, and Nusair (VWN) [13] and the Perdew-Zunger correlation functional [14], which are based on fitting to Monte Carlo data. The correlation energy per electron is given by a piecewise function that depends on the Wigner-Seitz radius r_s, which is related to the electron density by:

$$r_s = \left(\frac{3}{4\pi\rho}\right)^{1/3}$$

For spin-unpolarized systems, the Perdew-Zunger correlation energy per electron can be expressed as:

$$\epsilon_{\text{C}}(r_s) = \begin{cases} A + Br_s \ln(r_s) + Cr_s \ln(r_s) + Dr_s & \text{for } r_s \leq 1 \\ \gamma/(1 + \beta_1\sqrt{r_s} + \beta_2 r_s) & \text{for } r_s > 1 \end{cases} \tag{8.17}$$

and the correlation potential is

$$V_{\text{C}}(r_s) = \begin{cases} A \ln r_s + (B - \frac{A}{3}) + \frac{2}{3}Cr_s \ln(r_s) + \frac{2D-C}{3} r_s & \text{for } r_s \leq 1 \\ \epsilon_{\text{C}}(1 + \frac{7}{6}\beta_1\sqrt{r_s} + \frac{4}{3}\beta_2 r_s)/(1 + \beta_1\sqrt{r_s} + \beta_2 r_s) & \text{for } r_s > 1 \end{cases} \tag{8.18}$$

where the relevant parameters are A=0.0311, B=-0.048, C=0.002, D=-0.0116, γ = -0.1423, β_1=1.0529 and β_2 = 0.3334.

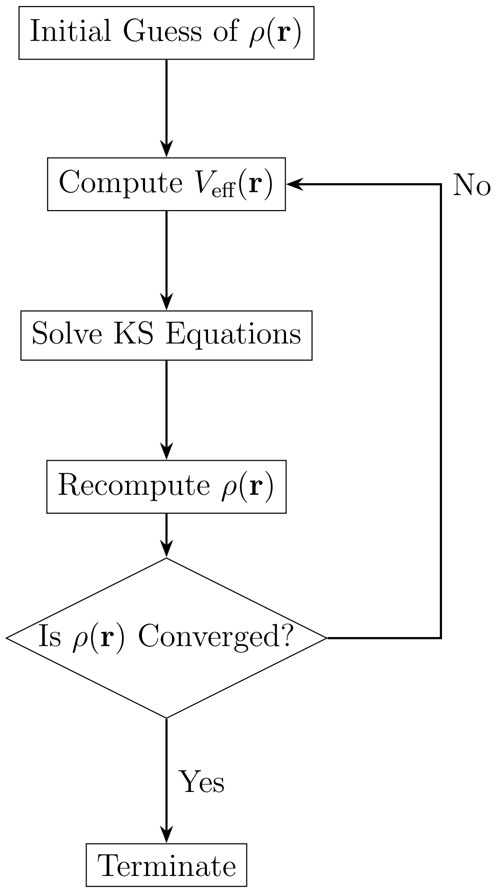

Initial Guess of $\rho(\mathbf{r})$
Compute $V_{\text{eff}}(\mathbf{r})$
No
Solve KS Equations
Recompute $\rho(\mathbf{r})$
Is $\rho(\mathbf{r})$ Converged?
Yes
Terminate

Figure 8.3: Iterative process to solve the Kohn-Sham equations.

8.3.4 Iterative Update of Electron Density

After choosing the form of E_{XC}, the total energy can be expressed as electron density. Since the electron density becomes the only concern of interest, one can start with an initial guess of the wavefunction and then solve the KS equations. From each individual Kohn-Sham Equation, we get $\phi_i(\mathbf{r})$ and ϵ_i. After knowing each $\phi_i(\mathbf{r})$, the **Total Electron Density** can be re-estimated as follows:

$$\rho(\mathbf{r}) = \sum_i^{\text{occ}} |\phi_i(\mathbf{r})|^2$$

One can then use the updated $\rho(\mathbf{r})$ to run another iteration. Repeating the iterations several times, one expect to the convergence of $\rho(\mathbf{r})$ and total energy.

8.3.5 Practical Workflow

In short, DFT proposed two important concepts to solve the many electron problems.

1. The simplification from many electron equations to a set of single electron KS equations. To enable this transition, one needs to seek an optimal effective potential for the single KS equations. This requires the introduction of Hartree potential(V_{Hartree}) and Exchange-Correlation potential (V_{XC}) on top of the traditional single electron problems based on $(T + V_{\text{external}})$ only.

2. When solving the KS equations, we use the electron density (ρ) to construct the $\hat{H}$. Thus all potential terms needs to be expressed as the function of ρ.

After knowing ρ, the tentative solution of KS energy and orbital can be obtained. From the orbital, we can get the updated ρ and V_{eff} and then repeat the process iteratively. When the system reaches a steady state, i.e., the ρ and V_{eff} no longer change, we can terminate this calculation. The entire procedure can be summarized in Fig. 8.3.

8.4. Summary

In this chapter, we began with the foundational quantum mechanical framework for electronic structure calculations, starting with the Schrödinger equation. For single-electron systems, we demonstrated how to numerically solve the equation using finite-difference methods, providing insights into energy levels, wavefunctions, and electron densities.

Transitioning to multi-electron systems, we highlighted the limitations of directly solving the many-body Schrödinger equation due to the complexity introduced by electron-electron interactions. As a solution, DFT was introduced as a groundbreaking approach to simplify the many-electron problem. The Hohenberg-Kohn theorems provided the theoretical foundation for DFT, demonstrating that the ground-state properties of a system can be uniquely determined by its electron density. Additionally, the Kohn-Sham equations offered a practical framework for approximating the ground-state electron density through non-interacting electrons and the concept of the effective potential.

Key aspects of the DFT formalism, including the Hartree energy, exchange-correlation energy, and the iterative procedure for achieving self-consistency, were discussed in detail. We also outlined the practical workflow for solving the Kohn-Sham equations, emphasizing the importance of convergence and the role of approximations in exchange-correlation functionals. Mastering these concepts lays the groundwork for realistic calculations of the electronic properties of materials, which will be explored in the following chapters.

9. DFT Simulation of the Hydrogen Molecule

Having learned the basic concept of DFT, we will continue to apply it to simulate a more complex system than a single-electron system: the H_2 molecule. This small size is ideal for demonstrating the DFT method in a manageable way.

By solving for the ground-state energy and electron density of H_2, we hope to better understand the numerical aspects of the DFT method. In addition, we will analyze the results to understand the bonding in H_2 and the electron distribution.

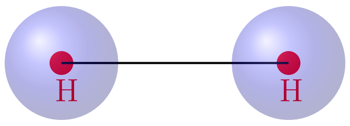


Figure 9.1: Representation of an H_2 Molecule

9.1. Basic Setup

In an H_2 molecule, we need to consider the following variables in the Kohn-Sham equation.

- Nuclei: Two protons located at positions R_1 and R_2.
- Electrons: Two electrons interacting with the protons and each other.

The Hamiltonian is given by:

$$\hat{H} = \hat{T}_e + \hat{V}_{\text{external}} + \hat{V}_{\text{ee}} + \hat{V}_{\text{nn}} \tag{9.1}$$

where:

- $\hat{T}_e$: Kinetic energy of the electrons.
- $\hat{V}_{\text{external}}$: External potential due to nuclei.
- $\hat{V}_{\text{ee}}$: Electron-electron interaction.
- $\hat{V}_{\text{nn}}$: Nucleus-nucleus interaction.

In the Kohn-Sham formalism, this reduces to a single Kohn-Sham equation. For H_2, both electrons occupy the same Kohn-Sham orbital.

$$\left[-\frac{\hbar^2}{2m}\nabla^2 + V_{\text{eff}}(\mathbf{r})\right]\phi_i(\mathbf{r}) = \epsilon_i\phi_i(\mathbf{r}) \tag{9.2}$$

9.2. Effective Potentials

9.2.1 External Potential

For an H_2 molecule, the external potential is given by the Coulomb interaction with the two protons:

$$V_{\text{external}}(\mathbf{r}) = -\frac{1}{\sqrt{|\mathbf{r}-\mathbf{R}_1|^2+\alpha^2}} - \frac{1}{\sqrt{|\mathbf{r}-\mathbf{R}_2|^2+\alpha^2}} \tag{9.3}$$

where α is a small softening parameter to avoid singularities.

9.2.2 Hartree Potential and Energy

The Hartree potential represents the classical electrostatic potential at a point $\mathbf{r}$ due to the electron density $\rho(\mathbf{r})$. It is defined as:

$$V_{\text{Hartree}}(\mathbf{r}) = \int \frac{\rho(\mathbf{r}')}{|\mathbf{r}-\mathbf{r}'|} d\mathbf{r}' \tag{9.4}$$

Computing this integral directly in real space can be computationally expensive, especially for large systems. To address this, the Hartree potential can be calculated more efficiently using a Fourier-space approach. In reciprocal space, the Hartree potential is expressed as:

$$V_{\text{Hartree}}(\mathbf{k}) = \frac{4\pi\rho(\mathbf{k})}{|\mathbf{k}|^2} \tag{9.5}$$

where $\rho(\mathbf{k})$ is the Fourier transform of the electron density. The corresponding Hartree energy is then given by:

$$E_{\text{Hartree}} = \frac{1}{2}\int \rho(\mathbf{r})V_{\text{Hartree}}(\mathbf{r})d\mathbf{r} \tag{9.6}$$

This Fourier-space method significantly reduces the computational cost of evaluating the Hartree potential, making it well-suited for numerical simulations involving a large number of grid points.

9.2.3 Exchange and Correlation

The exchange-correlation energy, E_{XC}, accounts for the quantum interactions between electrons. The exchange-correlation potential, $V_{\text{XC}}(\mathbf{r})$, is defined as:

$$V_{\text{XC}}(\mathbf{r}) = \frac{\delta E_{\text{XC}}[\rho]}{\delta\rho(\mathbf{r})} \tag{9.7}$$

Using the Local Density Approximation (LDA), the exchange-correlation energy is:

$$E_{\text{XC}}^{\text{LDA}}[\rho] = \int \epsilon_{\text{XC}}(\rho(\mathbf{r}))\rho(\mathbf{r})\,d\mathbf{r} \tag{9.8}$$

9.3. Python Implementation

Based our previous implementation on single electron system, we create the following Python code to compute the ground state of an H_2 molecule using DFT though the self-consistent field (SCF) calculation.

```python
import numpy as np
from scipy.sparse import kron, eye
from scipy.sparse import diags, csr_matrix
from scipy.fft import fftn, ifftn
from scipy.sparse.linalg import eigsh
import matplotlib.pyplot as plt

# Define 3D grid parameters
N = 80 #120     # Grid size along each dimension
L = 6  #40      # Simulation box size in Bohr
dx = L / N
x = np.linspace(-L/2, L/2, N)
y = np.linspace(-L/2, L/2, N)
z = np.linspace(-L/2, L/2, N)
X, Y, Z = np.meshgrid(x, y, z, indexing="ij")

# Positions of the two protons in H2
R1 = np.array([-0.7, 0, 0])
R2 = np.array([0.7, 0, 0])
num_electrons = 2.0
E_nuc = 1.0 / (R2[0]-R1[0])

# Softened Coulomb potential for two protons
alpha = 0.1
V_ext = -1 / np.sqrt((X - R1[0])**2 + (Y - R1[1])**2 + (Z - R1[2])**2 +
    alpha**2)
V_ext += -1 / np.sqrt((X - R2[0])**2 + (Y - R2[1])**2 + (Z - R2[2])**2
   + alpha**2)

# Initial guess for electron density
rho = np.exp(-1 * ((X - R1[0])**2 + (Y - R1[1])**2 + (Z - R1[2])**2))
rho += np.exp(-1 * ((X - R2[0])**2 + (Y - R2[1])**2 + (Z - R2[2])**2))
total_density = np.sum(rho) * dx**3
rho *= num_electrons / total_density

# Define FFT-based Poisson solver for Hartree potential
def compute_hartree_potential(rho):
    kx = 2 * np.pi * np.fft.fftfreq(N, d=dx)
    ky = 2 * np.pi * np.fft.fftfreq(N, d=dx)
    kz = 2 * np.pi * np.fft.fftfreq(N, d=dx)
    KX, KY, KZ = np.meshgrid(kx, ky, kz, indexing="ij")
    k2 = KX**2 + KY**2 + KZ**2
    k2[0, 0, 0] = 1     # Avoid division by zero

    rho_k = fftn(rho)
    V_H_k = 4 * np.pi * rho_k / k2
    V_H_k[0, 0, 0] = 0  # Set the zero-frequency term to zero

    V_H = np.real(ifftn(V_H_k))
    return V_H #* 2

def kinetic_energy_operator(N, dx):
    # Define 1D kinetic energy finite difference operator
    main_diag = -2 * np.ones(N)
    side_diag = np.ones(N - 1)
    T_1D = diags([main_diag, side_diag, side_diag],
                 [0, -1, 1],
                 shape=(N, N))
    T_1D /= dx**2

    # Build 3D kinetic energy operator using Kronecker products
    I = eye(N, format="csr")  # Identity matrix for each dimension
    T = kron(kron(T_1D, I), I) + kron(kron(I, T_1D), I) + kron(kron(I,
   I), T_1D)

    return -0.5 * T  # Scale by -0.5

# SCF loop parameters
tolerance = 1e-6
max_iterations = 100
damping_factor = 0.5

# Kinetic energy operator (sparse)
T = kinetic_energy_operator(N, dx); print(T.shape)

for iteration in range(max_iterations):
    # Compute Hartree and XC potentials
    V_H = compute_hartree_potential(rho)
    V_XC = -0.75 *(3 / np.pi)**(1 / 3) * rho**(1 / 3)
    V_eff = V_ext + V_H + V_XC

    # Construct H
    V_eff_flat = V_eff.flatten()
    H = T + diags(V_eff_flat, 0, shape=(N**3, N**3))

    # Solve the Kohn-Sham equation with a sparse eigenvalue solve
    energies, orbitals = eigsh(H, k=1, which="SA")
    psi = orbitals[:, 0].reshape((N, N, N))
    psi /= np.sqrt(np.sum(np.abs(psi)**2) * dx**3)

    # Update electron density with normalization
    rho_new = num_electrons * np.abs(psi)**2

    # Calculate energies for this iteration
    T_s = 2* np.sum(orbitals[:, 0] * T.dot(orbitals[:, 0])) * dx**3
    E_H = 0.5 * np.sum(rho * V_H) * dx**3
    E_ext = np.sum(rho * V_ext) * dx**3
    E_XC = np.sum(rho * V_XC) * dx**3
    E_total = T_s + E_ext + E_H + E_XC + E_nuc

    # Print the energies
    print(f"Iteration {iteration + 1}:")
    print(f"Kinetic Energy       (T)    = {T_s:.6f} Hartree")
    print(f"External Energy      (E_ext)= {E_ext:.6f} Hartree")
    print(f"Hartree Energy       (E_H)  = {E_H:.6f} Hartree")
    print(f"Exchange-Correlation (E_XC) = {E_XC:.6f} Hartree")
    print(f"Total Effective Energy      = {E_total:.6f} Hartree")
    print(f"Solver KS HOMO Energy       = {energies[0]:.6f} Hartree\n")
    # Damping update
    rho = (1 - damping_factor) * rho + damping_factor * rho_new
```

```python
    total_density = np.sum(rho) * dx**3

    # Check convergence
    if np.linalg.norm(rho_new - rho) < tolerance:
        print(f"Converged after {iteration + 1} iterations")
        break
else:
    print("Did not converge within the maximum number of iterations")
```

The code iteratively solves the KS equations. In each iteration, the electron density is updated from the orbitals, and the effective potential is recalculated using the new density. This process continues until the electron density converges (i.e., when the difference between the new and old density is smaller than a set tolerance).

Here the most expensive part of calculation lies in the solving the eigenvalue of Hamiltonian matrix. Due to the numerical grid setting, the size of Hamiltonian has a cubic relation with respect to the number of grids in each dimension. The function eigsh from `Scipy` is used to solve the eigenvalue problem for the Hamiltonian with the sparsification technique, returning the Kohn-Sham energies and orbitals. Once the Hamiltonian is constructed and solved, the Kohn-Sham orbitals $\phi_i(\mathbf{r})$ are used to update the electron density $\rho(\mathbf{r})$. The SCF loop continues until the electron density converges.

An example output looks like the following

```
Iteration 1:
Kinetic Energy       (T)    = 1.263639 Hartree
External Energy      (E_ext)= -3.557743 Hartree
Hartree Energy       (E_H)  = 0.545992 Hartree
Exchange-Correlation (E_XC) = -0.819774 Hartree
Total Effective Energy      = -1.853602 Hartree
Solver HOMO Energy          = -1.046968 Hartree

Iteration 2:
Kinetic Energy       (T)    = 1.256039 Hartree
External Energy      (E_ext)= -3.599140 Hartree
Hartree Energy       (E_H)  = 0.545473 Hartree
Exchange-Correlation (E_XC) = -0.812328 Hartree
Total Effective Energy      = -1.895671 Hartree
Solver HOMO Energy          = -1.046764 Hartree

Iteration 3:
Kinetic Energy       (T)    = 1.253817 Hartree
External Energy      (E_ext)= -3.614615 Hartree
Hartree Energy       (E_H)  = 0.543885 Hartree
Exchange-Correlation (E_XC) = -0.809045 Hartree
Total Effective Energy      = -1.911672 Hartree
Solver HOMO Energy          = -1.047270 Hartree
```

To provide a comparison for convergence values in the Kohn-Sham DFT iterations, here are the typical reference values for the energy components of the hydrogen molecule [15]:

Clearly, the results qualitatively agree with the DFT results found in the literature, except for a notable underestimation of the Hartree energy. If time allows, one can repeat

Table 9.1: Energy Components of the H_2 Molecule Simulation

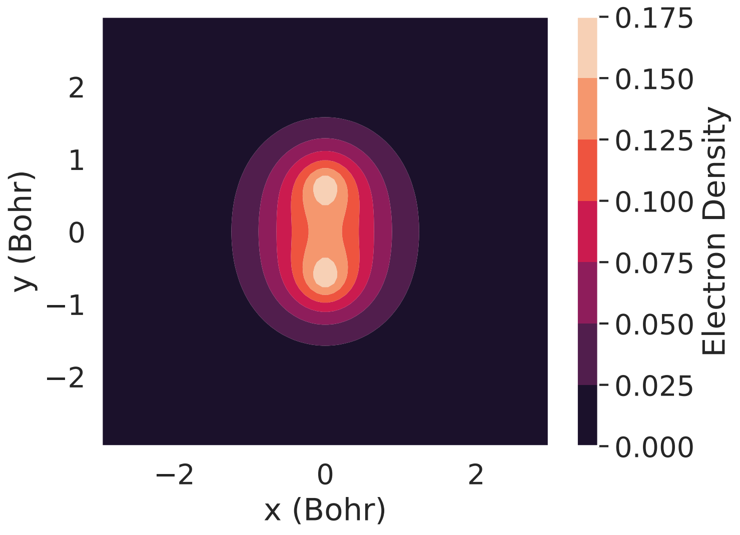
Energy Component	**Value Range (Hartree)**
Kinetic Energy (T)	1.25 ± 0.05
External Energy (E_{external})	-3.60 ± 0.20
Hartree Energy (E_{Hartree})	1.25 ± 0.10
Exchange-Correlation Energy (E_{XC})	-0.85 ± 0.05
Total Energy	-1.19 ± 0.02
HOMO Energy	-0.60 ± 0.05

the simulation by increasing L and N helps mitigate edge effects, especially for long-range interactions such as the Hartree term. Although it requires a higher computational cost, this should provide a more accurate representation of the continuum electron density, ensuring that calculations remain closer to these benchmark values.

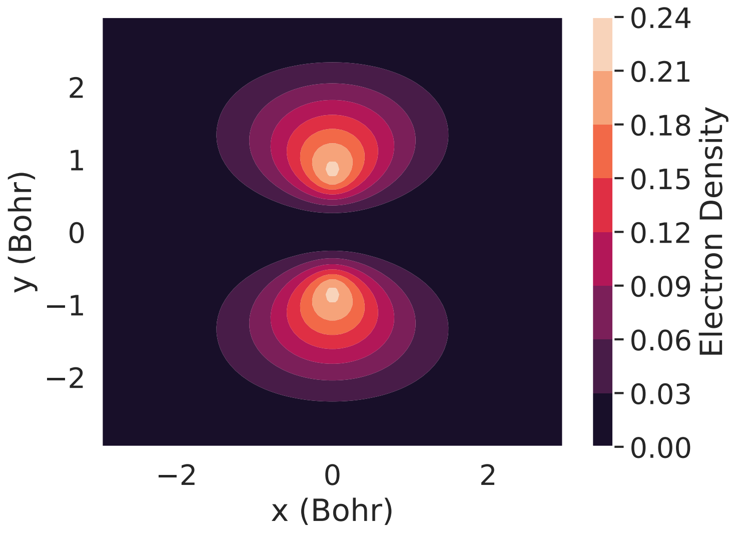
9.4. Physical Interpretation

The electron density can also be analyzed to verify whether the calculation successfully reproduces the well-known bonding and antibonding characteristics of the ground and first excited states. The following code provides a straightforward implementation for recalculating the electron density from the eigenfunctions and visualizing it in the 2D xy-plane.

```python
# Get the first two solutions
energies, orbitals = eigsh(H, k=2, which="SA")

# Plot the ground state at z=0 plane
psi = orbitals[:, 0].reshape((N, N, N))  # Ground state orbital
psi /= np.sqrt(np.sum(np.abs(psi)**2) * dx**3)
rho = np.abs(psi)**2

plt.contourf(rho[:, :, N//2],
             extent=(-L/2, L/2, -L/2, L/2),
             origin="lower")
plt.colorbar(label="Electron Density")
plt.xlabel("x (Bohr)")
plt.ylabel("y (Bohr)")
plt.savefig("H2-bond.png")
plt.close()

# Plot the first excited state at z=0 plane
psi = orbitals[:, 1].reshape((N, N, N))
psi /= np.sqrt(np.sum(np.abs(psi)**2) * dx**3)
rho = np.abs(psi)**2
rho *= num_electrons / (np.sum(rho) * dx**3)

plt.contourf(rho[:, :, N//2],
             extent=(-L/2, L/2, -L/2, L/2),
             origin="lower")
plt.colorbar(label="Electron Density")
plt.xlabel("x (Bohr)")
plt.ylabel"y (Bohr)")
plt.savefig("H2-antibond.png")
```



(a) Bonding



(b) Antibonding

Figure 9.2: The DFT simulated electron density for bonding and antibonding states in a H_2 molecule.

The final results are shown in Fig. 9.2. For the ground state, there is a strong overlap between the electron densities around the two hydrogen nuclei, indicating the formation of a bonding state. In contrast, for the first excited state, the electron density is repelled from the region between the two nuclei, suggesting an anti-bonding state for the hydrogen molecule. These results are consistent with the well-known molecular orbital theory, which describes bonding and antibonding states as arising from constructive and destructive interference of atomic orbitals, respectively.

9.5. Summary

We successfully developed a computational code to simulate the H_2 molecule using the Kohn-Sham formalism within a numerical grid framework. The final numerical results demonstrate that most energy components quantitatively agree with previously reported DFT results in the literature, with the exception of the Hartree energy. It is anticipated that adopting a finer grid resolution could improve the accuracy of Hartree energy at the expense of higher computational cost. Despite these limitations, the simplicity of the script implemented provides an accessible platform to gain hands-on experience with the numerical procedures involved in solving the Kohn-Sham equations and to deepen the understanding of chemical bonding in simple molecules such as H_2.

10. Efficient DFT via the Localized Basis Set

In the previous chapter, we have implemented a straightforward DFT approach to compute the electronic structure of the H_2 molecule. Although it can reproduce the bonding and antibonding state, the calculation, based on numerical grids on the real space, required diagonalizing a large Hamiltonian matrix, which can be computationally expensive for systems with many electrons. Moreover, the Hamiltonian matrix is typically sparse because the electron density decays exponentially far from the nucleus, leading to negligible overlap for basis functions centered on distant atoms. This sparsity suggests opportunities for optimization.

One such optimization involves the use of localized basis sets, which allow us to represent wavefunctions more efficiently. In this chapter, we will introduce the concept of localized basis sets, starting with the Slater-type orbital, and discuss their role in reducing computational costs while maintaining accuracy.

10.1. The Slater-type Orbital Basis Set

The Slater-type orbital (STO) basis set is a class of atomic orbital basis functions widely used in quantum chemistry to approximate the wavefunctions of electrons in atoms and molecules. These basis functions are named after John C. Slater, who introduced them to capture the essential features of atomic orbitals derived from solutions to the Schrödinger equation for hydrogen-like atoms.

The general form of an STO basis function is:

$$\chi^{nlm}(\mathbf{r}) = N r^{n-1} e^{-\zeta r} Y_{lm}(\theta, \phi) \tag{10.1}$$

where:

- N is a normalization constant.
- r is the radial distance from the nucleus.
- n is the principal quantum number (usually a positive integer).
- ζ is an exponent (often called the orbital exponent) that controls the decay rate of the function with distance from the nucleus.
- $Y_{lm}(\theta, \phi)$ is a spherical harmonic function.

In this formula, only ζ is a constant related to the effective charge of the nucleus, the nuclear charge being partly shielded by electrons. Historically, the effective nuclear charge was estimated by Slater's rules.

1. Electrons in the same n-shell shield 0.35 (except $1s$, where it's 0.30).
2. Electrons in the (n-1)-shell shield 0.85.
3. Electrons in shells n-2 or lower shield 1.0.

Empirically determined ζ values are often tabulated in quantum chemistry and physics resources. Some example values are shown in the following table.

Table 10.1: STO ζ values for selected elements and orbitals

Element	**Orbital**	ζ
H	$1s$	1.00
He	$1s$	1.69
C	$1s$	6.00
C	$2s$	1.72
C	$2p$	1.62

10.1.1 Numerical Behaviors

According to Slater, the STO is designed to decay exponentially with distance r from the nucleus, mimicking the behavior of atomic orbitals.

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

# Define the radial part of the STO for s, p, and d orbitals
def sto_radial_s(r, zeta, n):
    """
    Slater-type orbital for s orbitals.
    """
    return r**(n-1) * np.exp(-zeta * r)

def sto_radial_p(r, zeta, n):
    """
    Slater-type orbital for p orbitals.
    """
    return r**(n-1) * np.exp(-zeta * r) * r

# Parameters
r = np.linspace(0, 10, 200)  # Radial distance from the nucleus
zeta_1s = 1.0
zeta_2s = 0.5
zeta_2p = 0.5

# Radial wave functions
radial_1s = sto_radial_s(r, zeta_1s, 1)
radial_2s = sto_radial_s(r, zeta_2s, 2)
radial_2p = sto_radial_p(r, zeta_2p, 2)

```

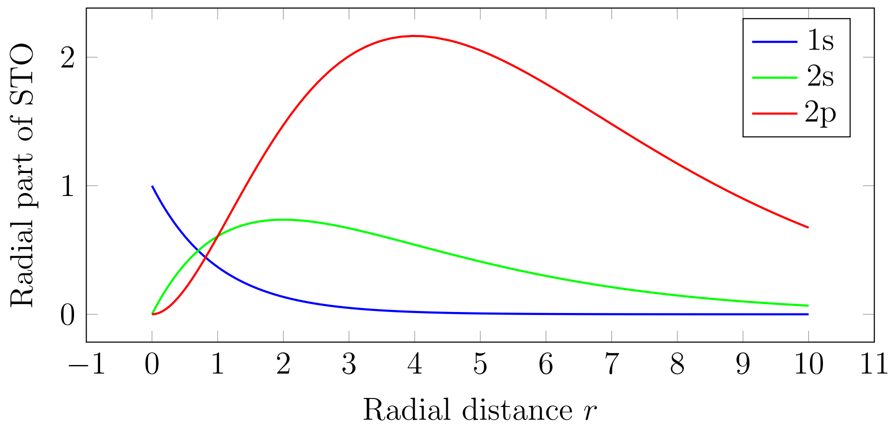


Figure 10.1: Slater-type Orbitals (STO) for $1s$, $2s$, $2p$.

```python
# Plot the radial functions
plt.figure(figsize=(10, 8))
plt.plot(r, radial_1s, label="1s")
plt.plot(r, radial_2s, label="2s")
plt.plot(r, radial_2p, label="2p")
plt.xlabel("Radial distance $r$")
plt.ylabel("Radial part of STO")
plt.title("Slater-type Orbitals (STO) for 1s, 2s, 2p")
plt.legend()
plt.show()
```

In addition, STOs include spherical harmonics $Y_{lm}(\theta, \phi)$ to account for the angular dependence, allowing the STOs to represent orbitals with various shapes, such as.

1. s-orbitals ($l = 0$): Spherically symmetric.

2. p-orbitals ($l = 1$): Dumbbell-shaped.

3. d-orbitals ($l = 2$) and higher angular momentum orbitals: More complex shapes.

This combination of radial and angular components makes STOs flexible and capable of approximating a wide range of orbital types.

10.1.2 Limitations

Although STOs are a good approximation to the shape of atomic orbitals, they are rarely used directly in practical quantum chemistry calculations. This is because the evaluation of two-electron integrals (which are required in Hartree-Fock and post-Hartree-Fock methods) with STOs is computationally expensive. Specifically, the overlap, kinetic, and electron repulsion integrals involving STOs do not have simple analytical solutions and must be evaluated numerically, which is slow and inefficient. In general, STOs are not orthogonal to each other. This lack of orthogonality arises because STOs do not inherently satisfy the orthogonality condition; they are designed primarily to approximate the shape and decay of hydrogen-like atomic orbitals rather than to be orthogonal.

10.2. Gaussian-type orbitals

Gaussian-type orbitals (GTOs) are widely used in quantum chemistry as numerical approximations to STOs. Their popularity stems from their ability to enable analytical solutions for integrals, making computations significantly faster and more efficient. However, GTOs differ from STOs in key aspects: they do not decay as quickly with distance from the nucleus and do not resemble the shape of atomic orbitals as closely. To overcome this limitation, Gaussian basis sets are constructed by combining multiple GTOs to better approximate STOs, a method known as STO-nG, where each STO is represented by n Gaussians.

10.2.1 Linear Combination of Multiple Gaussian

A common Gaussian takes the following form,

$$\chi^{nlm}_{\text{GTO}}(r) = br^{n-1}e^{-\alpha r^2}Y_{l,m}(\theta, \phi) \tag{10.2}$$

where b is the coefficients to normalize wavefunction, n is a parameter that defines the orbital's angular momentum (s-, p-, or d-type orbitals), α is a Gaussian exponent that controls the spread of the Gaussian. The exponential term, $e^{-\alpha r^2}$, causes the Gaussian function to decay more rapidly than an STO, making a single GTO unsuitable for directly replicating the decay behavior of STOs.

Gaussian functions decay more quickly than Slater functions, so a single Gaussian function cannot perfectly approximate an STO. However, by combining several Gaussian functions with different exponents (α) and coefficients (b), we can closely approximate the radial part of a STO.

$$\chi^{n}_{\text{STO-3G}}(r) = \sum_{i=1}^{3} b_i N_i r^{n-1} e^{-\alpha_i r^2}, \tag{10.3}$$

The values of b_i and α_i are chosen to best approximate the shape of the Slater-type orbital for each type of atomic orbital ($1s$, $2s$, $2p$, etc.). N_i is the normalization constants to ensure

$$\int Ne^{-\alpha \mathbf{r^2}} dr^3 = 1.0.$$

As we will discuss later, $N = (2\alpha/\pi)^{3/4}$.

For a $1s$ atomic orbital, the STO-3G function would be:

$$\chi_{\text{STO-3G}}(r) = b_1N_1e^{-\alpha_1 r^2} + b_2N_2e^{-\alpha_2 r^2} + b_3N_3e^{-\alpha_3 r^2}, \tag{10.4}$$

where the coefficients b_i and exponents α_i are predefined values from Least-squares fitting o match the characteristics of a $1s$ Slater-type orbital as closely as possible [16]. Note that Y_{lm} is omitted for the s-orbital. Table 10.2 lists a few selected atomic orbitals.

More parameters can be found at `https://www.basissetexchange.org`. These basis sets are commonly used for smaller molecules and are considered minimal basis sets, meaning they use the minimum number of orbitals required to represent each electron in the system. Fig. 10.2 shows the comparison between different STO-nG basis set in reproducing the original hydrogen's $1s$ orbital. Clearly, the more Gaussian functions (higher n) used, the better the approximation to the original STO, but at a higher computational cost.

Table 10.2: STO-3G Basis Set Parameters for selected atomic orbitals.

	H (1s)	**He** (1s)	**C** (1s)	**C** (2s)	**C** (2p)
α_1	3.425250914	6.362421394	71.61683735	2.941249355	2.941249355
α_2	0.6239137298	1.158922999	13.04509632	0.6834830964	0.6834830964
α_3	0.1688554040	0.3136497915	3.530512160	0.2222899159	0.2222899159
b_1	0.1543289673	0.1543289673	0.1543289673	-0.09996722919	0.1559162750
b_2	0.5353281423	0.5353281423	0.5353281423	0.3995128261	0.6076837186
b_3	0.4446345422	0.4446345422	0.4446345422	0.7001154689	0.3919573931

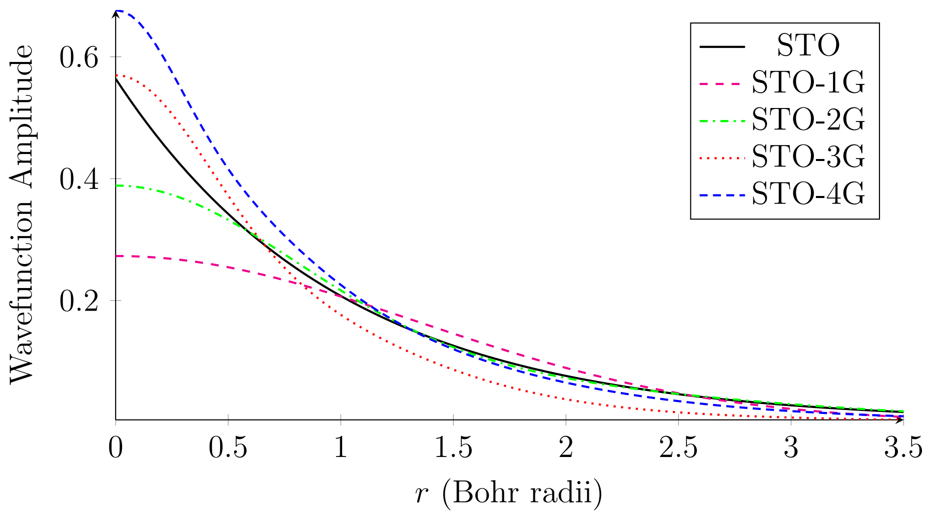


Figure 10.2: Comparison of STO-nG approximations for the hydrogen 1s orbital.

10.3. Other flavors of Basis Sets

In addition to STO-nG, there also exists other flavors of Basis Sets. Among them, the most popular one is Pople Basis Sets, which are named by specific conventions. They include,

- Split-Valence Basis Sets: They split the core and valence orbitals. For instance, 3-21G uses three Gaussians for core orbitals, and a combination of two and one Gaussian for the valence orbitals.
- Polarized Basis Sets: Include additional functions to account for electron correlation. For example, 6-31G(d) adds a d-type polarization function to improve flexibility.
- Diffuse Functions: Add extra Gaussians with low exponents for loosely bound electrons (like anions or excited states), denoted as 6-31+G or 6-31++G.

10.4. Mathematical Properties of Gaussians

The choice of Gaussian is mainly motivated by several key properties of Gaussian function, including

- The integral of Gaussian can be analytically derived.
- The derivative of Gaussian remains a Gaussian
- The product of two Gaussian functions remains Gaussian.

Before the actual physical discussion, let us first review the integral calculation from mathematical perspective.

10.4.1 Integral of Single Gaussian

For a Gaussian function in one dimension is typically written as:

$$f(x) = Ae^{-\alpha x^2}$$

The integral is:

$$\int_{-\infty}^{\infty} Ae^{-\alpha x^2} dx = A\sqrt{\frac{\pi}{\alpha}}$$

Extend it to n-dimension space $\mathbb{R}^n$, the integral is

$$\int_{\mathbb{R}^n} Ae^{-\alpha|\mathbf{r}|^2} d^n\mathbf{r} = A\left(\frac{\pi}{\alpha}\right)^{n/2} \tag{10.5}$$

Another important integral is the **error function**, often denoted as erf(x), is a mathematical function used to measure the probability of a value falling within a certain range of a normal distribution (Gaussian distribution). The error function is defined as:

$$\text{erf}(x) = \frac{2}{\sqrt{\pi}} \int_0^x e^{-t^2}\, dt \tag{10.6}$$

10.4.2 Integral of Gaussian Product

Consider two Gaussian functions centered at different points $\mathbf{R}_1$ and $\mathbf{R}_2$:

$$g_1(\mathbf{r}) = e^{-\alpha_1|\mathbf{r}-\mathbf{R}_1|^2}, \quad g_2(\mathbf{r}) = e^{-\alpha_2|\mathbf{r}-\mathbf{R}_2|^2}$$

The product of these two functions is:

$$\begin{aligned} g_1(\mathbf{r})g_2(\mathbf{r}) &= e^{-\alpha_1|\mathbf{r}-\mathbf{R}_1|^2} e^{-\alpha_2|\mathbf{r}-\mathbf{R}_2|^2} \\ &= e^{-\left(\alpha_1|\mathbf{r}-\mathbf{R}_1|^2+\alpha_2|\mathbf{r}-\mathbf{R}_2|^2\right)}. \end{aligned}$$

Expand the terms inside the exponent:

$$|\mathbf{r}-\mathbf{R}_1|^2 = (\mathbf{r}-\mathbf{R}_1)\cdot(\mathbf{r}-\mathbf{R}_1) = |\mathbf{r}|^2 - 2\mathbf{r}\cdot\mathbf{R}_1 + |\mathbf{R}_1|^2,$$

$$|\mathbf{r}-\mathbf{R}_2|^2 = (\mathbf{r}-\mathbf{R}_2)\cdot(\mathbf{r}-\mathbf{R}_2) = |\mathbf{r}|^2 - 2\mathbf{r}\cdot\mathbf{R}_2 + |\mathbf{R}_2|^2,$$

Substituting them to the product

$$\begin{aligned} g_1(\mathbf{r})g_2(\mathbf{r}) &= \exp\left[-\alpha_1|\mathbf{r}|^2 + 2\alpha_1\mathbf{r}\cdot\mathbf{R}_1 - \alpha_1|\mathbf{R}_1|^2 - \alpha_2|\mathbf{r}|^2 + 2\alpha_2\mathbf{r}\cdot\mathbf{R}_2 - \alpha_2|\mathbf{R}_2|^2\right] \\ &= \exp\left[-(\alpha_1+\alpha_2)|\mathbf{r}|^2 + 2(\alpha_1\mathbf{R}_1+\alpha_2\mathbf{R}_2)\cdot\mathbf{r} - \alpha_1|\mathbf{R}_1|^2 - \alpha_2|\mathbf{R}_2|^2\right]. \end{aligned}$$

The exponent can be rewritten in a form that shows the resulting function is Gaussian:

$$g_1(\mathbf{r})g_2(\mathbf{r}) = \exp\left[-\gamma\,|\mathbf{r}-\mathbf{P}|^2\right] \times \exp\left[\frac{-\alpha_1\alpha_2}{\gamma}|\mathbf{R}_1-\mathbf{R}_2|^2\right], \tag{10.7}$$

where

$$\gamma = \alpha_1 + \alpha_2$$
$$\mathbf{P} = \frac{\alpha_1\mathbf{R}_1 + \alpha_2\mathbf{R}_2}{\gamma}$$

Hence, the product can be viewed as a Gaussian centered at $\mathbf{P}$ with an exponent of γ and a constant factor dependent only on the relative positions of $\mathbf{R}_1$ and $\mathbf{R}_2$. The constant factor accounts for the overlap between the original Gaussian. Note that notations of γ and $\mathbf{P}$ are also extensively used in literature and computer codes for handling such integrals. This is concept can be graphically shown below in Fig. 10.3.

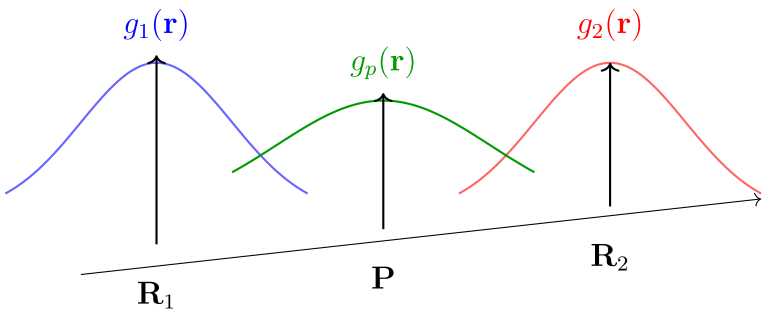


Figure 10.3: Schematic illustration of two Gaussian products.

10.4.3 Inverse R integral and the Boys Function

Following the definition of P and γ in eq. 10.7, another common integral is

$$\int \frac{e^{-\gamma|\mathbf{r}-\mathbf{P}|^2}}{|\mathbf{r}-\mathbf{R}_A|}\,d^3\mathbf{r} \tag{10.8}$$

To simplify the integral, shift the coordinate system so that the Gaussian center $\mathbf{P}$ becomes the origin:

$$\mathbf{r}' = \mathbf{r} - \mathbf{P}.$$

In this new coordinate system

$$|\mathbf{r}-\mathbf{P}|^2 = |\mathbf{r}'|^2, \quad |\mathbf{r}-\mathbf{R}_A| = |\mathbf{r}'-(\mathbf{R}_A-\mathbf{P})| = |\mathbf{r}'-\mathbf{R}'_A|$$

The integral becomes:

$$\int \frac{e^{-\gamma|\mathbf{r}'|^2}}{|\mathbf{r}'-\mathbf{R}'_A|}d^3\mathbf{r}'.$$

This integral is a standard form in quantum chemistry and can be evaluated analytically using the Boys function. The result is:

$$\int \frac{e^{-\gamma|\mathbf{r}\prime|^2}}{|\mathbf{r}'-\mathbf{R}'_A|}\,d^3\mathbf{r}' = \frac{2\pi}{\gamma}\,\mathrm{erf}\left(\sqrt{\gamma}|\mathbf{R}'_A|\right)e^{-\gamma|\mathbf{R}'_A|^2}.$$

Substitute $\mathbf{R}'_A = \mathbf{R}_A - \mathbf{P}$ and $\gamma = \alpha_1 + \alpha_2$. The full integral becomes:

$$I = e^{-\frac{\alpha_1\alpha_2}{\gamma}|\mathbf{R}_1-\mathbf{R}_2|^2} \cdot \frac{2\pi}{\gamma} \operatorname{erf}\left(\sqrt{\gamma}|\mathbf{R}_A - \mathbf{P}|\right) e^{-\gamma|\mathbf{R}_A-\mathbf{P}|^2}. \tag{10.9}$$

The general form of **Boys function** is

$$F_n(T) = \int_0^1 x^{2n} e^{-Tx^2}\, dx,$$

When $n = 0$

$$F_0(T) = \frac{\sqrt{\pi}}{2}\frac{\operatorname{erf}(\sqrt{T})}{\sqrt{T}}. \tag{10.10}$$

where

$$T = \gamma|\mathbf{P} - \mathbf{R}_A|^2.$$

10.4.4 Two-electron Integral

Finally, one needs to deal with the so called two-electron integral in order to compute the Hartree potential (i.e., the classical Coulomb interactions between electrons),

$$J_{ik,jl} = \iint \frac{g_i(\mathbf{r}_1;\alpha_i,\mathbf{R}_i)g_k(\mathbf{r}_2;\alpha_k,\mathbf{R}_k)g_j(\mathbf{r}_1;\alpha_j,\mathbf{R}_j)g_l(\mathbf{r}_2;\alpha_l,\mathbf{R}_l)}{|\mathbf{r}_1 - \mathbf{r}_2|} d^3\mathbf{r}_1\, d^3\mathbf{r}_2.$$

This integral involves four Gaussians over the $\mathbf{r}_1$ and $\mathbf{r}_2$ space.

Using the Gaussian product theorem, we can write it as the two new Gaussians g_p and g_q.

$$g_i(\alpha_i,\mathbf{R}_i)g_k(\alpha_k,\mathbf{R}_k) = g_P(\gamma_p,\mathbf{R}_p)e^{\frac{-\alpha_i\alpha_k}{\gamma_p}|\mathbf{R}_i-\mathbf{R}_k|^2}$$
$$g_j(\alpha_j,\mathbf{R}_j)g_l(\alpha_l,\mathbf{R}_l) = g_Q(\gamma_q,\mathbf{R}_q)e^{\frac{-\alpha_j\alpha_l}{\gamma_q}|\mathbf{R}_j-\mathbf{R}_l|^2}$$

where,

$$\mathbf{R}_p = \frac{\alpha_i\mathbf{R}_i + \alpha_k\mathbf{R}_k}{\alpha_i\alpha_k}, \quad \gamma_p = \alpha_i + \alpha_k,$$
$$\mathbf{R}_q = \frac{\alpha_j\mathbf{R}_j + \alpha_l\mathbf{R}_l}{\alpha_j\alpha_l}, \quad \gamma_q = \alpha_j + \alpha_l$$

Let us denote the following factor $\mathcal{M}$ that is independent of $\mathbf{r}_1$ and $\mathbf{r}_2$,

$$\mathcal{M} = e^{\frac{-\alpha_i\alpha_k}{\gamma_p}|\mathbf{R}_i-\mathbf{R}_k|^2} e^{\frac{-\alpha_j\alpha_l}{\gamma_q}|\mathbf{R}_j-\mathbf{R}_l|^2}$$

The integral becomes

$$J_{ik,jl} = \mathcal{M}\iint \frac{e^{-\gamma_p|r_1-\mathbf{R}_p|^2}e^{-\gamma_q|r_2-\mathbf{R}_q|^2}}{|r_1 - r_2|} d^3\mathbf{r}_1 d^3\mathbf{r}_2$$

Obviously, g_p and g_q can be merged with another Gaussian, and then we can solve the inverse-R integral again. So the final expression becomes

$$J_{ik,jl} = \mathcal{M}\frac{2\pi^{5/2}}{\gamma_p\gamma_q(\gamma_p+\gamma_q)^{1/2}} F_0\left(\frac{\gamma_p\gamma_q}{\gamma_p+\gamma_q}|\mathbf{R}_p - \mathbf{R}_q|^2\right) \tag{10.11}$$

These analytical results will be used heavily in the subsequent integral calculation, thus making GTOs computationally feasible and attractive.

10.5. Solving the Hydrogen Molecule with STO-3G

To illustrate the power of the basis set, let us revisit the problem of H_2 molecule with the use of STO-3G basis set as follows.

10.5.1 STO-3G basis for a H_2 molecule

For each hydrogen atom, the STO-3G basis set provides a single basis function to represent the 1*s* orbital. For each 1*s* orbital, STO-3G approximates the orbital as a combination of three Gaussian functions. This means that for the entire H_2 molecule, we have two basis functions (ϕ_1 for the 1st hydrogen and ϕ_2 for the 2nd hydrogen), each represented by three Gaussian functions. The basis functions are centered on the nuclei of the hydrogen atoms, which are located at positions $\mathbf{R}_1$ and $\mathbf{R}_2$.

In the STO-3G basis, we first express ϕ_1 and ϕ_2 as Gaussian expansions: Each STO basis function in STO-3G is represented by three Gaussian primitives:

$$\phi_1(\mathbf{r}) = \sum_{n=1}^{3} b_n |\mathbf{r} - \mathbf{R}_1|^{n-1} e^{-\alpha_k |\mathbf{r}-\mathbf{R}_1|^2} \tag{10.12}$$

$$\phi_2(\mathbf{r}) = \sum_{n=1}^{3} b_n |\mathbf{r} - \mathbf{R}_2|^{n-1} e^{-\alpha_k |\mathbf{r}-\mathbf{R}_2|^2} \tag{10.13}$$

To solve for the molecular orbitals of H_2, we form a linear combination of the two atomic basis functions:

$$\psi_k = \sum_{i=1}^{2} c_{ik}\phi_i \tag{10.14}$$

By solving the electronic Schrödinger equation on this basis, we compute the coefficients C (c_{ik}), which describe the molecular orbitals as combinations of ϕ_1 and ϕ_2.

10.5.2 Density Matrix and Electron Density

To compute the electron density $\rho(\mathbf{r})$ from the STO-3G basis set, one first obtains the density matrix D, which is defined in terms of these coefficients and gives the electron density in the basis of atomic orbitals. Each element D_{ij} of the density matrix is computed as:

$$D_{ij} = \sum_k f_k \, c_{ik} \, c^*_{jk} \tag{10.15}$$

where f_k is the occupation number (usually 1 for occupied orbitals and 0 for unoccupied ones in the Kohn–Sham approach). D has the same dimension of the coefficient matrix C.

The electron density $\rho(\mathbf{r})$ is then a weighted sum of the products of basis functions, where the weights are given by the elements of the density matrix D:

$$\rho(\mathbf{r}) = \sum_{i,j} D_{ij}\phi_i(\mathbf{r}) \, \phi_j(\mathbf{r}) \tag{10.16}$$

For an initial guess of H_2, we can assume that D is an identity matrix.

10.5.3 Overlap Matrix

The atomic orbitals (especially when approximated by Gaussian functions) are not generally orthogonal to each other. For two Gaussian functions centered at different positions $\mathbf{R}_i$ and $\mathbf{R}_j$, the overlap integral S is given by:

$$S(g_{ik}, g_{jl}) = \int g_{ik}(\mathbf{r}) g_{jl}(\mathbf{r}) d^3\mathbf{r}$$

Combining eq. 10.5 and eq. 10.7, the integral can be analytically computed as:

$$S(g_{ik}, g_{jl}) = b_{ik} b_{jl} \left(\frac{2\sqrt{\alpha_{ik}\alpha_{jl}}}{\alpha_{ik} + \alpha_{jl}} \right)^{3/2} e^{-\frac{\alpha_{ik}\alpha_{jl}}{\alpha_{ik}+\alpha_{jl}} |\mathbf{R}_i - \mathbf{R}_j|^2} \tag{10.17}$$

Here, α_{ik} and α_{jl} are the exponents of the Gaussian functions g_{ik} and g_{jl}.

10.5.4 Kinetic Energy

For a system consisting of N atomic orbitals, we expect to solve a $N \times N$ Hamiltonian matrix. For the case of H_2, this should result in a 2×2 square matrix. We first consider the kinetic energy T on the basis of the chosen atomic orbitals. For a basis function pair ϕ_i and ϕ_j, the kinetic energy matrix element in the atomic unit is given by:

$$T_{ij} = -\frac{1}{2} \int \phi_i(\mathbf{r}) \nabla^2 \phi_j(\mathbf{r}) \, d\mathbf{r}$$

To compute this, one needs to calculate a set of integrals like

$$T(g_{ik}, g_{jl}) = \int g_{ik}(\mathbf{r}) \nabla^2 g_{jl}(\mathbf{r}) d\mathbf{r}$$

The Laplacian of a Gaussian function yields another Gaussian:

$$\nabla^2 \left(e^{-\alpha r^2} \right) = (4\alpha^2 r^2 - 2\alpha) e^{-\alpha r^2}$$

Hence, we expect to arrive an analytic solution similar to the overlap integral S.

$$\begin{aligned} T(g_{ik}, g_{jl}) &= -\frac{1}{2} \int b_{ik} e^{-\alpha_{ik}|\mathbf{r}-\mathbf{R}_i|^2} \left(4\alpha_{jl}^2 |\mathbf{r} - \mathbf{R}_j|^2 - 2\alpha_{jl}\right) b_{jl} e^{-\alpha_{jl}|\mathbf{r}-\mathbf{R}_j|^2} \, d\mathbf{r} \\ &= \frac{\alpha_{ik}\alpha_{jl}}{\alpha_{ik} + \alpha_{jl}} \left(3 - \frac{2\alpha_{ik}\alpha_{jl}}{\alpha_{ik} + \alpha_{jl}} |\mathbf{R}_i - \mathbf{R}_j|^2 \right) S(g_{ik}, g_{jl}) \end{aligned} \tag{10.18}$$

Finally, the total is the sum of each combination of primitives:

$$T_{ij} = \sum_{k=1}^{3} \sum_{l=1}^{3} d_{ik} d_{jl} T(g_{ik}, g_{jl}) \tag{10.19}$$

10.5.5 External Potential

For the H_2 molecule, $V_{\text{external}}(\mathbf{r})$ at any point $\mathbf{r}$ due to nuclei at $\mathbf{R}_1$ and $\mathbf{R}_2$ would be:

$$V_{\text{external}}(\mathbf{r}) = -\frac{1}{|\mathbf{r}-\mathbf{R}_1|} - \frac{1}{|\mathbf{r}-\mathbf{R}_2|}$$

On the basis set space, this potential needs to be evaluated by

$$\langle \phi_i | V_{\text{external}} | \phi_j \rangle = \sum_{k=1}^{3}\sum_{l=1}^{3} d_{ik} d_{jl} \langle g_{ik} | V_{\text{external}} | g_{jl} \rangle \tag{10.20}$$

For each pair Gaussian integral,

$$\langle g_{ik} | V_{\text{external}}(\mathbf{r}) | g_{jl} \rangle = \int g_{ik}(\mathbf{r}) V_{\text{external}}(\mathbf{r}) g_{jl}(\mathbf{r})\, d^3\mathbf{r}$$

Thus, the problem reduces to evaluating the electron-nuclear attraction integral for each nucleus:

$$I_{\text{nuclear}} = \sum_{m=1}^{2} \int \frac{g_{ik}(\mathbf{r}) g_{jl}(\mathbf{r})}{|\mathbf{r}-\mathbf{R}_m|}\, d^3\mathbf{r}$$

Using eq. 10.7, we can transform $g_{ik}g_{jl}$ to a new Gaussian centered at $e^{-\gamma|\mathbf{r}-\mathbf{P}|^2}$,

$$I_{\text{nuclear, }m} = \int e^{-\gamma|\mathbf{r}-\mathbf{P}|^2} \frac{1}{|\mathbf{r}-\mathbf{R}_m|} d^3\mathbf{r},$$

The final expression is

$$V_{\text{external}}(g_{ik}, g_{jl}) = \sum_{m=1}^{2} \frac{-2\pi}{\gamma} e^{-\frac{\alpha_{ik}\alpha_{jl}}{\gamma}|\mathbf{R}_i - \mathbf{R}_j|^2} F_0\left(\gamma |\mathbf{R}_P - \mathbf{R}_m|^2\right) \tag{10.21}$$

10.5.6 Hartree Potential

The Hartree potential matrix on the basis functions can be expressed as

$$V_{\text{Hartree}}(g_{ik}, g_{jl}) = \sum_{k,l} D_{kl} J_{ij,kl}.$$

For each ij component, it requires solving the following integral

$$J_{ij,kl} = \iint \frac{\phi_i(\mathbf{r}_1)\phi_k(\mathbf{r}_1)\phi_j(\mathbf{r}_2)\phi_l(\mathbf{r}_2)}{|\mathbf{r}_1 - \mathbf{r}_2|}\, d^3\mathbf{r}_1\, d^3\mathbf{r}_2.$$

Using eq. 10.11, we get the expression as

$$J_{ij,kl} = \sum_{p,q,r,s} b_{ip} b_{jq} b_{kr} b_{ls} \frac{(2\pi)^{5/2}\exp(T_1)}{\gamma_1\gamma_2\sqrt{\gamma_1+\gamma_2}} F_0(T_2). \tag{10.22}$$

where:

$$\gamma_1 = \alpha_{ip} + \alpha_{kr} \to \mathbf{P}_1 = \frac{\alpha_{ip}\mathbf{R}_{ip} + \alpha_{kr}\mathbf{R}_{kr}}{\gamma_1} \xrightarrow{\text{Prefactor}} T_1 = -\frac{\alpha_{ip}\alpha_{kr}}{\gamma_1}|\mathbf{R}_{ip} - \mathbf{R}_{kr}|^2 \to \exp(T_1)$$

$$\gamma_2 = \alpha_{jq} + \alpha_{ls} \to \mathbf{P}_2 = \frac{\alpha_{jq}\mathbf{R}_{jq} + \alpha_{ls}\mathbf{R}_{ls}}{\gamma_2} \xrightarrow{\text{Boys Func.}} T_2 = \frac{\gamma_1\gamma_2}{\gamma_1+\gamma_2}|\mathbf{P}_1 - \mathbf{P}_2|^2 \quad \to F_0(T_2).$$

10.5.7 XC potential

To evaluate V_{XC} matrix, we need to return to real space to compute $V_{\mathrm{XC}}(r)$ from the basis as follows:

$$V_{\mathrm{XC}}[i,j] = \int \phi_i(r)\, V_{\mathrm{XC}}(r)\phi_j(r) d^3r$$

We still assume a LDA exchange functional as used in the previous chapter,

$$V_{\mathrm{XC}}(\mathbf{r}) = -\frac{3\rho(\mathbf{r})}{\pi}^{1/3}$$

This integral represents the contribution of the XC potential in the (i,j)-th element of the matrix. For efficiency, it's typical to use numerical quadrature or Gaussian quadrature methods if the basis functions are Gaussian.

This integral is challenging to evaluate analytically. However, numerical techniques can be applied. One can sample points r_p and weights w_p in real space, then approximate the integral as a weighted sum:

$$V_{\mathrm{XC}}(g_{ik}, g_{jl}) \approx \sum_p w_p \phi_i(\mathbf{r}_p) \frac{-3\rho(\mathbf{r}_p)}{\pi}^{1/3} \phi_j(\mathbf{r}_p) \tag{10.23}$$

10.5.8 Orthogonalization and SCF

In a self-consistent field (SCF) calculation for density functional theory (DFT), the molecular orbital coefficients ($\mathbf{c}_{\mathrm{matrix}}$) are obtained by solving the Kohn-Sham equations. These equations involve constructing and diagonalizing the Hamiltonian matrix H, which is defined as:

$$H = T + V_{\mathrm{external}} + V_{\mathrm{Hartree}} + V_{\mathrm{XC}} \tag{10.24}$$

To ensure an orthogonal basis, the overlap matrix S between basis functions is used to transform the H matrix.

$$H\mathbf{c} = \epsilon S\mathbf{c}$$

To solve this generalized eigenvalue problem, the basis functions are orthogonalized using a transformation involving the overlap integral S. Specifically, the overlap matrix is diagonalized, and a transformation matrix $\mathbf{X}$ is applied:

$$\mathbf{X} = S^{-1/2} \tag{10.25}$$

The Hamiltonian is then transformed into an orthogonalized basis:

$$\tilde{H} = \mathbf{X}^\dagger H \mathbf{X}, \quad \tilde{S} = \mathbf{X}^\dagger S \mathbf{X} = \mathbf{I} \tag{10.26}$$

The eigenvalue problem is now simplified to:

$$\tilde{H}\mathbf{c} = \epsilon \mathbf{c} \tag{10.27}$$

This ensures that the molecular orbitals obtained are orthogonal. Solving the eigenvalue problem will give both the eigenvalues (energies) and eigenvectors (coefficients) of the molecular orbitals. The coefficients can be used to recompute density matrix (eq.

10.15), electron density (eq. 10.16, kinetic energy (eq. 10.19), external potential (eq. 10.20), Hartree potential (eq. 10.22) and XC potential (eq. 10.23) iteratively until it reaches a convergence.

Algorithm 3 DFT-SCF Algorithm using STO-3G

```
1: Step 1: Initial Guess. Start with an initial guess of $D$ and the given $\phi_i$.
2: repeat
3:     Step 2: Construct Hamiltonian.
         • Evaluate $T$, $V_{\text{external}}$, $V_H$, and $V_{XC}$ using the current $\rho(\mathbf{r})$.
         • Compute $H = T + V_{\text{external}} + V_H + V_{XC}$.
4:     Step 3: Solve the Kohn-Sham Equations.
         • Diagonalize $H$ (after orthogonalization with $S$).
         • Obtain molecular orbital coefficients $\mathbf{c}$ and eigenvalues $\epsilon$.
5:     Step 4: Recompute $D$ and $\rho(\mathbf{r})$.
6:     Step 5: Update $V_H$ and $V_{XC}$.
7:     Step 6: Check Convergence.
         • Evaluate the energy: $E_{\text{total}} = E_{\text{kinetic}} + E_{\text{NN}} + E_{\text{Hartree}} + E_{\text{XC}}$
         • Check if the change in $E_{\text{total}}$ or $\rho(\mathbf{r})$ is below a predefined threshold.
8: until Convergence is achieved.
```

10.6. PySCF Exercise in simulating H_2

There are many packages available for performing DFT simulations. The PySCF package is a particularly useful tool to implement such calculations due to its flexibility and Python-based interface. PySCF is a powerful computational chemistry library designed for quantum chemistry and materials science, providing efficient tools for ab initio calculations, including Hartree-Fock, post-Hartree-Fock, and DFT methods. Below is an example to simulate H_2 based on PySCF.

```python
from pyscf import gto, dft
import numpy as np

# Calculation setup
mol = gto.Mole()
mol.atom = 'H 0 0 0; H 0 0 0.74'     # Create H2
mol.basis = 'sto-3g'                 # Define the basis
mol.build()
mf = dft.RKS(mol)                    # Setup DFT
mf.xc = 'lda'                        # Set up XC functional
mf.kernel()                          # Execute calculation

# Integrals from the initial guess of density matrix
dm = np.eye(2)
print("\nOverlap\n", mf.get_ovlp())
print("\nV_kinetic\n", mf.mol.intor("int1e_kin"))
print("\nExternal\n", mf.mol.intor("int1e_nuc"))
print("\nV_Hartree\n", mf.get_j(mol, dm))
print("\nV_XC\n", mf.get_veff(mol, dm) - mf.get_j(mol, dm))
```

```
#print("hcore\n", mf.get_hcore())
#print("V_eff\n", mf.get_veff(mol, dm))

# XC details regarding grid points and weights
#coords = mf.grids.coords
#weights = mf.grids.weights
```

Compared to other languages, one important advantage of Python is that one can conveniently output the intermediate variables. For example, the above script computes various integral terms related to the molecular Hamiltonian, such as overlap, kinetic energy, nuclear attraction, Hartree, and exchange-correlation potentials.

```
converged SCF energy = -1.02500812637085

Overlap
 [[1.         0.65987312]
 [0.65987312 1.        ]]

V_kinetic
 [[0.76003188 0.23696027]
 [0.23696027 0.76003188]]

External
 [[-1.88099134 -1.19633604]
 [-1.19633604 -1.88099134]]

V_Hartree
 [[1.34460083 0.88918225]
 [0.88918225 1.34460083]]

V_XC
 [[-0.39170033 -0.25292459]
 [-0.25292459 -0.39170033]]
```

These values can be useful understand the overall structure of each potential term and the simulation process. In addition, we will use these values as the reference to check our own Python code implementation in the following section. For different kinds of calculation setup and data analysis in using PySCF, please refer to its online tutorials.

10.7. Python Code Implementation from the Scratch

While most researchers mostly rely on the existing package to perform electronic structure calculations for realistic systems, it remains tremendously valuable to implement a small piece of code by hand if one aims to gain more first hand experiences. In the following, we will attempt to set up a Python script for this chapter.

10.7.1 Initial Planning

For DFT calculations with local basis sets, the coding requirements are significantly more complex than any other codes we have discussed thus far. To address this complexity, it is crucial to adopt a structured approach to manage a larger and more intricate coding project effectively.

First, a logical starting point is to modularize the code by separating the handling of GTOs and their associated integrals into distinct modules. This separation is especially useful because these calculations are mathematically intensive and require a clean and reusable design. Within Python's framework, such objectives can be elegantly achieved using **classes**. In the context of this discussion, a **GTO Class** to manage the construction of Gaussian basis functions, normalization, and various integral calculations (e.g., overlap, kinetic, and nuclear attraction integrals) would be desirable. By encapsulating these functionalities, we can create a reusable and modular design for the mathematical backbone of the DFT code. If the goal is to handle various types of molecules as input, it is also beneficial to design a separate **Molecule** class. This class can manage molecular data such as atomic coordinates, atomic numbers, and connectivity, providing a clean interface for interacting with different molecular systems.

Second, careful planning is necessary to validate each numerical routine and ensure that the code produces the desired results for a few well-known systems. Outputs from established libraries like `PySCF` can serve as excellent benchmarks for this purpose, helping to verify the correctness and accuracy of the implementation.

Third, sometimes efficiency matters. Python, like many other languages, cannot run nested loops efficiently. It is recommended to first write the plain version of code, and then vectorize if needed.

Finally, it is likely that many code components will need to be written in a way that supports extensibility and modularity. This approach will not only facilitate debugging and testing but also make the codebase more adaptable to future enhancements or changes.

10.7.2 The GTO class

To start, we first write a GTO class to handle Gaussian basis functions and some relatively easy integral calculations.

```python
import numpy as np
import scipy.special as sp

def boys_function(T, threshold=1e-4):
    """
    Compute the Boys function F_0(T) for given values of T.

    Args:
        T (np.ndarray): Input parameter T (non-negative).

    Returns:
        np.ndarray: Boys function F_0(T).
    """
    F0 = np.zeros_like(T)

    # For small T, use Taylor expansion
    st = T < threshold
    if np.any(st):
        t_ = T[st]
        F0[st] = 1 - t / 3 + t**2 / 10 - t**3 / 42 + t**4 / 108

    # For larger T, use the error function
    lt = ~small_t
    if np.any(lt):
        t = T[lt]
        F0[lt] = 0.5 * np.sqrt(np.pi / t) * sf.erf(np.sqrt(t))

    return F0

class GTO:
    """
    Gaussian-Type Orbital (GTO) Class
    Handles GTO basis sets generation and integrals.
    """

    def __init__(self, alpha, coeff, center):
        """
        Initialize a GTO.

        Args:
            alpha (list): Exponents of the Gaussian primitives.
            coeff (list): Coefficients of the Gaussian primitives.
            center (array-like): Coordinates of the GTO center.
        """
        self.alpha = np.array(alpha)
        self.coeff = np.array(coeff)
        self.center = np.array(center)
        self.norms = self.compute_norms()

    def compute_norms(self):
        """
        Compute normalization constants for the Gaussian primitives.

        Returns:
        List of normalization constants.
        """
        norm_constants = []
        for alpha in self.alpha:
            N = (2 * alpha / np.pi) ** (3 / 4)
            norm_constants.append(N)
        return np.array(norm_constants)

    def evaluate(self, r):
        """
        Evaluate the GTO at a given position r.

        Args:
            r (array-like): Position where the GTO is evaluated.

        Returns:
            float: Value of the GTO at r.
        """
        r = np.array(r)
        r_diff = np.linalg.norm(r - self.center)
        value = 0.0
        for c, a, norm in zip(self.coeff, self.alpha, self.norms):
            value += c * norm * np.exp(-a * r_diff**2)
        return value

    def overlap_integral(self, g2, return_matrix=False):
        """
        Compute the overlap integral between this and another GTO.

        Args:
            g2 (GTO): Another GTO object.
            return_matrix (bool): return matrix for kinetic integral

        Returns:
            float: Overlap integral.
        """
        g1 = self
        p = g1.alpha[:, None] + g2.alpha[None, :]
        q = g1.alpha[:, None] * g2.alpha[None, :]
        R_diff = np.linalg.norm(g1.center - g2.center)**2
        coefs = np.outer(g1.coeff, g2.coeff)
        S = (2 * np.sqrt(q) / p)**(3 / 2) * np.exp(-q/p * R_diff)
        if return_matrix:
            return S * coefs
        else:
            return np.sum(S * coefs)

    def kinetic_integral(self, g2):
        """
        Compute kinetic energy integral between this and another GTO.

        Args:
            g2 (GTO): Another GTO object.

        Returns:
            float: Kinetic energy integral.
        """
        g1 = self
        S = self.overlap_integral(g2, return_matrix=True)
        p = g1.alpha[:, None] + g2.alpha[None, :]
        q = g1.alpha[:, None] * g2.alpha[None, :] / p
        R_diff = np.linalg.norm(g1.center - g2.center)**2
        coef = q * (3 - 2 * q * R_diff)
        return np.sum(S * coef)

    def external_integral(self, g2, RA, Z=1.0):
        """
        Compute the nuclear attraction integral.

        Args:
            g2 (GTO): Another GTO object.
            RA (array-like): Position of the nucleus.
            Z (float): Nuclear charge.

        Returns:
            float: Nuclear attraction integral.
        """
        g1 = self
        RA = np.array(RA)
        R_diff = np.linalg.norm(g1.center - g2.center)
        p = g1.alpha[:, None] + g2.alpha[None, :]
        q = g1.alpha[:, None] * g2.alpha[None, :] / p

        # Broadcast verion of new Gaussian centers
        g1s = g1.alpha[:, None] * g1.center[None, :]
        g2s = g2.alpha[:, None] * g2.center[None, :]
        P = (g1s[None, :, :] + g2s[:, None, :]) / p[:, :, None]

        # Plain Python implemenation
        #P = np.zeros([len(g1.alpha), len(g2.alpha), 3])
        #for i in range(len(g1.alpha)):
        #    for j in range(len(g2.alpha)):
        #        P[i, j, :] = g1.alpha[i] * g1.center
        #        P[i, j, :] += g2.alpha[j] * g2.center
        #        P[i, j, :] /= p[i, j]
        #        P[i, j, :] -= RA

        F0_t = boys_function(p * np.linalg.norm(P - RA, axis=-1)**2)
        V = 2 * np.pi / p * np.exp(-q * R_diff**2) * F0_t
        coefs = np.outer(g1.norms * g1.coeff, g2.norms * g2.coeff)

        return -Z * np.sum(V * coefs)


if __name__ == "__main__":
    gtos = [
        GTO(alpha=[3.42525091, 0.62391373, 0.16885540],
            coeff=[0.15432897, 0.53532814, 0.44463454],
            center=[0.0, 0.0, 0.00000000]),
        GTO(alpha=[3.42525091, 0.62391373, 0.16885540],
            coeff=[0.15432897, 0.53532814, 0.44463454],
            center=[0.0, 0.0, 1.39839733])
       ]

    # Compute integrals
    S = np.zeros([2, 2])
    T = np.zeros([2, 2])
    V_ext = np.zeros([2, 2])
    for i in range(2):
        for j in range(2):
            S[i, j] = gtos[i].overlap_integral(gtos[j])
            T[i, j] = gtos[i].kinetic_integral(gtos[j])
            for k in range(2):
                term = gtos[i].V_ext_integral(gtos[j], gtos[k].center)
                V_ext[i, j] += term

    print(f"Overlap:\n {S}")
    print(f"Kinetic:\n {T}")
    print(f"Nuclear:\n {V_ext}")
```

The GTO class consists the following:

GTO attributes.

- *alpha*: List of exponents for Gaussian primitives.
- *coeff*: List of coefficients for the linear combination of these primitives.
- *center*: The 3D spatial center of the Gaussian orbital.
- *norms*: Normalization constants for the Gaussian primitives.

GTO methods.

- **compute_norms**: Calculates normalization constants for each Gaussian.
- **evaluate**: Evaluates the GTO at a specific spatial point r.
- **overlap_integral**: computes the overlap integral from eq. 10.17
- **kinetic_integral**: computes the kinetic energy integral from eq. 10.19
- **external_integral**: Computes the the external potential from eq. 10.20

In the main routine, we construct two GTO instances use the same parameters for α and coefficients but are centered at H atom positions in a H_2 molecule. Then, we call the GTO methods to compute S, T, V_{external} and obtain the following results.

```
Overlap:
 [[0.99999999 0.65987312]
 [0.65987312 0.99999999]]
Kinetic:
 [[0.76003188 0.23696026]
 [0.23696026 0.76003188]]
Nuclear:
 [[-1.88099132 -1.19633603]
 [-1.19633603 -1.88099132]]
```

Clearly, the results are consistent with the previous PySCF results, thus encouraging us to continue to build the rest functions. One may also check the commented parts in **external_integral** method, which shows a thought process regarding how to prototype the beginning code with nested for loops and then optimize it with the broadcast approach with the help of Numpy.

10.7.3 Hartree Potential

As discussed in the previous section, the most expensive calculation to compute the Hartree potential that involves the evaluation of two-electron integrals. Using eq. 10.22, we created the following two functions.

```python
def v_hartree(gtos, density_matrix):
    """
    Compute the Hartree potential matrix using the Gaussian basis set.

    Args:
        gtos (list of GTO): List of Gaussian basis set.
        density_matrix (np.ndarray): Density matrix (D) of the system.

    Returns:
        np.ndarray: Hartree potential matrix.
    """
    n_basis = len(gtos)
    V_h = np.zeros((n_basis, n_basis))

    # Compute the Hartree potential matrix
    for i, g1 in enumerate(gtos):
        for j, g2 in enumerate(gtos):
            for k, g3 in enumerate(gtos):
                for l, g4 in enumerate(gtos):
                    # Compute the 2-electron integral
                    integral = compute_J_integral(gtos, i, k, j, l)
                    V_h[i, j] += density_matrix[k, l] * integral
    return V_h

def compute_J_integral(basis_set, i, j, k, l, verbose=False):
    """
    Compute the Coulomb integral J_{ij,kl} for the basis set.

    Args:
        basis_set (list of GTO): List of Gaussian basis functions.
        i, j, k, l (int): Indices of the basis functions.

    Returns:
        float: Coulomb integral J_{ij,kl}.
    """
    gi, gj = basis_set[i], basis_set[j]
    gk, gl = basis_set[k], basis_set[l]

    J = 0.0
    for ci, ai, ni in zip(gi.coeff, gi.alpha, gi.norms):
        ri = gi.center
        for ck, ak, nk in zip(gk.coeff, gk.alpha, gk.norms):
            rk = gk.center
            for cj, aj, nj in zip(gj.coeff, gj.alpha, gj.norms):
                rj = gj.center
                for cl, al, nl in zip(gl.coeff, gl.alpha, gl.norms):
                    rl = gl.center
                    # Compute combined exponents and centers
                    gamma_1 = ai + ak
                    gamma_2 = aj + al
                    c1 = (ai * ri + ak * rk) / gamma_1
                    c2 = (aj * rj + al * rl) / gamma_2
                    RAC = np.linalg.norm(ri - rk) ** 2
                    RBD = np.linalg.norm(rj - rl) ** 2
                    RPQ = np.linalg.norm(c1 - c2) ** 2

                    # Contribution to the Coulomb integral
                    prefactor = (
                    ci * ck * cj * cl * ni * nk * nj * nl *
                    (2 * np.pi**2.5) /
                    (gamma_1 * gamma_2 * np.sqrt(gamma_1 + gamma_2))
                    )
                    T1 = -ai * ak * RAC / gamma_1
                    T1 -= aj * al * RBD / gamma_2
                    T2 = gamma_1 * gamma_2 * RPQ / (gamma_1 + gamma_2)
                    boys_val = boys_functions(np.array([T2]))[0]

                    J += prefactor * np.exp(exp1 + exp2) * boys_val
    return J

dm = np.eye(2)
V_H = v_hartree(gtos, dm)
print(f"V_H:\n {V_H}")
```

The corresponding outputs are

```
External:
 [[1.3446008  0.88918223]
[0.88918223 1.3446008 ]]
```

Clearly, both functions require four nested for loops. This is fine for a simple case of H_2 molecule. For more complicated examples, one may need to consider more efficient code vectorization or the use of more efficient libraries.

10.7.4 LDA Exchange

The last term is to construct the effective potential in DFT is to treat the Exchange Correlation term. Here we follow eq. 10.23 to complete the following two functions.

```python
def lda_exchange_potential(rho):
    """
    Compute the LDA exchange potential for a given density rho.
    """
    # Avoid division by zero for very small densities
    rho = np.maximum(rho, 1e-10)
    V_X = -(3 / np.pi)**(1 / 3) * (rho**(1 / 3))
    return V_X

def compute_V_xc(gtos, dm, grid_points, grid_weights):
    """
    Compute the LDA XC potential matrix V_XC on the basis set.

    Args:
        gtos (list of GTO): List of Gaussian basis functions.
        dm (np.ndarray): Density matrix P_kl.
        grid_points (np.ndarray): Points for numerical integration.
        grid_weights (np.ndarray): Weights for numerical integration.

    Returns:
        np.ndarray: XC potential matrix V_XC.
    """
    n_basis = len(gtos)
    V_XC = np.zeros((n_basis, n_basis))

    # Loop over grid points
    for p, w in zip(grid_points, grid_weights):
        # Compute the electron density at the grid point
        rho = 0.0
        phi_kp = gtos[k].evaluate(p)
        phi_lp = gtos[l].evaluate(p)
        for k in range(n_basis):
            for l in range(n_basis):
                rho += dm[k, l] * phi_kp * phi_lp

        # Compute the XC potential at the grid point
        V_X = lda_exchange_potential(rho)

        # Compute contributions to V_XC matrix elements
        for i in range(n_basis):
            for j in range(n_basis):
                V_XC[i, j] += w * phi_kp * phi_lp * V_X

    return V_XC

from pyscf import gto
from pyscf.dft.gen_grid import Grids
```

```python
# Calculation setup
mol = gto.Mole()
mol.atom = 'H 0 0 0; H 0 0 0.74'
mol.build()
grids = Grids(mol)
grids.build()
coords = grids.coords
weights = grids.weights
dm = np.eye(2)
V_XC = compute_V_xc(gtos, dm, coords, weights)
print("XC Potential Matrix:\n", V_XC)
```

Executing it requires the setup of grid coordinates and weights. For convenience, here we just use the grid scheme from `PySCF` to test our function. It returns

```
XC Potential Matrix:
 [[-0.39170032 -0.25292459]
 [-0.25292459 -0.39170032]]
```

10.7.5 SCF class

Finally, let us implement the SCF process as follows.

```python
def scf(gtos, dm0, grid_coords, grid_weight, max_iter=100, tol=1e-6):
    """
    Main routine to perform self-consistent field (SCF) calculation.

    Args:
        gtos (list of GTO): List of Gaussian-type orbitals.
        dm0 (np.ndarray): Initial density matrix.
        grid_coords (np.ndarray): Grid points for numerical integration
    .
        grid_weights (np.ndarray): Weights for numerical integration.
        max_iter (int): Maximum number of SCF iterations.
        tol (float): Convergence threshold.

    Returns:
        dict: Contains energies, density matrix, and molecular orbitals
    .
    """
    from scipy.linalg import eigh

    # Initialize density matrix and energies
    dm = dm0
    energies = []

    # Precompute electron-independent integrals
    n_basis = len(gtos)
    S = np.zeros([n_basis, n_basis])
    T = np.zeros([n_basis, n_basis])
    V_ext = np.zeros([n_basis, n_basis])
    for i in range(n_basis):
        for j in range(n_basis):
            S[i, j] = gtos[i].overlap_integral(gtos[j])
            T[i, j] = gtos[i].kinetic_integral(gtos[j])
            for k in range(2):
                term = gtos[i].external_integral(gtos[j], gtos[k].
    center)
```

```
                V_ext[i, j] += term

    E_nuc = 1.0 / np.linalg.norm((gtos[0].center - gtos[1].center))

    for cycle in range(max_iter):
        # update T and v_eff
        V_hartree = compute_v_hartree(gtos, dm)
        V_xc = compute_v_xc(gtos, dm, grid_coords, grid_weight)
        V_eff = V_hartree + V_xc

        # update H
        H = T + V_ext + V_eff

        # solve H
        mo_energy, mo_coeff = eigh(H, S)

        # Update density matrix
        dm_new = np.zeros_like(dm)
        for i in range(n_basis):
            for j in range(n_basis):
                for k in range(n_basis):
                    if mo_energy[k] < 0:
                        dm_new[i, j] += mo_coeff[i, k] * mo_coeff[j, k]
        dm_new *= 2

        # Compute energy components
        T_energy = np.einsum("ij,ij", dm.real, T)
        E_ext = np.einsum("ij,ij", dm.real, V_ext)
        E_hartree = 0.5 * np.einsum("ij,ij", dm.real, V_hartree)
        E_xc = np.einsum("ij, ij", dm.real, V_xc)
        E_total = T_energy + E_ext + E_hartree + E_xc + E_nuc

        # Output energy details
        print(f"Cycle {cycle + 1}:")
        print(f"  Kinetic Energy (T):      {T_energy:.6f}")
        print(f"  External Energy (E_ext): {E_ext:.6f}")
        print(f"  Hartree Energy (E_H):    {E_hartree:.6f}")
        print(f"  XC Energy (E_XC):        {E_xc:.6f}")
        print(f"  Total Energy (E_tot):    {E_total:.6f}")
        print(f"  HOMO Energy (KS_energy): {mo_energy.min():.6f}\n")

        # Check for convergence
        if np.linalg.norm(dm_new - dm) < tol:
            print(f"SCF converged in {cycle + 1} iterations.")
            break
        else:
            dm = dm_new

if __name__ == "__main__":
    from pyscf import gto
    from pyscf.dft.gen_grid import Grids

    # Calculation setup
    mol = gto.Mole()
    mol.atom = "H 0 0 0; H 0 0 0.74"    # Create H2
    mol.build()
    grids = Grids(mol)
    grids.build()
    coords = grids.coords
    weights = grids.weights
    #print(coords)

    gtos = [
        GTO(alpha=[3.42525091, 0.62391373, 0.16885540],
            coeff=[0.15432897, 0.53532814, 0.44463454],
            center=[0.0, 0.0, 0.00000000]),
        GTO(alpha=[3.42525091, 0.62391373, 0.16885540],
            coeff=[0.15432897, 0.53532814, 0.44463454],
            center=[0.0, 0.0, 1.39839733])
      ]

    dm = np.eye(2)
    results = scf(gtos, dm, coords, weights)
```

The above SCF function iteratively solves the Kohn-Sham equations to minimize the total energy of the system and obtain the converged electron density matrix and energy values. The process begins by initializing the system with an initial guess for the density matrix (D), followed by the computation of all necessary integrals (overlap, kinetic, external potential, Hartree, and exchange-correlation). By iteratively refining the density matrix, the SCF loop converges to a stable solution, returning the final electron density, molecular orbital energies, and total energy. During each iteration, the energy components are computed and printed for monitoring.

The following results were obtained by running the SCF function:

```
Cycle 1:
  Kinetic Energy (T):       1.520064
  External Energy (E_ext): -3.761983
  Hartree Energy (E_H):     1.344601
  XC Energy (E_XC):         -0.783401
  Total Energy (E_tot):     -0.965614
  HOMO Energy (KS_energy): -0.295912

Cycle 2:
  Kinetic Energy (T):       1.201287
  External Energy (E_ext): -3.707907
  Hartree Energy (E_H):     1.349512
  XC Energy (E_XC):         -0.777340
  Total Energy (E_tot):     -1.219343
  HOMO Energy (KS_energy): -0.292468

SCF converged in 2 iterations.
```

Clearly, the final output energy values are significantly more accurate compared to results obtained using a numerical grid approach. This is because the use of GTOs ensures a more compact and efficient representation of the molecular orbitals compared to the finite-difference grid methods. In addition, the entire calculation completes in just a few seconds, demonstrating the computational efficiency of using Gaussian basis sets combined with a self-consistent field method.

10.8. Conclusions

In this chapter, we presented a detailed implementation of DFT using Gaussian-Type Orbitals (GTOs) as the basis set. The SCF algorithm iteratively solved the Kohn-Sham equations, starting with an initial guess for the density matrix and refining it until convergence. Key integrals—including overlap, kinetic, external potential, Hartree, and exchange-correlation—were thoroughly discussed and numerically implemented within the Gaussian framework.

Compared to grid-based numerical approaches, the use of GTOs significantly reduced computational cost while improving accuracy. This efficiency stems from the compact representation of molecular orbitals and the optimized computation of integrals in the Gaussian basis. These features underscore the advantages of GTO-based methods for quantum chemistry simulations.

This implementation serves as a foundational exercise in DFT, providing insights into the numerical and theoretical aspects of solving the Kohn-Sham equations. It also paves the way for exploring advanced topics such as higher-level exchange-correlation functionals (e.g., GGA or hybrid methods), multi-electron systems, and extended systems like periodic structures. Repeating and building upon this exercise will enable a deeper understanding of the core principles of DFT and its practical computational framework.

11. Electronic Structure of the Periodic System

Having known how to compute the electronic properties of molecule, it is natural to extend it to handle the crystal. However, it is not straightforward to extend DFT from molecule to crystals. Unlike molecules, crystals are periodic and extend infinitely. We need methods that respect this periodicity and allow us to work with finite-sized models.

11.1. The Bloch Theorem

Let's start with a simplest possible periodic system, i.e., a stacking of the same atoms in one dimension with a unit distance of a,

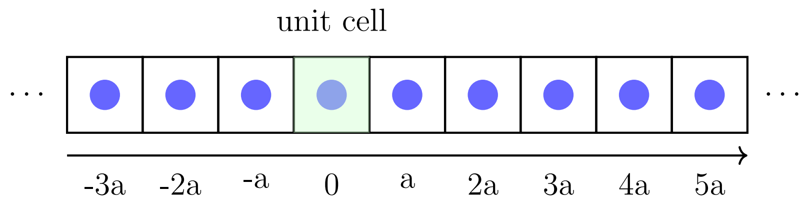


Figure 11.1: The schematic 1D crystal model with a unit distance of a.

The wavefunction must satisfy the time-independent Schrodinger equation:

$$H\psi(r) = E\psi(r) \tag{11.1}$$

Due to the periodic boundary condition, the Hamiltonian is periodic:

$$H(x+a) = H(x) \tag{11.2}$$

So the wave function must be somewhat periodic as well. The immediate solution would be $\psi(x+a) = \psi(x)$. However, quantum mechanics allows for a more flexible condition. Instead of requiring the wavefunction to be identical after shifting by a phase angle θ, quantum mechanics only requires that the probability density $|\psi(x)|^2$ remains periodic, as this is what directly relates to observable quantities.

$$|\psi(x+a)|^2 = |e^{i\theta}\psi(x)|^2 = |\psi(x)|^2 \tag{11.3}$$

Similarly, we can vary the length of periods to get

$$\begin{aligned}
\psi(x+2a) &= e^{i2\theta}\psi(x), \\
\psi(x+3a) &= e^{i3\theta}\psi(x), \\
&\vdots \\
\psi(x+na) &= e^{in\theta}\psi(x).
\end{aligned}$$

From these expressions, we can find that the wavefunction satisfy the above periodic constraints must satisfy the following relation,

$$\psi_k(x) = e^{ikx}u_k(x), \tag{11.4}$$

where

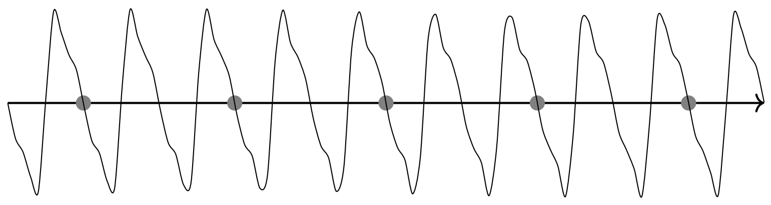
- k is an index to describe the periodic operations of applying the phase factor.
- e^{ikx} is a plane wave factor that describes how the wavefunction changes as you move through space,
- $u_k(x)$ is a function that has the same periodicity as the crystal lattice, meaning $u_k(x+a) = u_k(x)$,

This is called **Bloch theorem** in solid state physics, which states that for a particle in a periodic potential, the wavefunction can be expressed as a general wavefunction in the center unit, and the general wave function multiply a phase factor in the periodic unit cell.

With this relation, it is not difficult to find that $\psi_k(x+a)$ is also related to $\psi_k(x)$ via the phase factor e^{ika}

$$\psi_k(x+a) = e^{ik(x+a)}u_k(x+a) = e^{ika}e^{ikx}u_k(x) = e^{ika}\psi_k(x) \tag{11.5}$$

It suggests a picture like the following,



11.2. The Tight Binding Model

11.2.1 Solution of 1D monoatomic crystal

For this 1D model, we may assume that the wavefunction solution is the linear combination of localized atomic orbitals.

$$\psi_k(r) = \sum_i c_i \phi(r - R_i),$$

where c_i are the coefficients that describes the contribution of the i-th atomic orbital.

Plugging this into eq.11.1 and multiply $\phi^*(r-R_j)$ on both sides,

$$\sum_i c_i \int \phi^*(r-R_i)H\phi(r-R_j)dr = E\sum_i c_i \int \phi^*(r-R_j)\phi(r-R_i)dr.$$

The equation consists of three kinds of integrals

1. $\epsilon = \int \phi^*(r-R_i)H\phi(r-R_i)dr$ as the on-site energy (when $i=j$, representing the energy of an electron when it is localized on a single atomic site
2. $t = \int \phi^*(r-R_j)H\phi(r-R_i)dr$ as the hopping integral (when $i \neq j$, representing the interaction energy between atomic orbitals.
3. $S = \int \phi^*(r-R_j)\phi(r-R_i)dr$, representing the overlap between two orbitals.

If we assume that two atomic orbitals are nearly orthogonal, this can be simplified to a matrix form as follows

$$\sum_j H_{ij}c_j = Ec_i$$

And H_{ij} is the matrix element.

$$H_{ij} = \begin{cases} \epsilon & \text{if } i=j, \\ t & \text{if } i \neq j. \end{cases}$$

For the case of a 1D chain, we further assume that the interaction is only limited to neighboring atoms. Hence, t is only nonzero for $j = i \pm 1$.

Now we plug in the Bloch theorem,

$$c_{i+1} = c_i e^{ika}, \quad c_{i-1} = e^{-ika}$$

The Schrodinger equation becomes,

$$\epsilon c_i + tc_{i+1} + tc_{i-1} = Ec_i \quad \rightarrow \quad \epsilon c_i + tc_i e^{ika} + tc_i e^{-ika} = Ec_i$$

Canceling the c_i,

$$\epsilon + t(e^{ika} + e^{-ika}) = E \quad \rightarrow \quad \epsilon + 2t\cos(ka) = E$$

This reveals that E and k are related the following **dispersion relation**

$$E(k) = \epsilon + 2t\cos(ka) \tag{11.6}$$

Bloch's theorem states that the wavefunction in a periodic crystal lattice can be written as:

$$\psi_k(r) = e^{ikr}u_k(r) \quad \rightarrow \quad \psi_k(r+a) = e^{ik(r+a)}u_k(r+a)$$

Using $u_k(r+a) = u_k(r)$,

$$\phi_k(r+a) = e^{ik(r+a)}u_k(r) = e^{i(k+\frac{2\pi}{a})a}u_k(r) = \phi_{k+\frac{2\pi}{a}}(r)$$

The equation suggests that e^{ika} represents a phase factor. Similarly, one can find

$$\phi_k(r+na) = e^{ik(r+na)}u_k(r) = e^{i(k+n\frac{2\pi}{a})a}u_k(r) = \phi_{k+n\frac{2\pi}{a}}(r)$$

This suggest that moving $n\times a$ length in the R space is equivalent to moving $n\times(2\pi/a)$ in the k space, both of which simply apply a phase factor of $\exp(in\times a)$. Correspondingly, $2\pi/a$ is the unit length in the k space, we define it as the reciprocal unit length.

$$G = \frac{2\pi}{a} \quad \text{reciprocal unit length}$$

Hence, the periodicity of the lattice means that the allowed wave vectors k can be translated by reciprocal lattice vectors $\mathbf{G}$ without changing the physical properties of the system:

$$E(k) = E(k + nG), \quad n \in \mathcal{Z}$$

As a result, any k outside the range $-\pi/a$ to π/a can be mapped back into this range by subtracting or adding reciprocal lattice vectors. Thus, the dispersion relation can be formal written as,

$$E(k) = \epsilon + 2t\cos(ka), \quad k \in \left(\frac{\pi}{a}, -\frac{\pi}{a}\right] \tag{11.7}$$

This relation is graphically shown in Fig. 11.2. It suggests that the E reaches the maximum value $\epsilon + 2t$ at $k = 0$, and the minimum value $\epsilon - 2t$ at the edge $k = \pm\pi/a$. The periodicity of the cosine function ensures that the dispersion relation $E(k)$ repeats itself in adjacent Brillouin zones, reflecting the inherent periodicity of the crystal lattice in reciprocal space.

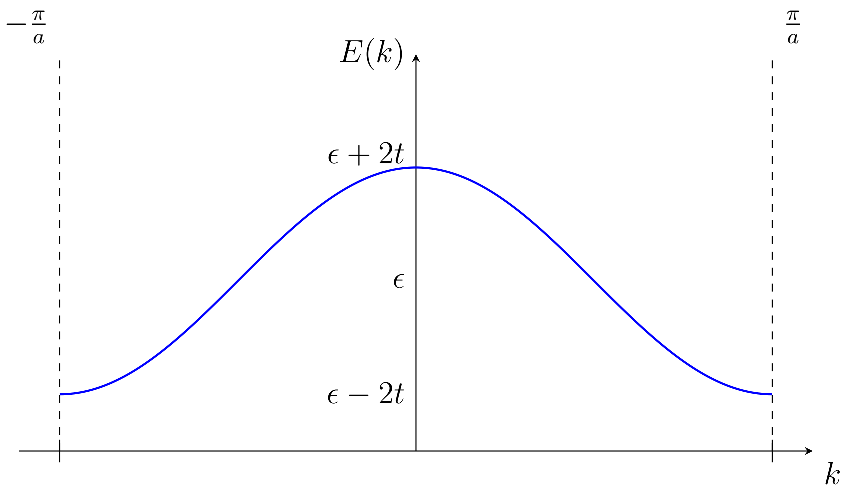


Figure 11.2: The $E - k$ relation for the 1D chain model.

This behavior also provides physical insight into the role of the hopping term t, which is responsible for introducing the dispersion. The width of the dispersion, quantified as $4t$ (from $\epsilon - 2t$ to $\epsilon + 2t$), is directly determined by the strength of the hopping interaction. Larger hopping integrals result in broader bands, reflecting a higher degree of electron delocalization, while smaller hopping terms indicate more localized electrons and narrower bands. As we will discuss in the later section, this dispersion suggests that the whole electron would form a series of energy bands in a periodic system.

11.2.2 Solution of 1D diatomic crystal

To make the model more complex, consider a 1D chain with two alternating atoms per unit cell. Let these atoms have different on-site energies, ϵ_A and ϵ_B , and hopping terms t_{AB}.

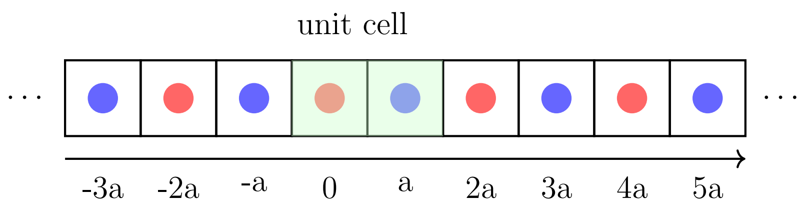


Figure 11.3: The schematic 1D diatomic crystal model with a unit distance of $2a$.

Using Bloch's theorem, the wavefunction can be written as:

$$\psi_k(r) = \sum_n \left[c_{nA}\phi_A(r - R_{nA}) + c_{nB}\phi_B(r - R_{nB})\right],$$

where c_{nA} and c_{nB} are the coefficients for the atomic orbitals ϕ_A and ϕ_B in the n-th unit cell. Hence, the lattice positions are:

$$R_{nA} = 2na, \quad R_{nB} = (2n + 1)a.$$

Substituting this wavefunction into the Schrödinger equation and applying Bloch's theorem:

$$c_{n+1,A} = c_{nA}e^{ik(2a)}, \quad c_{n-1,A} = c_{nA}e^{-ik(2a)}.$$

The resulting equations for the coefficients become:

$$\begin{aligned} \epsilon_A c_{nA} + t_{AB}c_{nB} + t_{AB}c_{nB}e^{ik(-2a)} &= Ec_{nA}, \\ \epsilon_B c_{nB} + t_{AB}c_{nA} + t_{AB}c_{nA}e^{ik(2a)} &= Ec_{nB}. \end{aligned}$$

The further simplification leads to

$$\begin{aligned} \epsilon_A c_{nA} + t_{AB}(1 + e^{-2ka})c_{nB} &= Ec_{nA}, \\ \epsilon_B c_{nB} + t_{AB}(1 + e^{2ka})c_{nA} &= Ec_{nB}. \end{aligned}$$

These equations can be written in matrix form:

$$\begin{bmatrix} \epsilon_A & t_{AB}(1 + e^{-2ika}) \\ t_{AB}(1 + e^{i2ka}) & \epsilon_B \end{bmatrix} \begin{bmatrix} c_{nA} \\ c_{nB} \end{bmatrix} = E \begin{bmatrix} c_{nA} \\ c_{nB} \end{bmatrix}.$$

The energy eigenvalues E can be obtained by solving the determinant:

$$\det \begin{bmatrix} \epsilon_A - E & t_{AB}(1 + e^{-i2ka}) \\ t_{AB}(1 + e^{i2ka}) & \epsilon_B - E \end{bmatrix} = 0.$$

Expanding the determinant:

$$(\epsilon_A - E)(\epsilon_B - E) - \left[t_{AB}^2(1 + e^{i2ka} + e^{-i2ka})\right] = 0.$$

Simplifying:

$$E^2 - (\epsilon_A + \epsilon_B)E + \epsilon_A\epsilon_B - 2t_{AB}^2[1 + \cos(2ka)] = 0.$$

This is a quadratic equation for E, and the solutionss:

$$E_\pm(k) = \frac{\epsilon_A + \epsilon_B}{2} \pm \sqrt{\left(\frac{\epsilon_A - \epsilon_B}{2}\right)^2 + 2t_{AB}^2[1 + \cos(2ka)]}.$$

Two representative solutions are

$$E_\pm(0) = \frac{\epsilon_A + \epsilon_B}{2} \pm \sqrt{\left(\frac{\epsilon_A - \epsilon_B}{2}\right)^2 + (2t_{AB})^2}, \quad E_\pm(\pm\pi/2a) = \frac{\epsilon_A + \epsilon_B}{2} \pm \sqrt{\left(\frac{\epsilon_A - \epsilon_B}{2}\right)^2}.$$

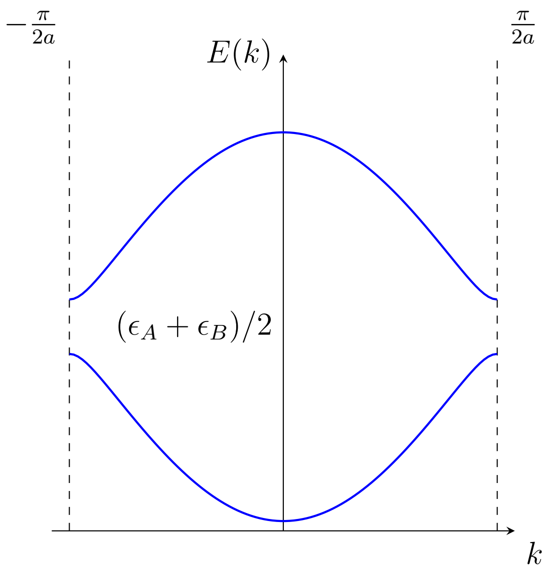


Figure 11.4: The $E - k$ relation for the 1D diatomic chain model.

The results are graphically shown in Fig. 11.4, where the two solutions correspond to the energy bands arising from the diatomic 1D chain model. Considering the general equation,

$$\epsilon_A c_A + 2t_{AB}c_B = Ec_A \quad \rightarrow \quad \frac{c_B}{c_A} = -\frac{\epsilon_A - E}{2t_{AB}}$$

For an extreme case when ϵ_A is very close to ϵ_B, the c_B/c_A ratio for $E_\pm$ states are

$$\frac{c_B}{c_A}(-) = \frac{\epsilon_A - (\epsilon_A + \epsilon_B)/2}{2t_{AB}} \geq 0, \quad \frac{c_B}{c_A}(+) = \frac{\epsilon_A - (\epsilon_A + \epsilon_B)/2 - 2t_{AB}}{2t_{AB}} \leq 0$$

Hence, these solutions represent two distinct states,

- In E_- state (like $\phi_A + \phi_B$), the electron wavefunctions on neighboring atoms interfere constructively. This means that the probability amplitude of finding an electron between the atoms is increased, enhancing the bond between them. The bonding state corresponds to a symmetric combination of atomic orbitals.

- In E_+ state (like $\phi_A - \phi_B$), the electron wavefunctions interfere destructively, reducing the probability amplitude between the atoms and weakening the bond. The antibonding state corresponds to an antisymmetric combination of atomic orbitals.

Unlike the monoatomic chain model, this diatomic model introduces additional complexity due to the alternating atoms, leading to a periodicity of π/a in reciprocal space. The extrema of the cosine term, $\cos(2ka)$, occur at k=0 (maximum) and $k = \pm\pi/(2a)$ (minimum). These points define the highest and lowest energy states within the respective bands. Notably, a band gap opens at the edge states ($k = \pm\pi/(2a)$), reflecting the energy difference between the bonding and antibonding states at those points. This gap indicates the presence of a forbidden energy region, a characteristic feature of materials with alternating atomic sites.

The opening of this gap and the distinct dispersion curves suggest that the system's electronic properties are significantly influenced by the alternating atomic potentials. This provides valuable insight into band formation and energy gaps, which are critical in understanding semiconductors and insulators.

11.2.3 Remarks on the Tight Binding Model

The tight-binding model is a simple yet powerful approach to describing the electronic structure of solids. It provides valuable insights into the formation of energy bands, band gaps, and dispersions, which are essential for understanding the electronic properties of materials.

Despite its strengths, the tight-binding model has limitations in accuracy. It assumes a fixed basis set of atomic orbitals and simplifies the calculation of overlap integrals, neglecting variations in orbital shapes due to bonding or hybridization. Consequently, the model is less effective in systems where electron-electron interactions, spin-orbit coupling, or long-range interactions are significant. Nonetheless, its simplicity and utility make it an essential tool in the study of solid-state physics and materials science.

11.3. The Plane Wave Model

To improve the accuracy of the tight-binding model, we may consider a better choice of basis set to expand the wavefunction. From Bloch's Theorem, it is natural to link the wavefunction with plane waves, as they inherently respect the periodicity of the crystal lattice.

A plane wave is a mathematical representation of a wave with constant frequency and wavelength, where the oscillations—whether they pertain to electric or magnetic fields in classical physics or quantum wavefunctions in solid-state physics—are uniform across planes perpendicular to the direction of wave propagation. A general plane wave in 3D traveling in the direction of a wave vector $\mathbf{k}$ at time t can be written as:

$$\psi(\mathbf{r}, t) = Ae^{i(\mathbf{k}\cdot\mathbf{r}-\omega t)} \tag{11.8}$$

Hence the 1D version is

$$\psi(x, t) = Ae^{i(\mathbf{k}\cdot x-\omega t)} \tag{11.9}$$

The direction of $\mathbf{k}$ determines the direction in which the wave propagates. For example, if $\mathbf{k}$ points along the x-axis, then the wave propagates in that direction. In quantum mechanics, the plane wave represents a delocalized particle, meaning it is spread out across all space, with no specific location but a definite momentum. If we omit the impact of time, the plane wave is reduced to

$$\psi(\mathbf{r}, t) = Ae^{i\mathbf{k}\cdot\mathbf{r}} \tag{11.10}$$

$$\psi(x, t) = Ae^{i\mathbf{k}\cdot x} \tag{11.11}$$

11.3.1 Using Plane Waves as the Basis Set

According to the Bloch's theorem, we find an expression for $\psi_k(x)$. To solve it, we still need to figure out $u_k(x)$. Numerically, we use a Fourier series expansion of plane waves to approximate it.

$$u_k(x) = \sum_n c_n e^{inGx}, \quad \text{where} \quad G = 2\pi/a,\ n \in \mathcal{Z}. \tag{11.12}$$

The choice is based on the fact that $u_k(x)$, being periodic with the same period as the crystal lattice, can be expressed as a sum of plane waves with wavevectors that are integer multiples of G.

Each plane wave in the expansion of the wavefunction has an associated kinetic energy:

$$E_{\text{kinetic}} = \frac{\hbar^2 |G|^2}{2m}$$

In practical calculations, only plane waves with a kinetic energy below a certain cutoff E_{cut} are included. This means that only plane waves satisfying the following conditions are used in the basis set.

$$\frac{\hbar^2 \|G\|^2}{2m} \leq E_{\text{cut}} \quad \rightarrow \quad \|G_{\text{max}}\|^2 \leq 2E_{\text{cut}} \quad \text{(in a.u.)} \tag{11.13}$$

When we use a high kinetic energy cutoff, we're allowing for more plane waves in the basis set, which improves the wavefunction's ability to adapt to rapid changes in the potential within the unit cell. Plane waves with very higher kinetic energy ignored because they contribute less to the wavefunction and are computationally more expensive to include.

11.3.2 The Numerical Solution

Now, to solve the 1D atomic chain model, the Schrödinger equation is:

$$\left(-\frac{\hbar^2}{2m}\frac{d^2}{dx^2} + V(x)\right)\psi_k(x) = E\psi_k(x) \tag{11.14}$$

Using Bloch's form of the wavefunction, substitute $\psi_k(x) = e^{ikx}u_k(x)$ into the Schrödinger equation. We then separates the wavefunction into a part that depends on k and a part that is periodic. This yields an equation for $u_k(x)$:

$$\left(-\frac{\hbar^2}{2m}\left(\frac{d}{dx} + ik\right)^2 + V(x)\right)u_k(x) = Eu_k(x)$$

Using the plane wave basis set, we break the equation into

$$\sum_n c_n \left(\frac{\hbar^2(k+G_n)^2}{2m}\right) e^{i(k+G_n)x} + \sum_n c_n \sum_l V(G_m) e^{i(k+G_n+G_l)x} = E \sum_n c_n e^{i(k+G_n)x},$$

where V_{G_m} are the Fourier components of the potential:

$$V(x) = \sum_l V(G_l) e^{iG_l x}.$$

To find the values of c_n, we can match coefficients of each term $e^{i(k+G_n)x}$ on both sides of the equation. This yields a system of linear equations:

$$\left(\frac{\hbar^2(k+G_n)^2}{2m} - E\right) c_n + \sum_l V(G_n - G_l) c_l = 0.$$

This equation can be rewritten in matrix form as:

$$\mathbf{Hc} = E\mathbf{c}$$

where $\mathbf{H}$ is a matrix representing the Hamiltonian, $\mathbf{c}$ is a vector of the Fourier coefficients c_n, and E is the energy eigenvalue.

Each element H_{ij} of the matrix $\mathbf{H}$ is given by:

$$H_{nl} = \frac{\hbar^2(k+G_n)^2}{2m}\delta_{nl} + V(G_n - G_l) \tag{11.15}$$

Since an infinite number of reciprocal lattice vectors G_n would make this a large system, one needs to truncate it by considering only a finite number of G_n values. For example, you might choose n from -N to N. This results in a finite matrix of size $(2N+1) \times (2N+1)$.

Each equation in this system corresponds to a different value of n, representing a different reciprocal lattice vector G_n. Solving this set of equations will give you the values of c_n series and the energy eigenvalues E for each k value. The wavefunciton $\psi_k(x)$ can be reconstructed as:

$$\psi_k(x) = e^{ikx} \sum_n c_n e^{inGx} = \sum_n c_n e^{i(k+nG)x}$$

For each k -point, solving this eigenvalue problem gives a spectrum of eigenvalues E. The lowest eigenvalue is typically associated with the first (lowest) energy band, the next eigenvalue with the second band, and so on. This provides a set of energy bands at each k-point, showing the band structure.

11.4. Extension to 3D system

For a 3D system, we need to extend the basis set to describe the 3D periodicity. Within the periodic boundary condition, the repetitive atomic packing is defined as a crystal. Choosing the smallest unit cell, we first define the lattice vector $\mathbf{a_1}$, $\mathbf{a_2}$ and $\mathbf{a_3}$.

The lattice vector $\mathbf{R}$ describes the periodic translation

$$\mathbf{R} = n_1\mathbf{a_1} + n_2\mathbf{a_2} + n_3\mathbf{a_3}, \quad n_i \in \mathcal{Z}$$

According to the translation symmetry,

$$H(\mathbf{r}+\mathbf{R}) = H(\mathbf{r})$$

$$\psi_k(\mathbf{r}) = e^{ik\cdot\mathbf{r}}u_{\mathbf{k}}(\mathbf{r}) \tag{11.16}$$

11.4.1 Reciprocal Lattice in 3D

Similar to the 1D case, we use the reciprocal lattice to describe the periodicity of the crystal in reciprocal space. So the Fourier series in 3D is

$$u_{\mathbf{k}}(\mathbf{r}) = \sum_{|\mathbf{G}|<G_{\max}} c_{\mathbf{k}}(\mathbf{G})e^{i\mathbf{G}\cdot\mathbf{r}} \tag{11.17}$$

The reciprocal lattice is defined by three reciprocal lattice vectors $\mathbf{b_1}$, $\mathbf{b_2}$, $\mathbf{b_3}$, which are related to the real-space lattice vectors $\mathbf{a_1}$, $\mathbf{a_2}$, $\mathbf{a_3}$ as:

$$\begin{aligned}
\mathbf{b_1} &= 2\pi\frac{\mathbf{a_2}\times\mathbf{a_3}}{\mathbf{a_1}\cdot(\mathbf{a_2}\times\mathbf{a_3})},\\
\mathbf{b_2} &= 2\pi\frac{\mathbf{a_3}\times\mathbf{a_1}}{\mathbf{a_1}\cdot(\mathbf{a_2}\times\mathbf{a_3})},\\
\mathbf{b_3} &= 2\pi\frac{\mathbf{a_1}\times\mathbf{a_2}}{\mathbf{a_1}\cdot(\mathbf{a_2}\times\mathbf{a_3})}.
\end{aligned}$$

Accordingly, the reciprocal lattice vector $\mathbf{G}$ is a linear combination of the reciprocal basis vectors:

$$\mathbf{G} = m_1\mathbf{b_1} + m_2\mathbf{b_2} + m_3\mathbf{b_3}, \quad m_i \in \mathbb{Z},$$

where m_1, m_2, m_3 are integers, representing the number of reciprocal lattice translations along $\mathbf{b_1}$, $\mathbf{b_2}$, and $\mathbf{b_3}$, respectively.

11.4.2 3D Hamiltonian on the Plane Wave Basis

The Hamiltonian matrix in 3D is,

$$H_{nl} = \frac{\hbar^2(\mathbf{k}+\mathbf{G}_i)^2}{2m}\delta_{nl} + V(\mathbf{G}_n - \mathbf{G}_l) \tag{11.18}$$

Thus, the final expression of wavefunction at k becomes

$$\psi_k(r) = \sum_{|\mathbf{G}|<G_{\max}} c_{\mathbf{k}}(\mathbf{G})e^{i(\mathbf{k}+\mathbf{G})\cdot\mathbf{r}}u_{\mathbf{k}}(\mathbf{r}) \tag{11.19}$$

11.5. Energy Bands and Brillouin Zone

11.5.1 Energy Bands

So far, we have learned that the eigenstates of each electron in the crystal depend on the wave vector $\mathbf{k}$, which implies that the ground state energy $E(\mathbf{k})$ of each electron varies with different **k** choices. This variation leads to an energy dispersion, which is fundamentally different from the molecular case. In molecules, two electrons can occupy a single energy level, constrained by the Pauli Exclusion Principle. In contrast, in a crystal, electrons occupy energy bands that arise from the periodic potential of the atomic lattice.

At absolute zero Kelvin, the energy bands that are completely filled with electrons are known as the **valence bands**, whereas the next higher energy band, which is either partially filled or completely empty, is called the **conduction bands**.

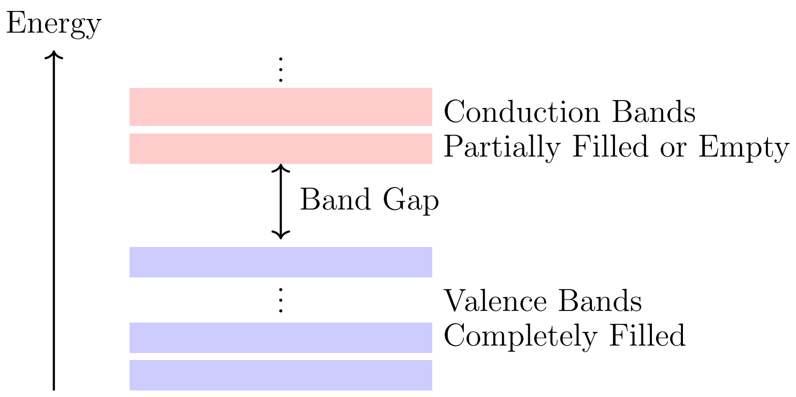


Figure 11.5: The concepts of valence and conduction bands.

A material's conductive properties depend on the arrangement of these bands. In conductors, the valence and conduction bands overlap, or the conduction band is partially filled. In semiconductors and insulators, a band gap separates the valence and conduction bands, preventing electron flow at low temperatures.

11.5.2 The Brillouin Zone

Since the eigenvalue depends on the choice of **k**, how do we select **k** in the practical calculation? In a crystal, the wave vector **k** represents the periodicity of the electron's wavefunction, with **k** values lying in the reciprocal space.

Considering the Schrödinger equation for electrons in a crytal:

$$\left[\frac{-\hbar^2}{2m}\nabla^2 + V(\mathbf{r})\right]\psi_{\mathbf{k}}(\mathbf{r}) = E_{\mathbf{k}}\psi_{\mathbf{k}}(\mathbf{r}),$$

The wave vector **k** appears in $\psi_{\mathbf{k}}(\mathbf{r})$ only as a phase factor $e^{i\mathbf{k}\cdot\mathbf{r}}$. Because of the periodicity of $V(\mathbf{r})$, the solutions $E_{\mathbf{k}}$ and $\psi_{\mathbf{k}}(\mathbf{r})$ are periodic in **k**-space, with a periodicity defined by the reciprocal lattice vectors:

$$E_{\mathbf{k}+\mathbf{G}} = E_{\mathbf{k}}.$$

Hence, we don't need to sample all **k** points. Instead, we mathematically define a smallest region that

$$\|\mathbf{k}\| < \|\mathbf{k}+\mathbf{G}\| \quad \forall \mathbf{G} \neq 0,$$

This is called **First Brillouin Zone** (FBZ). Below we discuss how to construct FBZ for a simple cubic structure.

The First Brillouin Zone of a Simple Cubic Structure.

In a simple cubic structure with unit length of a, the lattice vectors are:

$$\mathbf{a}_1 = a\hat{x}, \quad \mathbf{a}_2 = a\hat{y}, \quad \mathbf{a}_3 = a\hat{z}.$$

The reciprocal lattice is also a simple cubic lattice with lattice constant $2\pi/a.$:

$$\mathbf{b}_1 = \frac{2\pi}{a}\hat{x}, \quad \mathbf{b}_2 = \frac{2\pi}{a}\hat{y}, \quad \mathbf{b}_3 = \frac{2\pi}{a}\hat{z}.$$

To construct the FBZ, we look for the maximum values around the origin of the reciprocal lattice. For the SC lattice, the reciprocal lattice points are:

$$\mathbf{G} = h\mathbf{b}_1 + k\mathbf{b}_2 + l\mathbf{b}_3,$$

The nearest points to the origin are:

$$\pm\frac{2\pi}{a}\hat{x}, \quad \pm\frac{2\pi}{a}\hat{y}, \quad \pm\frac{2\pi}{a}\hat{z}.$$

Next, construct planes that are perpendicular to the vectors connecting the origin to each nearest reciprocal lattice point. These planes are positioned at the midpoint between the origin and the reciprocal lattice points.
The planes:

$$x = \pm\frac{\pi}{a} \quad (\text{for } \mathbf{b}_1) \qquad y = \pm\frac{\pi}{a} \quad (\text{for } \mathbf{b}_2) \qquad z = \pm\frac{\pi}{a} \quad (\text{for } \mathbf{b}_3).$$

Hence, the intersection of these planes forms a cube centered at the origin with side length $2\pi/a$.

$$-\frac{\pi}{a} \leq k_x \leq \frac{\pi}{a}, \quad -\frac{\pi}{a} \leq k_y \leq \frac{\pi}{a}, \quad -\frac{\pi}{a} \leq k_z \leq \frac{\pi}{a}.$$

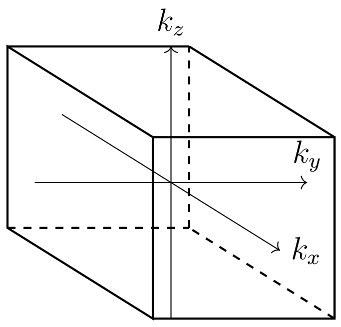


Figure 11.6: The first Brillouin Zone of simple cubic structure.

Since the FBZ contains all unique **k**-points, computational calculations of properties like band structures or density of states can be limited to the FBZ, avoiding redundant evaluations. In addition, the symmetry of the crystal can further reduce the number of **k**-points that need to be sampled. In the crystallography book, the high symmetry points and paths are also commonly used. These information can be found in either database or literature [17]. Only the irreducible wedge of the FBZ is required for most calculations.

While it is relatively straightforward to compute the FBZ for simple crystal structures such as the simple cubic or face-centered cubic lattices, constructing the FBZ for more complex lattices can be challenging. In such cases, specialized tools and software can simplify the process. One widely used tool is `Seek-Path`, available at Materials Cloud (`https://www.materialscloud.org/work/tools/seekpath`). This tool provides automatic construction of the FBZ for a given crystal structure. In addition, it generates the high-symmetry points and paths for band structure calculations, as well as convenient visualization of the FBZ and the irreducible wedge.

11.5.3 Practical Band Structure Visualization

The concept of the First Brillouin Zone provides a systematic framework for understanding and visualizing the band dispersion of electrons in a crystal. In principle, the electronic band structure is characterized by the energy $E_{\mathbf{k}}$ as a function of the wave vector $\mathbf{k}$, which exists in the 3D reciprocal space defined by the FBZ. However, directly visualizing $E_{\mathbf{k}}$ for all $\mathbf{k}$-points in 3D space is impractical due to the complexity of plotting such a multidimensional dataset.

To address this challenge, a more convenient and widely adopted method is to project the band dispersion along high-symmetry paths within the FBZ. These paths connect high-symmetry points (e.g., Γ, X, L, etc.), which are representative of the crystal's periodicity and symmetry. By plotting $E(\mathbf{k})$ along these paths, one captures the key features of the electronic structure, including:

1. whether the material is a metal, semiconductor or insulator.
2. if not a metal, what the band gap is between valence and conduction bands.
3. Curvature of the bands, which reflects the effective mass of electrons.
4. Dispersion characteristics (e.g., narrow or wide bands) indicating the degree of electron localization or delocalization.

This approach simplifies the visualization of band structures while retaining the essential physics that arise from the periodic potential of the atomic lattice. The resulting plots not only reveal critical information about a material's electronic properties but also provide insights into its conductivity, optical behavior, and response to external fields.

In addition, such a two-dimensional band plot is often coupled with a density of states (DOS) plot, which provides complementary information about the distribution of electronic states over energy. The DOS can be considered as a histogram of $E_{\mathbf{k}}$ values, showing the number of electronic states available at each energy level. Fig. 11.7 shows a combined plot for Silicon's band structure and DOS.

11.6. Empirical Pseudopotential Method

To obtain an accurate band structure, Empirical Pseudopotential Method (EPM) is a widely used approach for calculating the electronic band structure of crystals [18, 19]. It simplifies the problem of solving the Schrödinger equation for electrons in a periodic potential by replacing the full ionic potential with an effective potential, referred to as the pseudopotential. This effective potential accounts for the complex interactions between

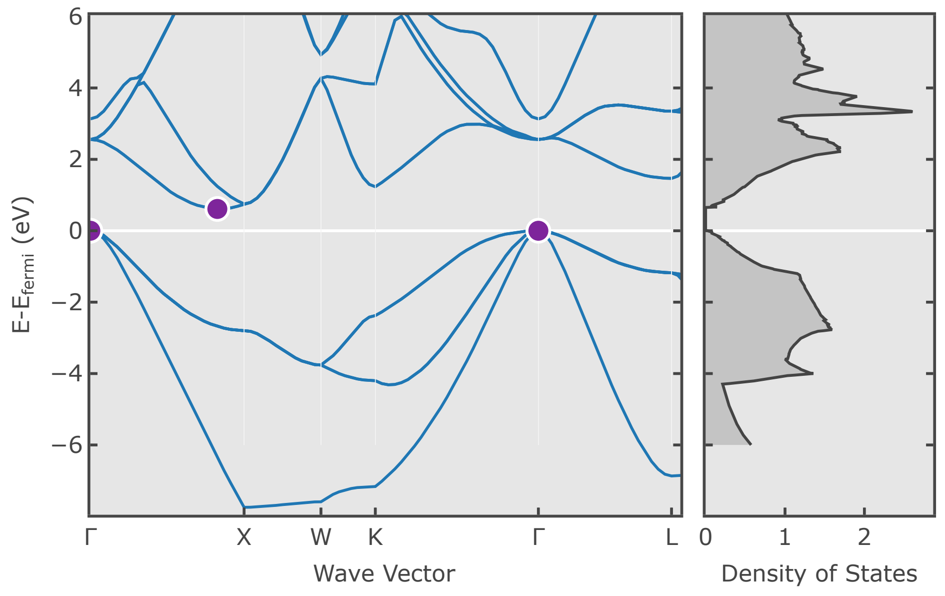


Figure 11.7: An example of band structure of silicon with the combined DOS plot (downloaded from `https://next-gen.materialsproject.org/materials/mp-149`).

valence electrons and the ion cores, allowing the computational focus to be directed primarily on valence electrons while avoiding the need to explicitly treat core electrons.

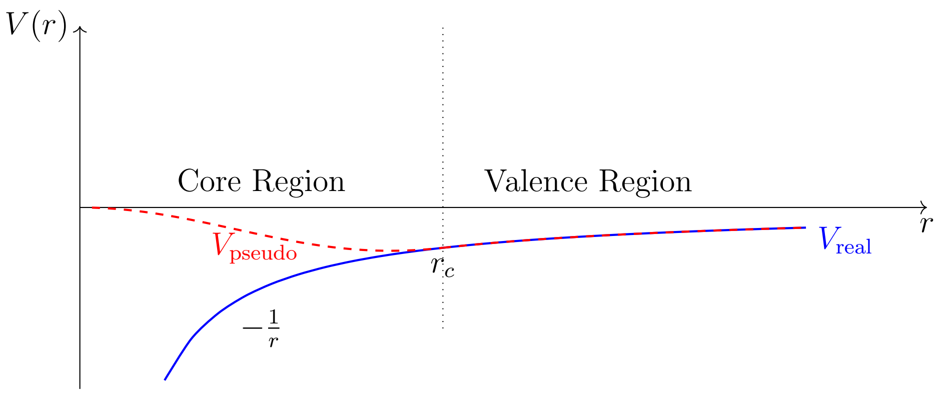


Figure 11.8: The idea of pseudopotential.

The Hamiltonian for an electron in the crystal consists of a kinetic-energy potential which depends on position.

$$\left(\frac{p^2}{2m} + V(\mathbf{r})\right)\phi_{\mathbf{k}}(\mathbf{r}) = E_k\phi_{\mathbf{k}}(\mathbf{r})$$

Here $V(\mathbf{r})$ is a pseudopotential, which depends on only valence electrons. Hence $V(\mathbf{r})$ are only small perturbations. Correspondingly, the solution $\phi_{\mathbf{k}}$ can expanded into a sum of plane waves.

$$|\phi_{\mathbf{k}}\rangle = \sum_{\mathbf{G}} \alpha_{\mathbf{G}} \exp(i\mathbf{k}\cdot\mathbf{G}) = \sum_{\mathbf{G}} \alpha_{\mathbf{G}} |\mathbf{k}+\mathbf{G}\rangle$$

where α are the coefficients and $|\mathbf{k}+\mathbf{G}\rangle$ denote the planewaves. The coefficients can be determined from

$$\det\left|\left[\frac{\hbar^2\mathbf{k}^2}{2m}-E_\mathbf{k}\right]\delta_{\mathbf{k},\mathbf{k}+\mathbf{G}}+\langle\mathbf{k}|V(\mathbf{r})|\mathbf{k}+\mathbf{G}\rangle\right|=0$$

where

$$\langle\mathbf{k}|V(\mathbf{r})|\mathbf{k}+\mathbf{G}\rangle=\left[\frac{1}{N}\sum_j e^{-i\mathbf{G}\cdot\mathbf{R}}\right]\frac{1}{\Omega}\int V(\mathbf{r})e^{-i\mathbf{G}\cdot\mathbf{r}}d\mathbf{r}$$

Hence, this includes two terms. The first term is called **form factor** V_f

$$V(\mathbf{G})=\frac{1}{\Omega}\int V(\mathbf{r})e^{-i\mathbf{G}\cdot\mathbf{r}}d\mathbf{r} \tag{11.20}$$

and the 2nd term is called **structure factor** (S)

$$S(\mathbf{G})=\frac{1}{N}\sum_{j=1}^{N}\exp(-i\mathbf{G}\cdot\mathbf{R_j}) \tag{11.21}$$

The pseudopotential $V(\mathbf{r})$ can be expressed in terms of the structure factor and the form factors by

$$V(\mathbf{r})=\sum_\mathbf{G}V_f(\mathbf{G})S(\mathbf{G})e^{-i\mathbf{G}\cdot\mathbf{r}} \tag{11.22}$$

Following the 1D case, the Schrodinger equation for the given k has a matrix form below

$$H_{mn}(\mathbf{k})=\frac{\hbar^2}{2m_e}|\mathbf{k}+\mathbf{G}_m|^2\delta_{mn}+V_f(\mathbf{G}_m-\mathbf{G}_n)S(\mathbf{G}_m-\mathbf{G}_n) \tag{11.23}$$

Clearly, this matrix has two features.

- For the diagonal part (when $m=n$) only depends on the $\mathbf{k}$, $\mathbf{G}$, $V_f(000)$, $S(000)$ values.
- When $m\neq n$, the kinetic energy can be neglected, However, one needs to know the series of $V_f(hkl)$ and $S(hkl)$ values.

Fortunately, $S(hkl)$ directly depends on crystal symmetry and can be zero for many cases, which can effectively eliminate many H_{mn} elements and thus simplify the computation of eigenvalue problems. Below we will explain how to derive a simplified model on high symmetric diamond crystal.

11.6.1 Structure Factor of Representative Structures

Before proceeding to the analysis of diamond structure, let us first do a few calculations for two representative high symmetry structures.

Body-centered cubic (BCC)

For the the body-centered cubic Bravais lattice, we have two points (0, 0, 0) and (1/2, 1/2, 1/2) in the unit cell. Hence

$$S(hkl) = e^{-i(0+0+0)} + e^{-i\pi(h+k+l)}.$$

Clearly, it follows

$$S_{hkl} = \begin{cases} 2, & h+k+l = \text{even} \\ 0, & h+k+l = \text{odd} \end{cases}$$

Face-centered cubic (FCC)

For the the body-centered cubic Bravais lattice, we four points (0, 0, 0), (1/2, 1/2, 0), (1/2, 0, 1/2), (0, 1/2, 1/2) in the unit cell. Hence,

$$S(hkl) = e^{-i(0+0+0)} + e^{-i\pi(h+k)} + e^{-i\pi(k+l)} + e^{-i(h+l)}).$$

Clearly, it follows

$$S(hkl) = \begin{cases} 4, & h, k, l \text{ are all odd or even} \\ 0, & \text{otherwise} \end{cases}$$

Using these calculation, we can also immediately find that translation ****.

11.6.2 Application to the Diamond Crystal

For the case of diamond structure, it can be counted as two FCC sublattice, including (0, 0, 0), (1/2, 1/2, 0), (1/2, 0, 1/2), (0, 1/2, 1/2), as well as the uniform (1/4, 1/4, 1/4) shift, resulting another 4 atoms at (1/4, 1/4, 1/4), (3/4, 3/4, 1/4), (3/4, 1/4, 3/4) and (1/4, 3/4, 3/4).

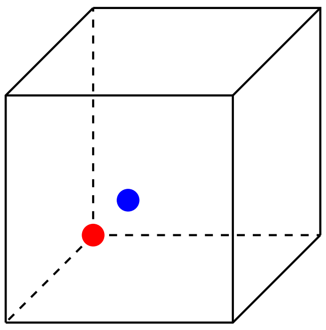


Figure 11.9: The diamond lattice with two atoms at (0, 0, 0) and (1/4, 1/4, 1/4). Note that other atoms are intentionally removed for clarity.

For the 1st group of 4 FCC atoms:

$$S^1(hkl) = e^{-i(0+0+0)} + e^{-i\pi(h+k)} + e^{-i\pi(k+l)} + e^{-i(h+l)}.$$

For the 2nd group of 4 atoms, it can be considered as the original fcc multiply a factor of $e^{(-iG_{hkl}\cdot[1/4,1/4,1/4]}$

$$S^2(hkl) = e^{(-i\pi\frac{h+k+l}{2})}[e^{-i(0+0+0)} + e^{-i\pi(h+k)} + e^{-i\pi(k+l)} + e^{-i(h+l)}].$$

Hence the total is

$$S(hkl) = S^1(hkl) + S^2(hkl) = (1 + e^{(-i\pi\frac{h+k+l}{2})})S_{\text{FCC}}.$$

And it satisfies the following

$$S_{hkl} = \begin{cases} 8, & h+k+l = 4N \\ 4(1+i), & h+k+l = 2N+1 \\ 0, & h+k+l = 2N+2 \end{cases}$$

Immediately, we can find the following that the following $h_1^2 + k_1^2 + l_1^2$ values may lead to either zero or non-existing S values.

$$\begin{aligned}
h^2+k^2+l^2 &= 1 &\leftarrow\quad & S(\pm1,0,0) = 0\\
h^2+k^2+l^2 &= 2 &\leftarrow\quad & S(\pm1,\pm1,0) = 0\\
h^2+k^2+l^2 &= 4 &\leftarrow\quad & S(\pm2,0,0) = 0\\
h^2+k^2+l^2 &= 5 &\leftarrow\quad & S(\pm2,\pm1,0) = 0\\
h^2+k^2+l^2 &= 6 &\leftarrow\quad & S(\pm2,\pm1,\pm1) = 0\\
h^2+k^2+l^2 &= 7 &\leftarrow\quad & \text{no valid h, k, l}\\
h^2+k^2+l^2 &= 9 &\leftarrow\quad & S(\pm3,0,0) = 0\\
h^2+k^2+l^2 &= 10 &\leftarrow\quad & S(\pm3,\pm1,0) = 0\\
h^2+k^2+l^2 &= 12 &\leftarrow\quad & \text{no valid h, k, l}\\
h^2+k^2+l^2 &= 13 &\leftarrow\quad & S(\pm3,\pm2,0) = 0\\
h^2+k^2+l^2 &= 14 &\leftarrow\quad & S(\pm3,\pm2,\pm1) = 0\\
h^2+k^2+l^2 &= 15 &\leftarrow\quad & \text{no valid h, k, l}\\
h^2+k^2+l^2 &= 16 &\leftarrow\quad & S(\pm4,0,0) = 0\\
h^2+k^2+l^2 &= 17 &\leftarrow\quad & S(\pm4,\pm1,0) = 0\\
h^2+k^2+l^2 &= 18 &\leftarrow\quad & S(\pm4,\pm1,\pm1) = 0
\end{aligned}$$

Note that the structure factors are generally complex values and their modulus won't change if one applies a uniform shift on the atomic coordinates. So the above (hkl) can be safely neglected. And the structure factor $S(\mathbf{G})$ is nonzero only for $|\mathbf{G}|^2 = 0, 3, 8, 11, 19, \ldots$. These correspond to

$$\begin{aligned}
h^2+k^2+l^2 &= 0 & \mathbf{G} &= (0,0,0)\\
h^2+k^2+l^2 &= 3 & \mathbf{G} &= (\pm1,\pm1,\pm1)\\
h^2+k^2+l^2 &= 8 & \mathbf{G} &= (\pm2,\pm2,0),(\pm2,0,\pm2),(0,\pm2,\pm2)\\
h^2+k^2+l^2 &= 11 & \mathbf{G} &= (\pm3,\pm1,\pm1),(\pm1,\pm3,\pm1),(\pm1,\pm1,\pm3)\\
h^2+k^2+l^2 &= 19 & \mathbf{G} &= (\pm3,\pm3,\pm1),(\pm3,\pm1,\pm3),(\pm1,\pm3,\pm3)
\end{aligned}$$

For the remaining nonzero hkl choices, S may take complex values when $h + k + l = 2N + 1$ (e.g., $S(111)$). To get rid of the complex values, we can shift the atoms in the diamond unit cell by (-1/8, -1/8, -1/8). Upon shifting, nonzero $S(hkl)$ values can be adjusted while maintaining the same modulus, However, zero $S(hkl)$ values remain zero.

For convenience, we choose the center between $\mathbf{R}_1$ and $\mathbf{R}_2$ as origin, and the coordinates becomes $\mathbf{R}'_1$ = (-1/8, -1/8, -1/8) $\times a$ and $\mathbf{R}'_2$ = (1/8, 1/8, 1/8) $\times a$.

$$\begin{aligned} S(\mathbf{G}) &= A \times S_{\mathrm{FCC}} \times \frac{1}{2}\left(e^{-i\mathbf{G}\cdot\mathbf{R}'_1} + e^{-i\mathbf{G}\cdot\mathbf{R}'_2}\right) \\ &= A \times S_{\mathrm{FCC}} \times \cos(\mathbf{G}\cdot\mathbf{R'_1}) \\ &= A \times S_{\mathrm{FCC}} \times \cos[\pi(h+k+l)/4]) \end{aligned}$$

This formula can ensure that we only need to deal with real valued $S(hkl)$ from now on.

Further, we enforce that the pseudopotential at the real space will expand spherically, leading to that $V_f(h_1, k_1, l_1) = V_f(h_2, k_2, l_2)$ when $h_1^2 + k_1^2 + l_1^2 = h_2^2 + k_2^2 + l_2^2$. As a result, equivalent reciprocal lattice vectors (e.g., $\mathbf{G} = (\pm1, \pm1, \pm3)$) share the same potential value, such as $V(\sqrt{11})$. Consequently, only the following potential values need to be considered: $V(0), V(\sqrt{3}), V(\sqrt{8}), V(\sqrt{11}), V(\sqrt{19})\dots$ are needed to express the Hamiltonian.

11.6.3 Model Hamiltonian

According to eq. 11.23, the Hamiltonian H is constructed by including contributions from both the kinetic energy $\hat{T}$ and the potential energy $\hat{V}$. Assuming the use of N_p plane waves within a cutoff defined by $\mathbf{G}_{\max}$, the resulting Hamiltonian matrix H is of size $N_p \times N_p$, including

- Diagonal Elements ($\mathbf{H}_{mm}$): These terms are caused by kinetic energy contribution and offset energy at $G = 0$, namely $\hbar^2\mathbf{k}/2m_e + V_f(0)S(0)$.

- Off-Diagonal Elements ($\mathbf{H}_{mn}$): These terms arise from the potential energy $V(\mathbf{q})$ (where $\mathbf{q} = |\mathbf{G}_m - \mathbf{G}_n|$). If we let $V_f(\mathbf{q})$ include the common factor of $A \times S_{\mathrm{FCC}}$, the expression becomes $V_f(q) \times \cos(\frac{\pi}{4} \times \mathbf{1}^\top\mathbf{q})$, where $\mathbf{1}^\top\mathbf{q}$ is simply the vector sum.

In practice, $V(0)$ is typically set to zero because it merely shifts the overall energy levels $E_\mathbf{k}$ without affecting the relative band structure.

Next, a higher hkl index should have less contribution and they can be omitted. So we can truncate the potential energy matrix to include reciprocal lattice vectors satisfying $h^2 + k^2 + l^2 < 19$. As such, only three independent parameters, $V(\sqrt{3})$, $V(\sqrt{8})$, and $V(\sqrt{11})$ are needed to construct a full $\mathbf{H}$ matrix. These values are typically fitted empirically from experimental data. For more details, please refer to Chelikowsky and Cohen's seminal work [19]. The specific values used in this calculation are summarized in Table 11.1.

11.6.4 Python Implementation

Following the discussions in the previous section, we can write a Python Code as shown below to compute the band structure of Si.

Table 11.1: Parameters employed in the band structure calculation of Si and Ge.

System	$V(\sqrt{3})$ (Ry)	$V(\sqrt{8})$ (Ry)	$V(\sqrt{11})$ (Ry)	a (Å)
Si [19]	-0.2241	-0.0520	-0.0724	5.43
Ge [20]	-0.2210	0.0190	0.0560	5.66

```python
import numpy as np
from scipy import constants as c
import matplotlib.pyplot as plt
import seaborn as sns
sns.set(font_scale=1.8)

# Function to generate Miller indices (h, k, l) within a given range
def generate_hkls(N=3):
    hkls = []
    for h in range(-N, N+1):
        for k in range(-N, N+1):
            for l in range(-N, N+1):
                #square = h**2 + k**2 + l**2
                #if square <= g_max:
                hkls.append([h, k, l])
    return np.array(hkls)

# Function to construct the Hamiltonian matrix for the system
def hamiltonian(k, a, form_factors, g_basis, hkls):

    kinetic_c = (2 * np.pi / a)**2 * c.hbar**2 / (2 * c.m_e * c.e)
    N = len(hkls)
    h = np.zeros([N, N])

    # Loop through rows and columns of the Hamiltonian
    for i in range(N):
        for j in range(N):
            if i == j: # kinetic energy
                g = (hkls[i] + k) @ g_basis
                h[i, i] = kinetic_c * g @ g
            else: # potential energy
                g = (hkls[i] - hkls[j]) @ g_basis
                hkl_square = int(np.sum(g * g))
                if hkl_square in form_factors.keys():
                    factor = form_factors[hkl_square]
                    h[i, j] = factor * np.cos(np.pi/4*sum(g))
    return h

# Constants and Parameters
a = 5.43e-10
form_factors = {
    3: -0.2241*13.6059, #-0.2241),
    8:  0.0551*13.6059, #(0.0551),
   11:  0.0724*13.6059, #(0.0724)
   }

g_basis = np.array([[-1, 1, 1], [1, -1, 1], [1, 1, -1]])
hkls = generate_hkls(2)
```

```

# https://www.materialscloud.org/work/tools/seekpath
# Define high-symmetry k-points for band structure calculation
pts = {
       "G": [0, 0, 0],
       "X": [1/2, 0, 1/2],
       "L": [1/2, 1/2, 1/2],
       "W": [1/2, 1/4, 3/4],
       "K": [3/8, 3/8, 3/4],
      }

N_pts = 25
k_paths = None
k_paths_x = None
k_paths_labels = []
for path in ["GX", "XW", "WL", "LG", "GK"]:
    b, e = pts[path[0]], pts[path[1]]
    lines = [np.linspace(_b, _e, N_pts) for _b, _e in zip(b, e)]
    k_path = np.stack(lines, axis=-1)
    k_length = np.linalg.norm(k_path[0] - k_path[-1])
    k_path_x = k_length * np.linspace(0, 1, N_pts)
    if k_paths is not None:
        x_max = k_paths_x[-1]
        k_paths = np.vstack((k_paths, k_path))
        k_paths_x = np.hstack((k_paths_x, k_path_x+x_max))
    else:
        k_paths = k_path
        k_paths_x = k_path_x
        k_paths_labels.append((k_paths_x[0], path[0]))
    k_paths_labels.append((k_paths_x[-1], path[1]))

# Solve the eigenvalue for each band
bands = []
for kpt in k_paths:
    H = hamiltonian(kpt, a, form_factors, g_basis, hkls)
    eigvals = np.linalg.eigvals(H).real
    eigvals.sort()
    bands.append(eigvals[:8])

bands = np.stack(bands, axis=-1)
bands -= max(bands[3]) # shift to E_f = 0

# Plot the band structure
fig = plt.figure(figsize=(8, 6))
xmin, xmax = k_paths_x[0], k_paths_x[-1]
ymin, ymax = min(bands[0])-0.1, 6
plt.xlim(xmin, xmax)
plt.ylim(ymin, ymax)

# Plot bands
for band in bands:
    plt.plot(k_paths_x, band, lw=2.0)

# Add high-symmetry labels and vertical lines
for pos_x, label in k_paths_labels:
    if label == "G": label = "$\Gamma$"
    plt.text(pos_x, ymin-1.0, label, ha='center')
    plt.axvline(x=pos_x, color="k", linestyle="--", lw=1.0)
```

```python

plt.ylabel("Energy (eV)")
plt.xticks([])
plt.tight_layout()
plt.savefig("Fig11-Si.pdf")
```

In the above Python code, we calculate and visualize the band structure of a silicon crystal using a simplified model. The main steps are outlined as follows:

1. Generate Reciprocal Lattice Vectors (hkl): The **generate_hkls** function generates all possible combinations of Miller indices (h, k, l) within a defined range. These indices represent reciprocal lattice vectors for constructing the Hamiltonian.

2. Construct the Hamiltonian: The **hamiltonian** function builds the Hamiltonian matrix (H) for each wavevector $\mathbf{k}$.

3. High-symmetry points in the Brillouin zone (Γ, X, W, L, K) are defined. And linear interpolation generates k-paths connecting these points, creating a smooth trajectory for band structure calculations.

4. Plot the Band Structure. We then shift the bands so that the maximum energy of the 3rd band aligns with the Fermi energy for a clearer comparison. The bands are plotted along the defined k-path, with labels and vertical lines marking the high-symmetry points (Γ, X, W, L, K).

The final band structure is presented in Fig. 11.10. Clearly, the results capture the essential features of the silicon band structure, showcasing the energy dispersion along the high-symmetry directions $\Gamma \to X \to W \to L \to \Gamma \to K$ within the first Brillouin zone. Comparing with the results from more expensive DFT calculation as shown in Fig. 11.7, the band energies are remarkably similar. However, our simplified pseudopotential approach use only 3 parameters with 125 plane waves, demonstrating the effectiveness of this model in reproducing key qualitative trends.

11.7. Summary

In this chapter, we introduced the electronic structure calculations for periodic systems based on Bloch's Theorem. The tight-binding approach, which utilizes linear combinations of localized atomic orbitals, was employed to provide a qualitative understanding of band structure dispersion. This approach highlights the fundamental relationship between energy levels and wave vectors, as well as the roles of on-site energies and hopping integrals in determining the band structure.

Subsequently, we transitioned to the plane wave basis, combined with the pseudopotential method, to achieve a more accurate description of the band structure. The use of plane waves ensures periodicity and simplifies the representation of electronic wavefunctions, while pseudopotentials effectively reduce the computational complexity by focusing on valence electrons.

The example of silicon served as a practical case study, illustrating how these methods can be applied to real materials. By working through these exercises, one can gain a deeper understanding of band physics, computational frameworks, and the interplay between model simplifications and numerical accuracy.

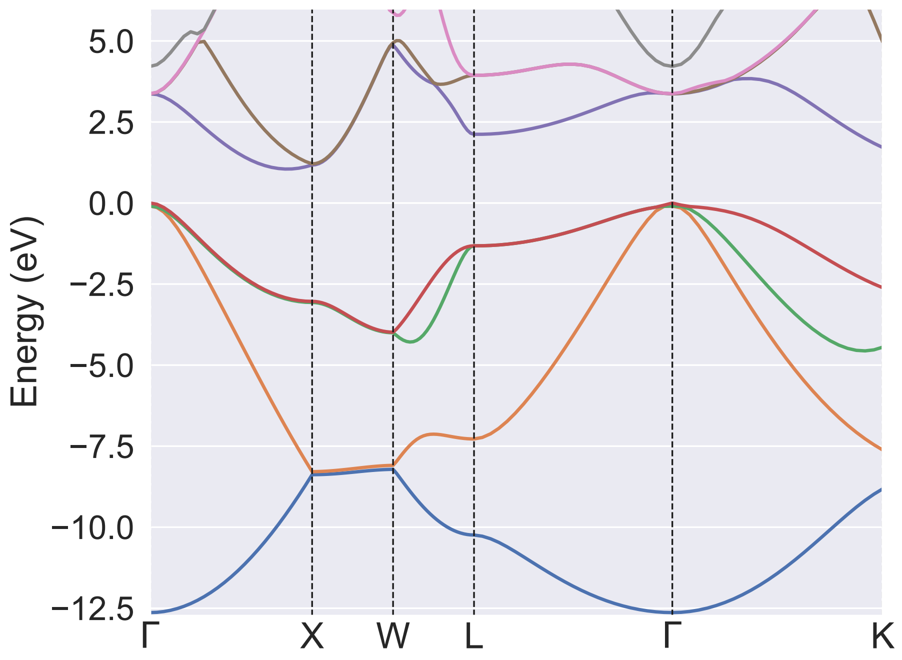


Figure 11.10: Calculated band structure of silicon, showing energy dispersion along high-symmetry directions in the Brillouin zone. (QZ: to add dosplot and codes.)

Both the tight-binding and empirical pseudopotential methods have been widely used to study the electronic structure of materials. These methods have played a crucial role in providing qualitative and semi-quantitative insights into band structures, particularly during the early development of solid-state physics. While modern, more accurate methods such as DFT have become the standard for electronic structure calculations, these low-cost approaches still hold value in specific contexts such as massive materials screening, large-scale systems and topological physics studies.

12. DFT Simulation of Crystals with Plane Waves

In the previous chapter, we explored methods for computing the ground state of many-electron systems in crystals using simplified approaches such as the tight-binding model and the empirical pseudopotential method. While these techniques provide valuable insights into electronic band structures, they often fall short in simultaneously capturing total energy and band dispersion with high accuracy.

In this chapter, we shift our focus to a more comprehensive approach—utilizing the plane-wave basis set—to investigate periodic systems in greater detail.

12.1. Pseudopotentials

For atoms with many electrons, explicitly calculating the interactions of every electron—particularly core electrons—can be computationally expensive. Pseudopotentials simplify this process by replacing the effects of the core electrons with an effective potential, allowing only the valence electrons to be explicitly considered.

From a practical perspective, the wavefunctions of core electrons oscillate rapidly near the nucleus due to the strong Coulomb attraction. Accurately representing these oscillations requires either a dense spatial grid or a large number of basis functions (such as plane waves), significantly increasing computational costs.

Pseudopotentials address this challenge by smoothing out the wavefunction in the core region while retaining accuracy in the valence region, making the calculations more computationally efficient without sacrificing essential physics.

12.1.1 Norm-Conserving Pseudopotentials

The concept of norm conservation was first introduced by Topp and Hopfield [21] in the context of empirical pseudopotentials and later extended to ionic potentials by Starkloff and Joannopoulos [22].

Norm-conserving pseudopotentials are widely used in electronic structure calculations due to their straightforward implementation and ability to accurately describe valence electrons. The core idea is to replace the core electrons with a pseudopotential that simplifies the behavior of valence electrons while satisfying the following conditions:

1. Outside the core radius r_c : The pseudo-wavefunction ψ_{pseudo} exactly matches the real wavefunction ψ_{real}.
2. Inside the core radius: Although ψ_{pseudo} differs from ψ_{real} within the core region, the integral of the probability density up to r_c (known as the **norm**) is conserved.

Mathematically, for each angular momentum l, the norm-conserving condition states:

$$\int_0^{r_c} |\psi_{\text{pseudo}}(r)|^2 r^2 dr = \int_0^{r_c} |\psi_{\text{real}}(r)|^2 r^2 dr \tag{12.1}$$

This condition ensures that the total probability density (norm) within the core region is the same for both the real and pseudo-wavefunctions.

12.1.2 The Goedecker-Teter-Hutter Pseudopotential

The Goedecker-Teter-Hutter (GTH) pseudopotential [23] is a widely-used approach in DFT simulations. It is particularly valued for its computational efficiency and smooth representation, making it well-suited for plane-wave basis sets.

The GTH pseudopotential is expressed in a separable form, which simplifies its evaluation in reciprocal space and significantly reduces the computational cost associated with non-local potentials.

Specifically, the GTH pseudopotential employs a Gaussian expansion to represent both the local potential and the non-local projectors. This formulation not only ensures smooth behavior but also enhances compatibility with plane-wave basis sets, enabling accurate and efficient simulations of periodic systems.

$$V_{\text{external}}(\mathbf{r}) = \sum_i \frac{Z_i}{\mathbf{r} - \mathbf{R}_i} \quad \rightarrow \quad V_{\text{ps}}(\mathbf{r}) = V_{\text{local}}(\mathbf{r}) + V_{\text{non-local}}(\mathbf{r}) \tag{12.2}$$

The local part is represented as a radial polynomial within a cutoff radius r_c:

$$V_{\text{local}}(r) = -\frac{Z_{\text{val}}}{r} \operatorname{erf}\left(\frac{r}{\sqrt{2}r_{\text{local}}}\right) + \exp\left[-\frac{1}{2}\left(\frac{r}{r_{\text{local}}}\right)^2\right] \sum_{i=0}^{N_g} C_i \left(\frac{r}{r_{\text{local}}}\right)^{2i}. \tag{12.3}$$

where Z_{val} is the valence electron charge, r_{local} is the distance, C_i are is the Gaussian coefficient, and N_g is number of Gaussian terms used in the expansion. In the expression, the leading term denotes the **long range interaction** with a decay of $1/r$ relation, whereas the second term carries the **short range information** in terms of Gaussian.

Conversely, the non-local part is represented as a sum of angular momentum-dependent projectors,

$$V_{\text{non-local}}(\mathbf{r}, \mathbf{r}') = \sum_{i=1}^{3} \sum_{j=1}^{3} \sum_{m=-l}^{+l} Y_{lm}(\hat{\mathbf{r}}) p_i^l(\hat{\mathbf{r}}) h_{i,j}^l p_j^l(\mathbf{r}') Y_{lm}^*(\hat{\mathbf{r}'}), \tag{12.4}$$

where Y_{lm} are the spherical hamonics, i and j are the indices of the radial projectors for a given l, h_{ij}^l is the nonlocal coupling matrix elements specific to l, and p_i^l is the radial projector functions in Gaussian:

$$p_i^l(r) = \frac{\sqrt{2} r^{l+2(i-1)} \exp\left(-\frac{r^2}{2r_l^2}\right)}{r_l^{l+(4i-l)/2} \sqrt{\Gamma\left(l + \frac{4i-1}{2}\right)}} \tag{12.5}$$

where Γ denotes the gamma function and r_l are the distance for reciproocal space.

12.1.3 Pseudopotential at the reciprocal space

With the plane wave basis, we also need to evaluate the V_{ps} at the reciprocal space ($\mathbf{G}$). The reciprocal space representation of the local pseudopotential is defined as:

$$V^{\mathrm{PS}}_{\mathrm{local}}(\mathbf{G}) = \int V^{\mathrm{PS}}_{\mathrm{local}}(\mathbf{r}) e^{-i\mathbf{G}\cdot\mathbf{r}} d^3\mathbf{r}$$

For a spherically symmetric function $V^{\mathrm{PS}}_{\mathrm{local}}(r)$, this reduces to:

$$V^{\mathrm{PS}}_{\mathrm{local}}(G) = 4\pi \int_0^\infty V^{\mathrm{PS}}_{\mathrm{local}}(r) \frac{\sin(Gr)}{Gr} r^2 dr$$

Using eq. 12.3, the Fourier transform of long range term $-Z_{\mathrm{val}}/r \times \operatorname{erf}\left(r/\sqrt{2}\right)$ is:

$$-\frac{4\pi Z_{\mathrm{val}}}{G^2} \exp\left(-\frac{G^2 r^2_{\mathrm{local}}}{2}\right)$$

For the short-range part, each term in the polynomial contributes to the Fourier transform in the form:

$$\int_0^\infty r^n e^{-ar^2} \frac{\sin(Gr)}{Gr} r^2 dr$$

Combining the Fourier transforms of the long-range and short-range components, the final expression is

$$V_{\mathrm{local}}(G) = -\frac{4\pi Z_{\mathrm{val}}}{\Omega G^2} \exp\left(-\frac{x^2}{2}\right) + \sqrt{\frac{8\pi^3}{\Omega}} r^3_{\mathrm{local}} \exp\left(-x^2\right) \times \\ [C_1 + C_2(3 - x^2) + C_3(15 - 10x^2 + x^4) + C_4(105 - 105x^2 + 21x^4 - x^6)] \tag{12.6}$$

where $x = Gr_{\mathrm{local}}$ and the polynomial terms involve factors of $(3 - x^2), (15 - 10x^2 + x^4), \ldots$. These arise from the Gaussian expansions and the Fourier transform of the radial powers $r^2, r^4, \ldots$.

In most cases, we evaluate the pseudopotential directly in the G-space and perform an inverse Fourier transform to real space only when necessary.

The nonlocal part of GTH pseudopotential at G space can be described by

$$V_{\text{non-local}}(\mathbf{G}, \mathbf{G}') = \sum_{i=1}^{3} \sum_{j=1}^{3} \sum_{m=-l}^{+l} Y_{lm}(G) p^l_i(G) h^l_{ij} p^l_j(G') Y^*_{lm}(G'), \tag{12.7}$$

where the first few projectors can be analytically derived as discussed in the original paper [24].

$$p^{l=0}_1(G) = \frac{1}{\sqrt{\Omega}} 4\sqrt{2r^3_l} \pi^{5/4} \exp[-(1/2)(Gr_l)^2], \tag{12.8}$$

$$p^{l=0}_2(G) = \frac{1}{\sqrt{\Omega}} 8\sqrt{\frac{2r^3_s}{15}} \pi^{5/4} \exp[-(1/2)(Gr_l)^2] \left[3 - (Gr_l)^2\right], \tag{12.9}$$

$$p^{l=1}_2(G) = \frac{1}{\sqrt{\Omega}} 8\sqrt{\frac{r^5_l}{3}} \pi^{5/4} \exp[-(1/2)(Gr_l)^2] G \tag{12.10}$$

12.1.4 Expression of Structural Local Potentials

For a periodic structure, the local potential $V_{\text{local}}(\mathbf{G})$ and nonlocal potential $V_{\text{non-local}}(\mathbf{G}, \mathbf{G}')$ must be modified to account for periodic boundary conditions. This is achieved by multiplying them with the structure factor $\exp(i\mathbf{G} \cdot \mathbf{R}_n)$, where $\mathbf{R}_n$ represents the positions of the atomic nuclei within the unit cell. This factor effectively imposes periodicity in the simulation.

Using eq. 12.6, the local potential for the whole structure at $\mathbf{G}$ space,

$$V^{\text{ps}}_{\text{local}}(\mathbf{G}) = \sum_{\mathbf{G}} \sum_{n} V_{\text{local}}(G) \exp(i\mathbf{G} \cdot \mathbf{R}_n). \tag{12.11}$$

The counterpart at the $\mathbf{R}$ space can be evaluated via an Inverse Fast Fourier Transform (IFFT).

$$V^{\text{ps}}_{\text{local}}(\mathbf{R}) = \text{IFFT}\left[V^{\text{ps}}_{\text{local}}(\mathbf{G})\right]. \tag{12.12}$$

To compute the nonlocal potential, we first define structural projector function

$$\beta_{ilm,n}(\mathbf{G}) = p^l_i(G) Y_{lm}(\hat{\mathbf{G}}) e^{i\mathbf{G}\cdot\mathbf{R}_n} \tag{12.13}$$

For simplicity, $\beta_{ilm,n}$ will be shortened as β_{in} in which i is the index of projector function and n is the index of atoms in the structure. When the wavefunction becomes available, we first compute the projection of the wavefunction onto each projector β_i and then evaluate the interactions,

$$V^{\text{ps}}_{\text{non-local}}(\mathbf{G}) = \sum_{i,i'} h^l_{ii'} \langle \beta_i | \psi \rangle \beta_{i'}(\mathbf{G})$$

Considering the number of atoms, the formal expression is

$$V^{\text{ps}}_{\text{non-local}}(\mathbf{G}) = \sum_{n} \sum_{i} \sum_{i'} \beta^*_{in}(\mathbf{G}) h^l_{ii'} \left[\sum_{\mathbf{G}'} \beta_{i'n}(\mathbf{G}') \psi(\mathbf{G}') \right] \tag{12.14}$$

In a practical calculation, one can pre-compute $V^{\text{ps}}_{\text{local}}$ and β_{in} before hand. $V^{\text{ps}}_{\text{non-local}}$ needs to be updated when wavefunction changes.

12.1.5 The Example of Si's GTH potential

A typical example file for silicon is shown below from Ref. [24].

```
Si GTH-PADE-q4 GTH-LDA-q4
    2    2
     0.44000000    1    -7.33610297
    2
     0.42273813    2     5.90692831    -1.26189397
                                        3.25819622
     0.48427842    1     2.72701346
```

This file includes the following parameters in arbitrary units,

- The 1st line specifies the element. The **PADE** refers to the fitting function for the exchange-correlation functional, and **q4** specifies that the pseudopotential represents 4 valence electrons (the $3s^2 3p^2$ electrons for silicon).

- The 2nd line **2 2** indicates the number of radial functions in the local part of the pseudopotential (or the Gaussian expansions for the local potential) and the number of angular momentum components l included in the non-local part of the pseudopotential.

- The 3rd line lists the local potential parameters $r_{\text{local}} = 0.44$, c_0=-7.336.

- The rest lines define two project functions. For $l = 0$ at $r_0 = 0.42273813$, it has $h^0_{11} = 90692831$, $h^0_{12} = -1.26189397$, $h^0_{22} = 3.25819622$. For $l = 1$ at r_1=0.48427842, only nonzero nonlocal coupling matrix element is h^1_{11}=2.72701346.

This example demonstrates reading and parsing the GTH parameter file, where the component parameters can be mapped to components of V_{local} and $V_{\text{non-local}}$ as shown in eqs. 12.3 and 12.4. Typically, the pseudopotential parameters depends a number of factors, such as element, XC correlation functional and number of valence electrons.

12.2. Brillouin Zone Sampling

When studying crystalline solids in DFT, we need to account for the periodic nature of the crystal lattice. The behavior of electrons in a crystal is described in terms of Bloch states, which means that their wavefunctions depend on the wavevector k in reciprocal space. The reciprocal space of a crystal is divided into regions known as Brillouin zones, and solving the DFT equations over the entire Brillouin zone is crucial to accurately capture the electronic properties of the system.

In periodic solids, the electronic properties (such as energy levels and densities) depend on the wavevector $\mathbf{k}$. Since the wavevector can take continuous values within the Brillouin zone, we cannot solve the Kohn-Sham equations for every possible $\mathbf{k}$ point. For a property F (e.g., energy, density of states or charge density), the averaged quantity can be expressed analytically by integrating the whole Brillouin zone.

$$I(\epsilon) = \frac{1}{\Omega_{\text{BZ}}} \int_{\text{BZ}} F(\epsilon)\delta(\epsilon_{n\mathbf{k}} - \epsilon) d\mathbf{k} \tag{12.15}$$

Instead, we need to sample the Brillouin zone at a discrete set of points and integrate over the zone to compute quantities like the total energy, charge density, and density of states.

$$I(\epsilon) \approx \sum_{\mathbf{k}}^{\text{BZ}} w_{i\mathbf{k}} F(\epsilon)\delta(\epsilon_{n\mathbf{k}} - \epsilon) \tag{12.16}$$

Brillouin zone sampling is typically done by choosing a grid of $\mathbf{k}$-points that represent the possible electronic states within the zone. The accuracy of the DFT calculation depends on how well the Brillouin zone is sampled. The more $\mathbf{k}$-points you use, the more accurate your results will be, but it will also increase the computational cost. To reduce the cost, it is also common to use crystal symmetry constraints to perform calculation at only the irreducible $\mathbf{k}$-points, and then multiply the weights based on its multiplicity.

The irreducible k-points in a Simple Cubic Structure.

For a simple cubic lattice, the reciprocal lattice is also simple cubic. A $3 \times 3 \times 3$ grid in the first Brillouin zone (BZ) includes $3^3 = 27$ points:

$$\begin{aligned}&(0,0,0), \quad \left(\pm\tfrac{1}{3},\, 0,\, 0\right), \quad \left(0,\, \pm\tfrac{1}{3},\, 0\right), \quad \left(0,\, 0,\, \pm\tfrac{1}{3}\right), \\ &\left(\pm\tfrac{1}{3},\, \pm\tfrac{1}{3},\, 0\right), \quad \left(\pm\tfrac{1}{3},\, 0,\, \pm\tfrac{1}{3}\right), \quad \left(0,\, \pm\tfrac{1}{3},\, \pm\tfrac{1}{3}\right), \quad \left(\pm\tfrac{1}{3},\, \pm\tfrac{1}{3}, \pm\tfrac{1}{3}\right).\end{aligned}$$

Under cubic symmetries, many of the k-points become equivalent. Hence, one can find a few *representative* points under all symmetry operations.

1. Γ: $\boldsymbol{k} = (0,0,0)$. All symmetry operation leads to the same solution. Hence it has just 1 point.
2. **Edge:** $\boldsymbol{k} = \left(\frac{1}{3},0,0\right)$. By applying all rotations and inversion, there are 6 equivalent points.
$$\left(\pm\tfrac{1}{3},0,0\right), \quad \left(0,\pm\tfrac{1}{3},0\right), \quad \left(0,0,\pm\tfrac{1}{3}\right).$$
3. **Face:** $\boldsymbol{k} = \left(\frac{1}{3},\frac{1}{3},0\right)$. Under all symmetry operations, one gets 12 points:
$$(\pm\tfrac{1}{3},\, \pm\tfrac{1}{3},\, 0), \quad (\pm\tfrac{1}{3},\, 0,\, \pm\tfrac{1}{3}), \quad (0,\, \pm\tfrac{1}{3},\, \pm\tfrac{1}{3}).$$
4. **Body:** $\boldsymbol{k} = \left(\frac{1}{3},\frac{1}{3},\frac{1}{3}\right)$. Under sign flips, 8 points belong to the same family.
$$(\pm\tfrac{1}{3},\, \pm\tfrac{1}{3},\, \pm\tfrac{1}{3}).$$

Putting it all together, we have 4 irreducible k-points in total:

$$\underbrace{1}_{\Gamma} + \underbrace{6}_{\text{edge}} + \underbrace{12}_{\text{face}} + \underbrace{8}_{\text{body}} = 27.$$

When performing band-structure or total-energy calculations, one only needs to evaluate at these 4 k-points (with appropriate weights) instead of all 27 points.

If a band is completely filled the integral can be calculated accurately using a low number of k-points (this is the case for semiconductors and insulators). If an orbital is partially occupied in a metal, one may use the Fermi-Dirac smearing function,

$$f_{i\mathbf{k}} = \frac{1}{\exp\left(\frac{\epsilon_{i\mathbf{k}}-\mu}{k_B T}\right) + 1}$$

12.3. Hamiltonian on the Plane Wave Basis

Under the planewave basis and pseudopotential assumption, we can rewrite the total energy as follows

$$E = E_{\text{kinetic}} + E^{\text{ps}}_{\text{local}} + E^{\text{ps}}_{\text{non-local}} + E_{\text{Hartree}} + E_{\text{XC}} + E_{\text{NN}} \tag{12.17}$$

And the Hamiltonian operator is

$$\hat{H} = \hat{T} + V^{\text{ps}}_{\text{local}} + V^{\text{ps}}_{\text{non-local}} + V_{\text{Hartree}} + V_{\text{XC}} \tag{12.18}$$

In the context of DFT with the plane wave basis set, we seek to solve the Kohn-Sham equation

$$H\psi_{ik}(\mathbf{r}) = \epsilon_{ik}\psi_{ik}(\mathbf{r}),$$

where $\epsilon_{i\mathbf{k}}$ and $\psi_{i\mathbf{k}}$ denote the energy and wavefunction of the ith orbital at wavevector $\mathbf{k}$.

On the plane wave basis, the wavefunction is

$$\psi_{i\mathbf{k}}(\mathbf{r}) = \frac{1}{\sqrt{\Omega}} \sum_{\mathbf{G}} c_{i,\mathbf{G}+\mathbf{k}} \exp[i\mathbf{G} + \mathbf{k} \cdot \mathbf{r}] \tag{12.19}$$

where $c_{i,\mathbf{G}+\mathbf{k}}$ are the expansion coefficients to be solved.

After knowing $\psi_{i\mathbf{k}}(\mathbf{r})$, we can update the electron density $\rho(\mathbf{r})$ by counting the orbitals that are occupied.

$$\rho(\mathbf{r}) = \sum_{\mathbf{k}} \sum_{i=1}^{N_{\text{occ}}} w_{\mathbf{k}} f_{ik} \psi^*_{i\mathbf{k}}(\mathbf{r}) \psi_{i\mathbf{k}}(\mathbf{r}) \tag{12.20}$$

12.3.1 Kinetic Operator and Energy

The kinetic operator and energy can be computed from

$$\hat{T}_{\mathbf{k}} = -\frac{1}{2}\nabla^2\psi_{\mathbf{k}} = -\frac{1}{2}|\mathbf{G}+\mathbf{k}|^2\psi_k \tag{12.21}$$

$$E_{\text{kinetic}} = -\frac{1}{2}\sum_{\mathbf{k}}\sum_{i}^{N_{\text{occ}}} \int_{\Omega} \psi^*_{i\mathbf{k}} \nabla^2 \psi_{i\mathbf{k}}(\mathbf{r}) d\mathbf{r} = w_{\mathbf{k}} f_{i\mathbf{k}} \sum_{\mathbf{G}} c^2_{i,\mathbf{G}+\mathbf{k}} |\mathbf{G}+\mathbf{k}|^2 \tag{12.22}$$

12.3.2 Pseudopotential and energy

The external energy (i.e., electron-nuclear energy under the pseudo potential) can be evaluated separately. For the local part, it can be simply evaluated at the real grid space.

$$E^{\text{ps}}_{\text{local}} = \int \rho(\mathbf{r}) V^{\text{PS}}_{\text{local}}(\mathbf{r}) d\mathbf{r} = \frac{\Omega}{N_{\text{grids}}} \sum \rho(\mathbf{r}) V^{\text{PS}}_{\text{local}}(\mathbf{r}) \tag{12.23}$$

For the nonlocal part, the evaluation can be done directly at the $\mathbf{G}$ space.

$$E^{\text{ps}}_{\text{nonlocal}} = \sum_{k} \sum_{i,j}^{\text{ilm}} \sum_{n}^{N_{\text{occ}}} w_k f_n \langle \psi_n | \beta_{ilm} \rangle h^l_{ij} \langle \beta_{jlm} | \psi_n \rangle^* \tag{12.24}$$

12.3.3 Hartree Potential and Energy

The Hartree potential satisfies Poisson's equation:

$$\nabla^2 V_{\text{Hartree}}(\mathbf{r}) = -4\pi\rho(\mathbf{r})$$

Taking the Fourier transform

$$-G^2 V_{\text{Hartree}}(\mathbf{G}) = -4\pi\rho(\mathbf{G}).$$

Hence

$$V_{\text{Hartree}}(\mathbf{G}) = \frac{4\pi\rho(\mathbf{G})}{G^2} \quad \rightarrow \quad V_{\text{Hartree}}(\mathbf{r}) = \text{IFFT}[V_{\text{Hartree}}(\mathbf{G})] \tag{12.25}$$

And the Hartree energy is

$$E_{\text{Hartree}} = \frac{1}{2}\int_{\Omega} V_{\text{Hatree}}(\mathbf{r})\rho(\mathbf{r})d\mathbf{r} = \frac{\Omega}{2\times N_{\text{grids}}}\sum \rho(\mathbf{r})V_{\text{Hartree}} \tag{12.26}$$

12.3.4 Exchange and Correlation

For the XC potential, there exist multiple choices such as LDA, GGA or hybrid forms. The simplest form is Perdew-Zunger formula [14] as described in Chapter 8 (see equations in 8.15, 8.16, 8.17, 8.18).

Since there exist many difference choices of XC functional, it is more convenient to call it separately from `libXc` [25, 26], which provides a unified interface to evaluate XC energies and potentials for various approximations, including LDA, GGA, hybridfunctional and many others.

In practice, there is a Python libarary `pylibxc`, which acts as a wrapper of `libxc` library. This approach simplifies the computation of XC functionals, enabling flexibility and extensibility. Below is an example demonstrating how to use `pylibxc` for evaluating the LDA exchange functional:

```
# https://gitlab.com/libxc/libxc
# Import pylibxc and numpy
>>> import pylibxc
>>> import numpy as np
# Build the LDA_X function
>>> func = pylibxc.LibXCFunctional("lda_x", "unpolarized")
# Create the input density
>>> inp = {"rho": np.random.random(3)}
>>> result = func.compute(inp)
# The result is a dictinary with 'zk' and 'vrho' values
>>> print(result["zk"])
[[-0.48574023]
 [-0.59986568]
 [-0.71961604]]
>>> print(result["vrho"])
[[-0.64765365]
 [-0.79982091]
 [-0.95948805]]
```

In short, one provides the electron density (plus other information if necessary) as the input to call the API (*pylibxc.LibXCFunctional*) to get the potential or energy values. Each function evaluation returns a dictionary of several items as follows,

1. "zk" (ϵ_X or ϵ_C): Represents the exchange energy density per unit volume, needed by energy evaluation.

2. "vrho" (V_X or V_C): Corresponds to the functional derivative of the exchange energy with respect to electron density, i.e., the exchange potential.

12.3.5 Conversion between Real and Reciprocal Space

When there is an update on electron density, the potential terms $V^{\text{ps}}_{\text{local}}$, V_{XC} and V_{Hartree} needs to be updated as well. To avoid the computational cost, they can be first evaluated at the **R**-space, and then the total sum is transformed back to **G**-space at once via the Fourier Transform.

For the rest potential terms such as T and $V^{\text{ps}}_{\text{nonlocal}}$, they can be directly evaluated at the **G**-space in the entire SCF process.

12.3.6 Nuclear–Nuclear Interaction

Finally, the nuclear–nuclear interaction energy in a periodic system can be obtained by Ewald summation [10]. The details about Ewald summation can be found in the Appendix C and eq. C.6.

12.4. Diagonalization

Given that many plane waves need to be used, we need to construct a large H matrix. However, we are only interested in the first few solutions of eigenvalues and eigen wavefunctions. Diagonalizing the entire H matrix can be rather expensive.

12.4.1 The Davidson Approach

In 1975, Davidson proposed an iterative algorithm for finding a few of the lowest eigenvalues and corresponding eigenvectors of a large, sparse symmetric matrix [27]. It is widely used in computational quantum mechanics and quantum chemistry, particularly for solving eigenproblems in large systems electronic structure calculations.

Instead of diagonalizing the full matrix H, Davidson's method focuses on building and diagonalizing a small subspace matrix iteratively. The method improves the subspace in each step using a correction vector based on the residual of the approximate eigenvector. This approach essentially involves the following steps,

1. Start with an initial guess for the eigenvalues ($\lambda_0 = \langle\psi|H|\psi\rangle$), where ψ is the trial wavefunction.

2. Evaluate the residual vector to measure deviations from the exact solution:
$$R = \lambda\psi - H\psi$$

3. R is first normalized to the unit vector and then preconditioned to improve convergence by reducing the high-frequency components:
$$R = \frac{R}{1 + |\mathbf{G}|^2}$$

4. The Hamiltonian (H) and Overlap (S) matrices for the expanded subspace (ψ, R) are computed separately,

$$H = \begin{pmatrix} \langle\psi|H|\psi\rangle & \langle\psi|H|R\rangle \\ \langle R|H|\psi\rangle & \langle R|H|R\rangle \end{pmatrix}, \quad S = \begin{pmatrix} \langle\psi|\psi\rangle & \langle\psi|R\rangle \\ \langle R|\psi\rangle & \langle R|R\rangle \end{pmatrix} \tag{12.27}$$

5. Solve the eigenvalue problem for the expanded subspace:

$$HC = SC\Lambda$$

and then update the $H\psi$ based on the new subspace eigenvectors:

$$\psi \to C^T\psi + C^T R$$

6. Iterate until the difference in eigenvalues between successive steps,

$$\Delta E = |\lambda^{(i+1)} - \lambda^{(i)}|,$$

falls below a predefined threshold (e.g., 10^{-6}).

12.4.2 Alternative Approaches

While the Davidson method is widely adopted for large-scale eigenvalue problems, several alternative diagonalization methods are available, such as:

1. Lanczos Method [28]: Efficient for finding a few extreme eigenvalues in sparse matrices. It constructs an orthogonal basis for the Krylov subspace and reduces the original matrix to a tridiagonal form, facilitating the extraction of eigenvalues. However, the method can suffer from numerical instability due to loss of orthogonality among the generated vectors.
2. LOBPCG (Locally Optimal Block Preconditioned Conjugate Gradient) [29]: Suitable for large Hermitian eigenproblems and often faster than Davidson. It operates by minimizing the Rayleigh quotient over a block of vectors, allowing simultaneous computation of multiple eigenpairs. The incorporation of preconditioning enhances convergence, making it suitable for large-scale problems.
3. Direct Diagonalization: This approach computes all eigenvalues and eigenvectors of a matrix simultaneously through methods such as QR decomposition. While it provides complete spectral information, the computational cost and memory requirements scale cubically with the matrix size, rendering it impractical for large matrices typically encountered in quantum mechanical simulations.

Diagonalization is often the bottleneck in modern electronic structure methods. Therefore, algorithmic improvements and parallel implementations are critical for tackling large systems effectively.

12.5. Self-Consistent Field

The SCF process is similar to what has been described in the previous chapters. For convenience, the main proceedure is outlined as follows,

1. Initial Guess for Wavefunction Ψ and Electron Density $\rho_0(\mathbf{r})$:
2. Construct the Hamiltonian H: Using the input density, construct the effective Hamiltonian that includes the following terms:

$$H = T + V^{\text{ps}}_{\text{local}} + V^{\text{ps}}_{\text{non-local}} + V_{\text{Hartree}} + V_{\text{XC}},$$

3. Solve the Kohn-Sham Equations via the diagonalization approach
4. Update Electron Density $\rho(\mathbf{r})$: Compute the new density using the occupied Kohn-Sham states, and update the total energy
5. Check for Convergence based on the change of Electron Density and total energy.

It must be noted that SCF may get stuck in some local energy minimum when the initial guess is far from the solution. To stabilize convergence, a density mixing scheme is applied. The updated density can be computed as:

$$\rho_{\text{new}} = (1-\beta)\rho_{\text{old}} + \beta\rho_{\text{out}}, \tag{12.28}$$

where β is the mixing parameter (typically 0.2–0.8).

More sophisticated methods like Pulay mixing [30] and Broyden's method [31, 32] can be used for accelerated convergence by maintaining a history of previous densities. .

12.6. A Plane Wave DFT Code for Silicon

In this section, we will continue to use diamond silicon as an example to demonstrate how to design DFT code on the plane wave basis set [33].

12.6.1 Initial Planning

Clearly, this would involve a lot of subroutines in order to make it. Similar to previous Gaussian basis code, we can predefine a few classes.

- **structure**:
  1. reads the lattice (L), element and atomic coordinates ($\mathbf{R}$);
  2. computes the volume (Ω) and reciprocal lattice (L_{rec}).
- **planewave**
  1. generates the grid points and valid $\mathbf{G}$ vectors from (E_{cut}, L_{rec});
  2. generates the initial plane wave functions $\psi_{i\mathbf{k}}$ from (N_{occ}, $\mathbf{k}$);
  3. orthonormalize the plane waves for each $\psi_{i\mathbf{k}}$
  4. convert the wavefuntion to 3d grid formats;
  5. computes the electron density $\rho(\mathbf{r})$ in 3d grid formats;
- **pseudopotential**
  1. initializes the GTH parameters (r_{local}, C_i, r_l, h_{ij}, Z_{val});
  2. evaluate $V_{\text{local}}(\mathbf{G})$ for the G vectors from eq. 12.6;
  3. get the $\beta_{ilm}(\mathbf{G})$ for the input i, l, m and the G vectors from eq. 12.13;
  4. get the real spherical harmonics (Y_{lm} for $\beta_{ilm}(\mathbf{G})$ calculations, see equations in the Appendix B)
  5. get the V_{local} for the input model and planewave grids from eq. 12.11-12.12

6. get V_{nonlocal} for the input model and planewave grids from eq. 12.14
7. get E_{local} for the whole structure from eq. 12.23
8. get E_{nonlocal} for the whole structure from eq. 12.24

- **hamiltonian**
  1. initialize the instance
  2. get the total Hamiltonian operator ($\hat{H}$) from eq. 12.18
  3. get $V^{\text{ps}}_{\text{local}}$ from the pseudopotential class (only called once);
  4. get $V^{\text{ps}}_{\text{local}}$ from the pseudopotential class;
  5. get the kinetic operator $\hat{T}$ from eq 12.21
  6. get the V_{Hartree} from eq. 12.25
  7. get the V_{XC} from eq. 8.15-8.18;
  8. get the total energy E_{total} from eq. 12.17;
  9. get the nuclear-nuclear energy E_{NN} (only called once) eq. C.6;
  10. get the kinetic energy E_{kinetic} from eq. 12.22;
  11. get the Hartree energy E_{Hartree} from eq. 12.26;
  12. get the XC energy E_{XC};
  13. Davidson's diagonalization to get reduced H from eq. 12.27
  14. Self-consistent-field (SCF) to get the converged $\Psi_{\boldsymbol{ik}}$, $\rho(\mathbf{r})$, E_{total}.

12.6.2 Structure and Plane Wave Setup

In this example, we aim to solve the ground state and compute the band structure for cubic diamond silicon with a lattice constant (a = 5.13155 Å), as introduced in the previous chapter. We set the energy cutoff (E_{cut}) to 15 Ry and use a $3 \times 3 \times 3$ k-point grid for sampling the Brillouin zone.

For the electron density, the cutoff energy is often set as twice of E_{cut} (30 Ry). As discussed earlier, a $3 \times 3 \times 3$ k-point grid contains 27 k-points in total, but symmetry reduces this number to just 4 irreducible k-points. However, the FCC lattice has a non-cubic primitive cell, hence the 4 irreducible points are changed to (0, 0, 0), (1/3, 0, 1/3), (1/3, 1/3, 1/3) and (1/3, 2/3, 1/3).

To start, we first implement and test the system related classes (structure and planewave-basis).

```python
import numpy as np
from scipy import linalg
from scipy.fft import fftn, ifftn

class Structure:
    def __init__(self, lattice, positions):
        """
        A class to represent a crystal structure.
        Args:
            lattice: array, The lattice vectors of the system.
            positions: array, The fractional atomic coordinates.
        """
        self.lattice = lattice
        self.rec_lattice = 2 * np.pi * np.linalg.inv(lattice)
        self.frac_positions = positions
        self.cart_positions = positions @ lattice
        self.volume = np.abs(np.linalg.det(lattice))

    def __str__(self):
        strs = "\nSystem Setup\n"
        strs += f"Model volume: {self.volume:.4f}\n"
        strs += "Model lattice (Bohr):\n"
        for r in self.lattice:
            strs += " ".join(f"{x:12.6f}" for x in r) + "\n"
        strs += "Model rec.lat (Bohr^-1):\n"
        for r in self.rec_lattice:
            strs += " ".join(f"{x:12.6f}" for x in r) + "\n"
        strs += "Model coordinates (frac):\n"
        for r in self.frac_positions:
            strs += " ".join(f"{x:12.6f}" for x in r) + "\n"
        strs += "Model coordinates (cart):\n"
        for r in self.cart_positions:
            strs += " ".join(f"{x:12.6f}" for x in r) + "\n"
        return strs

class PlaneWaveBasis:
    def __init__(self, model, Ecut, kpoints, kweights, occs):
        """
        Args:
            model: The structure class instance.
            Ecut: float, The energy cutoff for the plane-wave basis.
            kpoints: array, The k-points used in the calculation.
            kweights: array, The weights of the k-points.
            occs: array, The list of occupations
        """
        # System input
        self.Ecutwfc = Ecut
        self.Ecutrho = 4 * Ecut
        self.model = model
        self.occs = occs

        # Kpoints
        self.kpoints = kpoints @ self.model.rec_lattice
        self.kweights = kweights
        self.n_kpts = len(self.kpoints)

        # FFT and G_vector setup
        Gmax = 2 * np.sqrt(2 * self.Ecutwfc)
        inv_lat_t = np.linalg.inv(self.model.rec_lattice.T)
        norm = np.ceil(np.linalg.norm(inv_lat_t, axis=1) * Gmax)
        self.fx = int(norm[0])
        self.fy = int(norm[1])
        self.fz = int(norm[2])
        grids = np.array([self.fx, self.fy, self.fz], dtype=int)
        self.grids = 2 * grids + 1
        self.num_grids = np.prod(self.grids)
        self.get_g_vectors()
        self.num_gs = len(self.g_rhos)
        self.num_gws = [len(g) for g in self.g_wfcs]
        self.max_g2 = (self.g_rhos**2).sum(axis=1).max()
        self.max_g2w = [(g**2).sum(axis=1).max() for g in self.g_wfcs]


    def __str__(self):
        strs = "\nPlanewave Setup\n"
        strs += f"Cutoff Energy in planewave (Ry): {self.Ecutwfc}\n"
        strs += f"Cutoff Energy in density (Ry):   {self.Ecutrho}\n"
        strs += f"FFT Grid Size:                   {self.grids}\n"
        strs += f"Num g vectors in density:        {self.num_gs}\n"
        strs += f"Num g vectors in planewave:      {self.num_gws}\n"
        strs += f"Max g2 vectors in density:     "
        strs += f"{self.max_g2:10.4f}"
        strs += f"\nMax g2 vectors in planwwave:  "
        strs += " ".join(f"{x:10.4f}" for x in self.max_g2w)
        strs += "\n"
        strs += f"Num kpoints used:                {self.n_kpts}\n"
        for kpt, kw in zip(self.kpoints, self.kweights):
            strs += " ".join(f"{x:12.6f}" for x in kpt)
            strs += f"   weight: {kw:12.6f}\n"

        return strs

    def get_g_vectors(self):
        """
        Compute g-vectors and 3D masks for the FFT grid.
        Note we have two cutoff values for electron density
        and planewave expansions
        """
        n_kpts = self.n_kpts
        [gx, gy, gz] = self.grids

        g_rhos = []
        g_wfcs = [[] for _ in range(n_kpts)]
        g_masks_r = np.zeros([gx, gy, gz], dtype=int)
        g_masks_w = np.zeros([n_kpts, gx, gy, gz], dtype=int)

        # (0, 1, 2, .., N, N-1, N-2, .... -1)
        for i in range(gx):
            ii = i - gx if i > gx // 2 else i
            for j in range(gy):
                jj = j - gy if j > gy // 2 else j
                for k in range(gz):
                    kk = k - gz if k > gz // 2 else k
                    g = np.array([ii, jj, kk])
                    g = g @ self.model.rec_lattice
                    if np.sum(g**2) <= 2 * self.Ecutrho:
                        g_rhos.append(g)
                        g_masks_r[i, j, k] = 1
                        for l in range(n_kpts):
                            g1 = g + self.kpoints[l]
                            if np.sum(g1**2) <= 2 * self.Ecutwfc:
                                g_wfcs[l].append(g1)
                                g_masks_w[l, i, j, k] = 1
        self.g_wfcs = [np.array(g_wfc) for g_wfc in g_wfcs]
        self.g_rhos = np.array(g_rhos)
        self.g_masks_r = g_masks_r.astype(bool)
        self.g_masks_w = g_masks_w.astype(bool)
```

```python
    def orthonormalize(self, psi):
        """
        Make the wavefunction be orthonormal
        S = psi^+ psi =>  psi => S^(-1/2) psi
        """
        psi_sqrt = linalg.sqrtm(np.conj(psi) @ psi.T)
        return linalg.inv(psi_sqrt).T @ psi

    def random_guess(self):
        """
        Random wavefunction from the number of occupied states
        """
        n_states = len(self.occs)
        n_kpts = self.n_kpts
        psi_1d = [[] for _ in range(n_kpts)]

        for ik in range(n_kpts):
            n_gs = len(self.g_wfcs[ik])
            real_part = np.random.rand(n_states, n_gs)
            imag_part = np.random.rand(n_states, n_gs)
            psi = real_part + 1j * imag_part
            psi = self.orthonormalize(psi)
            psi_1d[ik] = psi
        self.psi_1d = psi_1d
        self.get_psi_3d()
        self.get_rho_r()

    def get_psi_3d(self):
        """
        Get psi in 3d grids format for a single kpoint

        Args:
            psi_1d: array, The wavefunction in 1D g-space.
            ik: int, The k-point
        """
        n_kpts = self.n_kpts
        psi_3d = [[] for _ in range(n_kpts)]
        for ik in range(n_kpts):
            psi_3d[ik] = self.get_psi_3d_single(self.psi_1d[ik], ik)
        self.psi_3d = psi_3d
        return psi_3d

    def get_psi_3d_single(self, psi_1d, ik):
        [gx, gy, gz] = self.grids
        mask = self.g_masks_w[ik]
        ns = len(psi_1d)

        psi_3d_k = np.zeros([ns, gx, gy, gz], dtype=complex)
        for i in range(ns):
            psi_3d_k[i][mask] += psi_1d[i]
        return psi_3d_k

    def get_rho_r(self):
        """
        Get electron density in real space
        """
        num_gs = len(self.g_wfcs)
        vol = self.model.volume
        rho = np.zeros(self.grids)

        for ik, kw in enumerate(self.kweights):
            for i, occ in enumerate(self.occs):
                psi_r = np.fft.ifftn(self.psi_3d[ik][i])
                psi_r *= np.sqrt(self.num_grids / vol)
                rho_r = np.real(psi_r * np.conj(psi_r))
                rho_r /= np.sum(rho_r)
                rho_r *= 2 * occ * kw * self.num_grids / vol
                rho += rho_r
        rho = np.maximum(rho, 2.22e-16)  # Avoid division by zero
        self.rho_r = rho

if __name__ == "__main__":
    # Setup random seed to ensure reproducibility
    np.random.seed(42)

    # System
    lattice = 5.13155 * np.array([[0, 1, 1], [1, 0, 1], [1, 1, 0]])
    positions = np.array([[0, 0, 0], [0.25, 0.25, 0.25]])
    model = Structure(lattice, positions)
    print(model)

    # Planewave (3*3*3 grid with symmetry reduction)
    kpoints=np.array([[0, 0, 0],
                      [1/3, 0, 1/3],
                      [1/3, 1/3, 1/3],
                      [1/3, 2/3, 1/3]])
    kweights=np.array([1., 6., 8., 12.])/27.
    occs = np.array([1, 1, 1, 1, 0, 0])
    pw = PlaneWaveBasis(model,
                        Ecut=15.0,
                        kpoints=kpoints,
                        kweights=kweights,
                        occs=occs)
    print(pw)

    # initialize random wavefunctions and electron density
    pw.random_guess()
    rho = pw.rho_r.sum() * pw.model.volume/pw.num_grids
    print(f"Total Number of electrons:      {rho}")
```

This code snippet provides a Python implementation for initializing a crystal structure and setting up a plane-wave basis for electronic structure calculations. The Structure class encodes the crystal structure's lattice, reciprocal lattice, atomic positions, and unit cell volume. These quantities can be directly accessed later. The PlaneWaveBasis Class represents the plane-wave basis set, including the energy cutoff, FFT grid, and reciprocal space G vectors. The implementation includes functionality to generate random wavefunctions and compute the electron density.

The *__str__* methods in both the Structure and PlaneWaveBasis classes are implemented to provide a human-readable summary of the object's key properties, enabling easy and informative printouts. This feature simplifies debugging, testing, and reporting by presenting the essential details of the initialized objects in a clear and structured format.

According to eq. 11.13, the number of grids $2N + 1$ in each direction is assigned. To ensure consistency with the Fast Fourier Transform algorithm, the indices have to be aligned as $(0, 1, 2, \ldots, N, N-1, \ldots, -1)$, where the 2nd half of grids represent the negative frequencies due to the discrete Fourier transform symmetry. Without this mapping, high-frequency terms (e.g., $N, N+1, \ldots$) will not be correctly interpreted, leading to incorrect transforms and aliasing artifacts.

Assuming the inputs are the silicon diamond crystal with 2 atoms in the unit cell, four valence electrons for each atom, and a total of six bands (four fully occupied and two empty), the outputs are shown below.

```
System Setup
Model volume: 270.2562
Model lattice (Bohr):
    0.000000     5.131550     5.131550
    5.131550     0.000000     5.131550
    5.131550     5.131550     0.000000
Model rec.lat (Bohr^-1):
   -0.612211     0.612211     0.612211
    0.612211    -0.612211     0.612211
    0.612211     0.612211    -0.612211
Model coordinates (frac):
    0.000000     0.000000     0.000000
    0.250000     0.250000     0.250000
Model coordinates (cart):
    0.000000     0.000000     0.000000
    2.565775     2.565775     2.565775


Planewave Setup
Cutoff Energy in planewave (Ry): 15.0
Cutoff Energy in density (Ry):   60.0
FFT Grid Size:                   [27 27 27]
Num g vectors in density:        5985
Num g vectors in planewave:      [749, 749, 757, 740]
Max g2 vectors in density:       119.9368
Max g2 vectors in planwwave:     29.9842 29.7760 29.9842 29.8176
Num kpoints used:                4
    0.000000     0.000000     0.000000   weight:     0.037037
    0.000000     0.408141     0.000000   weight:     0.222222
    0.204070     0.204070     0.204070   weight:     0.296296
    0.408141     0.000000     0.408141   weight:     0.444444
```

12.6.3 Pseudopotential Setup

Using the structural information of the system, we create a Pseudopotential class to model the ionic potential felt by valence electrons.

The implementation provided in this example is based on the HGH formalism. This pseudopotential class allows us to handle local and nonlocal potentials in both real space and reciprocal space.

```python
from scipy.special import erf, sph_harm

class PspHgh:
    """
    A class to represent a pseudopotential in the HGH form.
    Modification of the original code from DFTK.jl:
    https://github.com/JuliaMolSim/DFTK.jl
    Equations are taken from the original paper:
```

```python
    Hartwigsen, Goedecker and Hutter. Phys. Rev. B, 58, 3641, 1998

    Parameters:
        Z : float, The ionic charge.
        rloc : float, The local pseudopotential radius.
        cloc : array-like, The coefficients for the local pp
        rp : array-like, The projector radii.
        h : array-like, The projector coefficients.
    """

    def __init__(self, Z, rloc, cloc, rp, h):
        self.name = "Si"
        self.Z = Z
        self.rloc = rloc
        if len(cloc) < 4:
            self.cloc = np.pad(cloc, (0, 4 - len(cloc)), "constant")
        else:
            self.cloc = cloc
        self.lmax = len(h) - 1
        self.proj = ["s", "p", "d", "f"][:len(h)]
        self.rp = rp
        self.h = h
        self.ilm_indices = [(1, 0, 0),
                            (2, 0, 0),
                            (1, 1, -1),
                            (1, 1, 0),
                            (1, 1, 1)]
        self.num_ilms = len(self.ilm_indices)

    def __str__(self):
        strs = "\nPseudopotential Setup\n"
        strs += f"Element:                     {self.name}\n"
        strs += f"Number of electrons:         {self.Z}\n"
        strs += f"local radius:                {self.rloc:.6f}\n"
        strs += f"local coefficients:    "
        for cloc in self.cloc:
            if abs(cloc) > 0:
                strs += f"{cloc:12.6f}"
            else:
                strs += "\n"
                break
        for l in range(self.lmax+1):
            proj, rp, h = self.proj[l], self.rp[l], self.h[l]
            strs += f"Nonlocal Projector {proj}: {rp:12.6f}\n"
            strs += f"Coupling matrix\n"
            for c in h:
                strs += "                       "
                strs += " ".join(f"{x:12.6f}" for x in c) + "\n"
        return strs

    def eval_v_local_r(self, r):
        """
        Evaluate the local pseudopotential in real space, eq. (12.3)
        """
        cloc = self.cloc
        rr = r / self.rloc
        return (-self.Z / r * erf(rr / np.sqrt(2))
                + np.exp(-rr**2 / 2) * (cloc[0] +
                cloc[1] * rr**2 + cloc[2] * rr**4 + cloc[3] * rr**6))

    def eval_v_local_g(self, g):
        """
        Compute the local pseudopotential polynomial, eq. (12.6)
        """
        g = np.array(g)
        g2 = (g ** 2).sum(axis=1)
        V = np.zeros(len(g2), dtype=complex)

        # only deal with non-zero g vectors
        ids = g2 > 1e-2
        ids0 = g2 <= 1e-2
        g2 = g2[ids]

        rloc = self.rloc
        x2 = g2 * (rloc ** 2)
        Z = self.Z

        exp = np.exp(-0.5 * x2)

        term1 = -4 * np.pi * Z / g2 * exp

        P = (self.cloc[0]
             + self.cloc[1] * (3. - x2)
             + self.cloc[2] * (15. - 10. * x2 + x2**2)
             + self.cloc[3] * (105. - 105. * x2 + 21. * x2**2 - x2**3))
        term2 = np.sqrt(8.0 * np.pi **3) * rloc ** 3 * exp * P
        V[ids] = term1 + term2

        # Separately process g=0
        term0 = (2. * np.pi)**1.5 * rloc**3 * (self.cloc[0] +
                 3. * self.cloc[1] +
                 15. * self.cloc[2] +
                 105. * self.cloc[3])
        V[ids0] = 2 * np.pi * Z * rloc**2 + term0

        return V

    def eval_proj_g(self, g, i, l, vol):
        """
        Compute the projector polynomial, eq. (12.8)-(12.10)

        Args:
        i : int, The projector index.
        l : int, The angular momentum.
        """
        g1 = np.linalg.norm(g, axis=1)
        rp = self.rp[l]
        x2 = (g1 * rp)**2
        exp = np.exp(-0.5 * x2)
        prefactor = 4 * np.pi**(5 / 4) * np.sqrt(2**(l + 1) * rp**(2 * l + 3) / vol)
        common = exp * prefactor

        if [i, l] == [1, 0]:
            return common
        if [i, l] == [1, 1]:
            return common * g1 / np.sqrt(3)
        if [i, l] == [2, 0]:
            return common * 2. / np.sqrt(15.) * (3 - x2)

    def get_ylm_real(self, g, l, m):
        """
        Compute the real valued spherical harmonics, see eq (A.4).
        """
        ylms = np.zeros(len(g))
        gm = np.linalg.norm(g, axis=1) + 1e-7
        theta = np.arccos(g[:, 2] / gm)
        phi = np.arctan2(g[:, 1], g[:, 0])
        ylm = sph_harm(m, l, phi, theta)
        if m > 0:
            return np.sqrt(2) * (-1)**m * np.real(ylm)
        elif m < 0:
            return np.sqrt(2) * (-1)**m * np.imag(ylm)
        else:
            return np.real(ylm)

    def get_beta_nonlocal(self, pw):
        """
        Get beta_ilm for the given structure eq. (12.13)
        [N_kpt][N_gx, N_atom]

        Args:
            pw: planewave instance
        """
        vol = pw.model.volume
        pos = pw.model.cart_positions
        beta_ilms = [[] for _ in range(pw.n_kpts)]

        for ik in range(pw.n_kpts):
            gs = pw.g_wfcs[ik] + pw.kpoints[ik]
            sf = np.exp(1j* (gs @ pos.T))    # (ng, 3) (3, 2)
            beta_ilm = np.zeros([self.num_ilms, len(gs), len(pos)],
                                dtype=complex)

            for id, ilm in enumerate(self.ilm_indices):
                (i, l, m) = ilm
                proj = self.eval_proj_g(gs, i, l, vol)
                ylms = self.get_ylm_real(gs, l, m)
                proj *= ylms
                beta_ilm[id] = np.einsum('i,ij->ij', proj, sf)
                beta_ilm[id] *= (-1j)**l
            beta_ilms[ik] = beta_ilm
        self.beta_nl = beta_ilms

    def get_v_loc_r(self, pw):
        """
        Compute the structural V_ps_local, eq. (12.11)-(12.12)

        Args:
            pw: planewave instance
        """
        grids = pw.grids
        n_grids = pw.num_grids
        vol = pw.model.volume
```

```python
        g_masks = pw.g_masks_r
        g_vectors = pw.g_rhos
        v_loc_g = np.zeros(grids, dtype=complex)

        # get v_loc_g
        pos = (pw.model.cart_positions).T
        sf = np.exp(1j*g_vectors @ pos).sum(axis=1)
        v_loc_g_1D = self.eval_v_local_g(g_vectors)
        v_loc_g_1D *= sf / vol

        # convert to 3D
        v_loc_g[g_masks] = v_loc_g_1D.conj()

        # fft to real space
        self.v_loc_g = v_loc_g #_1D
        self.v_loc_r = np.fft.ifftn(v_loc_g).real * n_grids

    def get_v_nloc(self, pw, psi=None, ik=0):
        """
        Compute the structural V_ps_nonlocal with eq(12.14)

        Args:
            pw: planewave instance
            psi: array, The wavefunction in 1D g-space.
            ik: int, The k-point
        """
        if psi is None:
            psi = pw.psi_1d[ik] # (N_states, Ngx)

        V = np.zeros(psi.shape, dtype=complex)

        # h[i,j] * beta * < beta^* | psi>
        n_ilms = len(self.ilm_indices)
        beta = self.beta_nl[ik] # (N_ilm, Ngx, Natoms)
        # <beta|psi> => (N_ilm, Nat, Nst)
        # (N_ilm, Ngx, Nat) (Nst, Ngx)
        out = np.einsum('ijk,lj->ikl', beta, psi).conj()
        #out = np.einsum('ijk,lj->ikl', beta, psi)

        for id1 in range(n_ilms):
            (i1, l1, m1) = self.ilm_indices[id1]
            for id2 in range(n_ilms):
                (i2, l2, m2) = self.ilm_indices[id2]
                if [l1, m1] == [l2, m2]:
                    coef = self.h[l1][i1-1, i2-1]
                    tmp2 = np.einsum('ij,jk->ki', beta[id1], out[id2])
                    V += coef * tmp2.conj()
        return V

    def get_E_loc(self, pw):
        """
        Compute the structural E_ps_local, eq. (12.23)

        Args:
            pw: planewave instance
        """
        dvol = pw.model.volume / pw.num_grids
        E_loc = (self.v_loc_r * pw.rho_r).sum() * dvol
        return E_loc

    def get_E_nloc(self, pw):
        """
        Compute the structural E_ps_nlocal, eq. (12.24)

        Args:
            pw: planewave instance
        """
        E = 0
        occs = pw.occs
        for ik in range(pw.n_kpts):
            kw = pw.kweights[ik]
            n_ilms = len(self.ilm_indices)
            psi = pw.psi_1d[ik] # (N_states, Ngx)
            beta = self.beta_nl[ik] # (N_ilm, Ngx, Natoms)
            out = np.einsum('ijk,lj->ikl', beta, psi)

            for idx1 in range(n_ilms):
                (i1, l1, m1) = self.ilm_indices[idx1]
                for idx2 in range(n_ilms):
                    (i2, l2, m2) = self.ilm_indices[idx2]
                    if [l1, m1] == [l2, m2]:
                        coef = self.h[l1][i1-1, i2-1]
                        beta2 = (out[idx1] * out[idx2].conj()).real
                        beta2 = beta2 * occs[None, None, :]
                        E += kw * coef * np.sum(beta2)
        return 2*E
```

This class includes:

- Initialization: Handles input parameters for pseudopotential properties, including coefficients for local and nonlocal terms.
- Local Potential Evaluation: Computes V_{local} both in real and reciprocal space.
- Evaluates nonlocal contributions by projecting angular momentum channels onto wavefunctions.
- Calculates local and nonlocal energy contributions based on the electron density and wavefunctions.

In this class, we also need to compute the real valued spherical harmonics. To focus on the DFT implementation, we put the discussion in the Appendix B.

The outputs are shown below.

```
Pseudopotential Setup
Element:                   Si
Number of electrons:       4
local radius:              0.440000
local coefficients:       -7.336103
Nonlocal Projector s:      0.422738
Coupling matrix
                           5.906928    -1.261894
                          -1.261894     3.258196
Nonlocal Projector p:      0.484278
Coupling matrix
                           2.727013     0.000000
                          0.000000     0.000000

E_psp_local:             -9.555803
E_psp_nonlocal:           4.400774
```

12.6.4 The Hamiltonian Class

The Hamiltonian class is responsible for constructing the total energy functional and applying the Hamiltonian operator during SCF iterations. It incorporates contributions from kinetic energy, local and nonlocal pseudopotentials, Hartree energy, and exchange-correlation energy terms.

```python
class Hamiltionian:
    """
    A class to compute the hamiltonian

    Parameters:
        pw: the planewave instance
        psp: the pseudopotential instance
    """

    def __init__(self, pw, psp):
        # planewaves and pseudopotential
        self.pw = pw
        psp.get_v_loc_r(pw)
        psp.get_E_loc(pw)
        psp.get_beta_nonlocal(pw)
        self.psp = psp

        # potential terms
        self.V_ps_loc = psp.v_loc_r
        self.V_Hartree = np.zeros(pw.grids)
        self.V_XC = np.zeros(pw.grids)
        self.V_total = None

        # energie terms
        self.E_Kinetic = 0
        self.E_ps_loc = 0
        self.E_XC = 0
        self.E_Hartree = 0
        self.E_ps_nloc = 0
        self.E_total = 0
        self.E_NN = -8.3979274 # precomputed

    def __str__(self):
        strs = "\nHamiltonian"
        strs += f"\nE_Kinetic:   {self.E_Kinetic:12.6f}"
        strs += f"\nE_ps_local:  {self.E_ps_loc:12.6f}"
        strs += f"\nE_ps_nloc:   {self.E_ps_nloc:12.6f}"
        strs += f"\nE_Hartree:   {self.E_Hartree:12.6f}"
        strs += f"\nE_XC:        {self.E_XC:12.6f}"
        strs += f"\nE_NN:        {self.E_NN:12.6f}"
        strs += f"\nE_total:     {self.E_total:12.6f}"
        return strs

    def get_E_total(self):
        """
        Compute the total energy, eq. (12.17)
        """
        self.E_XC = self.get_E_XC()
        self.E_Hartree = self.get_E_Hartree()
        self.E_ps_loc = self.get_E_ps_loc()
        self.E_ps_nloc = self.get_E_ps_nloc()
        self.E_Kinetic = self.get_E_Kinetic()
        self.E_total = self.E_ps_loc + self.E_XC + self.E_Hartree
        self.E_total += self.E_ps_nloc + self.E_Kinetic + self.E_NN
        return self.E_total

    def get_H_op(self, ik=0, psi=None):
        """
        Compute the Hamiltonian operator, eq. (12.18)
        """
        if psi is None: psi = self.pw.psi_1d[ik]
        ns = len(psi)
        ngs = len(psi[0])
        mask = self.pw.g_masks_w[ik]

        # Don't update local potential in diag func
        if self.V_total is None:
            V_XC = self.get_V_XC()
            V_H = self.get_V_Hartree()
            self.V_XC = V_XC
            self.V_Hartree = V_H
            self.V_total = self.V_ps_loc.real + V_XC + V_H
        V = self.V_total

        # Local potential: V(r) => V(g)
        Vg = np.zeros([ns, ngs], dtype=complex)
        psi_3d = self.pw.get_psi_3d_single(psi, ik)
        for i, _psi in enumerate(psi_3d):
            psi_r = np.fft.ifftn(_psi)
            Vg[i] += np.fft.fftn(V * psi_r)[mask]

        # Update kinetic operator
        T = self.get_K_op(psi, ik)

        # Update nonlocal potential
        V_ps_nloc = self.psp.get_v_nloc(self.pw, psi, ik)

        # Get the H
        H = T + Vg + V_ps_nloc

        return H

    def get_K_op(self, psi_1d, ik):
        """
        Get kinetic operator in 1D g space (eq. 12.21)
        """
        k = self.pw.kpoints[ik]
        gs = self.pw.g_wfcs[ik]
        g2s = np.sum((gs + k)**2, axis=1)
        T = 0.5 * (g2s * psi_1d)
        return T

    def get_V_XC(self, backend=None):
        """
        Get Exchange-Correlation potential in 3D real space.
        Use the Perdew-Zunger (PZ81) for correlation (see eq. ).
        """
        rho = self.pw.rho_r

        if backend is not None:     # Call pylibxc
            import pylibxc

            func = pylibxc.LibXCFunctional("lda_x", "unpolarized")
            results = func.compute({"rho": rho})
            V_X, eps_X = results["vrho"], results["zk"]
            func = pylibxc.LibXCFunctional("lda_c_pz", "unpolarized")
            results = func.compute({"rho": rho})
            V_C, eps_C = results["vrho"], results["zk"]

        else:                       # from own code
            # Exchange
            eps_X = -0.75 * (3.0 * rho / np.pi) ** (1 / 3)
            V_X = (4 / 3) * eps_X

            # Correlation Potential (V_C) using PZ81
            # Compute Wigner-Seitz radius (rs)
            rs = (3 / (4 * np.pi * rho)) ** (1 / 3)
            rs = np.minimum(rs, 1e6)  # Avoid excessively large values

            # PZ81 Parameters
            Ah, Bh, Ch, Dh = 0.0311, -0.048, 0.002, -0.0116
            g, b1, b2 = -0.1423, 1.0529, 0.3334

            # Correlation energy per particle, eps_C(rs)
            eps_C = np.zeros_like(rs)
            V_C = np.zeros_like(rs)

            # High-density region (rs < 1)
            mask_h = rs < 1
            rh = rs[mask_h]
            logh = np.log(rh)
            eps_C[mask_h] = Ah * logh + Bh + Ch * rh * logh + Dh * rh
            V_C[mask_h] = Ah * logh + (Bh - Ah / 3.) + \
                            2./3. * Ch * rh * logh + \
                            (2. * Dh - Ch) / 3. * rh

            # Low-density region (rs >= 1)
            mask_l = ~mask_h
            rl = rs[mask_l]

            # Corrected formula for correlation energy and potential
            rs = np.sqrt(rl)
            ox = 1. + b1 * rs + b2 * rl
            dox = 1. + 7./.6 * b1 * rs + 4./3. * b2 * rl
            eps_C[mask_l] = g / ox
            V_C[mask_l] = eps_C[mask_l] * dox / ox

        V_XC = (V_X + V_C).reshape(self.pw.grids)
        eps_XC = (eps_X + eps_C).reshape(self.pw.grids)

        return V_XC, eps_XC
```

```python

    def get_V_Hartree(self):
        """
        Get Hartree potential in 3D real space (eq. 12.25)
        """
        mask = self.pw.g_masks_r
        gs = self.pw.g_rhos
        g2 = (gs**2).sum(axis=1)  + 1e-12
        rho_g = np.fft.fftn(self.pw.rho_r) * 4 * np.pi
        V_g = np.zeros(self.pw.grids, dtype=complex)
        V_g[mask] = rho_g[mask] / g2

        # Reset the gamma to 0
        V_g[0, 0, 0] = 0

        return np.real(np.fft.ifftn(V_g))

    def get_E_Kinetic(self):
        """
        Compute the kinetic energy, eq. (12.22)
        """
        E = 0.0
        occs = self.pw.occs
        for ik in range(self.pw.n_kpts):
            k = self.pw.kpoints[ik]
            kw = self.pw.kweights[ik]
            gs = self.pw.g_wfcs[ik]
            g2s = np.sum((gs + k)**2, axis=1)

            psi = self.pw.psi_1d[ik]
            factor = kw * occs[:, None] * g2s[None, :]
            E += ((psi.conj() * psi).real * factor).sum()

        return E

    def get_E_XC(self):
        """
        Compute the exchange-correlation energy from pylibxc
        """
        dvol = self.pw.model.volume / self.pw.num_grids
        E = (self.eps_XC * self.pw.rho_r).sum() * dvol
        return E

    def get_E_Hartree(self):
        """
        Compute the Hartree energy, eq. (12.26)
        """
        dvol = self.pw.model.volume / self.pw.num_grids
        E = 0.5 * (self.V_Hartree * self.pw.rho_r).sum() * dvol
        return E

    def get_E_ps_nloc(self):
        return self.psp.get_E_nloc(self.pw)

    def get_E_ps_loc(self):
        return self.psp.get_E_loc(self.pw)

    def diag(self, psi, ik=0):
        """
        Davidson dialgonalization
        """
        ns = len(psi)
        HX = self.get_H_op(ik, psi)

        # Initial guess eigenvalues
        eigval0 = (psi.conj() * HX).sum(axis=1).real  # HV

        # residuals R = eig*X - HX
        R = eigval0[:, None] * psi - HX  # (ns, ngs)
        residual = np.sqrt((R * R.conj()).sum(axis=1).real)
        residual = np.maximum(residual, 2e-16)  # Avoid division by 0

        for i in range(50):
            res_norm = 1.0 /residual
            R *= res_norm[:, None]

            # Precondition wavefunction based on g2 values
            R = R / (1 + (self.pw.g_wfcs[ik]**2).sum(axis=1))

            # H of R
            HR = self.get_H_op(ik, R)

            # Build H
            H1 = np.zeros([ns*2, ns*2], dtype=complex)
            if i == 0:
                H1[:ns, :ns] = psi.conj() @ HX.T
            else:
                np.fill_diagonal(H1, eigval0)

            H1[:ns, ns:] = psi.conj() @ HR.T
            H1[ns:, ns:] = R.conj() @ HR.T
            H1[ns:, :ns] = H1[:ns, ns:].conj().T

            # Build S
            S1 = np.zeros([ns*2, ns*2], dtype=complex)
            S1[:ns, :ns] = np.diag([1.+ 0.j] * ns)
            S1[:ns, ns:] = psi.conj() @ R.T
            S1[ns:, ns:] = R.conj() @ R.T
            S1[ns:, :ns] = S1[:ns, ns:].conj().T

            # Average
            H1 = 0.5 * (H1 + H1.T.conj())
            S1 = 0.5 * (S1 + S1.T.conj())
            lam_red, psi_red = linalg.eigh(H1, S1)

            # update eigvalue and psi
            eigval1 = lam_red[:ns].real
            psi = psi_red[:ns, :ns].T @ psi + psi_red[ns:, :ns].T @ R
            HX = psi_red[:ns, :ns].T @ HX + psi_red[ns:, :ns].T @ HR
            HX *= -1
            psi *= -1

            # get residual
            R = eigval1[:, None] * psi - HX  # (ns, ngs)
            residual = np.sqrt(np.einsum('ij,ij->i', R, R.conj()).real)

            # Check convergence
            d_eigval = np.abs(eigval1[:ns] - eigval0[:ns]).sum()
            #print(i, 'eigval', ik, eigval1[:4], d_eigval, residual[0])
            if d_eigval < 1e-6:
                break

            eigval0 = eigval1

        return eigval1, psi

    def scf(self, max_iter=50, beta=0.7, de_tol=1e-6):
        """
        Self-consistent field iteration

        Args:
            max_iter: int, The maximum number of iterations.
            beta: float, The mixing parameter.
            de_tol: float, The energy tolerance
        """
        n_kpts = self.pw.n_kpts
        ns = len(self.pw.psi_1d[0])
        dvol = self.pw.model.volume / self.pw.num_grids

        E = self.get_E_total()
        eigvals = np.zeros([n_kpts, ns])
        print(f"SCF: Init {E:12.8f}")

        for i in range(max_iter):
            self.rho_old = self.pw.rho_r.copy()
            self.E_old = E

            # update eigenwavefunctions
            for ik in range(self.pw.n_kpts):
                psi = self.pw.psi_1d[ik]
                eigval, psi = self.diag(psi, ik)
                self.pw.psi_1d[ik] = self.pw.orthonormalize(psi)
                eigvals[ik] = eigval

            self.pw.get_psi_3d()
            self.pw.get_rho_r()

            # mixing rho
            rho = self.pw.rho_r
            rho = rho * beta + self.rho_old * (1-beta)
            d_rho = np.sum(np.abs(rho - self.pw.rho_r)) * dvol
            self.pw.rho_r = rho

            # Update H
            V_XC = self.get_V_XC()
            V_H = self.get_V_Hartree()
            self.V_XC = V_XC
            self.V_Hartree = V_H
            self.V_total = self.V_ps_loc.real + V_XC + V_H
            E = self.get_E_total()
            dE = abs(E - self.E_old)
            print(f"SCF{i:3d} dE:{dE:.8f} E:{E:.6f} drho:{d_rho:.6f}")
            print(self)

            if dE < de_tol:
                print("\nSCF is Converged")
                break

        print(f"Final eigval {ns} states, {n_kpts} kpoints\n")
        print(eigvals.T)

if __name__ == "__main__":
    # Hamiltonian
    ham = Hamiltionian(pw, psp)

    V, eps = ham.get_V_XC()
    ham.V_XC, ham.eps_XC = V, eps
    E1 = ham.get_E_XC()

    V, eps = ham.get_V_XC(backend="pylibxc")
    ham.V_XC, ham.eps_XC = V, eps
    E2 = ham.get_E_XC()

    print(f"E_XC_from_owncode: {E1:.6f}")
    print(f"E_XC_from_pylibxc: {E2:.6f}")
```

The Hamiltonian class acts as the core computational module in DFT calculation. It integrates contributions from pseudopotentials, electron density, and exchange-correlation functionals, ensuring both numerical stability and efficiency for large systems.

In the current code, it generally implemented all required functions from the scratch except the computation of E_{NN}. E_{NN} only needs to be calculated once as long as the structure information is known. In order to focus on the DFT simulation, we put the discussion of E_{NN} in the Appendix C.

Before SCF iterations, it is useful to test the Hamiltonian using randomized wavefunctions. This preliminary test provides a baseline assessment of the initial energy state and helps identify any potential implementation issues.

In order to check LDA-XC functional implementation, we compared the E_{XC} results from our own code with `pylibxc` results as discussed previously. The results are shown as follows.

```
E_XC_from_owncode: -2.740286
E_XC_from_pylibxc: -2.740286

Hamiltonian (in Hartree)
E_Kinetic:       72.909390
E_ps_local:      -9.555803
E_ps_nloc:        4.400774
E_Hartree:        0.964267
E_XC:            -2.740286
E_NN:            -8.397927
E_total:         57.580415
```

Due to the random nature of the wavefunctions, the calculated total energy is expected to be high and non-physical. Such results can be cross-checked with other computational codes to verify consistency.

12.6.5 Test Run and Results Analysis

The final test run demonstrates the SCF procedure for a cubic diamond silicon structure with the following parameters:

- Energy cutoff (E_{cut}): 15 Ry
- K-point grid: $3 \times 3 \times 3$ reduced to 4 irreducible points
- Pseudopotential parameters specified for silicon
- SCF Convergence Tolerance: 10^{-8} Hatree in energy
- Davidson Diagonalization
- A simple mixing with β=0.6 for electron density update in SCF

```python
if __name__ == "__main__":

    np.random.seed(42)

    # System
    lattice = 5.13155 * np.array([[0, 1, 1], [1, 0, 1], [1, 1, 0]])
    positions = np.array([[0, 0, 0], [0.25, 0.25, 0.25]])
    model = Structure(lattice, positions)
    print(model)

    # Planewave (3*3*3 grid with symmetry reduction)
    kpoints=np.array([[0, 0, 0],
                      [1/3, 0, 1/3],
                      [1/3, 1/3, 1/3],
                      [1/3, 2/3, 1/3],
                      ])
    kweights=np.array([1., 6., 8., 12.])/27.
    occs = np.array([1, 1, 1, 1, 0, 0])

    pw = PlaneWaveBasis(model,
                        Ecut=15.0,
                        kpoints=kpoints,
                        kweights=kweights,
                        occs=occs,
                        )
    print(pw)
    pw.random_guess()
    rho = pw.rho_r.sum() * pw.model.volume/pw.num_grids
    print(f"Total Number of electrons:      {rho}")

    # Pseudopotential
    psp = PspHgh(Z=4, rloc=0.44000000,
                 cloc=np.array([-7.33610297, 0, 0, 0]),
                 rp=np.array([0.42273813, 0.48427842]),
                 h=np.array([[[5.90692831, -1.26189397],
                              [-1.26189397, 3.25819622]],
                             [[2.72701346, 0.00000000],
                              [0.00000000, 0.00000000]]]))
    print(psp)

    # Hamiltonian
    ham = Hamiltonian(pw, psp)
    ham.scf(200, beta=0.6, de_tol=1e-8)
```

The output is shown as follows,

```
Hamiltonian (in Hartree)
E_Kinetic:       72.909390
E_ps_local:      -9.555803
E_ps_nloc:        4.400774
E_Hartree:        0.964267
E_XC:            -2.740286
E_NN:            -8.397927
E_total:         57.580415
SCF: Init  57.58041491
SCF  0 dE:68.08150840 E_total:-10.501093 drho:2.808660

Hamiltonian (in Hartree)
E_Kinetic:        3.656224
E_ps_local:      -5.705757
E_ps_nloc:        1.866774
E_Hartree:        0.534247
E_XC:            -2.454654
E_NN:            -8.397927
E_total:        -10.501093
SCF  1 dE:1.29022749 E_total:-9.210866 drho:1.192831
SCF  2 dE:0.79025388 E_total:-8.420612 drho:0.367443
SCF  3 dE:0.31072227 E_total:-8.109890 drho:0.147906
SCF  4 dE:0.12214773 E_total:-7.987742 drho:0.058041
SCF  5 dE:0.04778611 E_total:-7.939956 drho:0.024031
SCF  6 dE:0.01878599 E_total:-7.921170 drho:0.010074
SCF  7 dE:0.00748915 E_total:-7.913681 drho:0.004305
SCF  8 dE:0.00304355 E_total:-7.910637 drho:0.001868
SCF  9 dE:0.00124751 E_total:-7.909390 drho:0.000827
SCF 10 dE:0.00053608 E_total:-7.908854 drho:0.000372
SCF 11 dE:0.00021527 E_total:-7.908638 drho:0.000170
SCF 12 dE:0.00009563 E_total:-7.908543 drho:0.000081
SCF 13 dE:0.00003921 E_total:-7.908504 drho:0.000040
SCF 14 dE:0.00001715 E_total:-7.908486 drho:0.000021
SCF 15 dE:0.00000740 E_total:-7.908479 drho:0.000012
SCF 16 dE:0.00000318 E_total:-7.908476 drho:0.000007
SCF 17 dE:0.00000141 E_total:-7.908474 drho:0.000004
SCF 18 dE:0.00000061 E_total:-7.908474 drho:0.000003
SCF 19 dE:0.00000027 E_total:-7.908474 drho:0.000002
SCF 20 dE:0.00000013 E_total:-7.908473 drho:0.000001
SCF 21 dE:0.00000005 E_total:-7.908473 drho:0.000001
SCF 22 dE:0.00000003 E_total:-7.908473 drho:0.000001
SCF 23 dE:0.00000001 E_total:-7.908473 drho:0.000000
SCF 24 dE:0.00000001 E_total:-7.908473 drho:0.000000
SCF 25 dE:0.00000000 E_total:-7.908473 drho:0.000000

SCF is Converged

Hamiltonian (in Hartree)
E_Kinetic:        3.283452
E_ps_local:      -2.598930
E_ps_nloc:        1.606884
E_Hartree:        0.636914
E_XC:            -2.438866
E_NN:            -8.397927
E_total:         -7.908473
[6 states, 4 kpoints] in Hatree
 -0.874910  -0.008945  -0.006864  -0.003463   0.194683   0.196131
 -0.737348  -0.373191  -0.178099  -0.175285   0.073554   0.169547
```

```
-0.776267   -0.387064   -0.078894   -0.076918    0.131446    0.266729
-0.641776   -0.497143   -0.313150   -0.156857    0.135196    0.441730
```

Clearly, we found the total energy quickly decreases, achieving a physical ground state energy of approximately -7.908473 Ha after 25 iterations, accompanied by the decay of density change ($d\rho$).

The final eigenvalues (6 states $\times$ 4 k-points) provide insight into the electronic structure of the system. In particular, the valence band maximum is located at (0, 0, 0) with -0.003463 Hartree, and the conduction band maximum is located at (1/3, 0, 1/3) with a value of 0.073554 Hartree, resulting in an indirect band gap of 1.0 eV. This closely reproduces the semiconducting behavior of silicon, although the band gap is underestimated compared to the experimental value (1.1 eV). This discrepancy is typical for calculations based on the LDA, which tends to underestimate band gaps due to the lack of exact exchange contributions.

It is expected that using more advanced XC functionals, such as GGA or hybrid functionals, can improve the accuracy of the computed band structure. These methods incorporate gradient corrections or a fraction of exact exchange, addressing some of the limitations inherent to LDA.

To implement improved functionals, the code needs to incorporate gradients of electron density and other advanced terms required by these functionals. This step opens pathways for studying more complex materials and properties, including excited states and response functions.

12.6.6 Limitation and Possible Extensions

While the current code successfully computes the electronic structure of a simple cubic system, it is primarily designed for instructional purposes rather than production-level simulations. Below, we outline its key limitations and suggest possible extensions to improve its accuracy, robustness, and scalability:

1. XC Functional. Only the simple LDA functionals (Perdew-Zunger) are supported in the current code. Therefore, the accuracy is limited for band structures and total energy calculations.

2. Pseudopotential choice. The current implementation uses HGH pseudopotentials, which are norm-conserving and optimized for efficiency in plane-wave basis sets. While HGH pseudopotentials are suitable for many applications, the parameters may need to be further tuned for different systems.

3. SCF Convergence. The SCF procedure may get stuck in cases with poor initial guesses or when using systems with complex symmetry.

4. Limitation of simple mixing scheme. The implementation uses a linear mixing scheme with a small mixing parameter (β). While this avoids divergence, it often leads to slow convergence. Larger β values may cause overshooting and instability.

5. Modern codes include advanced preconditioning techniques and density mixing algorithms (e.g., Pulay mixing) to accelerate convergence and avoid numerical instabilities. These features are not included in the current implementation.

6. The current implementation focuses primarily on total energy and band structures. Advanced analysis, such as density of states (DOS), projected DOS (PDOS), and charge density, are not included, despite that they can be easily implemented in Python.

7. Performance. Modern DFT codes leverage MPI-based parallelization or GPU to handle large-scale systems, significantly improving efficiency. Our implementation, however, operates on a single node, limiting scalability.

Despite its simplicity, this implementation captures the core principles of DFT and provides a foundational framework for exploring electronic structure calculations. With Python's flexibility and extensive libraries, it is well-suited for prototyping ideas, testing algorithms, and educational purposes.

Future developments can focus on extending functionalities, improving computational performance, and adopting modern algorithms to transform this code into a versatile research tool. Researchers are encouraged to experiment with extensions, optimize parallel performance, and explore new physical models to make the implementation even more powerful.

12.7. Forces and Stress Tensors

So far, we have learned how to perform total energy calculations in DFT using the plane-wave basis. However, knowing the ground-state energy alone is insufficient to determine the equilibrium structure of a material. Ideally, we also need to optimize the geometry to find the configuration that minimizes the total energy. This requires evaluating atomic forces and stress tensors to guide structural relaxation.

12.7.1 Forces on Atoms

The Hellmann-Feynman theorem provides a simple and efficient way to calculate forces on atoms in the system. For an atom at position $\mathbf{R}_I$, the force is computed as the negative derivative of the total energy with respect to its position:

$$\mathbf{F}_I = -\frac{\partial E_{\text{total}}}{\partial \mathbf{R}_I}$$

In the context of DFT, the total force consists of three contributions:

1. Electrostatic Force (Ionic Contribution): Due to the interaction between nuclei and electrons.

2. Pseudopotential Contribution: Arises from the derivative of the non-local pseudopotential terms.

3. Exchange-Correlation Contribution: Comes from variations in the exchange-correlation potential.

Explicitly, the force can be expressed as:

$$\mathbf{F}_I = \mathbf{F}_I^{\text{elec}} + \mathbf{F}_I^{\text{ps}} + \mathbf{F}_I^{\text{xc}} \tag{12.29}$$

12.7.2 Stress Tensors

The stress tensor describes the internal pressure in the system and is particularly important for lattice optimization and simulating systems under mechanical strain.

The stress tensor is computed as the derivative of the total energy with respect to a strain tensor ϵ:

$$\sigma_{\alpha\beta} = \frac{1}{\Omega} \frac{\partial E_{\text{total}}}{\partial \epsilon_{\alpha\beta}}$$

where α, β denote the Cartesian coordinates (e.g., x, y, z).

The stress tensor also includes kinetic energy, ionic interactions, Hartree potential, exchange-correlation energy, and pseudopotential contributions.

Forces and stress tensors extend DFT calculations to structural optimization, enabling accurate predictions of equilibrium geometries, elastic properties, and responses to external pressures. The readers are welcome to further enable this capability to strengthen the understanding of DFT simulation.

12.8. Summary

In this chapter, we introduced the fundamental principles and computational framework for performing DFT simulations using the plane-wave pseudopotential method. We provided a step-by-step explanation of constructing the Hamiltonian, solving the Kohn-Sham equations, and analyzing the resulting electronic band structure.

The example focused on a simple cubic diamond structure of silicon, demonstrating how to set up the pseudopotentials, define the plane-wave basis, and implement the SCF procedure to compute the ground state properties. By examining the valence and conduction bands, we highlighted the ability of DFT to describe the electronic structure and emphasized the role of exchange-correlation functionals in determining accuracy.

Despite the simplicity of this implementation, it captures the essential physics of silicon, including its semiconducting nature and band gap behavior. The example serves as a foundation for further exploration, enabling future extensions to incorporate improved functionals such as GGA and hybrid methods, as well as spin-polarized calculations and parallelization techniques.

For realistic simulations, the reader are recommended to explore widely-used planewave DFT packages such as `Quantum ESPRESSO` [34], `VASP` [35] based on Fortran, or `DFTK.jl` [36] and `PWDFT.jl` [33] based on Julia, which include more sophisticated algorithms and high-performance computing capabilities.

13. Phonon Calculation

In this chapter, we shift our focus from electronic structure to atomic motion. While the computation of atomic vibrations was previously discussed in the context of molecular dynamics simulations, our aim here is to explore vibrations from the perspective of lattice dynamics. Specifically, we delve into the fundamental concepts of phonon theory, starting with simple harmonic oscillators and progressing to more complex systems such as diatomic chains and three-dimensional crystals.

Phonons, the quantized vibrational modes of a crystal lattice, are essential for understanding a wide range of material properties, including thermal conductivity, heat capacity, and electron-phonon interactions. This chapter provides not only a theoretical foundation for the study of vibrational properties in solids but also practical numerical methods for calculating phonons in atomistic simulations. By bridging theory and computational techniques, this chapter equips the reader with tools to analyze and predict vibrational behavior in real-world materials.

13.1. A Simple Spring

In a simple spring with a force constant of k, Hooke's Law states that the force F exerted by a spring is proportional to the displacement x from its equilibrium position:

$$F = -kx$$

According to Newton's second law, the net force F acting on a mass m causes an acceleration a:

$$F = ma = m\frac{\partial^2 x}{\partial r^2}$$

Combining these two equations leads to the following differential equation that governs the motion of the spring-mass system:

$$m\frac{\partial^2 x}{\partial r^2} = -kx \tag{13.1}$$

Obviously, the general solution is a cosine function. In its most common form, the displacement $x(t)$ can be expressed as:

$$x(t) = A\cos(\omega t + \phi),$$

where A is the amplitude, ω is the angular frequency, and ϕ is the phase constant.

Substituting this solution into Eq. (13.1) gives:

$$-m\omega^2 x(t) = -kx(t) \quad \rightarrow w = \sqrt{\frac{k}{m}}$$

Thus, the vibration frequency f is related to the force constant k and the mass m as:

$$f = \frac{1}{2\pi}\sqrt{\frac{k}{m}}$$

13.2. The 1D Infinite Monoatomic Chain

Now let's consider a relatively more complex system with a infinite number of identical spring. This model, as shown in Fig. 13.1 helps in understanding a dispersion relation that connects the vibrational frequency of the system to the wavevector q. This forms the basis for understanding wave-like propagation of vibrations in a solid.

Consider a chain of identical atoms, each with mass m, connected by springs with force constant k. In this model, the atoms are spaced a distance a apart. Each atom oscillates around its equilibrium position.

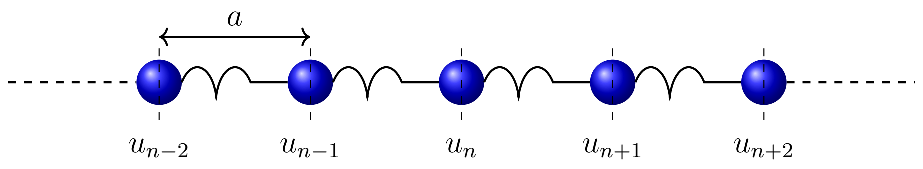


Figure 13.1: The schematic 1D monoatomic chain model.

Let $u_n(t)$ represent the displacement of the n-th atom from its equilibrium position at time t. For the n-th atom, the total force exerted on it comes from its interactions with its nearest neighbors, i.e., the $(n+1)$-th atom and the $(n-1)$-th atom.

According to Hooke's Law, the force on the n-th atom due to its neighbors is:

$$\begin{aligned} F_n &= -k\left[u_n(t) - u_{n-1}(t)\right) - k\left(u_n(t) - u_{n+1}(t)\right] \\ &= -k\left[2u_n(t) - u_{n+1}(t) - u_{n-1}(t)\right] \end{aligned}$$

Using $F_n = m\frac{d^2u_n}{dt^2}$, the equation of motion for the n-th atom becomes:

$$m\frac{d^2u_n}{dt^2} = -k\left[2u_n(t) - u_{n+1}(t) - u_{n-1}(t)\right] \tag{13.2}$$

13.2.1 Solution

Assume a wave-like solution,

$$u_n(t) = Ae^{i(nqa-\omega t)} \tag{13.3}$$

where A is the amplitude, q is the wavevector, ω is the angular frequency.

Substitute this into the equation of motion:

$$m\omega^2 Ae^{i(nqa-\omega t)} = -k\left[2Ae^{i(nqa-\omega t)} - Ae^{i((n+1)qa-\omega t)} - Ae^{i((n-1)qa-\omega t)}\right]$$

Simplifying the right-hand side:

$$kA\left(2 - e^{iqa} - e^{-iqa}\right) e^{i(nqa-\omega t)}$$

Using the identity $e^{iqa} + e^{-iqa} = 2\cos(qa)$, we obtain:

$$m\omega^2 A e^{i(nqa-\omega t)} = -kA\left[2 - 2\cos(qa)\right] e^{i(nqa-\omega t)}$$

Canceling common terms, we are left with:

$$m\omega^2 = 2k\left[1 - \cos(qa)\right] \tag{13.4}$$

So the general solution can be viewed as the following.

$$\omega^2(q) = \frac{2k}{m}\left[1 - \cos(qa)\right] = \frac{4k}{m}\sin^2(\frac{1}{2}aq) \tag{13.5}$$

This will lead to

$$\omega(q) = \pm 2\sqrt{\frac{k}{m}}\sin(\frac{1}{2}aq) \tag{13.6}$$

It suggests that ω is a function of q and behaves in a periodic manner. Given that the vibrational modes in the crystal do not depend on the sign of the wavevector q, forward and backward traveling waves have the same magnitude of oscillation frequency. Hence, we can rewrite it as

$$\omega(q) = 2\sqrt{\frac{k}{m}}\left|\sin(\frac{1}{2}aq)\right| \tag{13.7}$$

13.2.2 Choice of q for Infinite and Finite systems

Eq. 13.3 suggests that the function is periodic. Hence we can limit the choice of q within a periodic unit.

$$-\pi < qa \le \pi \quad \rightarrow \quad -\frac{\pi}{a} < q \le \frac{\pi}{a} \tag{13.8}$$

The above solution is suitable for a truly infinite long chain and q can take any values within this interval. However, we might consider a solution with a finite number of atoms N. This can be considered as a ring. So each atom still satisfy the previous motion of equation. But this introduce another constraint. That is, after N repetitions, the solution must be the same. After applying this boundary condition,

$$u_N(t) = Ae^{i(Nqa-\omega t)} = u_0(t) = Ae^{-i\omega t} \quad \rightarrow \quad e^{-iNaq} = 0$$

Combining eq. 13.8, q can only takes a series of integer values between $-\pi/a$ and π/a.

$$q = \frac{2\pi}{Na}\cdot h, \quad h = -\frac{N}{2}+1, -\frac{N}{2}+2, \ldots, \frac{N}{2} \tag{13.9}$$

This distinction between infinite and finite chains is critical in understanding the vibrational modes and their contributions to properties like heat capacity and thermal conductivity.

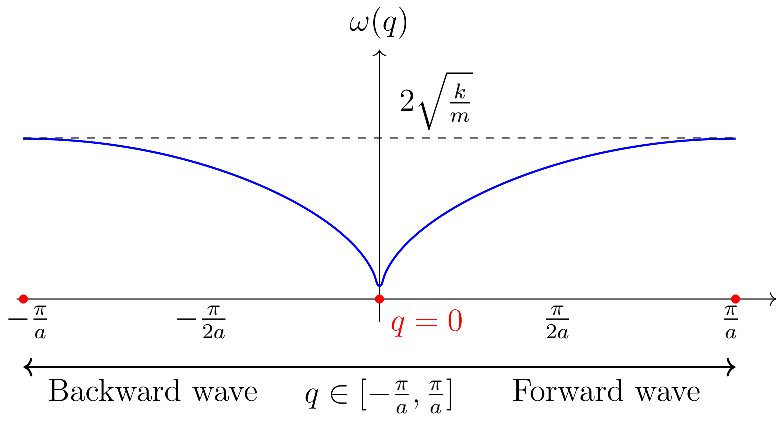


Figure 13.2: The $\omega - q$ relation for the 1D chain model.

13.2.3 Physical Insights

Fig. 13.2 illustrates the relation between ω and q. For an infinite system, q can take q can take continuous values, and the solutions form a continuous band. For a finite system, only discrete q values are allowed, which correspond to standing wave solutions that fit within the finite chain's boundary conditions.

Comparing the simple ring model and the 1D infinite chain model, we observe that the vibrations form a q-dependent dispersion relation. For small q, $\sin(qa/2) \approx qa/2$, so the frequency becomes approximately linear in q, i.e., $\omega(q) \propto q$, which is typical for acoustic phonons. At the edge of the Brillouin zone ($q = \pi/a$), the frequency reaches its maximum value.

What is the physical meaning of $u_n(q,t)$ at the small and big q values? Recall that

$$u_n(q,t) = Ae^{i(nqa-\omega t)}$$

It has both spatial and temporal characteristics.

- Spatial: The wave spreads in space with a wavelength $\lambda = 2\pi/q$, controlled by the wavevector q.
- Energy: The energy of the wave is governed by the angular frequency ω, which depends on q through the dispersion relation.

While $\lambda(q)$ and $E(\omega)$ are separately controlled, the dispersion relation ties q and ω together:

At $q = 0$, all u_n take the form of Ae^{iwt}, meaning that all atoms are collectively oscillate in the same phase with an infinitely long wavelength. In this case, the effective force constant (k) becomes zeros since the collective translation dose not change the total energy and each atom has a zero net force. This results in a zero frequency. When q is increased from 0 to a small value, the wavelength becomes smaller, and one thus expect to see an increasing of effective force constant and frequencies. This kind of wave is similar to a low-frequency sound wave propagating through the material.

Large q corresponds to short wavelengths ($\lambda \sim a$), where the wavelength approaches the interatomic spacing. At $q = \pi/a$, u_n and u_{n+1} has a different of half period. Hence, the atoms oscillate completely out of phase with their neighbors, meaning the displacement of one atom is maximally opposed by the displacement of its adjacent atoms. In this mode, each atom experiences the full restoring force from the two springs ($2k$) attached to it. This constructive addition of forces leads to the doubling of the restoring force and a maximum frequency of $\omega = 2\sqrt{k/m}$.

The behavior of $u_n(q,t)$ reveals the interplay between spatial characteristics (λ) and energy (ω). This understanding forms the foundation for analyzing phonon behavior in crystals and their contribution to thermal and electrical properties.

13.3. The 1D Diatomic Chain Model

Let us now analyze a more complex 1D chain model consisting of two alternating types of atoms, A and B, with masses m_A and m_B, respectively. As shown in Fig. 13.3 These atoms are connected by springs with a force constant k, and the distance between neighboring atoms is a.

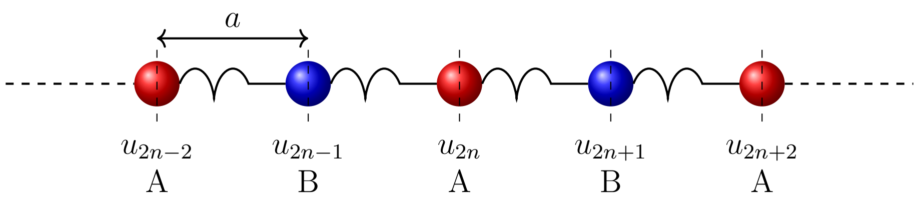


Figure 13.3: The schematic 1D diatomic chain model.

13.3.1 Equation of Motions

The displacement of atom A in the $2n$th position is denoted by $u_{2n}(t)$, and the displacement of atom B is $u_{2n+1}(t)$. And the forces on each atom come from the interactions with neighboring atoms. Using Hooke's law and Newton's second law, the equations of motion for atoms A and B are:

$$m_A \frac{d^2 u_{2n}}{dt^2} = -k\left(u_{2n} - u_{2n-1}\right) - k\left(u_{2n} - u_{2n+1}\right)$$
$$m_B \frac{d^2 u_{2n+1}}{dt^2} = -k\left(u_{2n+1} - u_{2n}\right) - k\left(u_{2n+1} - u_{2n+2}\right)$$

13.3.2 Solutions

We again assume wave-like solutions for the displacements of atoms A and B:

$$u_{2n}(t) = Ae^{i[2nqa-\omega t]}$$
$$u_{2n+1}(t) = Be^{i[(2n+1)qa-\omega t])}$$

Substituting these into the equations of motion, we get two coupled equations:

$$-m_A\omega^2 Ae^{i[2nqa-\omega t]} = -2kAe^{-i[2nqa-\omega t]} + kB\left(e^{iqa} - e^{iqa}\right)e^{i(2nqa-\omega t)}$$
$$-m_B\omega^2 Be^{i([2n+1]qa-\omega t)} = -2kBe^{-i[(2n+1)qa-\omega t]} + kA\left(e^{iqa} - e^{iqa}\right)e^{i[(2n+1)qa-\omega t]}$$

Similar to the previous 1D chain model, we simplify these to:

$$-m_A\omega^2 A = -2kA + kB\left(2\cos(qa)\right)$$
$$-m_B\omega^2 B = -2kB + kA\left(2\cos(qa)\right)$$

The above equations can be rewritten in matrix form, representing the dynamical matrix of the system:

$$\begin{pmatrix} m_A\omega^2 & 0 \\ 0 & m_B\omega^2 \end{pmatrix}\begin{pmatrix} A \\ B \end{pmatrix} = k\begin{pmatrix} 2 & -2\cos(qa) \\ -2\cos(qa) & 2 \end{pmatrix}\begin{pmatrix} A \\ B \end{pmatrix} \tag{13.10}$$

Reorganizing:

$$\begin{pmatrix} m_A\omega^2 - 2k & 2k\cos(qa) \\ 2k\cos(qa) & m_B\omega^2 - 2k \end{pmatrix}\begin{pmatrix} A \\ B \end{pmatrix} = 0$$

This is a standard **eigenvalue problem**. For non-trivial solutions $(A, B \neq 0)$, the determinant of the matrix must vanish:

$$\det\begin{pmatrix} m_A\omega^2 - 2k & 2k\cos(qa) \\ 2k\cos(qa) & m_B\omega^2 - 2k \end{pmatrix} = 0$$

Expanding the determinant:

$$\left(m_A\omega^2 - 2k\right)\left(m_B\omega^2 - 2k\right) - \left(2k\cos(qa)\right)^2 = 0 \tag{13.11}$$

Simplify:

$$\left(m_A\omega^2 - 2k\right)\left(m_B\omega^2 - 2k\right) = 4k^2\cos^2(qa)$$

Expanding and isolating ω^2:

$$m_A m_B\omega^4 - \left(2k(m_A + m_B)\right)\omega^2 + 4k^2\left(1 - \cos^2(qa)\right) = 0$$

This is a quadratic equation in ω^2:

$$\omega^4 + \frac{2k(m_A + m_B)}{m_A m_B}\omega^2 + \frac{4k^2}{m_A m_B}\sin^2(qa) = 0$$

Hence, the final solution is

$$\omega(\pm) = \sqrt{\frac{k(m_A + m_B) \pm k\sqrt{(m_A + m_B)^2 - 4m_A m_B\sin^2(qa)}}{m_A m_B}}. \tag{13.12}$$

Given the factor of $Ae^{i[2nqa - \omega t]}$, we see that the

$$-\pi < 2aq \leq \pi \quad \rightarrow \quad \frac{\pi}{2a} < q \leq \frac{\pi}{2a} \tag{13.13}$$

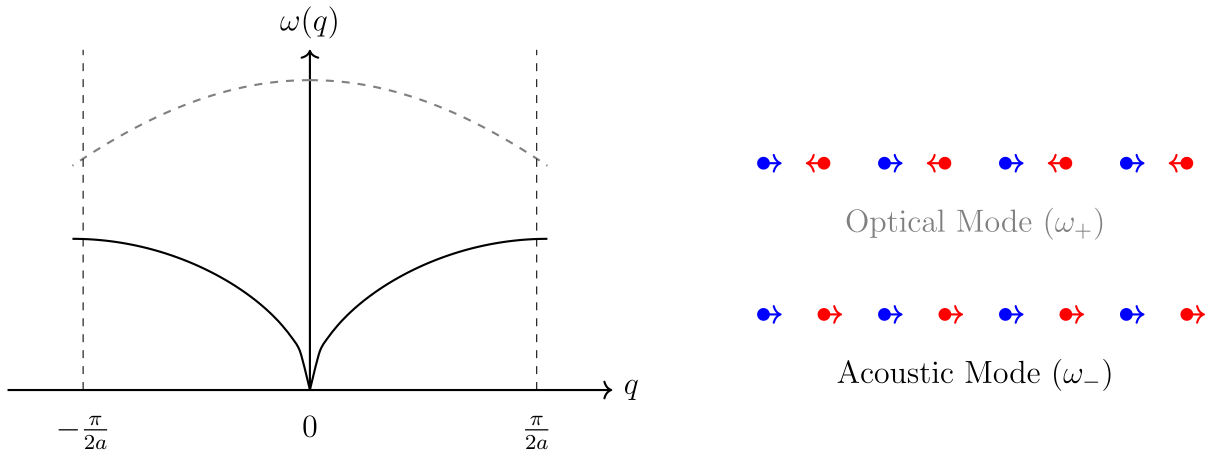


Figure 13.4: Dispersion relation for acoustic and optical modes (left) and schematic vibrations (right).

13.3.3 Dispersion Relation and Mode Analysis

Fig. 13.4 illustrates the relation between ω and q according to eq. 13.12. In the diatomic chain model, two distinct vibrational modes arise due to the existence of 2 distinguishable atoms in the unit cell. The first solution (ω_-) is very similar to that in a 1D atomic chain model, which is called **acoustic mode**, in which the ω increases from 0 to $q = \pm\pi/(2a)$. The second solution ω_+, called **optical mode**, behaves differently. It has a maximum at $q = 0$ and then decreases to $q = \pm\pi/(2a)$.

In order to understand their motions, let us evaluate the B/A ratio in both solutions at $q = 0$,

$$m_A\omega^2 A = 2k\left(A - B\cos(qa)\right) \rightarrow \frac{B}{A} = \frac{m_A\omega^2 - 2k}{-2k\cos(qa)}.$$

$$\omega_\pm^2(q=0) = \frac{k(m_A+m_B) \pm k(m_A+m_B)}{m_A m_B} \tag{13.14}$$

Hence

$$\left(\frac{B}{A}\right)_- = \frac{0-2k}{-2k} = 1,$$
$$\left(\frac{B}{A}\right)_+ = \frac{2k(m_A+m_B)/m_B - 2k}{-2k} = -\frac{m_A}{m_B}.$$

Therefore, the ω_- mode corresponds to the case where the atoms A and B oscillate almost in phase with each other, meaning their displacements are nearly synchronized. This type of mode mimics lattice oscillate in phase or nearly in phase, thus related to sound velocity and elastic constants. The behavior follows the mode as analyzed in the monoatomic chain model.

In the ω_+ mode, atoms A and B oscillate out of phase, meaning when A moves to the left, B moves to the right, and vice versa. At $q = 0$, the optical mode describes a uniform out-of-phase oscillation of all atoms: (1) atom A moves in one direction, and atom B moves in the opposite direction; (2) the displacement difference between neighboring atoms is maximized, resulting in the strongest restoring force. When q becomes nonzero, the wave introduces spatial variation in the relative displacements, reducing the restoring force and thus the frequency (see Fig. 13.4).

According to the left panel of Fig. 13.4, the acoustic and optical branches are at their maximum and minimum when $q = \pi/2a$,

$$\omega_{\text{acoustic\_max}} = \sqrt{\frac{4k}{m_A + m_B}}, \quad \omega_{\text{optical\_min}} = \sqrt{\frac{2k(m_A + m_B)}{m_A m_B}}$$

If $m_A = m_B = m$, both values are $\sqrt{2k/m}$ and the gap should be zero. Obviously, this gap arises due to the difference in mass between the two alternating atoms (m_A and m_B) and the interaction strength (k). This energy difference is a measure of how much energy is required to excite optical phonons compared to acoustic phonons.

To help the readers to understand the behavior, one can vary the following script to visualize the dispersions by playing with the parameters.

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

# Parameters in arbitary unit
k = 1.0      # Force constant
mA = 1.0     # Mass of atom A
mB = 2.0     # Mass of atom B
a = 1.0      # Lattice constant

# Dispersion relation for the acoustic and optical phonon modes
def q2omega(q, k, mA, mB, a):
    term1 = mA + mB
    term2 = np.sqrt((mA + mB)**2 - 4 * mA * mB * np.sin(q*a)**2)
    omega_acoustic = np.sqrt(k*(term1 - term2) / (mA * mB))
    omega_optical = np.sqrt(k*(term1 + term2) / (mA * mB))
    return omega_acoustic, omega_optical

# Generate q-values in the first Brillouin zone
q_values = np.linspace(-np.pi/a/2, np.pi/a/2, 100)

# Compute the corresponding phonon frequencies
omega_acoustic_values = []
omega_optical_values = []
for q in q_values:
    omega_acoustic, omega_optical = q2omega(q, k, mA, mB, a)
    omega_acoustic_values.append(omega_acoustic)
    omega_optical_values.append(omega_optical)

# Plot the acoustic and optical phonon dispersion relations
plt.plot(q_values, omega_acoustic_values, label='Acoustic Mode')
plt.plot(q_values, omega_optical_values, label='Optical Mode')
plt.xlabel(r'Wavevector $q$')
plt.ylabel(r'Angular frequency $\omega(q)$')
plt.title('Dispersion Relations in 1D Diatomic Chain')
plt.legend()
plt.show()
```

13.3.4 The Dynamical Matrix Approach

The method discussed in the previous section provides a straightforward way to analyze the system through two linear equations. However, it can also be formulated in a more systematic and generalized manner using the concept of the **Dynamical Matrix**.

From eq. 13.12, let us rewrite the equations of motion in matrix form :

$$k\begin{pmatrix} 2 & -2\cos(qa) \\ -2\cos(qa) & 2 \end{pmatrix}\begin{pmatrix} A \\ B \end{pmatrix} = \begin{pmatrix} m_A\omega^2 & 0 \\ 0 & m_B\omega^2 \end{pmatrix}\begin{pmatrix} A \\ B \end{pmatrix}$$

Dividing each row by m_A and m_B, respectively, yields:

$$\frac{k}{m_A m_B}\begin{pmatrix} 2m_B & -2m_B\cos(qa) \\ -2m_A\cos(qa) & 2m_A \end{pmatrix}\begin{pmatrix} A \\ B \end{pmatrix} = \omega^2\begin{pmatrix} A \\ B \end{pmatrix}.$$

This matrix equation allows us to define the **Dynamical Matrix** $D(q)$, which characterizes the interactions between atoms A and B in the lattice:

$$D(q) = \frac{k}{m_A m_B}\begin{pmatrix} 2m_B & -2m_B\cos(qa) \\ -2m_A\cos(qa) & 2m_A \end{pmatrix} \tag{13.15}$$

The determinant of $D(q)$ is

$$\left(\frac{k}{m_A m_B}\right)^2\left(2m_A m_B - 2m_A\cos(qa)2m_B\cos(qa)\right) = \frac{4k^2}{m_A m_B}\sin^2(qa) \tag{13.16}$$

To compute the vibrational frequencies of the system, we solve the corresponding **eigenvalue problem**:

$$D(q)\begin{pmatrix} A \\ B \end{pmatrix} = \omega^2\begin{pmatrix} A \\ B \end{pmatrix}$$

The **dynamical matrix** is a key concept in lattice dynamics and phonon theory. It arises in the context of solving the equations of motion for atoms in a crystal lattice, especially when dealing with harmonic vibrations in the lattice, such as in the 1D chain model with two types of atoms (diatomic chain). The dynamical matrix relates the forces on atoms to their displacements and encapsulates the vibrational properties of the system. Using the dynamical matrix, one expect to systematically solve the vibrations for more complex systems.

13.4. Extension to Realistic Systems

To compute phonon dispersions for a realistic 3D crystal, we need to extend the concepts from the 1D chain to a 3D lattice, considering all the interactions between atoms in the crystal unit cell.

In a 3D crystal, the phonon dispersion relation tells us how the phonon frequencies (or energies) depend on the wavevector **q** in different directions of the Brillouin zone. If there are N atoms in the unit cell, each atom has three degrees of freedom (x, y, z). Therefore, the system has $3N$ degrees of freedom, leading to $3N$ phonon modes at each **q** vector. The first 3 low-frequency vibrations where the entire unit cell moves in phase are called acoustic phonon modes, whereas the rest $3N$-3 higher-frequency modes where the atoms in the unit cell vibrate relative to each other are called optical phonon modes.

To compute these modes, we essentially need to obtain a dynamical matrix in 3D similar to the case of the 1D diatomic model like eq. 13.15.

Consider a crystal where atoms are displaced from their equilibrium positions due to vibrations. Let us first consider atoms in one unit cell with the lattice vector of $\mathbf{R}$. For the atom i in the direction of α, the force should be related to the sum of spring forces due to the displacement of other atoms. We first consider each displacement u of atom j in the direction β. The total force act on $u_{i\alpha}$ is the sum of all neighboring displacements via a force constant $\Phi^{ij}_{\alpha\beta}$. It can be expressed as

$$m_i \ddot{u}_{i\alpha}(\mathbf{R}, t) = -\sum_{j,\beta} \Phi^{ij}_{\alpha\beta} u_{j\beta}(\mathbf{R}, t) \tag{13.17}$$

Now we also consider the atoms from other neighboring unit cells. Hence, there is a need to sum over all lattice vectors $\mathbf{R}'$.

$$m_i \ddot{u}_{i\alpha}(\mathbf{R}) = -\sum_{j,\beta,\mathbf{R}'} \Phi^{ij}_{\alpha\beta}(\mathbf{R} - \mathbf{R}') u_{j\beta}(\mathbf{R}'), \tag{13.18}$$

According to the Bloch Theorem, all displacement should satisfy,

$$u(\mathbf{R}) = u(0) e^{ik\cdot\mathbf{R}} \tag{13.19}$$

Hence, we assume $u_{i\alpha}(\mathbf{R})$ can be written as plane waves, consistent with the periodicity of the lattice:

$$u_{i\alpha}(\mathbf{R}, t) = \frac{1}{\sqrt{m_i}} e^{i(\mathbf{q}\cdot\mathbf{R} - \omega t)} \, \tilde{u}_{i\alpha}(\mathbf{q}),$$
$$u_{j\beta}(\mathbf{R}', t) = \frac{1}{\sqrt{m_j}} e^{i(\mathbf{q}\cdot\mathbf{R}' - \omega t)} \, \tilde{u}_{j\beta}(\mathbf{q}),$$

Substituting this into the equation of motion, we get:

$$-\omega^2 \tilde{u}_{i\alpha}(\mathbf{q}) = \sum_{j,\beta} \sum_{\mathbf{R}'} \Phi^{ij}_{\alpha\beta}(\mathbf{R} - \mathbf{R}') \frac{1}{\sqrt{m_i m_j}} e^{i\mathbf{q}\cdot(\mathbf{R}' - \mathbf{R})} \tilde{u}_{j\beta}(\mathbf{q}). \tag{13.20}$$

In the middle term, it is only related to $\mathbf{R}' - \mathbf{R}$, we can simplify it to $\mathbf{R}$, and define it as the dynamic matrix.

$$D^{ij}_{\alpha\beta}(\mathbf{q}) = \frac{1}{\sqrt{m_i m_j}} \sum_{\mathbf{R}} \Phi^{ij}_{\alpha\beta}(\mathbf{R}) e^{i\mathbf{q}\cdot\mathbf{R}} \tag{13.21}$$

With the introduction of D, The final expression becomes

$$-\omega^2 \tilde{u}_{i\alpha}(\mathbf{q}) = D \sum_{j,\beta} \tilde{u}_{j\beta}(\mathbf{q}). \tag{13.22}$$

Group all equations to the matrix form:

$$\mathbf{D}(\mathbf{q})\tilde{\mathbf{u}} = \omega^2 \tilde{\mathbf{u}}. \tag{13.23}$$

Solving this eigenvalue problem gives the squared frequencies ω^2 and the eigenvectors $\tilde{\mathbf{u}}$, which represent the vibration modes.

13.4.1 Revisiting the Diatomic Chain Model

To understand the construction of the dynamical matrix $D(q)$, let us revisit the diatomic chain model. In this system, two atoms (A and B) are connected by a spring with an equilibrium length of a. The potential energy depends on the deviation of the relative displacement $(u_A - u_B)$ from a, where u_A and u_B represent the displacements of atoms A and B from their equilibrium positions, respectively. The total energy after displacements can be expressed as,

$$U = \frac{1}{2}k\sum_n \left[(u_{An} - u_{Bn})^2 + (u_{Bn} - u_{A,n+1})^2\right]$$

To construct the force constant matrix Φ, we compute the second derivatives of E with respect to u_A and u_B at the n-th cell.

For atom A

$$\Phi_{AA}^{nn} = \frac{\partial^2 U}{\partial u_{An}^2} = \frac{\partial^2}{\partial u_{An}^2}\left[\frac{1}{2}k(u_{An} - u_{Bn})^2 + \frac{1}{2}k(u_{An} - u_{B,n-1})^2\right] = 2k$$
$$\Phi_{BB}^{nn} = \frac{\partial^2 U}{\partial u_{Bn}^2} = \frac{\partial^2}{\partial u_{Bn}^2}\left[\frac{1}{2}k(u_{Bn} - u_{An})^2 + \frac{1}{2}k(u_{Bn} - u_{A,n+1})^2\right] = 2k$$
$$\Phi_{AB}^{nn} = \frac{\partial^2 U}{\partial u_{An}\partial u_{Bn}} = \frac{\partial^2}{\partial u_{An}\partial u_{Bn}}\left[\frac{1}{2}k(u_{Bn} - u_{An})^2\right] = -k$$
$$\Phi_{BA}^{nn} = \frac{\partial^2 U}{\partial u_{An}\partial u_{Bn}} = \frac{\partial^2}{\partial u_B \partial u_A}\left[\frac{1}{2}k(u_{Bn} - u_{An})^2\right] = -k$$

The force constant matrix Φ at the same unit cell ($R = 0$) becomes:

$$\Phi(0) = \begin{pmatrix} 2k & -k \\ -k & 2k \end{pmatrix}.$$

This can also be easily understood by simply counting the springs for each atom. For atom A in the current cell, it is connected to two B neighbors (one is in the same cell, the other is in the neighboring cell), so Φ_{AA} is $2k$ that counts for these two springs, and Φ_{AB} is -k when counting the contribution due to the atom B in the same cell, and Φ_{AB} = -$ke^{-iq\times 2a}$. Use the same reasoning, we get Φ_{BB} is $2k$, and Φ_{BA} has -k due to contribution of atom A in the same cell, and -$ke^{iq\times 2a}$ due to the contribution of atom A in the neighboring cell.

Thus, the force constant matrix from neighboring cells is:

$$\Phi_{\text{neighbors}} = \begin{pmatrix} 0 & -ke^{-i2qa} \\ -ke^{i2qa} & 0 \end{pmatrix}.$$

And the total dynamical matrix is the sum of the contributions from the same unit cell and neighboring cells, normalized by the atomic masses:

$$D(q) = \frac{1}{\sqrt{m_A m_B}}\left(\Phi(0) + \Phi_{\text{neighbors}}\right).$$

After combining all terms:

$$D(q) = \begin{pmatrix} 2k/m_A & -2k(1+e^{2iqa})/\sqrt{m_A m_B} \\ -k(1+e^{-2iqa})/\sqrt{m_A m_B} & 2k/m_B \end{pmatrix}$$
$$= \frac{k}{m_A m_B}\begin{pmatrix} 2m_B & -2\sqrt{m_A m_B}(1+e^{2iqa}) \\ -2\sqrt{m_A m_B}(1+e^{-2iqa}) & 2m_A \end{pmatrix}$$

Although this representation looks different from Eq. 13.15, they share the same trace and determinant.

The trace is the sum of diagonal elements as follows

$$\frac{2k}{m_A} + \frac{2k}{m_B}.$$

The determinant is given by:

$$\left(\frac{2k}{m_A}\right)\left(\frac{2k}{m_B}\right) - \left(-\frac{2k}{\sqrt{m_A m_B}}\right)^2 (1+e^{iq2a})(1+e^{-iq2a}) = \frac{4k^2}{m_A m_B}\sin^2(qa).$$

Since both representations share these invariants, they yield the same eigenvalues, ω^2, after diagonalization. This confirms that the two forms are mathematically equivalent.

13.4.2 Special Case at q=0

Another important observation is that when $|\mathbf{q}|$=0, D takes the following form after normalization

$$D(0) = \begin{pmatrix} 1 & -1 \\ -1 & 1 \end{pmatrix} \tag{13.24}$$

This can be generalized to any higher dimensional system. For a given row i, the sum of all elements D_{ij} represents the net force acting on atom i due to displacements of all other atoms. At $q = 0$, the sum of these forces must be zero because of the symmetry and force balance.

$$\sum_j D_{ij}(q=0) = 0, \quad \text{and by symmetry,} \sum_i D_{ij}(q=0) = 0.$$

The above conditions ensure that the system exhibits translational invariance, meaning the lattice remains stable under uniform translation. This invariance corresponds to the presence of acoustic phonon modes with zero frequency at $q = 0$.

This structure can be generalized to any higher-dimensional system. In $D(0)$, the rows (and columns) sum to zero, reflecting the absence of net restoring forces under uniform displacements. Hence, the q=0 matrix always includes zero eigenvalues corresponding to the translational acoustic modes. For a 1D system, there is one such mode; for a 3D system, there are three, corresponding to translations along x, y, and z. The remaining eigenvalues in $D(\mathbf{q} = \{0, 0, 0\})$ correspond to optical modes (if applicable) or nonzero restoring forces due to relative displacements.

This feature can also be helpful to check if your numerical code implementation is correct.

13.4.3 Application to the 3D System

One can follow the similar procedure to compute the dynamic matrix of a 3D system. After D is known, one then needs diagonalize it to obtain the phonon frequencies $\omega(\mathbf{q})$ for each mode as well as the eigenvector that denotes the vibrational directions.

Below are the steps to compute phonon in a 3D Crystal.

1. Get Atomic Positions $[N,\ 3]$: Obtain the positions of atoms in the unit cell and lattice vectors that define the crystal structure.
2. Compute $\sum_{\mathbf{R}} \Phi^{ij}_{\alpha\beta}$ $[3N, 3N, L]$: These describe the interaction between atoms and can either be obtained from ab initio calculations (using DFT) or from experimental data. The force constants can be represented as a matrix that relates atomic displacements to forces. The most straightforward way to displace each atom's x, y, z with a small amount and then compute the constant via the numerical gradient. Given that atoms in a crystal are related by symmetry operation, one can take the advantage of symmetry to reduce the number of displacements.
3. Construct $D^{ij}_{\alpha\beta}(\mathbf{q})$ $[3N, 3N]$. For each $\mathbf{q}$ in the Brillouin zone, construct the dynamical matrix $D(\mathbf{q})$ using the force constants.
4. Diagonalize $D^{ij}_{\alpha\beta}(\mathbf{q})$ $[3N, 3N]$. The eigenvalues of the matrix give the squared frequencies $\omega^2(\mathbf{q})$, and the eigenvectors describe the polarization vectors (atomic displacements) for the phonon modes.
5. Compute the $\omega(q)$ dispersion.

Clearly, computing $\sum_{\mathbf{R}} \Phi^{ij}_{\alpha\beta}(\mathbf{R})$ is a key to construct the dynamic matrix. In a real system, $\sum_{\mathbf{R}} \Phi^{ij}_{\alpha\beta}(\mathbf{R})$ needs to be calculated numerically. Namely, one perform a small displacement on each degree of freedom and then evaluate the 2nd derivative on other degrees in the same and neighboring unit cells. To ensure a numerical stability, it needs to include as many neighboring cells as possible.

13.4.4 Phonon Density of States

In addition to the phonon dispersion relation $\omega(\mathbf{q})$, which provides the vibrational frequencies as a function of wavevector $\mathbf{q}$, it is essential to understand the phonon density of states (DOS). The phonon DOS describes the distribution of vibrational states in energy (or frequency) space and plays a crucial role in understanding thermal and transport properties of materials, such as heat capacity and thermal conductivity.

The phonon DOS, $g(\omega)$, is effectively a histogram of the vibrational states across all modes and $\mathbf{q}$-points weighted by their frequencies. Mathematically, it is expressed as:

$$g(\omega) = \frac{1}{N_q} \sum_{\mathbf{q},j} \delta(\omega - \omega_j(\mathbf{q})), \tag{13.25}$$

where N_q is the total number of $\mathbf{q}$-points sampled in the Brillouin zone, $\delta(\omega - \omega_j(\mathbf{q}))$ is the Dirac delta function, which ensures that only modes with frequencies near ω contribute.

In practical numerical calculations, the Brillouin zone is discretized into a finite grid of $\mathbf{q}$-points, and the delta function is approximated using a broadening function, such as a Gaussian or Lorentzian. The procedure is as follows.

Algorithm 4 Phonon Density of States (DOS) Calculation

```
 1: Initialize: Set $g(\omega) \leftarrow 0$.
 2: Generate q Grid:
 3: for each $\mathbf{q}$ in the grid do
 4:     Compute and diagonalize $D(\mathbf{q})$.
 5:     Extract phonon frequencies $\omega_j(\mathbf{q})$ for all branches $j$.
 6: end for
 7: for each $\mathbf{q}$ in the grid do
 8:     for each branch $j$ do
 9:         $g(\omega) \leftarrow g(\omega) + \frac{1}{\sqrt{2\pi}\sigma} \exp\left(-\frac{(\omega-\omega_j(\mathbf{q}))^2}{2\sigma^2}\right)$.
10:     end for
11: end for
12: Normalize DOS: $g(\omega) \leftarrow g(\omega)/N_q$
```

13.5. Application to FCC Argon

Let us consider a face-centered cubic (FCC) crystal of argon with a Lennard-Jones potential. We aim to write a code to compute the dynamic matrix for this system. For simplicity, we'll consider only one atom in the primitive unit cell and take into account only the first, second and third nearest neighboring atoms contributing to the force constants.

13.5.1 System Setup

As shown in Fig. 13.5, an FCC lattice with a unit length a has the primitive lattice vectors in the real and reciprocal space defined as:

$$\mathbf{a}_1 = \frac{a}{2}(0,1,1), \qquad \mathbf{b}_1 = 2\pi\frac{\mathbf{a}_2 \times \mathbf{a}_3}{\mathbf{a}_1 \cdot (\mathbf{a}_2 \times \mathbf{a}_3)} = \frac{2\pi}{a}(1,-1,1),$$
$$\mathbf{a}_2 = \frac{a}{2}(1,0,1), \qquad \mathbf{b}_2 = 2\pi\frac{\mathbf{a}_3 \times \mathbf{a}_1}{\mathbf{a}_1 \cdot (\mathbf{a}_2 \times \mathbf{a}_3)} = \frac{2\pi}{a}(1,1,-1),$$
$$\mathbf{a}_3 = \frac{a}{2}(1,1,0), \qquad \mathbf{b}_3 = 2\pi\frac{\mathbf{a}_1 \times \mathbf{a}_2}{\mathbf{a}_1 \cdot (\mathbf{a}_2 \times \mathbf{a}_3)} = \frac{2\pi}{a}(-1,1,1).$$

Each atom at position $\mathbf{R}$ interacts with its neighbors at positions $\mathbf{R} + \boldsymbol{\delta}$, where $\boldsymbol{\delta}$ are the vectors to the neighbors.

- The 1st-nearest neighbors are $a/2 \times \{(\pm1,0,\pm1),(0,\pm1,\pm1),(\pm1,\pm1,0)\}$.
- The 2nd-nearest neighbors are $a/2 \times \{(\pm2,0,0),(0,\pm2,0),(0,0,\pm2)\}$.
- The 3rd-nearest neighbors are $a/2 \times \{(\pm1,\pm1,\pm2),(\pm1,\pm2,\pm1),(\pm2,\pm1,\pm1)\}$.

13.5.2 From Force Constants to Dynamical Matrix

According to Lennard-Jones potential, the total energy is

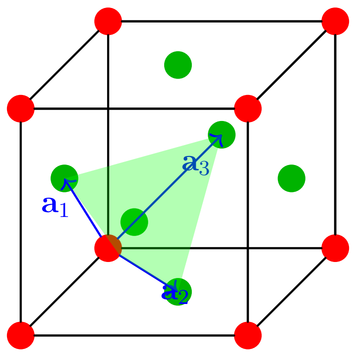


Figure 13.5: The schematic of FCC lattice. To enhance the clarity, the corner and face centered atoms are shown in different colors.

$$U(r) = \sum_{ij} 4\varepsilon \left[\left(\frac{\sigma}{r_{ij}} \right)^{12} - \left(\frac{\sigma}{r_{ij}} \right)^{6} \right],$$

where:

- ε is the depth of the potential well.
- σ is the finite distance at which the inter-particle potential is zero.
- r is the distance between two atoms.

For a given ij pair, the 1st and 2nd derivatives of energy with respect to the distance are

$$\frac{\partial U}{\partial r} = 4\epsilon \left(-12\frac{\sigma^{12}}{r^{13}} + 6\frac{\sigma^6}{r^7} \right).$$

$$\frac{\partial^2 U}{\partial r^2} = 4\epsilon \left(156\frac{\sigma^{12}}{r^{14}} - 42\frac{\sigma^6}{r^8} \right).$$

The force constant matrix elements for the interaction between two atoms via a pair-wise potential $U(r)$ are given by:

$$\Phi_{\alpha\beta} = \frac{\partial^2 U}{\partial r_\alpha \partial r_\beta}.$$

This can be expressed in terms of the total derivatives dU/dr and d^2U/dr^2 as:

$$\Phi_{\alpha\beta} = \hat{r}_\alpha \hat{r}_\beta \frac{d^2U}{dr^2} + \delta_{\alpha\beta}\frac{1}{r}\frac{dU}{dr} - \hat{r}_\alpha \hat{r}_\beta \frac{1}{r}\frac{dU}{dr},$$

where:

- $\hat{r}_\alpha$ and $\hat{r}_\beta$ are components of the unit vector $\hat{r}$.
- $\delta_{\alpha\beta}$ is the Kronecker delta (1 if $\alpha = \beta$, 0 otherwise).

For the diagonal terms ($\alpha = \beta$):

$$\Phi_{\alpha\alpha} = \hat{r}_\alpha^2 \frac{d^2U}{dr^2} + (1 - \hat{r}_\alpha^2)\frac{1}{r}\frac{dU}{dr}.$$

For the off-diagonal terms ($\alpha \neq \beta$):

$$\Phi_{\alpha\beta} = \hat{r}_\alpha \hat{r}_\beta \left(\frac{d^2U}{dr^2} - \frac{1}{r}\frac{dU}{dr} \right).$$

When looping over the neighbors for each atom i, we add $\Phi^{ij}_{\alpha\beta} e^{i\mathbf{q}\cdot\mathbf{R}}$ to $D^{ij}_{\alpha\beta}(\mathbf{q})$, which count the force contribution from the neighboring atoms. In addition, one must take into account that per force is equally applied to each atom i, j. Hence, the force contribution $-\Phi^{ij}_{\alpha\beta}$ should be added to $D^{ii}_{\alpha\beta}(\mathbf{q})$ to reflect the force contribution from the self-atom with a phase factor of 1 since $\mathbf{R} = (0, 0, 0)$.

13.6. Python Implementation

The following code calculates the phonon dispersion relation for an FCC Argon by evaluating the dynamical matrix at different wavevectors $\mathbf{q}$, accounting for contributions from the nearest neighbors in the lattice.

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

epsilon = 0.0103     # eV
sigma = 3.4          # Angstrom
mass = 39.948        # a.u
a = 5.3              # Lattice constant (Angstrom)
q_basis = 2* np.pi / a * np.array([[1, -1, 1], [1, 1, -1], [-1, 1, 1]])

# Nearest neighbors for FCC lattice
NN1 = np.array([
    [1, 1, 0], [-1, -1, 0], [-1, 1, 0],
    [1, -1, 0], [1, 0, 1], [-1, 0, -1],
    [-1, 0, 1], [1, 0, -1], [0, 1, 1],
    [0, -1, -1], [0, -1, 1], [0, 1, -1]]) * a / 2

NN2 = np.array([
    [1, 0, 0], [-1, 0, 0], [0, 1, 0],
    [0, -1, 0], [0, 0, 1], [0, 0, -1]]) * a

NN3 = np.array([
    [1, 1, 2], [-1, -1, -2], [1, 2, 1],
    [-1, -2, -1], [2, 1, 1], [-2, -1, -1],
    [1, -1, 2], [-1, 1, -2], [1, 2, -1],
    [-1, -2, 1], [2, 1, -1], [-2, -1, 1],
    [-1, -1, 2], [1, 1, -2], [-1, 2, -1],
    [1, -2, 1], [2, -1, -1], [-2, 1, 1],
    [1, 1, -2], [-1, -1, 2], [1, -2, 1],
    [-1, 2, -1], [-2, 1, 1], [2, -1, -1]]) * a / 2

# Compute derivatives
def lj_derivatives(r):
    sr6 = (sigma / r)**6
    sr12 = sr6**2
    dE_dR = 4 * epsilon * (12 * sr12 - 6 * sr6) / r
    d2E_dR2 = 4 * epsilon * (156*sr12 - 42* sr6) / r**2
    return dE_dR, d2E_dR2

```

```python
# Compute force constants
def fc(r, r_hat, dE_dR, d2E_dR2, alpha, beta):
    product = r_hat[alpha] * r_hat[beta]
    if alpha == beta:  # Diagonal terms
        return product * d2E_dR2  + (1 - product) * dE_dR / r
    else:  # Off-diagonal terms
        return product * (d2E_dR2 - dE_dR / r)

# Compute the dynamical matrix for a single atom
def compute_dynamical_matrix(q, nearest_neighbors, mass):
    d_matrix = np.zeros((3, 3), dtype=complex)
    for neighbor in nearest_neighbors:
        r = np.linalg.norm(neighbor)
        r_hat = neighbor / r
        phase = np.exp(1j * np.dot(q, neighbor))
        dE_dR, d2E_dR2 = lj_derivatives(r)

        for alpha in range(3):
            for beta in range(3):
                fc_value = fc(r, r_hat, dE_dR, d2E_dR2, alpha, beta)
                # Neighbor contribution
                d_matrix[alpha, beta] += phase * fc_value
                # self-interation
                d_matrix[alpha, beta] -= fc_value
    return d_matrix / mass

# Compute the phonon dispersion from (0, 0, 0) to (1, 0, 0)
NN = None
qxs = np.linspace(1.0, 0.0, 20)
freqs = np.zeros([4, len(qxs), 3])
colors = ['r', 'b', 'g']
for i, NNs in enumerate([NN1, NN2, NN3]):
    # Setup the truncation of neighbors
    if NN is None:
        NN = NNs
    else:
        NN = np.vstack((NN, NNs))

    # Compute vibrational frequencies
    for j, qx in enumerate(qxs):
        q0 = np.array([qx, 0, 0]) @ q_basis
        d_matrix = compute_dynamical_matrix(q0, NN, mass)
        eigenvalues, eigenvectors = np.linalg.eig(d_matrix)
        freq = np.sqrt(np.abs(eigenvalues.real)) / (2*np.pi) * 241.7991
        freq.sort()
        freqs[i, j, :] += freq

    plt.plot(qxs, freqs[i, :, 0], '-.', c=colors[i],
             label=f"$\omega$ ({len(NN)} Neighbors)")
    plt.plot(qxs, freqs[i, :, 1], '-o', c=colors[i])
    plt.plot(qxs, freqs[i, :, 2], '-d', c=colors[i])
plt.ylabel('Frequency (THz)')
plt.legend()
plt.show()
```

The initial section defines the key parameters, including the Lennard-Jones potential parameters for Argon (ϵ, σ), the FCC lattice constant ($a = 5.3$ Å), mass values, and the reciprocal lattice basis vectors for the FCC lattice, which are essential for computing wavevectors.

It then specifies the nearest neighbors (NN) for the FCC lattice:

- NN1: 12 first nearest neighbors at a distance of $\sqrt{2}/2 \times a$.
- NN2: 6 second nearest neighbors at a distance of a.
- NN3: 24 third nearest neighbors at a distance of $\sqrt{5}/2 \times a$.

Three functions are implemented to handle the calculations:

- **lj_derivatives**: Computes the first ($\partial E/\partial r$) and second derivatives ($\partial^2 E/\partial r^2$) of the Lennard-Jones potential with respect to the interatomic distance r.
- **fc**: Calculates contributions to the force constant matrix elements for given directions (indices α and β).
- **compute_dynamical_matrix**: Iterates over all neighbors to compute the contributions to the dynamical matrix $D(\mathbf{q})$.

In the main routine, the phonon frequencies are computed for 20 evenly spaced q-points from the (0, 0, 0] to (1, 0, 0) direction in reciprocal space. To investigate the role of neighbor contributions to the dynamical matrix, the program iterates over the nearest neighbors (NN1, NN2, and NN3) to calculate their cumulative effects on the dispersion relation. For each q-point, the dynamical matrix is constructed using all included neighbors up to the current level (NN), and eigenvalues are extracted to determine the phonon frequencies.

Fig. 13.6 illustrates the calculated phonon dispersion curves, revealing three acoustic branches: two degenerate transverse acoustic (TA) modes with a smaller dispersion and one longitudinal acoustic (LA) mode with a larger dispersion. The results are consistent with the expected behavior for a FCC lattice.

Additionally, as the number of included neighbors increases from NN1 to NN2 and NN3, only minor modifications to the dispersion curve are observed. This behavior aligns with the short-range nature of the LJ potential, which contributes negligibly at larger distances. As shown in Fig. 13.7, the derivative values of the LJ potential (both $\partial E/\partial r$ and $\partial^2 E/\partial r^2$) rapidly decrease with increasing r, emphasizing the dominance of first-neighbor interactions in determining the vibrational properties of the lattice.

To understand the physical meaning of each mode. One can directly check the D values at different $\mathbf{q}$ vectors and compute the corresponding eigenvectors and eigenvalues. Below is a case for $\mathbf{q}$=(0, 0, 0.1) near zero.

```python
# Compute an indivudal q point
q = np.array([0, 0, 0.1])
D = compute_dynamical_matrix(q, NN, mass)
eig_freqs, eig_vecs = np.linalg.eigh(D, UPLO='U')
eig_freqs = np.sqrt(eig_freqs.T/(2*np.pi) * 241.7991) # THz

print("Dynamical Matrix (Real Part):")
for row in D.real:
    formatted_row = " ".join(f"{value:8.4f}" for value in row)
    print(formatted_row)

for eig_freq, eig_vec in zip(eig_freqs, eig_vecs.T):
    vec = " ".join(f"{value:8.4f}" for value in eig_vec.real)
    print(f"Frequency: {eig_freq:.6f} => Eigenvector: {vec}")
```

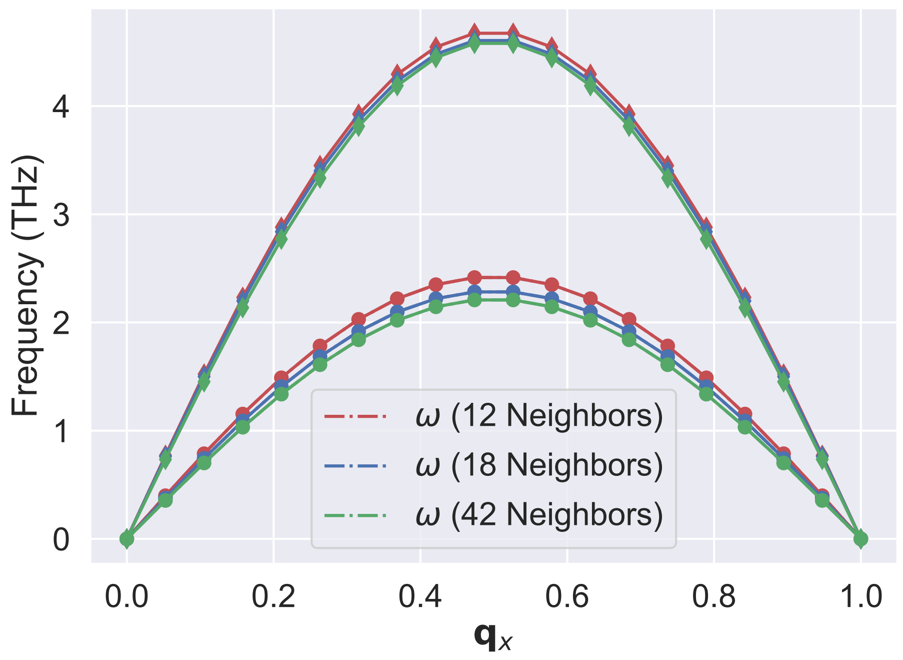


Figure 13.6: The calculation dispersion relation along ($\mathbf{q}_x$, 0, 0) for FCC argon using different cutoff distances.

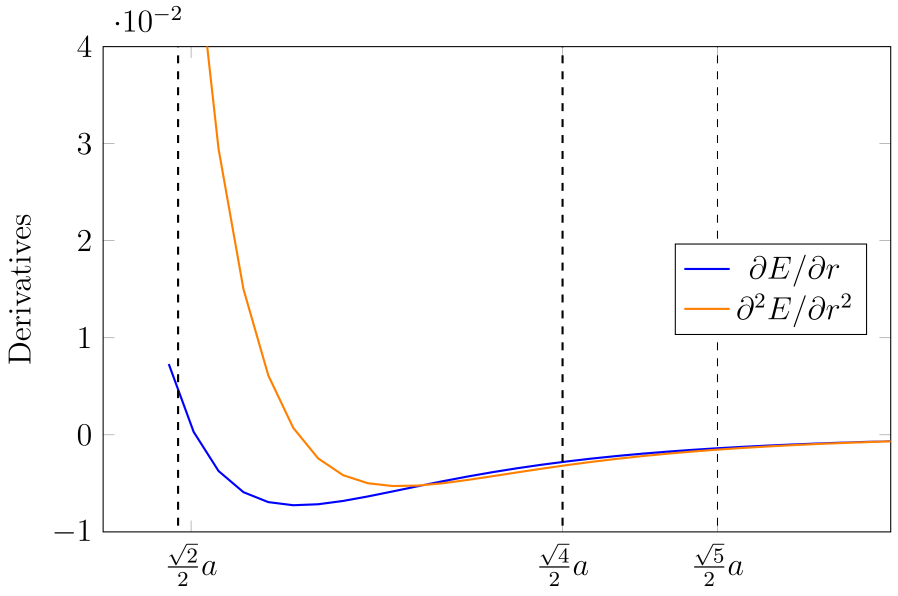


Figure 13.7: The radial dependence of LJ derivative values for Argon.

The outputs are the following.

```
Dynamical Matrix (Real Part):
  0.0001   0.0000  -0.0000
  0.0000   0.0001   0.0000
 -0.0000   0.0000   0.0002
Frequency: 0.066978 => Eigenvector:   0.7071  -0.7071   0.0000
Frequency: 0.066978 => Eigenvector:   0.7071   0.7071   0.0000
Frequency: 0.091297 => Eigenvector:   0.0000   0.0000   1.0000
```

Clearly, there are three distinct vibrational modes. Among these, the mode with the highest frequency (0.09 THz) corresponds to a vibration with the eigenvector (0, 0, 1) aligned along the wave vector. This is known as the longitudinal acoustic (LA) mode. The other two modes have eigenvectors perpendicular to the wave vector, and they are identified as transverse acoustic (TA) modes.

When two atoms are present in the unit cell, a similar vibrational pattern emerges. In addition to the acoustic modes, optical modes appear due to the relative motion between atoms in the unit cell. Specifically, there is one longitudinal optical (LO) mode, where vibrations align with the wave vector, and two transverse optical (TO) modes, where vibrations are perpendicular to it. These optical modes typically occur at higher frequencies compared to the acoustic modes and reflect the internal dynamics of the unit cell.

For a cross-validation, one can use the `ASE` package [37] to repeat the simulation for a few representative $\mathbf{q}$ points as listed in Table 13.1. Clearly, the results are very similar. The slight discrepancy is likely due to the use of numerical derivatives in `ASE`.

Table 13.1: Comparison of results from the `ASE` code and this work.

$\mathbf{q}$	ASE ($7 \times 7 \times 7$)	NN1	NN2	NN3
(0, 0, 0)	(0.00, 0.00, 0.00)	(0.00, 0.00, 0.00)	(0.00, 0.00, 0.00)	(0.00, 0.00, 0.00)
($\frac{1}{2}$, 0, 0)	(2.11, 2.11, 4.59)	(2.42, 2.42, 4.67)	(2.28, 2.28, 4.61)	(2.21, 2.21, 4.58)
($\frac{1}{2}$, $\frac{1}{2}$, 0)	(3.16, 3.16, 4.61)	(3.37, 3.37, 4.69)	(3.37, 3.37, 4.69)	(3.30, 3.30, 4.66)
($\frac{1}{2}$, $\frac{1}{2}$, $\frac{1}{2}$)	(2.11, 2.11, 4.59)	(2.42, 2.42, 4.67)	(2.28, 2.28, 4.61)	(2.21, 2.21, 4.58)

13.7. Summary and Outlook

Through the study of monoatomic and diatomic chain models, we uncovered the fundamental concepts of phonon dispersion and the emergence of acoustic and optical modes. The introduction of the dynamical matrix allowed us to generalize these ideas to more realistic three-dimensional systems, enabling the calculation of phonon frequencies and the density of states. For those interested in phonon and vibrational analysis, following the procedures outlined in this chapter offers valuable insights into the fundamental aspects of lattice dynamics. For modern applications, the Python library `phonopy` [38] provides a powerful framework for automating phonon calculations, streamlining the analysis of vibrational properties in a wide range of materials.

To illustrate the concept, we primarily focused on the supercell approach to collect contributions to the force constants. This method is effective when interatomic forces decay rapidly with increasing distance. However, for systems where long-range interactions or subtle force contributions are significant, this approach may lose accuracy. In such cases, more accurate methods, such as linear response theory, can provide enhanced precision by directly computing the dynamical matrix without relying on finite displacements. For a detailed explanation of this approach, please refer to Baroni et. al [39].

One limitation of the approaches discussed lies in the reliance on the harmonic approximation, which assumes that atomic displacements remain small. However, this approximation breaks down in cases where anharmonic effects become significant, such as at high temperatures or in light systems with strong lattice vibrations. In such scenarios, higher-order terms must be considered to account for anharmonicity.

The insights gained here form the basis for studying more complex phenomena, such as heat transport, electron-phonon interactions, and optical properties of materials. By combining theoretical approaches with numerical techniques like Brillouin zone sampling and dynamical matrix diagonalization, we establish a versatile framework for exploring vibrational behavior across a wide range of materials. This understanding is crucial for advancing applications in thermoelectrics, semiconductors, and other areas of material research.

14. Representing Local Atomic Environment

In atomic and molecular systems, the local environment around a particle plays a crucial role in determining its physical properties. Understanding how atoms or molecules are arranged in the 3D space provides insights into the material's structural characteristics, phase transitions, and dynamic behaviors. The local atomic environment can be described through various mathematical tools. In the previous chapter, we introduced Radial distribution function (RDF) as a fundamental tool for describing the local structure in a system of particles. It gives the probability of finding a particle at a distance r from a reference particle, normalized by the average particle density, while the RDF is a powerful tool for capturing the radial distribution of particles, it is unable to describe angular correlations between particles. As a result, systems with orientational order, such as liquid crystals or crystals with complex angular symmetries, cannot be fully characterized by the pair distribution function alone. In this chapter, we will explore the descriptors that can deal with both radial and angular information with mathematical rigor.

14.1. Orientational Order Parameter

To capture the missing angular information in systems with orientational order, we need to introduce additional descriptors, such as the orientational order parameter. This parameter quantifies the degree of angular ordering among neighboring particles. Unlike RDF, which focuses solely on the radial distances, orientational order parameters provide insights into how the bonds between particles are aligned in space. These parameters are particularly useful in distinguishing between phases with different degrees of symmetry, such as solid, liquid, or nematic phases.

14.1.1 Orientational Order in a 2D System

Let's first consider a two-dimensional system, where the angular relationships between a particle and its nearest neighbors can provide critical information about the system's structure. To quantify these angular relationships, the bond orientational order parameter ψ_m has been introduced to measures how the bonds between a particle and its neighbors are aligned with respect to a reference axis [40]. The order parameter ψ_m is given by the following equation:

$$\psi_m = \frac{1}{N}\sum_{j=1}^{N} e^{im\theta_j}. \tag{14.1}$$

In this expression, N is the number of neighbors around a given particle, θ_j is the angle formed between the bond connecting a particle to its neighbor j and some reference axis, and m is the symmetry index. For example, $m = 6$ is used for systems with hexagonal symmetry.

The value of ψ_m is a **complex number**, and it will depend on the degree of angular ordering in the system. To evaluate the numerical behaviors of $\psi_m(i)$ values, let us examine how they represent a simple geometry object of a cluster where the center atom is surrounded by 3 neighboring atoms with a nearly center distance ***.

Figures ***.

- **Magnitude**:
- **Phase**: The phase angle of $\psi_m(i)$ indicates the orientation of the bond order relative to the reference direction.

Hence, we can summarize that ϕ_m-series attempts to reflect the correlation between the target geometry and reference geometry (i.e., ****). The resulting magnitude $|\psi_m(i)|$ measures how well the local arrangement of neighbors conforms to an m-fold symmetric structure. If the neighbors are perfectly arranged in a hexagonal pattern, $|\psi_6(i)|$ will be close to 1. In disordered regions, the value of $|\psi_6(i)|$ will be closer to 0. On the other hand, the phase of $\psi_m(i)$ indicates the orientation of the bond order relative to the reference direction.

Thus, by examining how ψ_m evolves with temperature, pressure, or density, researchers can gain insights into the structural transformations occurring in the system.

14.1.2 Extension to 3D: Neighbor Density Function

In the real-world scenarios, we are primarily dealing with 3D systems. How can we extend the approach we discussed for 2D to 3D? This extension involves additional complexity because, with the introduction of an extra dimension, the alignment of atoms can no longer be described by a single variable, as in 2D. To capture this alignment in 3D, we need to be more rigorous with our mathematical description.

We define the **Atomic Neighbor Density Function** to describe the spatial distribution of atoms around a reference atom within a cutoff radius r_c. The atomic neighbor density function is expressed as:

$$\rho(\mathbf{r}) = \sum_{i}^{r_i \leq r_c} \delta(\mathbf{r} - \mathbf{r_i}) \tag{14.2}$$

Here, $\delta(\mathbf{r} - \mathbf{r}_i)$ is the Dirac delta function, which ensures that the function only contributes when a neighboring atom is located at $\mathbf{r}_i$, and the summation runs over all neighboring atoms within the cutoff radius r_c.

14.1.3 Expansion on the Spherical Harmonics

To capture the angular distribution of neighboring atoms, we can transform the spatial neighbor density function into another domain, similar to how the Fourier transform converts a time-domain signal into its frequency components. In this case, we are interested in projecting the atomic density distribution onto the unit sphere to study the angular arrangement of atoms.

A popular choice for the basis functions on the unit sphere is spherical harmonics, which are functions defined on the surface of a sphere. Therefore, we can expand the neighbor density function $\rho(\mathbf{r})$ as a series of spherical harmonics on the 2-sphere:

$$\rho(\mathbf{r}) = \sum_{l=0}^{+\infty} \sum_{m=-l}^{+l} c_{lm} Y_{lm}(\hat{\mathbf{r}}) \tag{14.3}$$

In this expression:

- $Y_{lm}(\hat{\mathbf{r}})$ are the spherical harmonics, which form a complete orthonormal basis on the sphere.
- $\hat{\mathbf{r}}$ is the normalized radial vector, with a unit length of 1.
- c_{lm} are the expansion coefficients, which describe the contribution of each spherical harmonic mode to the overall distribution.

The c_{lm} coefficients can be computed by projecting the neighbor density function onto the spherical harmonics:

$$c_{lm} = \langle Y_{lm}(\hat{\mathbf{r}}) | \rho(\mathbf{r}) \rangle = \int Y^*_{lm}(\hat{\mathbf{r}}) \rho(\mathbf{r}) d^3 r = \sum_{i}^{r_i \le r_c} Y_{lm}(\hat{\mathbf{r}}_\mathbf{i}) = \sum_{i}^{r_i \le r_c} N e^{im\phi} P_{lm} \cos(\theta) \tag{14.4}$$

Here, l denotes the total angular momentum, m represents its projection along a chosen axis, and P_{lm} is the associated Legendre Polynomial. The $Y_{lm}(\hat{\mathbf{r}})$ can be decomposed into a complex exponential and associated Legendre polynomials. From this expression one can clearly see a strong similarity between ψ_m in the 2D and c_{lm} in the 3D. Indeed, c_{lm} is an extension of ψ_m by adding the associated Legendre Polynomial P_{lm} to account for the distribution of angular momentum. Hence, the c_{lm} coefficients are complex numbers that capture the angular characteristics of the neighbor density. Similar to ψ_m, these coefficients are sensitive to rotations of the system. If the system is rotated, the values of c_{lm} will change, which is undesirable when trying to describe the local atomic environment in a way that is independent of orientation.

In practical applications, we aim to find a representation of the local atomic environment that is, similar to RDF, both *real-valued* and *invariant under translations and rotations* of the system. However, they should go beyond the pairwise two-body, *** reflect the three-body, four-body and even manybody distributions.

14.1.4 Rotation-Invariant Parameters

To address this challenge, Steinhardt introduced **bond order parameters** in 1983 [41], which use second- and third-order combinations of the expansion coefficients c_{lm} to quantify the order in liquids and glasses. These bond order parameters are rotationally invariant, making them useful for characterizing local atomic environments without being affected by the orientation of the system.

The bond order parameter p_l is defined as:

$$p_l = \sum_{m=-l}^{+l} c_{lm} c^*_{lm} \tag{14.5}$$

14.1.5 Applications and Limitations

Note that this was called Q_l in the original paper [41]. However, it was later found that bond order parameter is closely related to the power spectrum. Hence, we will call it p_l from now on. In signal processing, the power spectrum describes how the power of a signal is distributed across different frequency components. Similarly, in this context, p_l measures the **power** of the neighbor density when projected onto the angular frequency components represented by l. In general, the power spectrum p_l is the Fourier transform of the autocorrelation function, and it provides a frequency-domain representation of the dependencies captured by the autocorrelation.

When analyzing the p_l series for different structures, Steinhardt found that p_4 and p_6 were particularly useful in distinguishing between different crystal structures such as body-centered cubic (bcc), face-centered cubic (fcc), hexagonal close-packed (hcp), and icosahedral arrangements. These parameters have proven to be very useful for analyzing MD simulations, particularly when identifying structural differences between solids, liquids, and glasses. For instance, LAMMPS allows the computation of bond-orientational order parameters in several kinds of styles.

While the p_l series is useful to capture the feature on angular distribution, it does not contain any radial information. Additionally, it assumes a neighbor density in the form of a Dirac delta function. This may not be good for the purpose of measuring the similarities between two environments. Nevertheless, the idea of using spherical harmonics and power spectra to describe the local atomic environment has inspired many subsequent works and is still widely used in computational materials science today.

14.2. Manybody descriptors

In modern computational materials science and atomistic simulations, understanding the local atomic environment goes beyond simple pairwise interactions. Describing how groups of atoms are collectively arranged, including both radial and angular components, is critical for capturing the structural complexity of systems such as liquids, glasses, and complex crystals. Manybody descriptors provide a mathematical framework to represent these arrangements, offering a more detailed picture of the atomic environment than traditional pair distribution functions or angular descriptors alone.

Manybody descriptors, such as the power spectrum and bispectrum, help quantify the relative positions of multiple atoms in a way that is invariant to rotation and translation.

14.2.1 Radial Dependent Power Spectrum Descriptor

In practical applications, we often care about how atoms are spatially arranged in both radial and angular space. While previous approaches focused primarily on the angular distribution, the introduction of radial information allows for a more comprehensive description of the local atomic environment.

In 2012, Bartók et al. introduced an improved manybody descriptor that explicitly incorporates both radial and angular components [42]. This approach overcomes the limitation of describing neighbor density with a Dirac delta function by replacing the delta function with a Gaussian function of limited width α. This smoothing allows for a more realistic representation of how atoms are distributed around a reference atom.

The modified neighbor density function is given by:

$$\rho'(\mathbf{r}) = \sum_i^{r_i \leq r_c} e^{(-\alpha|\mathbf{r}-\mathbf{r_i}|^2)} = \sum_i^{r_i \leq r_c} e^{-\alpha(r^2+r_i^2)} e^{2\alpha\mathbf{r}\cdot\mathbf{r_i}} \tag{14.6}$$

Expanding the exponential of a dot product in spherical coordinates:

$$e^{2\alpha\mathbf{r}\cdot\mathbf{r_i}} = e^{2\alpha r r_i \cos(\gamma)} = 4\pi \sum_{l=0}^{\infty} \sum_{m=-l}^{l} I_l(2\alpha r r_i) Y_{lm}^*(\hat{\mathbf{r}}_\mathbf{i}) Y_{lm}(\hat{\mathbf{r}}). \tag{14.7}$$

In which, we used the general formula addition theorem for spherical harmonics,

$$e^{z\cos(\gamma)} = 4\pi \sum_{l=0}^{\infty} \sum_{m=-l}^{l} I_l(z) Y_{lm}^*(\hat{\mathbf{r}}_\mathbf{i}) Y_{lm}(\hat{\mathbf{r}}). \tag{14.8}$$

This expression can be further expanded as:

$$\rho'(\mathbf{r}) = \sum_i^{r_i \leq r_c} \sum_{lm} 4\pi e^{-\alpha(r^2+r_i^2)} I_l(2\alpha r r_i) Y_{lm}^*(\hat{\mathbf{r}}_\mathbf{i}) Y_{lm}(\hat{\mathbf{r}}_\mathbf{i}), \tag{14.9}$$

where the first part $I_l(2\alpha r r_i)$ is the modified spherical Bessel function of the first kind (governed by $2\alpha r r_i$), providing the radial dependence, and the second part captures the angular dependence of the vectors $\mathbf{r}$ and $\mathbf{r}_i$.

Bartók also introduced a set of polynomials, $g_n(r)$, which help describe the radial component in a more refined way:

$$\phi_\alpha(r) = (r_c - r)^{\alpha+2}/N_\alpha$$

where N_α is a normalization factor given by:

$$N_\alpha = \sqrt{\int_0^{r_c} r^2 (r_c - r)^{2(\alpha+2)} dr}$$

These polynomials are orthonormalized to ensure that the radial functions $g_n(r)$ form a basis. The orthonormalization process is performed through linear combinations of $\phi_\alpha(r)$, and the coefficients are obtained from

$$g_n(r) = \sum_{\alpha=1}^{n_{\max}} W_{n\alpha} \phi_\alpha(r),$$

where W is constructed from the inverse square root of the overlap matrix S,

$$\begin{aligned} S_{\alpha\beta} &= \int_0^{r_c} r^2 \phi_\alpha(r) \phi_\beta(r) dr \\ &= \frac{\sqrt{(2\alpha+5)(2\alpha+6)(2\alpha+7)(2\beta+5)(2\beta+6)(2\beta+7)}}{(5+\alpha+\beta)(6+\alpha+\beta)(7+\alpha+\beta)} \end{aligned}$$

As discussed in the previous DFT chapter with Gaussian basis set, the overlap matrix describes how different radial functions overlap with each other and ensures that the final radial basis functions $g_n(r)$ are orthonormal.

The neighbor density function $\rho'(\mathbf{r})$ can then be expanded in terms of both the radial basis $g_n(r)$ and the spherical harmonics:

$$\begin{aligned} c_{nlm} &= \langle g_n(r) Y_{lm}(\hat{\mathbf{r}}) | \rho'(\mathbf{r}) \rangle \\ &= \int d^3 r g_n(r) Y_{lm}(\hat{\mathbf{r}})) \sum_{r_i \le r_c} \sum_{l'm'} 4\pi e^{-\alpha(r^2+r_i^2)} I_{l'}(2\alpha r r_i) Y^*_{l'm'}(\hat{\mathbf{r}}_\mathbf{i}) Y_{l'm'}(\hat{\mathbf{r}}) \end{aligned} \tag{14.10}$$

When integrating over the angular variables $\hat{\mathbf{r}}$, only the terms with $l' = l$ and $m' = m$ will survive, due to orthogonality.

$$\begin{aligned} c_{nlm} &= 4\pi \sum_i^{r_i \le r_c} Y^*_{lm}(\hat{\mathbf{r}}_\mathbf{i}) \int_0^{r_c} r^2 g_n(r) e^{-\alpha(r^2+r_i^2)} I_l(2\alpha r r_i) dr \\ &= 4\pi \sum_i^{r_i \le r_c} e^{-\alpha r_i^2} Y^*_{lm}(\hat{\mathbf{r}}_\mathbf{i}) \int_0^{r_c} r^2 g_n(r) e^{-\alpha r^2} I_l(2\alpha r r_i) dr \end{aligned} \tag{14.11}$$

Finally, the **rotation-invariant power spectrum** is obtained by combining these expansion coefficients:

$$p_{n_1 n_2 l} = \sum_{m=-l}^{+l} c_{n_1 lm} c^*_{n_2 lm} \tag{14.12}$$

This rotation-invariant descriptor provides a comprehensive measure of the local atomic environment by accounting for both radial and angular information, making it a powerful tool for analyzing atomic structures in simulations and experiments.

14.2.2 Bispectrum on 4D Hyperspace

An alternative approach to capturing manybody interactions involves mapping the neighbor density function onto the surface of a 4D hypersphere. This method allows for a richer representation of the local atomic environment by incorporating angular information in 4D.

In this formalism, the coordinates (x, y, z, r) on the 4D hypersphere are given by:

$$\begin{aligned} s_1 &= r_0 \cos\omega \\ s_2 &= r_0 \sin\omega \cos\theta \\ s_3 &= r_0 \sin\omega \sin\theta \cos\phi \\ s_4 &= r_0 \sin\omega \sin\theta \sin\phi, \end{aligned}$$

$$\begin{aligned} r_0 &\ge r_c \\ \theta &= \arccos(z/r) \\ \phi &= \arctan(y/x) \\ \omega &= \pi r / r_0 \end{aligned}$$

where r_0 is a characteristic radius (related to the cutoff radius r_c), ω, θ, and ϕ are the spherical coordinates.

The atomic neighbor density function is expressed as:

$$\rho(\mathbf{r}) = \delta(\mathbf{r}) + \sum_i f_c(r)\delta(\mathbf{r} - \mathbf{r_i}) \tag{14.13}$$

The first term ensures that the density function remains well-behaved with respect to variations in ω, and f_c is a smooth function to ensure that the it gradually decays to 0 when $r \geq r_c$.

By expanding the atomic neighbor density function in terms of the Wigner-D matrix elements, which represent rotations in the angular coordinates, we obtain:

$$\rho(\mathbf{r}) = \sum_{j=0}^{+\infty} \sum_{m',m=-j}^{+j} c^j_{m',m} D^j_{m',m}(2\omega;\theta,\phi) \tag{14.14}$$

The coefficients $c^j_{m\prime,m}$ are determined by projecting the neighbor density function onto the Wigner-D matrix elements:

$$c^j_{m',m} = \langle D^j_{m',m} | \rho \rangle$$

$$= \int_0^\pi d\omega \, \sin^2\omega \, \omega \int_0^\pi d\theta \, \sin\theta \int_0^{2\pi} d\phi \, D^{*j}_{m',m}(2\omega;\theta,\phi) \, \rho(r)$$

$$= D^{*j}_{m',m}(0) + \sum_i f_{\text{cut}}(r_i) D^{*j}_{m',m}(r_i) \tag{14.15}$$

Finally, the **bispectrum components**, which capture three-body correlations, can be computed using the triple correlation of the expansion coefficients:

$$B_{j_1,j_2,j} = \sum_{m',m=-j}^{+j} c^{*j}_{m',m} \sum_{m'_1,m_1=-j_1}^{+j_1} c^{j_1}_{m'_1,m_1} \sum_{m'_2,m_2=-j_2}^{+j_2} c^{j_2}_{m'_2,m_2} C^{j_1 j_2 j}_{m_1 m_2 m} C^{j_1 j_2 j}_{m'_1 m'_2 m'} \tag{14.16}$$

Here, $C^{j_1 j_2 j}_{m_1 m_2 m}$ are the **Clebsch-Gordan coefficients**, which ensure proper angular momentum coupling in the bispectrum calculation.

14.2.3 Atomic Cluster Expansion

So far, the powerspectrum and bispectrum ideas are great in terms of strict orthonormality. In parallel to the spectrum descriptors, it is also popular to design descriptors that explicitly counts the many-body interactions with the choice of Gaussian basis set (for more details, please refer to Behler and Parrinello [43, 44]). But it often requires manual parametrization of the basis set. However, none of them can take into both aspects.

Drautz made an important step to provide a general framework based on Atomic Cluster Expansion (ACE) [45]. His idea is to break the energy of atoms in a cluster-expansion fashion.

$$
\begin{aligned}
E_i = V^{(0)} + V^{(1)}(\mathbf{r}_i) + \frac{1}{2}\sum_{ij} V^{(2)}(\mathbf{r}_i, \mathbf{r}_j) \\
+ \frac{1}{3!}\sum_{ijk} V^{(3)}(\mathbf{r}_i, \mathbf{r}_j, \mathbf{r}_k) \\
+ \frac{1}{4!}\sum_{ijkl} V^{(4)}(\mathbf{r}_i, \mathbf{r}_j, \mathbf{r}_k, \mathbf{r}_l) + \cdots \ ,
\end{aligned} \tag{14.17}
$$

where $\mathbf{r}_i$ is the position of atom i, $V^{(0)}$ is a constant offset value, and $V^{(1)}$, $V^{(2)}$ $\cdots$ are the potentials functions due to the 1-body, 2-body clusters.

For a cluster α with k neighbors surrounding the center atom i, the energy of the center atom should be uniquely determined by the neiboring distances ($\mathbf{r}_{j_1 i}$, $\mathbf{r}_{j_2 i}$, $\cdots$ $\mathbf{r}_{j_k i}$). We define a set of orthonormal basis ϕ_ν to expand them. Due to the orthonormality, ϕ-series satisfy

$$
\int \phi^*_\nu(\mathbf{r})\phi_u(\mathbf{r})d\mathbf{r} = \delta_{\nu u}
$$

$$
\sum_\nu \phi^*_\nu(\mathbf{r})\phi_u(\mathbf{r}')d\mathbf{r} = \delta(\mathbf{r} - \mathbf{r}')
$$

Then, the cluster-α's basis function is

$$
\Phi^\alpha_\nu = \phi_{v_1}(\mathbf{r}_{j_1 i})\phi_{v_2}(\mathbf{r}_{j_2 i}) \cdots \phi_{v_k}(\mathbf{r}_{j_k i})
$$

Similarly, $\Phi_{\alpha\nu}$ is orthonomal and the expansion coefficients can be obtained by projection,

$$
J^\alpha_\nu = \langle \Phi^\alpha_\nu | E_i \rangle
$$

And the energy can be expressed as

$$
\begin{aligned}
E_i = &\sum_j \sum_v J^{(1)}_v \phi_v(\mathbf{r}_{ji}) \\
&+ \frac{1}{2} \sum_{j_1 j_2}^{j_1 \neq j_2} \sum_{v_1 v_2} J^{(2)}_{v_1 v_2} \phi_{v_1}(\mathbf{r}_{j_1 i})\phi_{v_2}(\mathbf{r}_{j_2 i}) \\
&+ \frac{1}{3!} \sum_{j_1 j_2 j_3}^{j_1 \neq j_2, \cdots} \sum_{v_1 v_2 v_3} J^{(3)}_{v_1 v_2 v_3} \phi_{v_1}(\mathbf{r}_{j_1 i})\phi_{v_2}(\mathbf{r}_{j_2 i})\phi_{v_3}(\mathbf{r}_{j_3 i}) \\
&+ \cdots \ .
\end{aligned}
$$

One may rewrite it in unrestricted sums,

$$
\begin{aligned}
E_i = &\sum_j \sum_v c^{(1)}_v \phi_v(\mathbf{r}_{ji}) \\
&+ \frac{1}{2} \sum_{j_1 j_2} \sum_{v_1 v_2} c^{(2)}_{v_1 v_2} \phi_{v_1}(\mathbf{r}_{j_1 i})\phi_{v_2}(\mathbf{r}_{j_2 i}) \\
&+ \frac{1}{3!} \sum_{j_1 j_2 j_3} \sum_{v_1 v_2 v_3} c^{(3)}_{v_1 v_2 v_3} \phi_{v_1}(\mathbf{r}_{j_1 i})\phi_{v_2}(\mathbf{r}_{j_2 i})\phi_{v_3}(\mathbf{r}_{j_3 i}) \\
&+ \cdots \ .
\end{aligned} \tag{14.18}
$$

Obviously, evaluating this formula requires $O(N_c\nu) + O(N_c^2\nu^2) + O(N_c^3\nu^3) + \cdots$ cost, thus prohibiting its practical usage. However, one can the reordering trick. We can first evaluate $\sum_j \phi_\nu(\mathbf{r}_{ji})$. Physically, this term mimics the projection of the basis ϕ_ν to atom i's neighbor density function (ρ_i),

$$\rho_i = \sum_j \delta(\mathbf{r} - \mathbf{r}_{ji}) \quad \longrightarrow \quad A_{iv} = \langle \rho_i | \phi_v \rangle = \sum_j \phi_v(\mathbf{r}_{ji}),$$

Hence we define it as the atomic base $A_{i\nu}$ at atom i and the basis function ν. Using $A_{i\nu}$, one can further compute the double, triple and quadruple summations as well,

$$\sum_{j_1 j_2} \phi_{\nu_1}(\mathbf{r}_{j_1 i})\phi_{\nu_2}(\mathbf{r}_{j_2 i}) = A_{i\nu_1}A_{i\nu_2}$$

$$\sum_{j_1 j_2 j_3} \phi_{\nu_1}(\mathbf{r}_{j_1 i})\phi_{\nu_2}(\mathbf{r}_{j_2 i})\phi_{\nu_3}(\mathbf{r}_{j_3 i}) = A_{i\nu_1}A_{i\nu_2}A_{i\nu_3}$$

Hence, eq. 14.18 becomes

$$\begin{aligned} E_i(\boldsymbol{\sigma}) = & \sum_v c_v^{(1)} A_{iv} + \sum_{v_1 \geq v_2} c_{v_1 v_2}^{(2)} A_{iv_1} A_{iv_2} \\ & + \sum_{v_1 \geq v_2 \geq v_3} c_{v_1 v_2 v_3}^{(3)} A_{iv_1} A_{iv_2} A_{iv_3} + \cdots . \end{aligned} \tag{14.19}$$

As compared to the original formula, the new formula requires only $O(\nu) + O(\nu^2) + O(\nu^3) + \cdots$, that linearly scales with respect to number of neighbors within a cutoff.

In a practical calculation, the basis function should include both radial and angular components such that,

$$\phi_\nu(\mathbf{r}) = \sqrt{4\pi} R_{nl}(r) Y_{lm}(\hat{\mathbf{r}})$$

Therefore, ν needs to be collapsed into three indices (nlm), which n counts the radial basis and lm counts for the spherical harmonics.

From eq. 14.19, one can derive the rotational invariant products as follows

$$B_{in}^{(1)} = A_{in00}, \tag{14.20}$$

$$B_{in_1n_2l}^{(2)} = \sum_{m=-l}^{l} (-1)^m A_{in_1lm} A_{in_2l-m}, \tag{14.21}$$

$$B_{in_1n_2n_3}^{(3),l_1l_2l_3} = \sum_{m_1=-l_1}^{l_1} \sum_{m_2=-l_2}^{l_2} \sum_{m_3=-l_3}^{l_3} \begin{pmatrix} l_1 & l_2 & l_3 \\ m_1 & m_2 & m_3 \end{pmatrix} A_{in_1l_1m_1} A_{in_2l_2m_2} A_{in_3l_3m_3}, \tag{14.22}$$

$$B_{in_1n_2n_3n_4}^{(4),l_1l_2l_3l_4} = \sum_{m_1,\cdots,m_4} \begin{bmatrix} l_1 & l_2 & l_3 & l_4 \\ m_1 & m_2 & m_3 & m_4 \end{bmatrix} A_{in_1l_1m_1} A_{in_2l_2m_2} A_{in_3l_3m_3} A_{in_4l_4m_4}, \tag{14.23}$$

$$B_{in_1n_2n_3n_4n_5}^{(5),l_1l_2l_3l_4l_5} = \sum_{m_1,\cdots,m_5} \begin{bmatrix} l_1 & l_2 & l_3 & l_4 & l_5 \\ m_1 & m_2 & m_3 & m_4 & m_5 \end{bmatrix} A_{in_1l_1m_1} A_{in_2l_2m_2} A_{in_3l_3m_3} A_{in_4l_4m_4} A_{in_5l_5m_5}. \tag{14.24}$$

From the expression, $B^{(2)}$ is clearly very similar to the powerspectrum descriptor, while $B^{(3)}$ has a strong correlation with the bispectrum descriptor. In the mean time, the descriptors are explicitly tied to the 2-body, 3-body clusters, which is connected to the BP approaches. As such, one can consider the set of descriptors is a generalization of previously developed spectrum [42] and

14.3. Code Implementation

14.3.1 Reference environments

SC/Diamond/BCC/FCC/HCP/Icosahderal

14.3.2 The Bond Order Parameters

14.3.3 The Powerspectrum Descriptor

14.3.4 The ACE Descriptor

14.4. Applications

Manybody descriptors, such as the power spectrum and bispectrum, have found widespread applications in various fields of materials science, condensed matter physics, and machine learning, particularly in the analysis of atomic-scale structures. Their ability to describe both angular and radial components of atomic environments has made them essential tools for understanding complex materials and phenomena.

- **Machine Learning Interatomic Potentials**. One of the most prominent applications of many-body descriptors is in the development of machine-learning-based interatomic potentials. These models require descriptors that are invariant to translations, rotations, and permutations of atoms. By providing a compact and invariant representation of the local atomic environment, many-body descriptors allow machine learning models to predict atomic forces and energies with high accuracy, without the need for empirical fitting. This has revolutionized the simulation of large-scale systems, such as materials under extreme conditions or complex chemical reactions.

- **Materials Characterization in Dynamical Simulation**. Bond order parameters are widely used in MD simulations to distinguish between different crystal structures (e.g., fcc, bcc, hcp) and to identify phase transitions between solid, liquid, and amorphous states. These descriptors allow researchers to quantify the degree of local order or disorder in a material and monitor how this order evolves over time. This is particularly useful in the study of glasses, liquids, and amorphous materials, where traditional descriptors like the pair distribution function fail to capture the full complexity of the atomic arrangement.

- **Material Properties Prediction**. By representing atomic structures in a form that is both compact and invariant, these descriptors enable the construction of high-throughput screening models to predict material properties such as hardness, conductivity, and thermal stability. The use of descriptors like the bispectrum in machine learning pipelines has enabled researchers to explore vast chemical and structural spaces and identify novel materials with desired properties. Manybody descriptors are also employed to study nano-structures and catalytic surfaces, where the arrangement of atoms plays a key role in determining reactivity and stability.

14.5. Conclusion and Further Discussions

The development of manybody descriptors, such as the power spectrum and bispectrum, has provided researchers with powerful tools to describe complex atomic environments in a rigorous and invariant manner. These descriptors have addressed the limitations of simpler pairwise and angular metrics, enabling more accurate characterization of local atomic arrangements. Whether in the context of interatomic potentials, structural analysis, phase transitions, or materials discovery, manybody descriptors have significantly advanced our understanding of atomic-scale phenomena.

A. Linear Algebra and Python Implementation

Throughout the book, many computational techniques rely on the operations on the vectors and matrices. Therefore, the readers are assumed to know the fundamental concepts in linear algebra and their realization via Python. This chapter provides a brief overview of fundamental linear algebra concepts and their Python implementations.

A.1. Vector and Matrix

A vector is a one-dimensional array representing a point or direction in space. For instance:

$$\mathbf{v} = \begin{bmatrix} v_1 \\ v_2 \\ \vdots \\ v_n \end{bmatrix}$$

A matrix is a two-dimensional array that represents a linear transformation:

$$\mathbf{A} = \begin{bmatrix} a_{11} & a_{12} & \cdots & a_{1n} \\ a_{21} & a_{22} & \cdots & a_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ a_{m1} & a_{m2} & \cdots & a_{mn} \end{bmatrix}$$

A.2. Norm, Determinant and Inverse

The **norm** of a vector $\mathbf{v}$ measures its length:

$$\|\mathbf{v}\| = \sqrt{v_1^2 + v_2^2 + \cdots + v_n^2}$$

The **determinant** of a square matrix $\mathbf{A}$ quantifies its scale transformation:

$$\det(\mathbf{A}) = a_{11}(a_{22}a_{33} - a_{23}a_{32}) - a_{12}(a_{21}a_{33} - a_{23}a_{31}) + a_{13}(a_{21}a_{32} - a_{22}a_{31})$$

The **inverse** of a square matrix $\mathbf{A}$ satisfies:

$$\mathbf{A}^{-1}\mathbf{A} = \mathbf{I}$$

A.3. Multiplications

Scalar-Vector Multiplication

$$\mathbf{c} \cdot \mathbf{v} = \begin{bmatrix} cv_1 \\ cv_2 \\ \vdots \\ cv_n \end{bmatrix}$$

Matrix-Vector Multiplication

$$\mathbf{A}\mathbf{v} = \begin{bmatrix} a_{11}v_1 + a_{12}v_2 \\ a_{21}v_1 + a_{22}v_2 \\ \vdots \end{bmatrix}$$

Matrix-Matrix Multiplication

$$(\mathbf{A} \cdot \mathbf{B})_{ij} = \sum_k a_{ik} b_{kj}$$

Einstein Summations

A.4. Inner and Outer Product

The inner product of two vectors **a** and **b**:

$$\mathbf{a} \cdot \mathbf{b} = \sum_i a_i b_i$$

The outer product of two vectors **a** and **b**:

$$(\mathbf{a} \otimes \mathbf{b})_{ij} = a_i b_j$$

A.5. Dirac Notation

In quantum mechanics, vectors are represented in Dirac notation as kets $|\psi\rangle$. The inner product is written as:

$$\langle\phi|\psi\rangle = \sum_i \phi_i^* \psi_i$$

The outer product is represented as:

$$|\phi\rangle\langle\psi|$$

A.6. Eigenvalue Problem

For a square matrix **A**:

$$\mathbf{A}\mathbf{x} = \lambda\mathbf{x}$$

where λ is the eigenvalue and **x** is the eigenvector.

$$\det(\mathbf{A} - \lambda\mathbf{I}) = 0$$

A.7. Numerical Computation in Python

NumPy provide powerful numerical computation capabilities in Python, similar to MATLAB. NumPy offers efficient array operations and linear algebra functions. Here's a demonstration of basic linear algebra operations using NumPy:

```python
import numpy as np

# Vector and matrix definitions
v = np.array([1, 2, 3])
A = np.array([[1, 2], [3, 4]])

# Norm
norm = np.linalg.norm(v)
print("Norm:", norm)

# Determinant
det = np.linalg.det(A)
print("Determinant:", det)

# Inverse
inv = np.linalg.inv(A)
print("Inverse:\n", inv)

# Einstein summation for dot product
a = np.array([1, 2, 3])
b = np.array([4, 5, 6])
dot_product = np.einsum("i,i->", a, b)
print("Dot product (Einstein summation):", dot_product)

# Matrix multiplication using Einstein summation
C = np.einsum("ik,kj->ij", A, B)
print("Matrix multiplication (Einstein summation):\n", C)

# Eigenvalues and eigenvectors
eigvals, eigvecs = np.linalg.eig(A)
print("Eigenvalues:", eigvals)
print("Eigenvectors:\n", eigvecs)
```

In addition to Numpy, SciPy extends these capabilities with additional scientific computing tools. For instance, SciPy provides functions for optimization, integration, interpolation, special functions (e.g., spherical harmonics), and more. Both Numpy and SciPy are widely used in scientific computing and data analysis.

For symbolic mathematics, SymPy provides functionality similar to Mathematica and Maple, allowing for symbolic manipulation of mathematical expressions.

A.8. Advanced Usage Python

In Python and many other languages, summation can be made highly efficient through vectorization and broadcasting, which leverage NumPy's optimized C-implementation to perform operations on arrays without explicit loops. This avoids the overhead of Python loops and improves performance significantly.

```python
import numpy as np

```

```python
# Example: Summing elements of a vector
a = np.array([1, 2, 3, 4, 5])
total = np.sum(a)
print("Sum of vector elements:", total)

# Example: Summing a matrix with a vector
A = np.array([[1, 2, 3], [4, 5, 6], [7, 8, 9]])
b = np.array([10, 20, 30])

# Broadcasting the vector b along rows of A
result = A + b
print("Matrix and vector summation (broadcasted):\n", result)

# Example: Tensor summation using Einstein summation
C = np.einsum('ij,jk->ik', A, A.T)
print("Result of Einstein summation:\n", C)
```

B. Spherical Harmonics

Spherical harmonics play a crucial role in quantum mechanics and computational materials modeling. They are widely used to represent functions defined on the surface of a sphere, providing a natural basis for angular-dependent phenomena such as atomic orbitals, molecular vibrations, and scattering processes.

B.1. Mathematical Definition

Spherical harmonics, denoted as $Y_{lm}(\theta, \phi)$, are the eigenfunctions of the angular part of the Laplacian operator in spherical coordinates. They form a complete, orthonormal set of functions defined on the unit sphere, satisfying the spherical Schrödinger equation:

$$\nabla^2 Y_{lm}(\theta, \phi) + l(l+1) Y_{lm}(\theta, \phi) = 0 \tag{B.1}$$

where l is the degree (non-negative integer), m is the integer such that $-l \leq m \leq l$, θ and ϕ are the polar angle (0 to π) and the azimuthal angle (0 to 2π).

The general form of spherical harmonics can be written as:

$$Y_{lm}(\theta, \phi) = N_{lm} P_{lm}(\cos\theta) e^{i\phi} \tag{B.2}$$

where N_{lm} is a normalization constant ensuring orthonormality,

$$N_{lm} = \sqrt{\frac{(2l+1)(l-m)!}{4\pi(l+m)!}}$$

and P_{lm} are the associated Legendre polynomials

$$P_{lm}(x) = (-1)^m (1-x^2)^{m/2} \frac{d^m}{dx^m} P_l(x)$$

The associated Legendre polynomials satisfy an orthogonality condition:

$$\int_{-1}^{1} P_{lm}(x) P_{l'm}(x)\, dx = 0 \quad \text{for } l \neq l'.$$

In addition, it can be computed recursively using:

$$P_{l+1,m}(x) = \frac{(2l+1) x P_{lm}(x) - (l+m) P_{l-1,m}(x)}{l-m+1} \tag{B.3}$$

B.2. Examples of Real Spherical Harmonics

For specific values of l and m, the associated Legendre polynomials are

$$P_{lm}(x) = \begin{cases} 1, & l=0, m=0, \\ x, & l=1, m=0, \\ -\sqrt{1-x^2}, & l=1, m=1, \\ \frac{1}{2}(3x^2-1), & l=2, m=0, \\ -3x\sqrt{1-x^2}, & l=2, m=1, \\ 3(1-x^2), & l=2, m=2. \end{cases}$$

And the corresponding Y_{lm} values are,

$$Y_{lm}(\theta,\phi) = \begin{cases} \frac{1}{\sqrt{4\pi}}, & l=0, m=0, \\ \sqrt{\frac{3}{4\pi}}\sin\theta\sin\phi, & l=1, m=-1, \\ \sqrt{\frac{3}{4\pi}}\cos\theta, & l=1, m=0, \\ \sqrt{\frac{3}{4\pi}}\sin\theta\cos\phi, & l=1, m=1, \\ \frac{1}{4}\sqrt{\frac{5}{\pi}}(3\cos^2\theta-1), & l=2, m=0. \end{cases}$$

B.3. The Real Valued Spherical Harmonics

While spherical harmonics are inherently complex-valued, many physical problems require real-valued representations. These are constructed as linear combinations of the complex functions:

$$Y_{lm}^{\mathrm{Re}}(\theta,\phi) = \begin{cases} \sqrt{2}(-1)^m \mathrm{Im}[Y_{l|m|}(\theta,\phi)], & m<0 \\ Y_{l0}(\theta,\phi), & m=0 \\ \sqrt{2}(-1)^m \mathrm{Re}[Y_{lm}(\theta,\phi)], & m>0 \end{cases} \tag{B.4}$$

This formulation eliminates complex components, making them more suitable for numerical computation and physical interpretations, such as in DFT pseudopotentials and molecular modeling.

B.4. Calculation with Python

While it is relatively straightforward to implement spherical harmonics calculations manually, **SciPy** provides highly optimized and efficient methods for evaluating these functions. Below is an example demonstrating the use of *scipy.special.sph_harm* to compute the spherical harmonics for given quantum numbers.

```python
import numpy as np
from scipy.special import sph_harm

# Quantum numbers
l, m = 1, 0

```

```python
# Define angles in radians
phi = np.linspace(0, 2 * np.pi, 100)        # azimuthal angle
theta = np.linspace(0, np.pi, 100)          # polar angle

# Evaluate spherical harmonics
Y_lm = sph_harm(m, l, phi, theta[:, None])  # Broadcasting (theta, phi)

# Print the real and imaginary parts
print("Real part:\n", np.real(Y_lm))
print("Imaginary part:\n", np.imag(Y_lm))
```

C. Ewald Summation

Ewald summation is a computational technique used to efficiently evaluate the long-range Coulomb interactions in systems with periodic boundary conditions, such as crystals and charged systems. It is widely applied in both DFT and MD simulations.

C.1. Motivation

In a periodic system, we often need to compute the total electrostatic energy for charged particles. The general formula is

$$E = \frac{1}{2} \sum_{\mathbf{l}}^{*} \sum_{ij} \frac{Z_i Z_j}{|\mathbf{r}_{ij} + \mathbf{l}|} \tag{C.1}$$

where $\mathbf{l}$ denote the lattice vectors, i and j are the particle indices, Z_i and Z_j are the charges, and r_{ij} is the distance vector. To compute the total sum, one ideally needs to go over all lattice vectors until the contribution becomes negligible at a long distance. The * symbol on the top of first summation highlights that the summation should exclude the cases of self interaction, namely when $l = 0$ and $i = j$, as it will lead to division by zero of $1/r$.

However, the $1/r$ term decays very slowly even at a long distance. Direct summation over all periodic images converges very slowly or may even diverge.

C.2. Decomposition of $1/r$

We use the error function (erf) and its complementary function (erfc), defined as:

$$\mathrm{erf}(x) = \frac{2}{\sqrt{\pi}} \int_0^x e^{-t^2} dt,$$

$$\mathrm{erfc}(x) = 1 - \mathrm{erf}(x).$$

In these terms, $\mathrm{erf}(x)$ approaches 1 as $x \to \infty$ and 0 as $x \to 0$, while $\mathrm{erfc}(x)$ approaches 0 as $x \to \infty$ and 1 as $x \to 0$.

As such, we rewrite the $1/r$ term as follows

$$\frac{1}{r} = \frac{1}{r}\left[\mathrm{erfc}(\eta r) + \mathrm{erf}(\eta \mathrm{r})\right] = \frac{\mathrm{erfc}(\eta r)}{r} + \frac{\mathrm{erf}(\eta r)}{r}.$$

To compute the second term more efficiently, we shall convert it to the reciprocal space.

$$f(g) = 4\pi \int_0^\infty \frac{\mathrm{erf}(\eta \mathrm{r})}{r} \frac{\sin(gr)}{gr} r^2 dr = \frac{4\pi}{g} \int_0^\infty \mathrm{erf}(\eta \mathrm{r}) \sin(gr) dr$$

Use the definition of the error function,

$$\begin{aligned} f(g) &= \frac{4\pi}{g} \int_0^\infty \left[\frac{2}{\sqrt{\pi}} \int_0^{\eta r} e^{-u^2} du \right] \sin(gr) dr \\ &= \frac{8\sqrt{\pi}}{g} \int_0^\infty \int_0^{\eta r} e^{-u^2} \sin(gr) du dr \\ &= \frac{8\sqrt{\pi}}{g} \int_0^{\eta r} e^{-u^2} \left[\int_{r=u/\eta}^\infty \sin(gr) dr \right] du \\ &= \frac{8\sqrt{\pi}}{g} \int_0^{\eta r} e^{-u^2} \frac{\cos(gu/\eta)}{g} du \\ &= \frac{8\sqrt{\pi}}{g^2} \int_0^{\eta r} e^{-u^2} \cos(gu/\eta) du \end{aligned}$$

Now we use the known Gaussian–cosine integral (for $\alpha > 0$):

$$\int_0^\infty e^{-x^2} \cos(\alpha x)\, dx = \frac{\sqrt{\pi}}{2} \exp\left(-\frac{\alpha^2}{4} \right).$$

Here $\alpha = g/\eta$, plug it back

$$f(g) = \frac{8\sqrt{\pi}}{g^2} \frac{\sqrt{\pi}}{2} \exp\left(-\frac{g^2}{4\eta^2} \right) = \frac{4\pi}{g^2} \exp\left(-\frac{g^2}{4\eta^2} \right) \tag{C.2}$$

Hence the final expression of $1/r$ is

$$\frac{1}{r} = \frac{\mathrm{erfc}(\eta r)}{r} + \frac{4\pi}{g^2} \exp\left(\frac{g^2}{4\eta^2} \right) \tag{C.3}$$

Here, the first term $\mathrm{erfc}(\eta r)$ represents a short range behavior since it is large at a short distance and then rapidly converge to 0 at the long distance. On the other hand, the second term $\mathrm{erf}(\eta r)$ approaches to $1/r$ at large distances and hence it represent the long range interaction.

Figure C.1 illustrates the trends of the decay rates for the short-range and long-range terms as functions of η. The decay behavior of these terms is strongly influenced by the choice of η. Key observations include:

1. For a large η, the short-range term decays faster, reducing the real-space cutoff.

2. For a small η, the long-range term decays faster in reciprocal space, requiring fewer Fourier terms, but increasing cost.

Therefore, a good choice of η should balance the real-space and reciprocal-space contributions, thus optimizing computational cost while ensuring numerical accuracy.

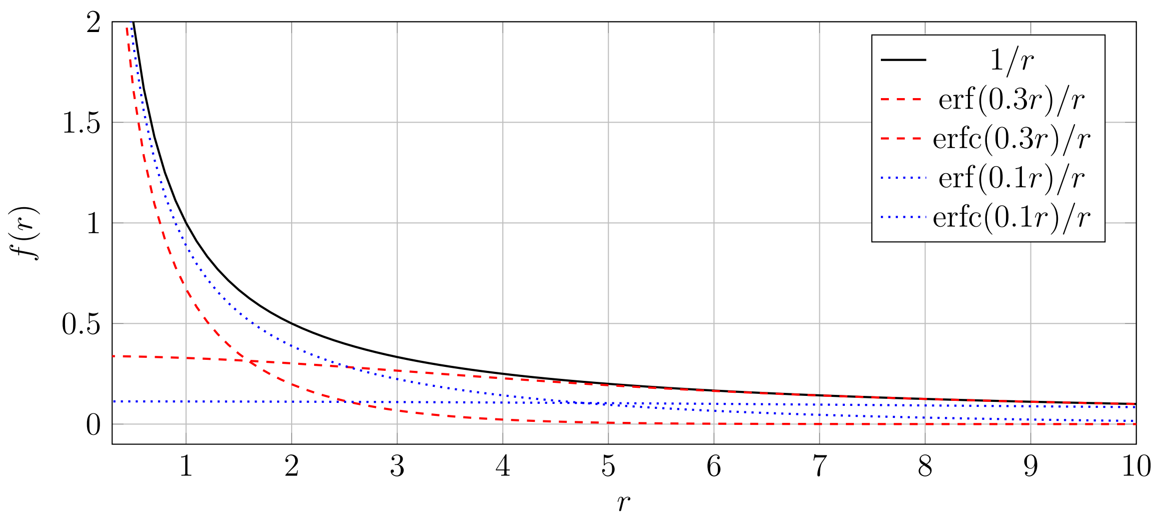


Figure C.1: The break down of $1/r$ term with different η choices.

C.3. Total Energy and Its Correction

Using the decomposition, eq. C.1 can be split into several terms.

The real space summation is straightforward to evaluate.

$$E_{\text{real}} = \frac{1}{2}\sum_{ij} Z_i Z_j \left[\sum_{\mathbf{l}} \frac{\mathrm{erfc}(\eta|\mathbf{R}_i - \mathbf{R}_j - \mathbf{l}|)}{|\mathbf{R}_i - \mathbf{R}_j - \mathbf{l}|}\right] \tag{C.4}$$

The reciprocal space sum is first expressed as,

$$E_{\text{reciprocal}} = \frac{1}{2}\int d^3r \int d^3r' \rho(r)\rho(r')\frac{\mathrm{erf}(\eta|\mathbf{r}-\mathbf{r}'|)}{|\mathbf{r}-\mathbf{r}'|}$$

where

$$\rho(\mathbf{r}) = \sum_{j=1}^{N} Z_j \delta(\mathbf{r} - \mathbf{r}_j)$$

This integral can be evaluated vi Fourier transform,

$$\rho(\mathbf{G}) = \int_{\Omega} \rho(\mathbf{r}) e^{-i\mathbf{k}\cdot\mathbf{r}} d^3\mathbf{r} = \sum_{j=1}^{N} Z_j e^{-i\mathbf{G}\cdot\mathbf{r_j}}$$

Combining it with eq. C.2

$$E_{\text{reciprocal}} = \frac{1}{2\Omega}\sum_{\mathbf{G}} \frac{4\pi}{G^2} \exp\left(-\frac{G^2}{4\eta^2}\right)\left|\rho(\mathbf{G})\right|^2$$

$$\left|\rho(\mathbf{G})\right|^2 = \sum_i \sum_j Z_i Z_j e^{i\mathbf{G}\cdot(\mathbf{r}_i - \mathbf{r}_j)} = \sum_{ij} Z_i Z_j [\cos[\mathbf{G}\cdot(\mathbf{r}_i - \mathbf{r}_j)] + i\sin[\mathbf{G}\cdot(\mathbf{r}_i - \mathbf{r}_j)]$$

Since the energy must be in real value, we can omit the sin term and get the final expression as

$$E_{\text{reciprocal}}(G \neq 0) = \frac{1}{2\Omega}\sum_{ij} Z_i Z_j \sum_{\mathbf{G}} \frac{4\pi}{G^2} \exp\left(-\frac{G^2}{4\eta^2}\right) \cos[\mathbf{G}\cdot(\mathbf{r}_i - \mathbf{r}_j)] \tag{C.5}$$

In addition, there still exist two issues in practice. First, this real space sum ignores the case of $r = 0$. Hence we need to subtract it. At $r = 0$, the self-energy per charge is:

$$E_{\text{self}} = \lim_{r\to 0} \frac{1}{2}\frac{\text{erfc}(\eta r)}{r}.$$

Using the Taylor expansion of erfc near $r = 0$:

$$\text{erfc}(\eta r) \approx 1 - \frac{2\eta r}{\sqrt{\pi}} \quad \rightarrow \quad \frac{\text{erfc}(\eta r)}{r} \approx \frac{1}{r} - \frac{2\eta}{\sqrt{\pi}}$$

The leading divergence is $1/r$ that cannot be counted, but the finite part is:

$$E_{\text{self}} = -\frac{2\eta}{\sqrt{\pi}} \sum Z_i^2$$

Second, periodic systems assume a neutralizing background charge to ensure charge neutrality. The interaction between the charges $\rho_{\text{pts}}(\mathbf{r})$ and this background ($\rho_{\text{bg}}(\mathbf{r}) = \sum_i Z_i/\Omega$) also needs to be corrected.

$$\begin{aligned}
E_{\text{pts-bg}} &= \frac{1}{2}\int d^3\mathbf{r} \int d^3\mathbf{r}' \rho_{\text{pts}}(\mathbf{r})\rho_{\text{bg}}(\mathbf{r}')\frac{\text{erf}(\eta|\mathbf{r}-\mathbf{r}'|)}{|\mathbf{r}-\mathbf{r}'|} \\
&= -\frac{\sum_i Z_i}{\Omega}\int_\Omega d^3\mathbf{r}\rho_{\text{pts}}(\mathbf{r})\int_\Omega d^3\mathbf{r}'\frac{\text{erf}(\eta|\mathbf{r}-\mathbf{r}'|)}{|\mathbf{r}-\mathbf{r}'|} \\
&- \frac{\sum_i Z_i}{\Omega}\sum_i Z_i \int_\Omega d^3\mathbf{r}'\frac{\text{erf}(\eta|\mathbf{r}-\mathbf{r}'|)}{|\mathbf{r}-\mathbf{r}'|}
\end{aligned}$$

For all points in a cubic box, the integral can be analytically evaluated

$$\int_\Omega d^3\mathbf{r}'\frac{\text{erf}(\eta|\mathbf{r}-\mathbf{r}'|)}{|\mathbf{r}-\mathbf{r}'|} = \frac{\pi}{\eta^2}$$

Using this relation,

$$E_{\text{pts-bg}} = -\frac{1}{\Omega}\frac{\pi}{\eta^2}\left(\sum_i Z_i\right)^2$$

Hence the final expression of energy is

$$\begin{aligned}
E &= E_{\text{real}} + E_{\text{reciprocal}} + E_{\text{self}} + E_{\text{pts-bg}} \\
&= \frac{1}{2}\sum_{i\neq j} Z_i Z_j\left[\frac{\text{erfc}(\eta|\mathbf{R}_i-\mathbf{R}_j-\mathbf{T}|)}{|\mathbf{R}_i-\mathbf{R}_j-\mathbf{T}|} + \frac{4\pi}{\Omega}\sum_{\mathbf{G}\neq 0}\frac{1}{\hat{\mathbf{G}}^2}\exp\left(\frac{\hat{\mathbf{G}}^2}{4\eta^2}\right)\cos[\mathbf{G}\cdot(\mathbf{R}_i-\mathbf{R}_j)]\right] \\
&- \frac{2\eta}{\sqrt{\pi}}\sum_i Z_i^2 - \frac{\pi}{\Omega\eta^2}\left(\sum_i Z_i\right)^2
\end{aligned} \tag{C.6}$$

This idea of Ewald summation is schematically shown in Fig. C.2

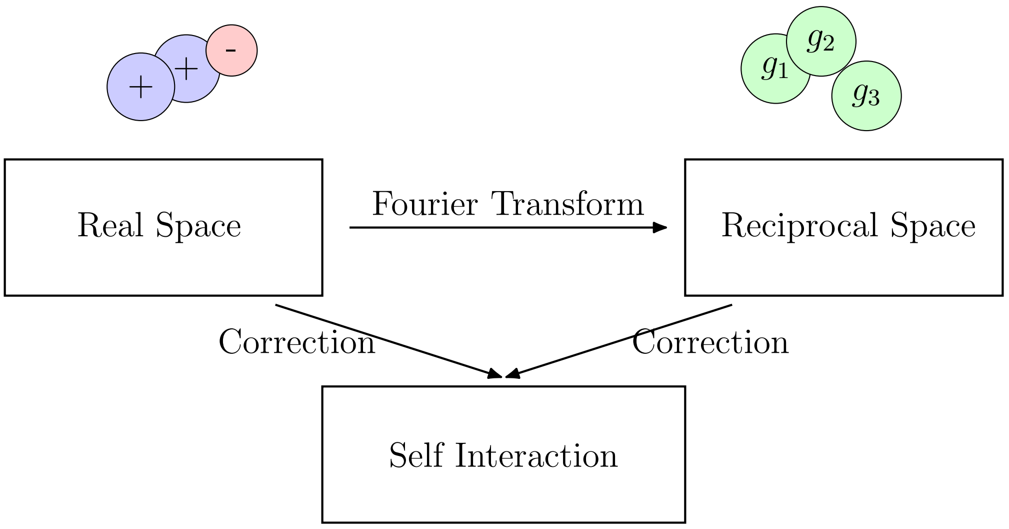


Figure C.2: The general idea of Ewald Summation.

C.4. Practical Setting of Parameters

The balance between real-space and reciprocal-space terms is controlled by η (the splitting parameter). For small systems, real-space sums can be computationally dominant because the number of image cells considered may be large. For large systems, reciprocal-space sums dominate because the number of reciprocal lattice vectors scales cubically with system size.

To select an appropriate value for η, we first need to define the truncation of $G_{\max}$, the maximum reciprocal lattice vector. This truncation ensures that both the short-range and long-range terms decay below a specified precision threshold when $G = G_{\max}$.

$$\exp\left(-\frac{G_{\max}^2}{4\eta^2}\right) = \epsilon \quad \rightarrow \quad \eta = \frac{G_{\max}}{2\sqrt{-\log\epsilon}}$$

$$\frac{\operatorname{erfc}(\eta t_{\max})}{t_{\max}} \leq \varepsilon \quad \rightarrow \quad t_{\max} \geq \frac{\sqrt{-\log\epsilon}}{\eta}$$

Hence, when $g_{\max} = 2.0$ and $\epsilon = 10^{-8}$, η can be approximated as

$$\eta = \frac{g_{\max}}{2\sqrt{-\log(10^{-8})}} \approx 0.2329953$$

C.5. Python Ewald Simulation for a Silicon Crystal

Below is the Python implementation for simulating the nuclear-nuclear energy for the cubic diamond silicon according to eq. C.6.

```python
import numpy as np
from scipy.special import erfc

def ewald(lattice, positions, Zvals, gmax=2.0, epsilon=1e-8):
    """
    Ewald summation of the nuclear-nuclear energy (eq. B.6)

    Args:
        lattice (np.ndarray): Lattice vectors
        positions (np.ndarray): Atomic positions
        Zvals (np.ndarray): Atomic charges
        gmax (float): Reciprocal space cutoff
    """
    # Initialize variables
    Natoms = len(positions)
    rec_lattice = 2 * np.pi * np.linalg.inv(lattice)
    t = lattice.T
    g = rec_lattice.T
    Omega = np.linalg.det(lattice)
    positions = positions @ lattice

    # Get eta and cutoff Parameters
    gexp = -np.log(epsilon)
    eta = np.sqrt(gmax**2 / gexp) / 2
    tmax = np.sqrt(0.5 * gexp) / eta

    # Self-energy correction
    ewald = -2 * eta * np.sum(Zvals ** 2) / np.sqrt(np.pi)

    # background charge subtraction
    ewald -= (np.pi * np.sum(Zvals)**2) / (Omega * eta ** 2)

    def get_lattice_translations(t, tmax):
        # Generate lattice translations
        tm = np.linalg.norm(t, axis=1)
        m = np.ceil(tmax / tm + 1.5).astype(int)
        I, J, K = np.meshgrid(np.arange(-m[0], m[0]+1),
                              np.arange(-m[1], m[1]+1),
                              np.arange(-m[2], m[2]+1), indexing='ij')
        T = I[..., None] * t[0] + J[..., None] * t[1] + K[..., None] *
  t[2]
        T = T.reshape(-1, 3)
        return T

    T = get_lattice_translations(t, tmax)
    G = get_lattice_translations(g, gmax)

    # Compute prefactor for E_reci
    G2 = np.sum(G**2, axis=1)
    mask = G2 > 1e-8  # Exclude G=0
    G = G[mask]
    G2 = G2[mask]
    exp_term = np.exp(-0.25 * G2 / eta ** 2)
    reci_factor = 4 * np.pi / Omega * exp_term / G2

    for i in range(Natoms):
        for j in range(Natoms):
            ZiZj = Zvals[i] * Zvals[j]
            dR = positions[i] - positions[j]

            # Real space
            rmag = np.linalg.norm(dR - T, axis=1)
            mask = (rmag > 1e-8) | (i != j)
            E_real = np.sum(erfc(rmag[mask] * eta) / rmag[mask])

            # Reciprocal space
            E_reci = np.sum(reci_factor * np.cos(G @ dR.T))
            ewald += ZiZj * (E_real + E_reci)

    return 0.5 * ewald

if __name__ == "__main__":

    lattice = 5.13155 * np.array([[0, 1, 1], [1, 0, 1], [1, 1, 0]])
    positions = np.array([[0, 0, 0], [0.25, 0.25, 0.25]])
    Zvals = np.array([4.0, 4.0])
    print(ewald(lattice, positions, Zvals))

    # Total Energy -8.397927400714138
```

For a simple system like cubic diamond with 2 atoms, the code should be sufficient to get a rather accurate result. For a relative large system in DFT calculation, further optimization through vectorization or parallelization will be needed.

C.6. Particle Mesh Ewald Summation

The standard Ewald summation technique, while accurate, involves computing pairwise interactions in both real space and reciprocal space, making it scale as $O(N^2)$, where N is the number of particles. For large systems of MD simulation involving millions of atoms, this computational cost becomes prohibitive.

The Particle Mesh Ewald (PME) method addresses this limitation by approximating the reciprocal-space sums ($E_{\text{reciprocal}}$) using Fast Fourier Transforms. In the mean time, B-splines is used to map charges onto the mesh, ensuring smooth transitions. Thanks to this simplification, computational complexity scales as $O(N \log N)$ due to the use of FFT. Hence it is highly scalable and suitable for large systems with millions of particles. For more details, please refer to the work by Darden and coworkers [46].

D. Wigner-D matrix and Clebsch-Gordan Coefficients

D.1. Introduction

In quantum mechanics, the Wigner-D matrix and Clebsch-Gordan coefficients are indispensable tools for analyzing angular momentum. The Wigner-D matrix describes the rotation operator in the angular momentum eigenstate basis, while the Clebsch-Gordan coefficients facilitate the coupling of angular momenta, which is crucial in multi-particle quantum systems.

D.2. Wigner-D matrix

The Wigner-D matrix $D^j_{m\prime m}(\alpha,\beta,\gamma)$ is the matrix element of the rotation operator $\hat{R}(\alpha,\beta,\gamma)$ acting on an angular momentum eigenstate $|j,m\rangle$:

$$D^j_{m\prime m}(\alpha,\beta,\gamma) = \langle j,m'|\hat{R}(\alpha,\beta,\gamma)|j,m\rangle$$

in which α,β,γ are the Euler angles and j is the total angular momentum quantum number, m and m' are the magnetic quantum numbers.

Explicitly, the Wigner-D matrix is:

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

Here, $d^j_{m'm}(\beta)$ is the reduced Wigner-d matrix.

D.2.1 Properties of the Wigner-D Matrix

The Wigner-D matrices satisfy the orthogonality relation:

$$\int_0^{2\pi}\int_0^{\pi}\int_0^{2\pi} D^{j*}m'm(\alpha,\beta,\gamma)D^{j'}m'`m'(\alpha,\beta,\gamma)\sin\beta\,d\alpha\,d\beta\,d\gamma = \frac{8\pi^2}{2j+1}\delta_{jj'}\delta_{mm'}\delta_{m'm''}$$

It is also symmetric

$$D^j_{m\prime m}(\alpha,\beta,\gamma) = (-1)^{m-m'}D^j_{-m,-m'}(\alpha,\beta,\gamma)$$

D.3. Clebsch-Gordan Coefficients

Clebsch-Gordan coefficients $\langle j_1, m_1; j_2, m_2 | j, m\rangle$ express the relationship between the coupled angular momentum basis $|j, m\rangle$ and the uncoupled basis $|j_1, m_1\rangle|j_2, m_2\rangle$:

$$|j, m\rangle = \sum_{m_1, m_2} \langle j_1, m_1; j_2, m_2 | j, m\rangle |j_1, m_1\rangle |j_2, m_2\rangle$$

The Clebsch-Gordan coefficients are nonzero only if the following selection rules are satisfied:

$$m = m_1 + m_2, \qquad |j_1 - j_2| \leq j \leq j_1 + j_2$$

Similar to Wigner-D matrix, the Clebsch-Gordan Coefficients also satisfy

$$\sum_{j,m} \langle j_1, m_1; j_2, m_2 | j, m\rangle \langle j_1, m_1'; j_2, m_2' | j, m\rangle = \delta_{m_1, m_1'} \delta_{m_2, m_2'}$$

$$\langle j_1, m_1; j_2, m_2 | j, m\rangle = (-1)^{j_1 + j_2 - j} \langle j_2, m_2; j_1, m_1 | j, m\rangle$$

D.4. Python Computation

Modern Python libraries provide straightforward methods to compute the Wigner-D matrix and Clebsch-Gordan coefficients.

```python
from scipy.special import wigner_d
d = wigner_d(j=1, mp=1, m=0, beta=np.pi/4)
print(d)

from sympy.physics.quantum.cg import CG
cg = CG(1/2, 1/2, 1/2, -1/2, 1, 0).doit()
print(cg)
```

E. List of Popular Codes and Tools

While this book primarily serves as an introduction to popular atomistic modeling techniques, readers may wish to further expand their knowledge or conduct productive simulations. To support continued learning and practical applications, we provide the following list of resources for the readers.

E.1. DFT packages

For large-scale and high-performance simulations, several well-established commercial and open-source DFT software packages are available:

1. `VASP`: `https://www.vasp.at`, an efficient planewave DFT code based on the projector-augmented wave (PAW) method, suitable for bulk materials, surfaces, and interfaces, and many others.

2. `Quantum Espresso`: `https://www.quantum-espresso.org`, a popular open-source planewave DFT code, highly extensible and integrates various modules for different functionalities.

3. `ABINIT`: `https://abinit.github.io/abinit_web/`, an open-source planewave DFT code, with excellent support for excited states calculations.

4. `CP2K`: `https://www.cp2k.org`, a highly versatile code designed for DFT-based MD simulations. It implements the Gaussian and planewave (GPW) method and can handle both localized and delocalized systems.

5. `PySCF`: `https://pyscf.org`, Python-based Simulations of Chemistry Framework based on localized basis set. In addition to DFT, it also supports other high-end electronic structure methods such as coupled-cluster, and configuration interaction calculations.

The following packages are particularly well-suited for educational purposes, algorithm development, and testing:

1. `DFTK.jl`: `https://docs.dftk.org/stable/`

2. `PWDFT.jl`: `https://github.com/f-fathurrahman/PWDFT.jl`

E.2. Classical MD packages

Several popular classical MD simulation toolkits are also publicly available,

1. `LAMMPS`: `https://www.lammps.org`, a versatile and highly scalable MD engine to support many classical force fields for different types of materials modeling.
2. `Gromacs`: `https://www.gromacs.org`, a highly efficient MD package for biomolecular systems.
3. `NAMD`: `https://www.ks.uiuc.edu/Research/namd/`, a large biomolecular simulations code for large-scale parallel computing.
4. `JAXMD`: `https://jax-md.readthedocs.io`, a Python-based MD library built on JAX, providing automatic differentiation and GPU/TPU acceleration. It is particularly suitable for machine learning applications in molecular simulations.

For educational purposes, one may also be interested in `ASE`'s MD module (`https://wiki.fysik.dtu.dk/ase/ase/md.html`) or `JAXMD`.

E.3. Tight-binding Packages

The tight-binding (TB) method provides a simplified framework for studying the electronic properties of materials, making it ideal for large-scale simulations and modeling electronic transport properties. Several software packages implement the tight-binding method, with varying degrees of complexity and functionality.

1. `DFTB+`: `https://www.dftbplus.org`, an efficient quantum-mechanical simulation package based on the tight-binding approximation to DFT.
2. `CP2K`: `https://www.cp2k.org`, also implements several semi-empirical tight-binding (SE-TB) and DFTB approaches.
3. `Wannier90`: `http://www.wannier.org`, designed to compute maximally localized Wannier functions (MLWFs) and tight-binding parameters from DFT calculations.

For educational purposes, `pythtb` (`https://www.physics.rutgers.edu/pythtb/`) also provides several tools for modeling tight-binding systems and topological physics studies.

E.4. Phonon Packages

Phonon calculations are essential for understanding vibrational properties, thermal conductivity, specific heat, and electron-phonon interactions in materials. Below, we list some widely-used phonon calculation packages:

1. `Phononpy`: `https://phonopy.github.io/phonopy/`, A widely-used Python-based package for phonon calculations based on force constants for realistic materials calculations.
2. `ASE`'s Phonon Module: `https://wiki.fysik.dtu.dk/ase/` provides tools for prototypical phonon calculations within `ASE` workflows.

E.5. Visualization Packages

Visualization plays a crucial role in analyzing atomic structures, simulation trajectories, density maps, and phonon modes. Below are some widely-used tools for visualizing atomistic and molecular simulations:

1. `Ovito`: `https://www.ovito.org`, a powerful tool for visualizing and analyzing molecular dynamics and atomistic simulation data at a large scale.
2. `MDAnalysis`: `https://www.mdanalysis.org`, a Python library for analyzing molecular dynamics trajectories and associated data.
3. `VESTA`: `https://jp-minerals.org/vesta/en/`, a versatile visualization program designed for crystallographic structures and volumetric data.

E.6. Tools for Structure Manipulation and Analysis

These tools assist in creating, manipulating, and analyzing atomic structures. They are widely used in material modeling, data processing, and simulation preparation:

1. `ASE`: `https://wiki.fysik.dtu.dk/ase/`, a Python library designed for creating, manipulating, and running simulations of atomic structures.
2. `Pymatgen`: `https://pymatgen.org/`, a Python library for analyzing and manipulating crystal structures with a focus on materials informatics.
3. `PyXtal`: `https://pyxtal.readthedocs.io/`, a Python package for generating and analyzing crystal structures with symmetry constraints.

E.7. Online Database

In addition to standalone simulation software, several online databases and tools have been developed to facilitate materials discovery and computational modeling. These platforms provide precomputed data, interactive visualizations, and simulation frameworks, enabling rapid exploration and analysis of materials properties. Below is a list of widely used online resources:

1. **Materials Project**: `https://materialsproject.org` A comprehensive database offering computed properties of thousands of materials, including band structures, elastic constants, and formation energies. It supports data downloads, structure visualization, and API access for programmatic queries.
2. **AflowLib**: `http://www.aflowlib.org` An extensive repository for high-throughput materials data, including thermodynamic, electronic, and structural properties. It provides symmetry analysis and machine learning capabilities to accelerate materials design.
3. `OQMD`: `http://oqmd.org` A database focused on DFT-computed properties of crystalline materials. It is particularly useful for phase stability analysis and materials screening based on formation energies.

4. **JARVIS**: `https://jarvis.nist.gov` Developed by NIST, JARVIS combines DFT, machine learning, and experimental data for materials discovery. It supports 2D materials, topological materials, and quantum computations.

5. **Nomad**: `https://nomad-lab.eu` A platform that stores and processes input and output files from a wide variety of computational codes. It also provides visualization tools and machine-learning-based models for materials prediction.

E.8. Miscellaneous Resources

This section provides a list of useful online tools and resources that support various aspects of materials modeling, data analysis, and simulation setup.

1. **Phonon Demo**: `https://lampz.tugraz.at/~hadley/ss1/phonons/1d/1d2m.php`. An interactive visualization tool to simulate and understand phonon modes in one-dimensional systems.

2. **Basis Set Exchange**: `https://www.basissetexchange.org`, A comprehensive online repository of atomic basis sets used in quantum chemistry and solid-state physics calculations. It allows users to download standardized basis sets for various elements and methods.

3. **OpenKIM**: `https://openkim.org`. A repository of interatomic potentials and force fields for molecular dynamics and lattice dynamics simulations. OpenKIM also offers tools for benchmarking and validating force fields, ensuring compatibility with popular simulation codes like `LAMMPS`.

4. **Bilbao Crystallographic Server**: `https://www.cryst.ehu.es`. A powerful platform offering various tools for crystallography and symmetry analysis, including space group determination, band structure paths, and symmetry-adapted modes.

5. `Libxc`: `https://gitlab.com/libxc/libxc`. A library of exchange-correlation functionals used in DFT codes. It provides an extensive database of functionals and is widely integrated into many major DFT codes.

6. `Spglib`: `https://spglib.github.io/spglib/`. A symmetry-finding library that detects and analyzes crystal symmetries, computes space groups, and generates symmetry operations. It is frequently used by many packages.

7. `Seek-Path`: `https://www.materialscloud.org/work/tools/seekpath`. A tool for automatic band structure path generation in reciprocal space. It is widely used in DFT studies for plotting band structures.

Bibliography

[1] John L Finney and Leslie V Woodcock. Renaissance of bernal's random close packing and hypercritical line in the theory of liquids. *J. Phys.: Condens. Matter*, 26(46):463102, 2014.

[2] AI Kitaigorodskii. Organic chemical crystallography, con sultants bureau: New york, 1961 (originally published in russian by the press of the academy of sciences of the ussr, moscow, 1955); spek, al, single-crystal structure validation with the program platon. *J. Appl. Crystallogr*, 36:7–13, 2003.

[3] Berni Julian Alder and Thomas Everett Wainwright. Phase transition for a hard sphere system. *J. Chem. Phys.*, 27(5):1208–1209, 1957.

[4] J. B. Gibson, A. N. Goland, M. Milgram, and G. H. Vineyard. Dynamics of radiation damage. *Phys. Rev.*, 120:1229–1253, 1960.

[5] A. Rahman. Correlations in the motion of atoms in liquid argon. *Phys. Rev.*, 136:A405–A411, 1964.

[6] Hans C Andersen. Molecular dynamics simulations at constant pressure and/or temperature. *J. Chem. Phys.*, 72(4):2384–2393, 1980.

[7] Nosé Shuichi. Constant temperature molecular dynamics methods. *Prog. Theor. Phys. Suppl.*, 103:1–46, 1991.

[8] William G. Hoover. Canonical dynamics: Equilibrium phase-space distributions. *Phys. Rev. A*, 31:1695–1697, 1985.

[9] M. Parrinello and A. Rahman. Crystal structure and pair potentials: A molecular-dynamics study. *Phys. Rev. Lett.*, 45:1196–1199, 1980.

[10] Daan Frenkel and Berend Smit. *Understanding molecular simulation: from algorithms to applications.* Elsevier, 2023.

[11] P. Hohenberg and W. Kohn. Inhomogeneous electron gas. *Phys. Rev.*, 136:B864–B871, 1964.

[12] W. Kohn and L. J. Sham. Self-consistent equations including exchange and correlation effects. *Phys. Rev.*, 140:A1133–A1138, 1965.

[13] Seymour H Vosko, Leslie Wilk, and Marwan Nusair. Accurate spin-dependent electron liquid correlation energies for local spin density calculations: a critical analysis. *Can. J. Phys.*, 58(8):1200–1211, 1980.

[14] J. P. Perdew and Alex Zunger. Self-interaction correction to density-functional approximations for many-electron systems. *Phys. Rev. B*, 23:5048–5079, 1981.

[15] Choirun Nisaa Rangkuti, Suci Faniandari, A Suparmi, and Yanoar Pribadi Sarwono. Density functional calculations on h2 using 1 s slater type orbitals. *J. Chem. Educ.*, 101:172–180, 2023.

[16] Warren J Hehre, Robert F Stewart, and John A Pople. Self-consistent molecular-orbital methods. i. use of gaussian expansions of slater-type atomic orbitals. *J. Chem. Phys.*, 51(6):2657–2664, 1969.

[17] Yoyo Hinuma, Giovanni Pizzi, Yu Kumagai, Fumiyasu Oba, and Isao Tanaka. Band structure diagram paths based on crystallography. *Comp. Mat. Sci.*, 128:140–184, 2017.

[18] Marvin L. Cohen and Volker Heine. The fitting of pseudopotentials to experimental data and their subsequent application. volume 24 of *Solid State Physics*, pages 37–248. Academic Press, 1970.

[19] James R. Chelikowsky and Marvin L. Cohen. Electronic structure of silicon. *Phys. Rev. B*, 10:5095–5107, 1974.

[20] Martin M. Rieger and P. Vogl. Electronic-band parameters in strained $\mathrm{si}_{1-x}\mathrm{ge}_x$ alloys on $\mathrm{si}_{1-y}\mathrm{ge}_y$ substrates. *Phys. Rev. B*, 48:14276–14287, 1993.

[21] William C. Topp and John J. Hopfield. Chemically motivated pseudopotential for sodium. *Phys. Rev. B*, 7:1295–1303, 1973.

[22] Th. Starkloff and J. D. Joannopoulos. Local pseudopotential theory for transition metals. *Phys. Rev. B*, 16:5212–5215, 1977.

[23] S. Goedecker, M. Teter, and J. Hutter. Separable dual-space gaussian pseudopotentials. *Phys. Rev. B*, 54:1703–1710, 1996.

[24] C. Hartwigsen, S. Goedecker, and J. Hutter. Relativistic separable dual-space gaussian pseudopotentials from h to rn. *Phys. Rev. B*, 58:3641–3662, 1998.

[25] Miguel AL Marques, Micael JT Oliveira, and Tobias Burnus. Libxc: A library of exchange and correlation functionals for density functional theory. *Comp. Phys. Comm.*, 183(10):2272–2281, 2012.

[26] Susi Lehtola and Miguel AL Marques. Reproducibility of density functional approximations: How new functionals should be reported. *J. Chem. Phys.*, 159(11), 2023.

[27] E. R. Davidson. The iterative calculation of a few of the lowest eigenvalues and corresponding eigenvectors of large real-symmetric matrices. *J. Comput. Phys*, 17:87–94, 1975.

[28] Cornelius Lanczos. An iteration method for the solution of the eigenvalue problem of linear differential and integral operators, 1950.

[29] Andrew V. Knyazev. Toward the optimal preconditioned eigensolver: Locally optimal block preconditioned conjugate gradient method. *SIAM Journal on Scientific Computing*, 23(2):517–541, 2001.

[30] Péter Pulay. Convergence acceleration of iterative sequences. the case of scf iteration. *Chem. Phys. Lett.*, 73(2):393–398, 1980.

[31] Charles G Broyden. A class of methods for solving nonlinear simultaneous equations. *Mathematics of Computation*, 19(92):577–593, 1965.

[32] D. D. Johnson. Modified broyden's method for accelerating convergence in self-consistent calculations. *Phys. Rev. B*, 38:12807–12813, 1988.

[33] Fadjar Fathurrahman, Mohammad Kemal Agusta, Adhitya Gandaryus Saputro, and Hermawan Kresno Dipojono. Pwdft. jl: A julia package for electronic structure calculation using density functional theory and plane wave basis. *Comp. Phys. Comm.*, 256:107372, 2020.

[34] Paolo Giannozzi, Stefano Baroni, Nicola Bonini, Matteo Calandra, Roberto Car, Carlo Cavazzoni, Davide Ceresoli, Guido L Chiarotti, Matteo Cococcioni, Ismaila Dabo, Andrea Dal Corso, Stefano de Gironcoli, Stefano Fabris, Guido Fratesi, Ralph Gebauer, Uwe Gerstmann, Christos Gougoussis, Anton Kokalj, Michele Lazzeri, Layla Martin-Samos, Nicola Marzari, Francesco Mauri, Riccardo Mazzarello, Stefano Paolini, Alfredo Pasquarello, Lorenzo Paulatto, Carlo Sbraccia, Sandro Scandolo, Gabriele Sclauzero, Ari P Seitsonen, Alexander Smogunov, Paolo Umari, and Renata M Wentzcovitch. Quantum espresso: a modular and open-source software project for quantum simulations of materials. *J. Phys. Condensed Matter*, 21(39):395502 (19pp), 2009.

[35] G. Kresse and J. Furthmüller. Efficient iterative schemes for ab initio total-energy calculations using a plane-wave basis set. *Phys. Rev. B*, 54:11169–11186, 1996.

[36] Michael F. Herbst, Antoine Levitt, and Eric Cancès. Dftk: A julian approach for simulating electrons in solids. *Proc. JuliaCon Conf.*, 3:69, 2021.

[37] Ask Hjorth Larsen, Jens Jørgen Mortensen, Jakob Blomqvist, Ivano E Castelli, Rune Christensen, Marcin Dułak, Jesper Friis, Michael N Groves, Bjørk Hammer, Cory Hargus, Eric D Hermes, Paul C Jennings, Peter Bjerre Jensen, James Kermode, John R Kitchin, Esben Leonhard Kolsbjerg, Joseph Kubal, Kristen Kaasbjerg, Steen Lysgaard, Jón Bergmann Maronsson, Tristan Maxson, Thomas Olsen, Lars Pastewka, Andrew Peterson, Carsten Rostgaard, Jakob Schiøtz, Ole Schütt, Mikkel Strange, Kristian S Thygesen, Tejs Vegge, Lasse Vilhelmsen, Michael Walter, Zhenhua Zeng, and Karsten W Jacobsen. The atomic simulation environment—a python library for working with atoms. *J. Phys. Condensed Matter*, 29(27):273002, 2017.

[38] Atsushi Togo, Laurent Chaput, Terumasa Tadano, and Isao Tanaka. Implementation strategies in phonopy and phono3py. *J. Phys. Condens. Matter*, 35(35):353001, 2023.

[39] Stefano Baroni, Stefano de Gironcoli, Andrea Dal Corso, and Paolo Giannozzi. Phonons and related crystal properties from density-functional perturbation theory. *Rev. Mod. Phys.*, 73:515–562, 2001.

[40] B. I. Halperin and David R. Nelson. Theory of two-dimensional melting. *Phys. Rev. Lett.*, 41:121–124, 1978.

[41] Paul J. Steinhardt, David R. Nelson, and Marco Ronchetti. Bond-orientational order in liquids and glasses. *Phys. Rev. B*, 28:784–805, 1983.

[42] Albert P. Bartók, Risi Kondor, and Gábor Csányi. On representing chemical environments. *Phys. Rev. B*, 87:184115, 2013.

[43] Jörg Behler and Michele Parrinello. Generalized neural-network representation of high-dimensional potential-energy surfaces. *Phys. Rev. Lett.*, 98:146401, 2007.

[44] Jörg Behler. Atom-centered symmetry functions for constructing high-dimensional neural network potentials. *J. Chem. Phys.*, 134(7), 2011.

[45] Ralf Drautz. Atomic cluster expansion for accurate and transferable interatomic potentials. *Phys. Rev. B*, 99:014104, 2019.

[46] Tom Darden, Darrin York, and Lee Pedersen. Particle mesh ewald: An n log (n) method for ewald sums in large systems. *J. Chem. Phys.*, 98(12):10089–10092, 1993.