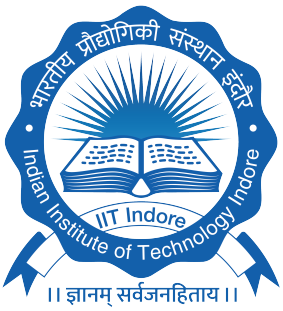
Solving Euler's equation for Non-Interacting Model Systems and Atoms

M.Sc. THESIS

By

ASHUTOSH PATEL
(2303131006)



DEPARTMENT OF CHEMISTRY
INDIAN INSTITUTE OF TECHNOLOGY
INDORE
May 2025

Solving Euler's equation for Non-Interacting Model Systems and Atoms

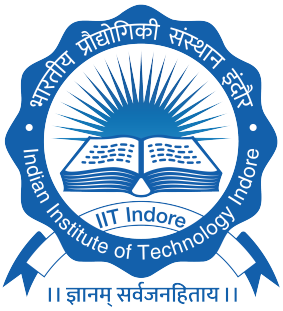
A THESIS

Submitted in partial fulfillment of the requirements for the award of the degree of

MASTER OF SCIENCE

By

ASHUTOSH PATEL



DEPARTMENT OF CHEMISTRY

INDIAN INSTITUTE OF TECHNOLOGY

INDORE

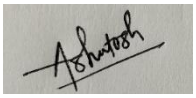
MAY 2025

INDIAN INSTITUTE OF TECHNOLOGY INDORE

CANDIDATE'S DECLARATION

I hereby certify that the work which is presented in the thesis entitled **SOLVING EULER'S EQUATION FOR NON-INTERACTING MODEL SYSTEMS AND ATOMS** in the partial fulfilment of the requirements for the award of the degree of **MASTER OF SCIENCE** and submitted in the **DEPARTMENT OF CHEMISTRY, Indian Institute of Technology Indore**, is an authentic record of my own work carried out during the time period from July, 2023 to May, 2025 under the supervision of **Prof. SATYA S. BULUSU, Professor**, Department of CHEMISTRY, Indian Institute of Technology Indore.

The matter presented in this thesis has not been submitted for the award of any other degree of this or any other institute.

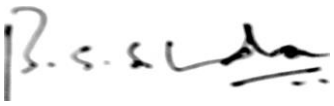


Signature of student with date

ASHUTOSH PATEL

- -

This is to certify that the above statement made by the candidate is correct to the best of my/our knowledge.

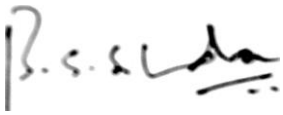


Signature of Thesis Supervisor with date

Prof. SATYA S. BULUSU

- -

ASHUTOSH PATEL, has successfully given his M.Sc. Oral Examination held on 15 May 2025.



16.05.2025

Signature of Supervisor of the M.Sc. thesis

Prof. Satya S. Bulusu

- -

Acknowledgements

I would like to thank my supervisor **Prof. Satya S. Bulsu**(Professor, Department of Chemistry, IIT Indore) for his continuous support, conviction, encouragement and invaluable advice during my lab work. I have been fortunate to have a supervisor who is deeply invested my work, consistently addressing my questions and concerns promptly.

I would like to acknowledge this assistance given to me by lab senior **Ms. Aparna Gangwar**. She helped me a lot by giving me guidance and encouragement whenever I was stuck in any problem. Their contribution was precious to me in this project. I feel lucky to work under her guidance. I also thank my other lab members for their valuable suggestions and help.

I would like to thank all my classmates and friends. Finally, I would like to thank my parents who have always been a constant source of support for me throughout my life.

(ASHUTOSH PATEL)

ABSTRACT

In this project, we develop a numerical approach to solve Euler's equation for many-body atomic systems within the framework of density functional theory (DFT) to calculate the electron density that minimizes the total energy functional. The energy functional includes contributions from kinetic energy, external potential, Hartree potential and exchange-correlation energy. The Pauli potential is incorporated to account for the quantum mechanical effects of fermionic antisymmetry, improving the representation of kinetic energy. Using a self-consistent field (SCF) method, the electron density is iteratively updated, starting from an initial guess, with the Hartree, exchange-correlation, and Pauli potentials computed at each step. The density is normalized to maintain the total number of electrons, and convergence is achieved when the error in density or energy falls below a defined threshold. This implementation employs Numerov methods for radial grid discretizations and numerical integration to evaluate potentials and energies. The results demonstrate the successful computation of electron density for multi-electron atomic systems, providing a robust framework for extending the method to more complex systems or incorporating advanced functionals.

Contents

ABSTRACT vii

1 Introduction 1

1.1 Overview . . . 1

1.2 Motivation . . . 3

1.3 Objective . . . 3

1.4 Organization of the Thesis . . . 4

2 Literature Review 5

2.1 Theory and Method . . . 5

2.1.1 Euler'ss equation for Non-Interacting Model systems 5

2.1.2 Gradient Descent Minimization and Energy Calculation . . . 6

2.1.3 Euler's equation for Interacting Atomic Systems . . . 7

3 Results and Discussion 11

3.1 Euler's Equation code: . . . 11

3.2 Euler's Equation code: . . . 16

3.2.1 Helium Atom . . . 16

3.2.2 Lithium, Beryllium and Neon Atom . . . 21

3.3 Results and Outputs . . . 26

3.3.1 Non-Interacting Model Systems . . . 26

3.3.2 Interacting Atomic System . . . 27

3.3.2.1 Helium Atom . . . 27

3.3.2.2 Lithium Atom . . . 28

3.3.2.3 Beryllium Atom 29
3.3.2.4 Neon Atom 30

4 Conclusion and Scope **31**

A Appendix Section **33**

A.0.1 Magnesium Atom 33
A.0.2 Nitrogen Atom 34
A.0.3 Argon Atom 35

List of Figures

3.1 Initial density for 2S 26
3.2 Optimized density for 2S 26
3.3 Initial density for 3S 26
3.4 Optimized density for 3S 26
3.5 Initial density for 4S 26
3.6 Optimized density for 4S 26
3.7 Radial density for Helium atom 27
3.8 Radial density for Lithium atom 28
3.9 Radial density for Beryllium atom 29
3.10 Radial density for Neon atom 30

A.1 Radial Density for Magnesium Atom 33
A.2 Radial density for Nitrogen atom 34
A.3 Radial density for Argon atom 35

List of Tables

3.1 Comparison of Reference and Calculated Energy and chemical potential (μ) 27

3.2 Comparison of Reference and Calculated Energy and chemical potential (μ) 28

3.3 Comparison of Reference and Calculated Energy and chemical potential (μ) 29

3.4 Comparison of Reference and Calculated Energy and chemical potential (μ) 30

A.1 Comparison of Reference and Calculated Energy and chemical potential (μ) 33

A.2 Comparison of Reference and Calculated Energy and chemical potential (μ) 34

A.3 Comparison of Reference and Calculated Energy and chemical potential (μ) 35

.

.

Chapter 1

Introduction

1.1 Overview

Density Functional Theory (DFT) has emerged as a crucial instrument in computational chemistry and materials research throughout the last half-century. In 1927, Thomas and Fermi originally introduced the idea of calculating electronic structure using electron density instead of the more complicated electronic wavefunction. The theoretical underpinnings of DFT were developed over thirty years later by Hohenberg and Kohn [1], who showed that the electron density of a many-electron system under an external potential is the only characteristic that can uniquely identify its ground state. The ability to represent the overall ground-state energy of a many-electron system as a functional of the electron density is a significant implication of this discovery. Furthermore, the Euler-Lagrange equation results from the ground-state density's adherence to the stationary energy principle:

$$\mu = \frac{\delta E[n]}{\delta n(r)} \tag{1.1}$$

where $n(r)$ denotes the electron density and μ represents the chemical potential, ensuring the conservation of the total number of electrons. The ground-state energy functional $E[n]$ is generally decomposed as:

$$E[n] = T[n] + E_{\text{ext}}[n] + E_{\text{coul}}[n] + E_{\text{XC}}[n] \tag{1.2}$$

where $T[n]$, $E_{\text{ext}}[n]$, $E_{\text{coul}}[n]$, and $E_{\text{XC}}[n]$ represent the non-interacting kinetic energy (KE), the interaction energy of the electron density with

the external potential, the classical Coulomb repulsion energy between the electrons, and the so-called exchange–correlation (XC) energy, respectively.

Unfortunately, only $E_{\text{ext}}[n]$ and $E_{\text{coul}}[n]$ have accurate functional forms. To do practical DFT-based calculations, one must rely on approximation forms for $T[n]$ [2] and $E_{\text{XC}}[n]$. Since the formal formulation of DFT, great work has been put into constructing XC functionals with increasing accuracy.

The Kohn-Sham formalism of DFT (KS-DFT) solves the difficulty of the KE functional by expressing it exactly in terms of a collection of non-interacting occupied orbitals. As a result of the availability of precise XC functionals throughout time, the Kohn-Sham formalism of DFT has become an essential tool for all modern first-principles-based electronic structure computing methods.

1.2 Motivation

The motivation behind solving Euler's equation in the context of Orbital-Free Density Functional Theory (OF-DFT) is to overcome the limitations of traditional quantum mechanical methods, such as Hartree-Fock and Kohn-Sham DFT, which require explicit computation of orbitals. These methods become computationally expensive and complex for large systems. Earlier implementations of OF-DFT also relied on approximate kinetic energy functionals, leading to inefficiencies and inaccuracies. By solving Euler's equation, derived from the variational principle of OF-DFT, we can directly minimize the total energy functional with respect to electron density, eliminating the need for orbitals. This approach offers a more computationally efficient and accurate method for calculating the radial electron density, particularly for spherically symmetric systems like atoms, reducing both mathematical and computational complexity while retaining the essential physical characteristics of the system.

1.3 Objective

- To solve Euler's equation for atoms using an exact form of the kinetic energy functional in an orbital-free density functional theory (OF-DFT) framework.
- To compute the radial electron density distribution for many-electron atomic systems.
- To implement a Numerov Method for the derived equations, ensuring proper boundary conditions and normalization constraints.
- To compare the obtained electron density distributions with the reference obtained from Gaussian for results validation.

1.4 Organization of the Thesis

The work done in the present thesis is organized in four chapters. The present chapter gives brief overview about the Density funtional Theory and Euler-Lagranze equation. Further it discusses the motivation towards solving Euler's equation for atoms.

- **Chapter 2,** presents the theory and methods behind the work.
- **Chapter 3,** sumarises the work done in thesis.
- **Chapter 4,** presents the conclusion of whole work and describe the future scope.

Chapter 2

Literature Review

2.1 Theory and Method

2.1.1 Euler'ss equation for Non-Interacting Model systems

The Density Functional Theory (DFT) offers a strong framework for characterizing systems of non-interacting fermions in addition to the numerical solution of the Schrödinger equation. DFT minimizes the system's total energy [3] in relation to the particle density. The particle density distribution can be ascertained by combining the external potential with the kinetic energy functional of non-interacting particles to create an energy minimization problem.

In this study, we concentrate on a system of N non-interacting fermions (where $N = 2, 3$) that are exposed to a Gaussian potential, which is provided by:

$$v(x) = -\sum_{k=1}^{3} \alpha_k \exp\left(-\frac{(x-\beta_k)^2}{2\gamma_k^2}\right), \tag{2.1}$$

where a random sample of the parameters α_k, β_k, and γ_k are taken from predetermined boundaries. It is contained within a one-dimensional box with x between -3.0 and 3.0. The eigenvalues E_i and eigenfunctions $\phi_i(x)$ are obtained by numerically solving the Schrödinger equation, and these orbitals are filled in accordance with the Pauli exclusion principle.

For $N = 2, 3$, we explore two configurations: one with two fermions having opposite spins in the lowest energy state (2S), and another where

the lowest two energy levels are singly occupied by spinless fermions (3S), and for $N = 3$) lowest three energy levels are singly occupied by spinless fermions (4S).

The electron density is given by:

$$n(x) = \sum_{i=1}^{N} f_i |\phi_i(x)|^2, \tag{2.2}$$

where f_i denotes the occupation number. The kinetic energy functional $T[n]$ is computed using the density $n(x)$, with the KE density $\tau(x)$ given by:

$$\tau(x) = \frac{1}{2} \sum_{i=1}^{N} |\nabla \phi_i(x)|^2 \tag{2.3}$$

The exact kinetic energy is computed as:

$$T_{\text{Exact}} = \int_{-\infty}^{\infty} \tau(x) dx \tag{2.4}$$

The functional derivative of the kinetic energy is:

$$\frac{\delta T_{\text{Exact}}[n]}{\delta n} = -v(x) + \mu \tag{2.5}$$

where μ is the chemical potential. The minimization [4] of the energy functional is carried out using an iterative gradient descent method, where the density is updated at each step:

$$n_{\text{new}}(x) = n_{\text{old}}(x) - \eta \frac{\delta L[n]}{\delta n} \tag{2.6}$$

Here, η is a constant step size, and the energy functional is minimized until the difference between the new and target densities is below a specified threshold.

2.1.2 Gradient Descent Minimization and Energy Calculation

In the iterative gradient descent method used to minimize the energy functional, the functional derivative of the Lagrangian with respect to the density is given by:

$$\frac{\delta L[n]}{\delta n} = \frac{\delta T_{\text{Exact}}[n]}{\delta n} + V(x) - \mu. \tag{2.7}$$

The iteration starts with a random initial density in order to determine the minimal energy density. Until the difference between the total

energy of the new density and the target density is less than 10^{-3}, the gradient descent process is repeated. With a step size of $\eta = 0.005$, the overall number of iterations is restricted to 100. The result discussion section displays the plots of the total energy minimum curves that were achieved during minimization for (2S , 3S and 4S) using the gradient descent method.

2.1.3 Euler's equation for Interacting Atomic Systems

The Hohenberg-Kohn theorem in density functional theory (DFT) says that the exact ground-state energy of a system of N electrons in an external potential may be formally stated as a functional $E[\rho]$ [5] solely in terms of the electron density $\rho(r)$. An essential consequence of this finding is that the total ground-state energy of a many-electron system may be represented as a function of electron density, and the ground-state density satisfies the energy stationary principle, yielding the Euler-Lagrange equation. This theorem uses the Euler equation to minimize $E[\rho]$ with the constraint condition $\int \rho(r)dr = N$. Euler's equation is given as follows:

$$\mu = \frac{\delta E[n]}{\delta n(r)} \tag{2.8}$$

The Kohn-Sham (KS) formalism of DFT introduces a partitioning of the total energy functional into different energy components:

$$E[\rho] = T_s[\rho] + E_{ee}[\rho] + E_{en}[\rho] + T_c[\rho] \tag{2.9}$$

where $T_s[\rho]$ [6] represents the non-interacting kinetic energy (KE) functional, $E_{ee}[\rho]$ represents the electron–electron interaction energy functional, and $E_{en}[\rho]$ denotes the energy functional corresponding to the interaction of the electrons with the external potential arising due to the electron–nucleus attractive interaction. The functional form for the electron–nuclear energy is given by:

$$E_{\text{en}}[\rho] = \int v_{\text{ext}}(r)\rho(r)dr \tag{2.10}$$

The final term in the equation, $T_c[\rho]$, represents the correlation kinetic energy, which accounts for the difference between the true kinetic energy and the non-interacting kinetic energy $T_s[\rho]$. This term, along with the exchange-correlation functional, plays a crucial role in determining the accuracy of density functional approximations.This approach allows for efficient computations of ground-state properties without the need for solving

the many-body Schrödinger equation directly. The Kohn-Sham (KS) formalism of DFT introduces a partitioning of the total energy functional into different energy components:

$$E[\rho] = T_s[\rho] + E_{ee}[\rho] + E_{en}[\rho] + T_c[\rho] \tag{2.11}$$

where $T_s[\rho]$ represents the non-interacting kinetic energy (KE) functional, $E_{ee}[\rho]$ is the electron-electron interaction energy functional, and $E_{en}[\rho]$ [7]denotes the energy functional corresponding to the interaction of electrons with the external potential. The energy associated with the electron-nuclear interaction is given by:

$$E_{en}[\rho] = \int \rho(\mathbf{r})\nu_{cn}(\mathbf{r})d\mathbf{r}, \tag{2.12}$$

where $\nu_{cn}(\mathbf{r})$ is the nuclear potential, which for an atom with nuclear charge Z is given by:

$$\nu_{en}(\mathbf{r}) = -\frac{Z}{|\mathbf{r}|}. \tag{2.13}$$

The kinetic energy functional $T_c[\rho]$ accounts for electron-electron correlation and is conventionally rewritten as:

$$E_{ee}[\rho] + T_c[\rho] = E_H[\rho] + E_{xc}[\rho], \tag{2.14}$$

where the Hartree energy $E_H[\rho]$ represents the classical Coulomb interaction of the electron density:

$$E_H[\rho] = \frac{1}{2}\iint \frac{\rho(\mathbf{r})\rho(\mathbf{r}')}{|\mathbf{r}-\mathbf{r}'|}d\mathbf{r}'d\mathbf{r}, \tag{2.15}$$

and $E_{xc}[\rho]$ represents the exchange-correlation energy, which includes exchange, Coulombic correlation, and kinetic correlation contributions. In the KS formalism, the exact form of the kinetic energy functional [8] $T_s[\rho]$ is not known in terms of density. Instead, it is computed using the KS orbitals $\phi_i(\mathbf{r})$ of N non-interacting electrons:

$$T_s[\rho] = \int t_s(\mathbf{r})d\mathbf{r} = \sum_i^N \int \frac{1}{2}|\nabla\phi_i(\mathbf{r})|^2 d\mathbf{r}, \tag{2.16}$$

where the electron density is obtained from the orbitals as:

$$\rho(\mathbf{r}) = \sum_i |\phi_i(\mathbf{r})|^2. \tag{2.17}$$

In contrast, orbital-free DFT (OF-DFT) seeks to bypass explicit orbital dependence by employing density-based approximations for the kinetic energy functional $T_s[\rho]$. In this approach, the kinetic energy functional is decomposed as:

$$T_s[\rho] = T_{\nu W}[\rho] + T_\theta[\rho], \tag{2.18}$$

where the first term, $T_{\nu W}[\rho]$, is the von Weizsäcker kinetic energy functional, given by:

$$T_{\nu W}[\rho] = \int t_{\nu W}(\mathbf{r}) d\mathbf{r} = \frac{1}{8} \int \frac{|\nabla \rho(\mathbf{r})|^2}{\rho(\mathbf{r})} d\mathbf{r}. \tag{2.19}$$

This term is exact for single orbital systems and bosonic systems. The second term, $T_\theta[\rho]$, represents the Pauli kinetic energy functional, which accounts for the many-body fermionic effects. Unlike the von Weizsäcker term, the exact form of $T_\theta[\rho]$ is unknown, and various approximate models have been proposed in the literature to represent it. Applying the variational principle under the constraint $\int \rho(\mathbf{r}) d\mathbf{r} = N$, a single KS-like equation is obtained for $\sqrt{\rho(\mathbf{r})}$:

$$\left(-\frac{1}{2}\nabla^2 + \nu_s([\rho];\mathbf{r}) + \nu_\theta([\rho];\mathbf{r}) \right) \sqrt{\rho(\mathbf{r})} = \mu \sqrt{\rho(\mathbf{r})}, \tag{2.20}$$

where μ is the chemical potential. Here, $\nu_s([\rho];\mathbf{r})$ is the KS potential, given by:

$$\nu_s([\rho];\mathbf{r}) = \nu_H([\rho];\mathbf{r}) + \nu_{XC}([\rho];\mathbf{r}) + \nu_{cn}([\rho];\mathbf{r}), \tag{2.21}$$

where $\nu_H([\rho];\mathbf{r})$, $\nu_{XC}([\rho];\mathbf{r})$, and $\nu_{cn}([\rho];\mathbf{r})$ are the Hartree, exchange-correlation, and electron-nuclear potentials, respectively. The second term, $\nu_\theta([\rho];\mathbf{r})$, is the Pauli potential derived from $T_\theta[\rho]$:

$$\nu_\theta([\rho];\mathbf{r}) = \frac{\delta T_\theta[\rho]}{\delta \rho(\mathbf{r})}. \tag{2.22}$$

Since the exact form of $\nu_\theta([\rho];\mathbf{r})$ in terms of density is unknown, various approximations are used in practical implementations of OF-DFT. Future work aims to develop improved kinetic energy functionals that accurately capture the effects of electronic correlations and shell structure in many-electron systems. The exact form of the Pauli potential $\nu_\theta([\rho];\mathbf{r})$ in terms of density $\rho(\mathbf{r})$ is unknown, but it can be expressed using KS orbitals and orbital energies as:

$$\nu_\theta([\rho];\mathbf{r}) = t_\theta(\mathbf{r}) + \sum_{i=1}^{N} (\varepsilon_N - \varepsilon_i) \frac{|\phi_i(\mathbf{r})|^2}{\rho(\mathbf{r})}, \tag{2.23}$$

where the Pauli kinetic energy (KE) density is given by:

$$t_\theta(\mathbf{r}) = t_s(\mathbf{r}) - t_{\nu W}(\mathbf{r}). \tag{2.24}$$

The total Pauli KE functional is:

$$T_\theta[\rho] = \int t_\theta(\mathbf{r}) \rho(\mathbf{r}) d\mathbf{r}. \tag{2.25}$$

An important constraint in orbital-free DFT [9]is the non-negativity of $T_\theta[\rho]$ and $\nu_\theta([\rho];\mathbf{r})$. The Pauli KE functional can also be computed from the density-potential relationship:

$$T_\theta[\rho] = -\frac{1}{2}\int \rho(\mathbf{r})\mathbf{r}\cdot\nabla\nu_\theta([\rho];\mathbf{r})d\mathbf{r}. \tag{2.26}$$

Since this formulation makes the KE functional non-unique and does not guarantee translational and rotational invariance, we focus on directly modeling $\nu_\theta([\rho];\mathbf{r})$ instead of $T_\theta[\rho]$.

Chapter 3

Results and Discussion

Here, we have numerically solved the 1-dimensional Schrödinger equation using Numerov and Finite- difference method . We calculated potentials,eigenvalue and eigenfunction for various potential systems and we solved solved Euler's equation for Non-Interacting system to generate automatic densities.

3.1 Euler's Equation code:

1. Code for Non-Interacting Systems

```
import numpy as np
import matplotlib.pyplot as plt
from itertools import islice
from scipy import integrate
import re
import sys

n = 4
file0 = open("density_initial.dat", "w")
file1 = open("initial_vext.dat", "w")
file2 = open("check.dat", "w")

with open('para_initial', 'r') as f:
    while True:
        next_n_lines = list(islice(f, 4))
        if not next_n_lines:
            print("here")
            break
        first_line = next_n_lines[0]
        number_line = first_line.rstrip('\n')
        del next_n_lines[0]
        lines_array = np.array(next_n_lines)
        a = []
        b = []
```

```
        c = []

        for line in lines_array:
            line = line.rstrip('\n')
            values = line.split()
            a.append(float(values[0]))
            b.append(float(values[1]))
            c.append(float(values[2]))

        a = np.asarray(a)
        b = np.asarray(b)
        c = np.asarray(c)
        a1, b1, c1 = a[0], b[0], c[0]
        a2, b2, c2 = a[1], b[1], c[1]
        a3, b3, c3 = a[2], b[2], c[2]

        def gaussian_potential(x, h, w, s):
            return h * np.exp(-0.5 * (((x - w) / s)
                ) ** 2)

        def double_well_potential(x):
            return -(gaussian_potential(x, a1, b1,
                c1) + gaussian_potential(x, a2, b2,
                c2) + gaussian_potential(x, a3, b3,
                c3))

        # Parameters
        x_min = -3.0
        x_max = 3.0
        h = 0.005  # Step size
        x = np.arange(x_min, x_max + h, h)
        n_points = len(x)

        # Construct Hamiltonian matrix
        V = double_well_potential(x)
        H = np.diag(1 / h**2 + V)
        H += np.diag(-0.5 / h**2 * np.ones(n_points
            - 1), k=1)
        H += np.diag(-0.5 / h**2 * np.ones(n_points
            - 1), k=-1)

        # Diagonalize Hamiltonian matrix
        eigenvalues, eigenvectors = np.linalg.eigh(
            H)

        # Find the ground state wavefunction
        ground_state_index = np.argmin(eigenvalues)
        ground_state_energy = eigenvalues[
            ground_state_index]
        ground_state_psi = eigenvectors[:,
            ground_state_index]

        # Find the first excited state wavefunction
        excited_state_index = np.argsort(
            eigenvalues)[1]  # Sort eigenvalues and
```

```
            choose the second smallest
70          excited_state_energy = eigenvalues[
               excited_state_index]
71          excited_state_psi = eigenvectors[:,
               excited_state_index]
72
73          # Ensure correct phase of the wavefunctions
74          reference_point = 0  # Choose a reference
               point
75          if ground_state_psi[np.abs(x -
               reference_point).argmin()] < 0:
76              ground_state_psi *= -1
77          if excited_state_psi[np.abs(x -
               reference_point).argmin()] < 0:
78              excited_state_psi *= -1
79
80          # Normalize the wavefunctions
81          ground_state_psi /= np.sqrt(integrate.
               trapezoid(ground_state_psi**2, x))
82          excited_state_psi /= np.sqrt(integrate.
               trapezoid(excited_state_psi**2, x))
83
84          ground_state_density = ground_state_psi**2
85          excited_state_density = excited_state_psi
               **2
86
87          integral1 = integrate.trapezoid(
               ground_state_density, x)
88          print(integral1)
89
90          integral2 = integrate.trapezoid(
               excited_state_density, x)
91          print(integral2)
92
93          totdens =  ground_state_density +
               excited_state_density
94          integral = integrate.trapezoid(totdens, x)
95          print(integral)
96
97          with np.printoptions(threshold=sys.maxsize,
                precision=4):
98              print(re.sub(r'␣*\n␣*', '\n', np.
                   array_str(np.c_[x, totdens,
                   ground_state_psi,
                   double_well_potential(x)]).replace('
                   [', '').replace(']', '').strip()),
                   file=file0)
99              print(file=file0)
100
101         with np.printoptions(threshold=sys.maxsize)
               :
102             print(re.sub(r'␣*\n␣*', '\n', np.
                   array_str(np.c_[x,
                   double_well_potential(x)]).replace('
                   [', '').replace(']', '').strip()),
```

```
                file=file1)
103             print(file=file1)
104
105         # Calculate kinetic energy functional
106         gradient_ground = np.gradient(
                ground_state_psi, x)
107         gradient_excited = np.gradient(
                excited_state_psi, x)
108
109         # Square of the gradients
110         gradient_ground_squared = gradient_ground
                **2
111         gradient_excited_squared = gradient_excited
                **2
112
113         # Sum the squared gradients for all states
114         kinetic_energy_integrand =  0.5*(
                gradient_ground_squared +
                gradient_excited_squared)
115
116         #T_exact = integrate.trapezoid(
                kinetic_energy_integrand, x)
117         #print("Exact Kinetic Energy (T_Exact):",
                T_exact)
118
119         # Calculate chemical potential (average
                energy per electron)
120         total_energy =  ground_state_energy +
                excited_state_energy
121         N = 2 # Example number of electrons
122         mu =  excited_state_energy   #total_energy
                / N
123
124         print("Chemical Potential (mu):", mu)
125
126         # Gradient descent parameters
127         eta = 0.005  # Step size
128         max_iterations = 500  # Maximum number of
                iterations
129         convergence_threshold = 1e-3  # Convergence
                criterion for density change
130
131         # Initialize the density (old density)
132         n_old = totdens  # This is the initial
                total density you already have
133
134         # Iterate until convergence
135         for iteration in range(max_iterations):
136             gradient_ground = np.gradient(
                    ground_state_psi, x)
137             gradient_excited = np.gradient(
                    excited_state_psi, x)
138
139             gradient_ground_squared =
                    gradient_ground**2
```

```
            gradient_excited_squared =
                gradient_excited**2

            kinetic_energy_integrand =0.5* (
                gradient_ground_squared +
                gradient_excited_squared)
            T_exact = integrate.trapezoid(
                kinetic_energy_integrand, x)

            potential_integral = integrate.
                trapezoid(n_old *
                double_well_potential(x), x)

            E_exact = T_exact + potential_integral

            L = E_exact - mu * (integrate.trapezoid
                (n_old, x) - N)

            gradient_T_exact = np.gradient(
                kinetic_energy_integrand, x)  #
                Numerical second derivative of
                density
            grad = double_well_potential(x) - mu

            grad_L = gradient_T_exact +
                double_well_potential(x) - mu

            n_new = n_old - eta * grad_L

            Energy_change = np.linalg.norm(E_exact
                - total_energy)
            if Energy_change <
                convergence_threshold:
                print(f"Converged after {iteration
                    + 1} iterations.")
                break

            # Update the old density for the next
                iteration
            n_old = n_new

        # Final density after gradient descent
        #print("Final density:", n_new)
        with np.printoptions(threshold=sys.maxsize)
            :
            print(re.sub(r' *\n *', '\n', np.
                array_str(np.c_[x, n_new,
                gradient_T_exact, grad]).replace('[
                ', '').replace(']', '').strip()),
                file=file2)
            print(file=file2)

        print(E_exact)
        print(total_energy)

```

```
175
176
177         # Optionally, plot the final density
178         plt.plot(x, n_new, label="Final Density")
179         plt.plot(x, totdens, label="initial Density
                ")
180         plt.xlabel('x')
181         plt.ylabel('Density')
182         plt.title('Final Density after Gradient
                Descent')
183         plt.legend()
184         plt.grid(True)
185         plt.show()
186
187         print("Ground state energy:",
                ground_state_energy)
188         print("First excited state energy:",
                excited_state_energy)
```

3.2 Euler's Equation code:

2. Code for Interacting Atomic Systems:

3.2.1 Helium Atom

```
1  import numpy as np
2  import matplotlib.pyplot as pl
3  from scipy.integrate import simpson
4  from scipy.integrate import trapezoid
5  from scipy.integrate import quad
6  from scipy import optimize
7  import time
8  from scipy.special import factorial
9
10 #def differentiate(f, x):
11
12 #assuming r = x*x
13 #R = u*phi, phi = x^(-3/2)
14 #in the remaining x is expressed by r
15 t1 = time.time()
16 #values of psi at nth and n-1th point
17 psi_n = 0.0
18 psi_n1 = 1e-8
19 #number of mesh points
20 n = 2500
21 lamb = 1.0 #0.2
22 Z = 2 #18.0
23 max_iter = 8000
24 mixing = 0.05
25 root = -500.0
```

```
26 #the mesh
27 r_c = 11.999
28 r = np.linspace(.01, np.sqrt(r_c), n)
29 d = r[1]-r[0]
30 #limit of integration
31
32 #some constants
33 Ck = (3.0*np.power(3.0*3.14159*3.14159, 2.0/3.0)
      /10.0)
34 Cx = (3.0*np.power(3.0/3.14159, 1.0/3.0)/4.0)
35
36 #load data file for v_q
37 file_str = "He.dat"
38 dat = np.loadtxt(file_str)
39 ##create v_q by interpolation
40 #v_q = np.interp(r*r, dat[:,0], dat[:,3])
41
42 #func for numerove step
43 def numerove(E, r, n, d, v_r):
44     u = np.zeros(n)
45     u[n-1] = psi_n
46     u[n-2] = psi_n1
47     f_r = -(E-v_r)*(8.0*r*r)/lamb
48     #print("right integration")
49     for i in range(2,n): #right integration
50         l = n-i
51         u[l-1] = (u[l+1]*(12.0- d*d*f_r[l+1]) -
              2.0*u[l]*(12.0 + 5.0*d*d*f_r[l]))/(d*d*
              f_r[l-1]-12.0)
52     #normalize
53     N = simpson(u*u*r*r, r)
54     #print(N)
55     u = u/np.sqrt(8*3.14159*N/Z)
56     return u[0]
57
58 #func for evaluating ground stae wavefunction with
      optimized energy
59 def wavefunction(E, r, n, d, v_r):
60     u = np.zeros(n)
61     u[n-1] = psi_n
62     u[n-2] = psi_n1
63     f_r = -(E-v_r)*(8.0*r*r)/lamb
64     #print("right integration")
65     for i in range(2,n): #right integration
66         l = n-i
67         u[l-1] = (u[l+1]*(12.0- d*d*f_r[l+1]) -
              2.0*u[l]*(12.0 + 5.0*d*d*f_r[l]))/(d*d*
              f_r[l-1]-12.0)
68     #normalize
69     N = simpson(u*u*r*r, r)
70     u = u/np.sqrt(8*3.14159*N/Z)
71     return u #only returns the u part of R
72
73 #actual self-consistent code execution stars here
74
```

```
#initial guess density
rho = np.zeros(n)
n_i = np.array([1,1,2,2])
a_i = np.array([1,2,1,2])

N_i = (1.0/np.sqrt(4.0*3.14159*factorial(2*n_i))) *
    np.power(2.0*a_i, n_i+0.5)
X_i = np.zeros((n, 4))

for i in range(0,4):
    X_i[:,i] = N_i[i] * np.power(r*r, n_i[i] - 1.0)
        * np.exp(-a_i[i] * r*r)
phi_1s = X_i[:,0] + X_i[:,1]
phi_2s = X_i[:,2] + X_i[:,3]
phi_2p = phi_2s

rho = 2*np.power(phi_1s, 2.0) #+ 1*np.power(phi_2s,
    2.0) #+ 3*np.power(phi_2p, 2.0)

#rho =4*r*r*np.exp(-r*r*r*r)
#pl.plot(r*r, 4*3.14*r*r*r*r*rho)
#pl.show()

M = 8*3.14159*simpson(rho*np.power(r,5), r)
print(M)
rho = rho/(M/Z)

v_h = np.zeros(n)
E_T= 0.0

for i in range(max_iter):
    #calculate potentials
    #calculate Vh
    for j in range(0,n):
        v_h[j] = 8.0*3.14159*(simpson(rho[j:n]*np.
           power(r[j:n], 3.0), r[j:n])
        - simpson(rho[j:n]*np.power(r[j:n], 5.0), r
           [j:n])/(r[j]*r[j])) + Z/(r[j]*r[j])

    v_x = -Cx*(4.0/3.0)*np.power(rho, 1.0/3.0)
    v_ext = -Z/np.power(r, 2.0)

    v_KS = v_x +  v_ext + v_h# + v_c
    #v_KS =  v_ext + v_h# + v_c
    v_r = v_KS +3.0*lamb/(32.0*r*r*r*r)

    #shooting method for rough guess; E_i should be
        suffieciently low, E_f sufficeintly high,
       N_E sufficiently large.
    E_i = root-1.5
    E_f = 1.0
    N_E = 1000
    E = np.linspace(E_i, E_f, N_E)
    u0 = numerove(E_i, r, n, d, v_r)
    flag = 0
```

```
123     for e in E[1:]:
124         temp = numerove(e, r, n, d, v_r)
125         if(temp*u0 < 0.0):
126             E_f = e
127             flag = 1
128             break
129         else:
130             u0 = temp
131             E_i = e
132 
133     #brent method for refining energy
134     if flag==1 :
135         root = optimize.brentq(numerove, E_i,E_f,
                args=(r, n, d, v_r))
136         u_new = wavefunction(root, r, n, d, v_r)
137 
138         rho_new = u_new*u_new*np.power(r, -3.0) #
                rho_new = R*R = x^-3 * u*u
139         M = 8*3.14159*simpson(rho_new*np.power(r,5)
                , r)
140 
141         rho_new = rho_new/(M/Z)
142         err = np.sum(np.power(rho_new - rho, 2.0))
143         rho = (1-mixing)*rho + mixing*rho_new
144 
145         #calculate the potential energy
146 
147         #E_T_new = 8.0*3.14159*simpson(rho*(v_ext +
                0.5*v_h + v_x*3.0/4.0+v_q*3.0/5.0)*np.
                power(r,5.0), r)
148         #calculate the kinetic energy
149         #grad = np.gradient(np.sqrt(rho), r,
                edge_order=2)
150         #grad2 = np.gradient(grad, d)
151         #ke = lamb*3.14159*simpson(np.power(grad
                ,2.0)*np.power(r,3.0), r)
152         #print(ke)
153         #E_T_new += ke
154         #E_T_new =E_T_new - 3.14159*simpson(r*r*r
                *(3.0*grad/r+grad2), r)
155         #E_T_new = E_T_new + 3.14159*simpson(r*r*r
                *(grad*grad), r)
156         E_T_new = root
157         deltaE = (E_T_new - E_T)
158         E_T = E_T_new
159         print("Iteration␣=␣{:d},␣energy␣=␣{:f},␣u
                (0)␣=␣{:e},␣error␣=␣{:e},␣root␣=␣{:f}".
160                 format(i, E_T, u_new[0], err, root))#
                    deltaE, root))
161 
162         if(err < 8.0e-0 or np.abs(deltaE) < 1e-6):
163             break
164     else:
165         print("Given␣initial␣range␣and␣mesh␣of␣
                energy␣is␣poor.␣Try␣lowering␣E_i␣or␣
```

```
            increasing␣N_E␣or␣E_f.")
166
167 pl.plot(r*r, 4*3.14*r*r*r*r*rho)
168 pl.plot(dat[:,0], dat[:,2])
169 pl.xlabel("r")
170 pl.ylabel("density")
171 #pl.savefig(file_str.split(".")[0]+".png")
172 pl.show()
173 print("time=",time.time()-t1, "␣s")
174
175 dens = np.interp(r * r, dat[:, 0], dat[:, 2])
176
177 output_data = np.column_stack((r * r, dens, 4 *
       3.14 * r * r * r * r * rho ))
178 output_file = "He_density.txt"
179 np.savetxt(output_file, output_data, comments='')
180
181 #output_data = np.column_stack((r * r, v_h, v_x,
       v_ext,v_r))
182 #output_file =  "He_Veff.txt"
183 #np.savetxt(output_file, output_data, comments='')
184
185
186 for j in range(0,n):
187         v_h[j] = 8.0*3.14159*(simpson(rho[j:n]*np.
               power(r[j:n], 3.0), r[j:n])
188         - simpson(rho[j:n]*np.power(r[j:n], 5.0), r
               [j:n])/(r[j]*r[j])) + Z/(r[j]*r[j])
189 #v_x = -Cx*(4.0/3.0)*np.power(rho, 1.0/3.0)
190 v_ext = -Z/np.power(r, 2.0)
191 E_ext = simpson(8.0*3.14159*rho*v_ext*np.power(r
       ,5.0), r)
192 print(E_ext)
193 E_H = simpson(4.0*3.14159*rho*(v_h)*np.power(r,5.0)
       , r)
194 print(E_H)
195 E_X = simpson(8.0*3.14159*rho*v_x*np.power(r,5.0)
       *3.0/4.0, r)
196 print(E_X)
197
198 E_T1 = (E_ext + E_H +E_X)/2.0
199 print(E_T1)
200 #print("M =", 8*3.14159*simpson(rho*np.power(r,5),
       r))
201 #data = np.zeros((n,3))
202 #data[:,0] = r*r
203 #data[:,1] = rho
204 #data[:,2] = np.sqrt(rho)
205 #np.savetxt(file_str.split(".")[0]+".csv", data,
       delimiter=",")
206
207 exit()
```

3.2.2 Lithium, Beryllium and Neon Atom

```
import numpy as np
import matplotlib.pyplot as pl
from scipy.integrate import simpson
from scipy.integrate import trapezoid
from scipy.integrate import quad
from scipy import optimize
import time
from scipy.special import factorial

#def differentiate(f, x):

#assuming r = x*x
#R = u*phi, phi = x^(-3/2)
#in the remaining x is expressed by r
t1 = time.time()
#values of psi at nth and n-1th point
psi_n = 0.0
psi_n1 = 1e-8
#number of mesh points
n = 2000
lamb = 1.0 #0.2
Z = 3 #4 #10
max_iter = 8000
mixing = 0.05
root = -500.0
#the mesh
r_c = 11.999
r = np.linspace(.01, np.sqrt(r_c), n)
d = r[1]-r[0]
#limit of integration

#some constants
Ck = (3.0*np.power(3.0*3.14159*3.14159, 2.0/3.0)
   /10.0)
Cx = (3.0*np.power(3.0/3.14159, 1.0/3.0)/4.0)

#load data file for v_q
file_str = "Li.dat"#"Ne.dat"#"Be.dat"
dat = np.loadtxt(file_str)
#create v_q by interpolation
v_q = np.interp(r*r, dat[:,0], dat[:,3])

#func for numerove step
def numerove(E, r, n, d, v_r):
    u = np.zeros(n)
    u[n-1] = psi_n
    u[n-2] = psi_n1
    f_r = -(E-v_r)*(8.0*r*r)/lamb
    #print("right integration")
    for i in range(2,n): #right integration
        l = n-i
        u[l-1] = (u[l+1]*(12.0- d*d*f_r[l+1]) -
           2.0*u[l]*(12.0 + 5.0*d*d*f_r[l]))/(d*d*
```

```
            f_r[l-1]-12.0)
    #normalize
    N = simpson(u*u*r*r, r)
    #print(N)
    u = u/np.sqrt(8*3.14159*N/Z)
    return u[0]

#func for evaluating ground stae wavefunction with optimized energy
def wavefunction(E, r, n, d, v_r):
    u = np.zeros(n)
    u[n-1] = psi_n
    u[n-2] = psi_n1
    f_r = -(E-v_r)*(8.0*r*r)/lamb
    #print("right integration")
    for i in range(2,n): #right integration
        l = n-i
        u[l-1] = (u[l+1]*(12.0- d*d*f_r[l+1]) -
           2.0*u[l]*(12.0 + 5.0*d*d*f_r[l]))/(d*d*
           f_r[l-1]-12.0)
    #normalize
    N = simpson(u*u*r*r, r)
    u = u/np.sqrt(8*3.14159*N/Z)
    return u #only returns the u part of R

rho = np.zeros(n)
n_i = np.array([1,1,2,2])
a_i = np.array([1,2,1,2])

N_i = (1.0/np.sqrt(4.0*3.14159*factorial(2*n_i))) *
    np.power(2.0*a_i, n_i+0.5)
X_i = np.zeros((n, 4))

for i in range(0,4):
    X_i[:,i] = N_i[i] * np.power(r*r, n_i[i] - 1.0)
        * np.exp(-a_i[i] * r*r)
phi_1s = X_i[:,0] + X_i[:,1]
phi_2s = X_i[:,2] + X_i[:,3]
phi_2p = phi_2s

#rho = 2*np.power(phi_1s, 2.0) + 1*np.power(phi_2s,
    2.0) #+ 3*np.power(phi_2p, 2.0)

#rho = r*r*np.exp(-0.5*r*r)
#rho = r*r*np.exp(-r*r)
#rho = r*np.exp(-r*r)
rho = 2*r*np.exp(-r**4)

#rho = r*np.exp(-r*r)
#pl.plot(r*r, 4*3.14*r*r*r*r*rho)
#pl.show()

M = 8*3.14159*simpson(rho*np.power(r,5), r)
print(M)
rho = rho/(M/Z)
```

```
100
101 v_h = np.zeros(n)
102 E_T= 0.0
103
104 for i in range(max_iter):
105     #calculate potentials
106     #calculate Vh
107     for j in range(0,n):
108         v_h[j] = 8.0*3.14159*(simpson(rho[j:n]*np.
                power(r[j:n], 3.0), r[j:n])
109         - simpson(rho[j:n]*np.power(r[j:n], 5.0), r
                [j:n])/(r[j]*r[j])) + Z/(r[j]*r[j])
110         #v_h[j] = 8*3.14159*(simpson(rho[j:]*np.
                power(r[j:], 3.0), r[j:])
111         #+ simpson(rho[0:j+1]*np.power(r[0:j+1],
                5.0), r[0:j+1])/(r[j]*r[j]))
112
113
114     #v_q = Ck*(5.0/3.0)*np.power(rho, 2.0/3.0)
115     v_x = -Cx*(4.0/3.0)*np.power(rho, 1.0/3.0)
116     v_ext = -Z/np.power(r, 2.0)
117     #rs_r = np.power(3.0/(4.0*3.14159*rho),
            1.0/3.0)
118     #v_c = -0.0666*np.log(1+11.4/rs_r)
119     #a = 9.81
120     #b = 21.437
121     #c = 28.582667
122     #v_c = - (a+c*np.power(rho, -1.0/3.0))/np.power
            (a+b*np.power(rho, -1.0/3.0), 2.0)
123     v_KS = v_x +  v_ext + v_h# + v_c
124     v_r = v_q + v_KS +3.0*lamb/(32.0*r*r*r*r)
125
126     #shooting method for rough guess; E_i should be
            suffieciently low, E_f sufficeintly high,
            N_E sufficiently large.
127     E_i = root-1.5
128     E_f = 1.0
129     N_E = 1000
130     E = np.linspace(E_i, E_f, N_E)
131     u0 = numerove(E_i, r, n, d, v_r)
132     flag = 0
133     for e in E[1:]:
134         temp = numerove(e, r, n, d, v_r)
135         if(temp*u0 < 0.0):
136             E_f = e
137             flag = 1
138             break
139         else:
140             u0 = temp
141             E_i = e
142
143     #brent method for refining energy
144     if flag==1 :
145         root = optimize.brentq(numerove, E_i,E_f,
                args=(r, n, d, v_r))
```

```
146         u_new = wavefunction(root, r, n, d, v_r)
147 
148         rho_new = u_new*u_new*np.power(r, -3.0) #
               rho_new = R*R = x^-3 * u*u
149         M = 8*3.14159*simpson(rho_new*np.power(r,5)
               , r)
150 
151         rho_new = rho_new/(M/Z)
152         err = np.sum(np.power(rho_new - rho, 2.0))
153         rho = (1-mixing)*rho + mixing*rho_new
154 
155         #calculate the potential energy
156 
157         #E_T_new = 8.0*3.14159*simpson(rho*(v_ext +
                0.5*v_h + v_x*3.0/4.0+v_q*3.0/5.0)*np.
               power(r,5.0), r)
158         #calculate the kinetic energy
159         #grad = np.gradient(np.sqrt(rho), r,
               edge_order=2)
160         #grad2 = np.gradient(grad, d)
161         #ke = lamb*3.14159*simpson(np.power(grad
               ,2.0)*np.power(r,3.0), r)
162         #print(ke)
163         #E_T_new += ke
164         #E_T_new =E_T_new - 3.14159*simpson(r*r*r
               *(3.0*grad/r+grad2), r)
165         #E_T_new = E_T_new + 3.14159*simpson(r*r*r
               *(grad*grad), r)
166         E_T_new = root
167         deltaE = (E_T_new - E_T)
168         E_T = E_T_new
169         print("Iteration = {:d}, energy = {:f}, u
               (0) = {:e}, error = {:e}, root = {:f}".
170               format(i, E_T, u_new[0], err, root))#
                  deltaE, root))
171 
172         if(err < 8.0e-0 or np.abs(deltaE) < 1e-6):
173             break
174     else:
175         print("Given initial range and mesh of
               energy is poor. Try lowering E_i or
               increasing N_E or E_f.")
176 pl.plot(r*r, 4*3.14*r*r*r*r*rho)
177 pl.plot(dat[:,0], dat[:,2])
178 pl.xlabel("r")
179 pl.ylabel("density")
180 pl.savefig(file_str.split(".")[0]+".png")
181 pl.show()
182 print("time=",time.time()-t1, " s")
183 
184 dens_inter = np.interp(r * r, dat[:, 0], dat[:, 2])
185 
186 output_data = np.column_stack((r * r, dens_inter, 4
    * 3.14 * r * r * r * r * rho ))
```

```
187 output_file = file_str.split(".")[0] + "_density.
    txt"
188 np.savetxt(output_file, output_data, comments='')
189
190 output_data = np.column_stack((r * r, v_h, v_x,
    v_ext, v_q, v_r))
191 output_file = file_str.split(".")[0] + "_Veff.txt"
192 np.savetxt(output_file, output_data, comments='')
193
194
195 for j in range(0,n):
196         v_h[j] = 8.0*3.14159*(simpson(rho[j:n]*np.
               power(r[j:n], 3.0), r[j:n])
197         - simpson(rho[j:n]*np.power(r[j:n], 5.0), r
               [j:n])/(r[j]*r[j])) + Z/(r[j]*r[j])
198 v_x = -Cx*(4.0/3.0)*np.power(rho, 1.0/3.0)
199 v_ext = -Z/np.power(r, 2.0)
200 E_ext = simpson(8.0*3.14159*rho*v_ext*np.power(r
    ,5.0), r)
201 print(E_ext)
202 E_H = simpson(4.0*3.14159*rho*(v_h)*np.power(r,5.0)
    , r)
203 print(E_H)
204 E_X = simpson(8.0*3.14159*rho*v_x*np.power(r,5.0)
    *3.0/4.0, r)
205 print(E_X)
206 E_T1 = (E_ext + E_H + E_X)/2.0
207 print(E_T1)
208 #print("M =", 8*3.14159*simpson(rho*np.power(r,5),
    r))
209 data = np.zeros((n,3))
210 data[:,0] = r*r
211 data[:,1] = rho
212 data[:,2] = np.sqrt(rho)
213 np.savetxt(file_str.split(".")[0]+".csv", data,
    delimiter=",")
214
215
216 exit()
```

3.3 Results and Outputs

3.3.1 Non-Interacting Model Systems

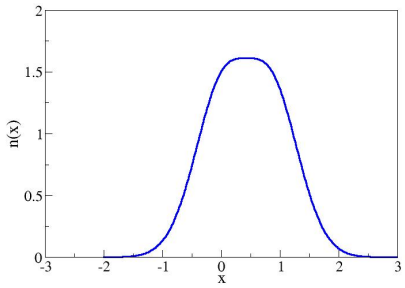


Figure 3.1: Initial density for 2S

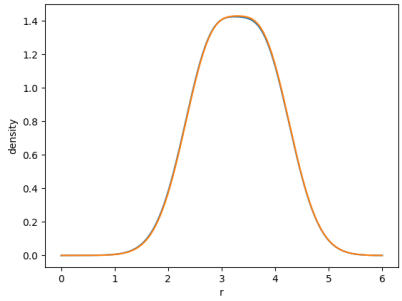


Figure 3.2: Optimized density for 2S

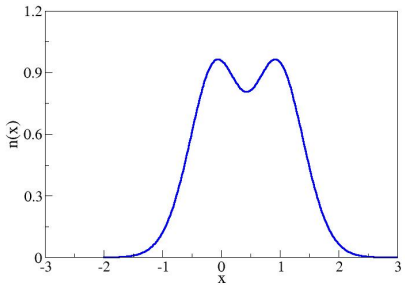


Figure 3.3: Initial density for 3S

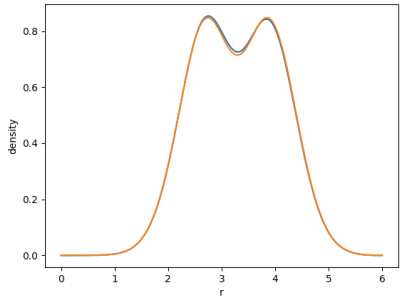


Figure 3.4: Optimized density for 3S

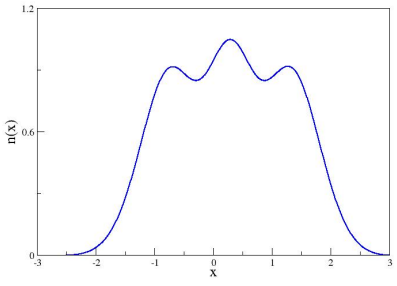


Figure 3.5: Initial density for 4S

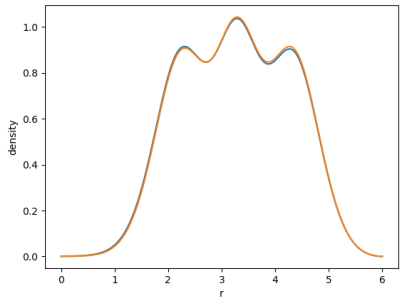


Figure 3.6: Optimized density for 4S

3.3.2 Interacting Atomic System

3.3.2.1 Helium Atom

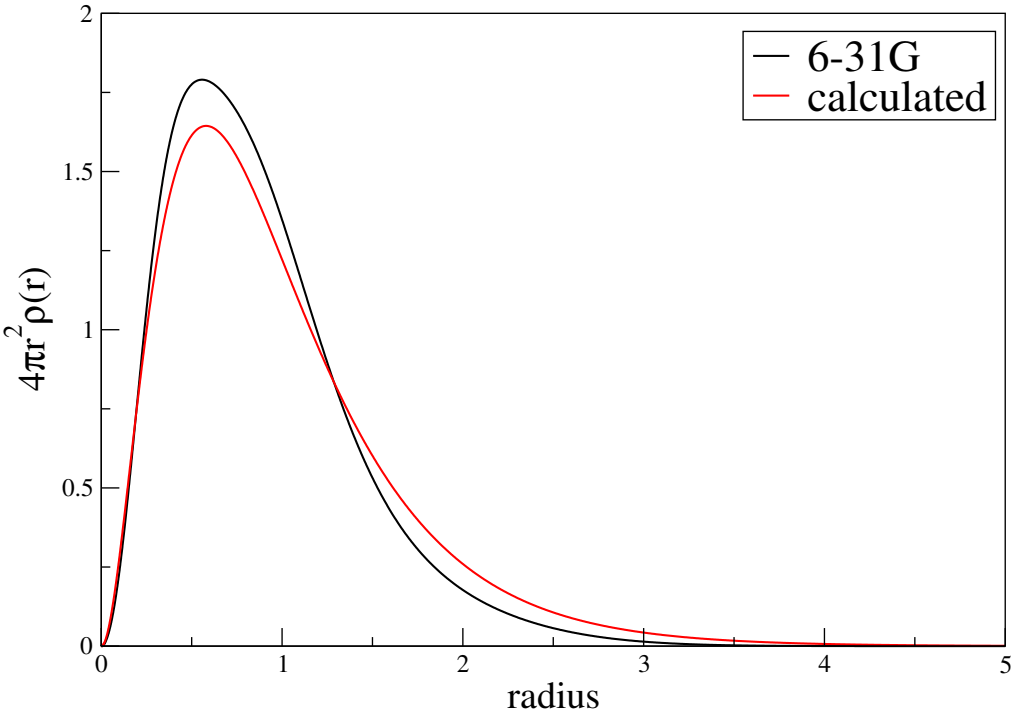


Figure 3.7: Radial density for Helium atom

Property	**Predicted (6-31G)**	**Calculated**
Energy (E_n)	-2.714	-2.681
μ (a.u.)	-0.517	-0.526

Table 3.1: Comparison of Reference and Calculated Energy and chemical potential (μ)

3.3.2.2 Lithium Atom

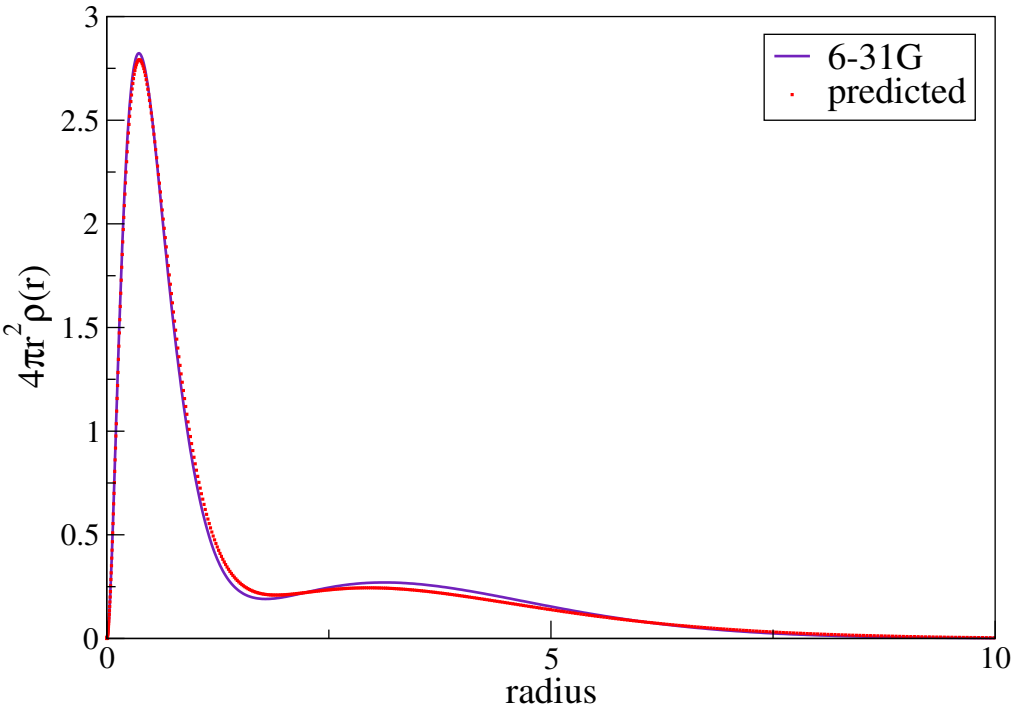


Figure 3.8: Radial density for Lithium atom

Property	**Predicted (6-31G)**	**Calculated**
Energy (E_n)	-7.16	-7.18
μ (a.u.)	-0.374	-0.432

Table 3.2: Comparison of Reference and Calculated Energy and chemical potential (μ)

3.3.2.3 Beryllium Atom

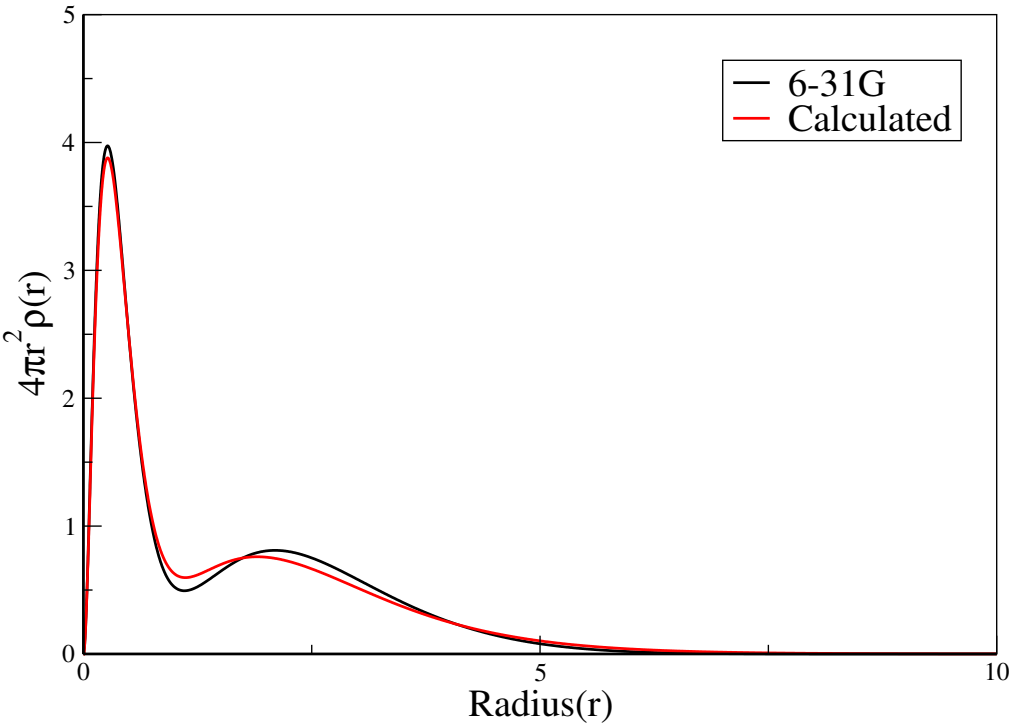


Figure 3.9: Radial density for Beryllium atom

Property	**Predicted (6-31G)**	**Calculated**
Energy (E_n)	-0.168	-0.170
μ (a.u.)	-14.182	-14.214

Table 3.3: Comparison of Reference and Calculated Energy and chemical potential (μ)

3.3.2.4 Neon Atom

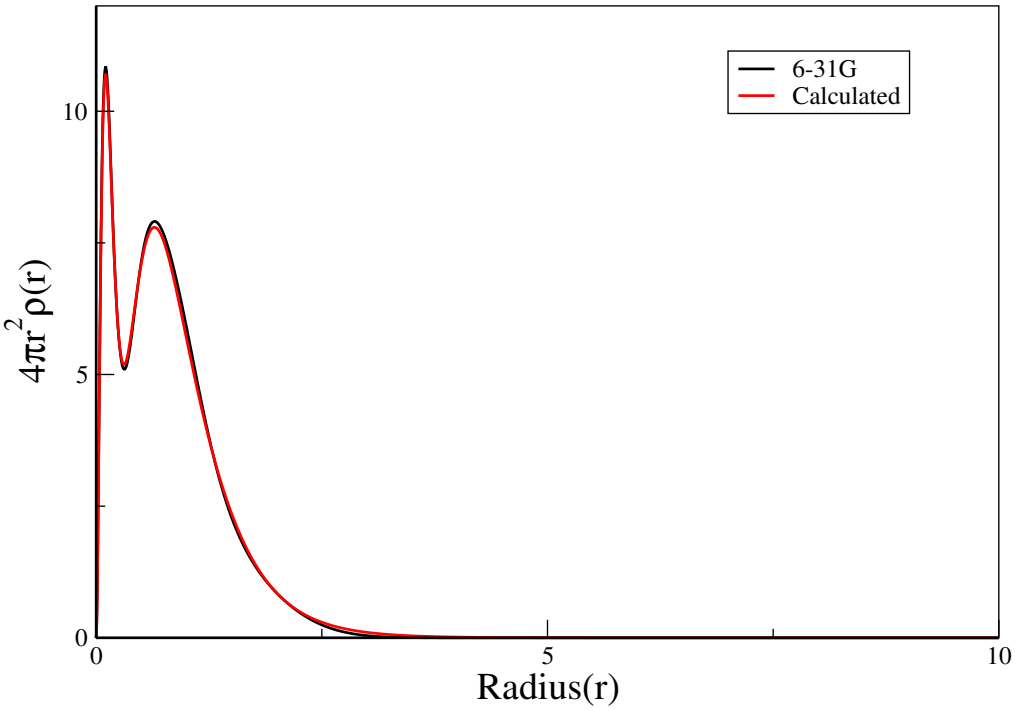


Figure 3.10: Radial density for Neon atom

Property	**Predicted (6-31G)**	**Calculated**
Energy (E_n)	-127.394	-126.206
μ (a.u.)	-0.444	-0.437

Table 3.4: Comparison of Reference and Calculated Energy and chemical potential (μ)

Chapter 4

Conclusion and Scope

In this project, we used a method to calculate the radial electron density of atoms by solving the Euler–Lagrange equation derived from the variational principle within framework of density functional theory (DFT). By using the exact Pauli potential extracted from Kohn–Sham DFT calculations, we accurately included the kinetic energy contribution without explicitly solving for orbitals. The Euler equation, reformulated in terms of the square root of the density, was solved numerically using the Numerov method, ensuring stability and precision.

The resulting radial densities closely matched those obtained from full Kohn–Sham solutions gaussian software which validating the approach. This demonstrates that incorporating the exact Pauli potential into orbital-free frameworks can reproduce high-quality densities with significantly reduced computational effort. The success of this method highlights its potential for extending orbital-free DFT techniques to larger systems, where traditional Kohn–Sham methods become computationally expensive.

Future scope will focus on further enhancing orbital-free density functional theory (OF-DFT) by refining the approximate forms of the Pauli potential $\nu_\theta([\rho]; \mathbf{r})$, which demonstrated improved accuracy and numerical stability in this work. These refinements will be coupled with machine learning techniques and line-integration methods to construct functionals that are invariant under translations and rotations, ensuring better adaptability across various atomic and molecular systems. By improving these approximations, the proposed approach will continue to offer a computationally efficient alternative to traditional Kohn–Sham DFT, making it suitable for large-scale electronic structure calculations in materials science and quantum chemistry.

.

Appendix A

Appendix Section

A.0.1 Magnesium Atom

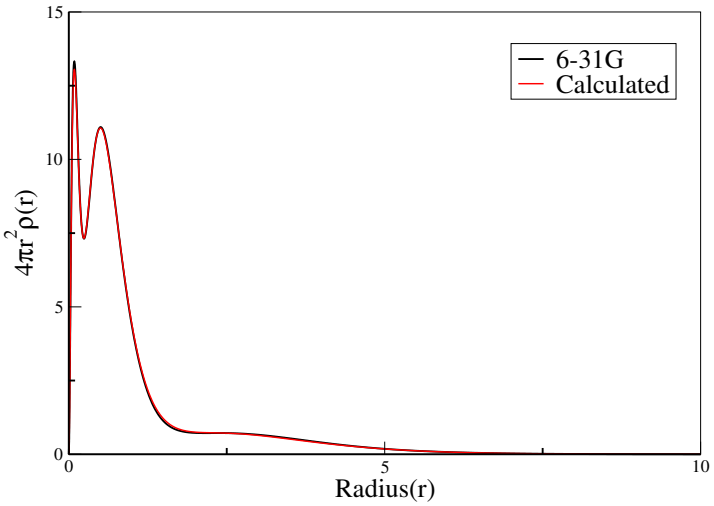


Figure A.1: Radial Density for Magnesium Atom

Property	**Predicted (6-31G)**	**Calculated**
Energy (E_n)	-195.601	-194.501
μ (a.u.)	-0.140	-0.1

Table A.1: Comparison of Reference and Calculated Energy and chemical potential (μ)

A.0.2 Nitrogen Atom

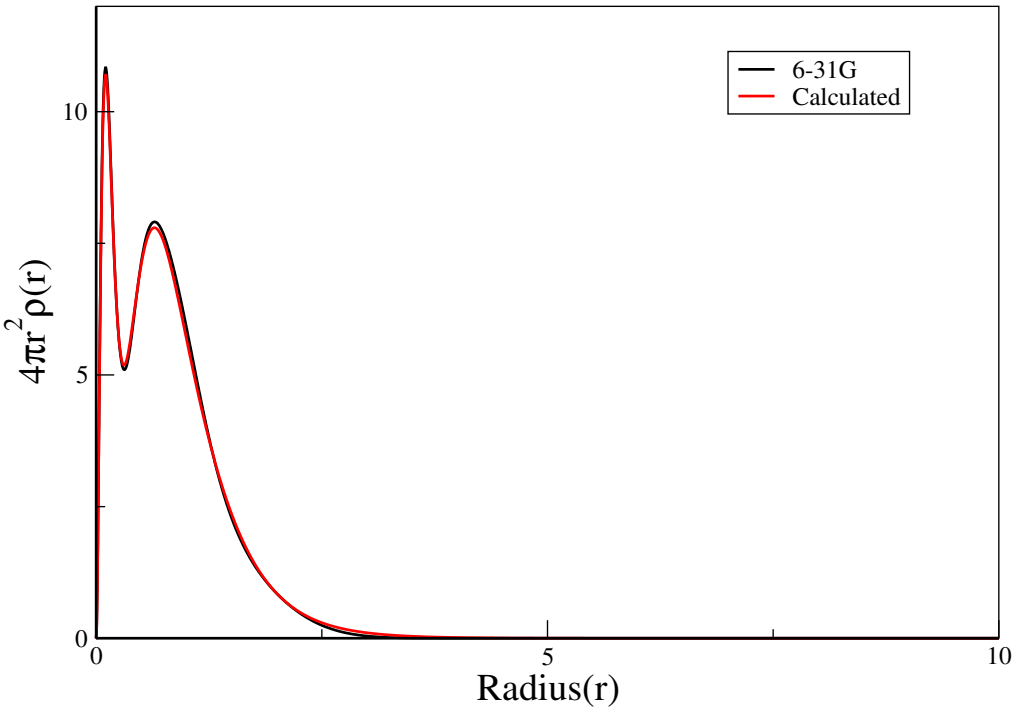


Figure A.2: Radial density for Nitrogen atom

Property	**Predicted (6-31G)**	**Calculated**
Energy (E_n)	-0.221	-0.226
μ (a.u.)	-53.680	-53.391

Table A.2: Comparison of Reference and Calculated Energy and chemical potential (μ)

A.0.3 Argon Atom

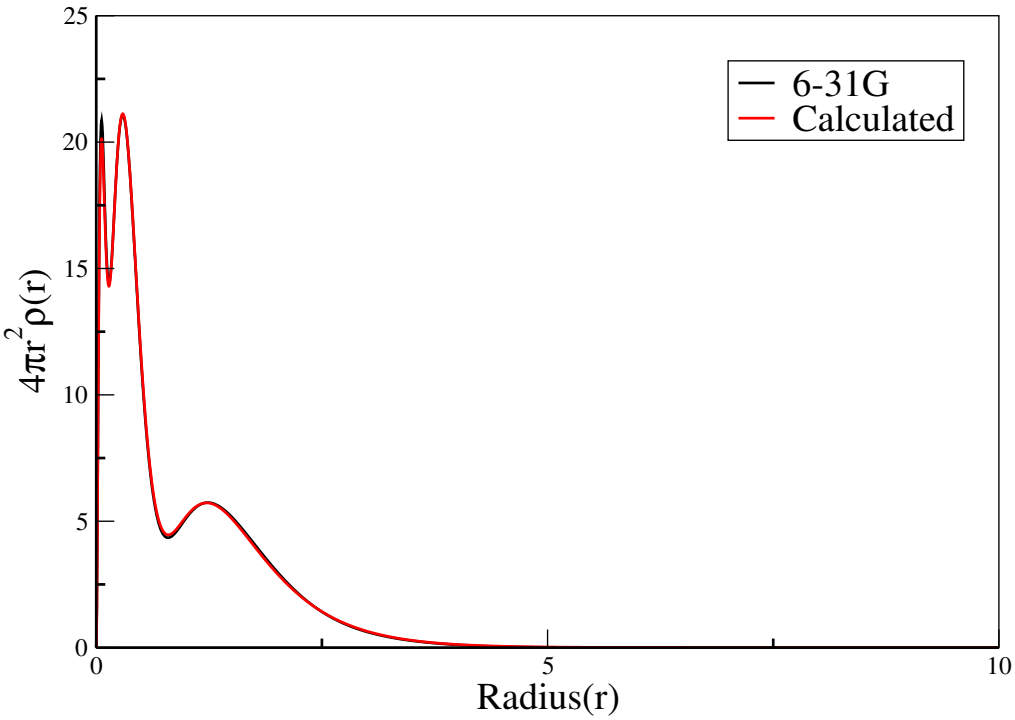


Figure A.3: Radial density for Argon atom

Property	**Predicted (6-31G)**	**Calculated**
Energy (E_n)	-524.452	-512.73
μ (a.u.)	-0.334	-0.333

Table A.3: Comparison of Reference and Calculated Energy and chemical potential (μ)

Bibliography

[1] Aparna Gangwar, Satya S Bulusu, and Arup Banerjee. Feed-forward neural networks for fitting of kinetic energy and its functional derivative. *Chemical Physics Letters*, 801:139718, 2022.

[2] Aparna Gangwar, Satya S Bulusu, and Arup Banerjee. Neural network learned pauli potential for the advancement of orbital-free density functional theory. *The Journal of Chemical Physics*, 159(12), 2023.

[3] P Hohenberg and WJPR Kohn. Density functional theory (dft). *Phys. Rev*, 136(1964):B864, 1964.

[4] BM Deb and SK Ghosh. New method for the direct calculation of electron density in many-electron systems. i. application to closed-shell atoms. *International Journal of Quantum Chemistry*, 23(1):1–26, 1983.

[5] Aparna Gangwar, Satya S Bulusu, Amit Kumar Das, and Arup Banerjee. Self-consistent electron density with shell structure using neural network-based pauli potential. *The Journal of Chemical Physics*, 162(3), 2025.

[6] Timo Graen and Helmut Grubmüller. Nusol—numerical solver for the 3d stationary nuclear schrödinger equation. *Computer Physics Communications*, 198:169–178, 2016.

[7] Robert J Hinde. Quantum chemistry, (by ira n. levine). *Journal of Chemical Education*, 77(12):1564, 2000.

[8] SDG Martinz and RV Ramos. Numerical solution of the 1d-schr\" odinger equation with pseudo-delta barrier using numerov method. *arXiv preprint arXiv:1507.03708*, 2015.

[9] Libero J Bartolotti and Ken Flurchick. An introduction to density functional theory. *Reviews in computational chemistry*, pages 187–216, 1996.