

Advanced Computational Quantum Mechanics: Mathematical Foundations and Numerical Implementation

A Comprehensive Technical Reference

Contents

| | | |
|----------|--|-----------|
| 1 | Quantum Mechanics and Computational Quantum Mechanics | 3 |
| 1.1 | Mathematical Foundations | 3 |
| 1.1.1 | Hilbert Space and State Vectors | 3 |
| 1.1.2 | Fundamental Operators | 3 |
| 1.1.3 | Time-Independent Schrödinger Equation | 4 |
| 1.2 | Analytical Solutions: Hydrogen Atom | 4 |
| 1.3 | The Many-Body Problem | 5 |
| 1.4 | Born-Oppenheimer Approximation | 6 |
| 1.5 | Density Functional Theory | 6 |
| 1.5.1 | Hohenberg-Kohn Theorems | 6 |
| 1.5.2 | Kohn-Sham Equations | 7 |
| 2 | Programming and Algorithmic Analysis | 8 |
| 2.1 | Computational Complexity | 8 |
| 2.2 | Python Scientific Computing Stack | 8 |
| 2.2.1 | NumPy — Numerical Linear Algebra | 8 |
| 2.2.2 | SciPy — Advanced Algorithms | 8 |
| 2.2.3 | PySCF — Production Quantum Chemistry | 9 |
| 2.3 | Parallel Computing | 9 |
| 2.3.1 | Shared Memory Parallelism (OpenMP) | 9 |
| 2.3.2 | Distributed Memory (MPI) | 9 |
| 2.3.3 | GPU Acceleration | 9 |
| 3 | Computational Quantum Dynamics | 11 |
| 3.1 | Time-Dependent Schrödinger Equation | 11 |
| 3.2 | Split-Operator Method | 11 |
| 3.3 | Runge-Kutta Methods | 12 |
| 3.3.1 | Fourth-Order Runge-Kutta (RK4) | 12 |
| 3.4 | Chebyshev Polynomial Expansion | 12 |
| 4 | Implementation: Detailed Code Examples | 14 |
| 4.1 | Time-Independent Schrödinger Equation Solver | 14 |
| 4.1.1 | Problem: 1D Quantum Harmonic Oscillator | 14 |
| 4.1.2 | Finite Difference Discretization | 14 |
| 4.1.3 | Complete Python Implementation | 14 |
| 4.1.4 | Convergence Analysis and Error Scaling | 18 |
| 4.2 | Time-Dependent Schrödinger Equation: Split-Operator Method | 18 |
| 4.2.1 | Problem: Gaussian Wave Packet in Harmonic Potential | 18 |

| | | |
|----------|---|-----------|
| 4.2.2 | Split-Operator Algorithm Implementation | 18 |
| 4.2.3 | Numerical Stability and Error Analysis | 21 |
| 4.3 | Hartree-Fock Self-Consistent Field Method | 21 |
| 4.3.1 | Theoretical Foundation | 21 |
| 4.3.2 | Implementation: Helium Atom | 21 |
| 4.3.3 | Analysis of HF Approximation | 24 |
| 5 | Advanced Topics and Extensions | 26 |
| 5.1 | Basis Set Selection and Convergence | 26 |
| 5.2 | Post-Hartree-Fock Methods | 26 |
| 5.3 | Density Functional Theory in Practice | 27 |
| 5.4 | Future Directions | 27 |
| 5.5 | Method Comparison Summary | 28 |
| A | Useful Constants and Conversion Factors | 29 |
| B | Numerical Methods Reference | 30 |

Chapter 1

Quantum Mechanics and Computational Quantum Mechanics

1.1 Mathematical Foundations

1.1.1 Hilbert Space and State Vectors

The state of a quantum system is represented by a vector $|\psi\rangle$ in a complex Hilbert space \mathcal{H} . For a single particle in three dimensions, the wavefunction $\psi(\mathbf{r}, t) \in L^2(\mathbb{R}^3)$ must belong to the space of square-integrable functions, satisfying the normalization condition:

$$\int_{-\infty}^{\infty} \int_{-\infty}^{\infty} \int_{-\infty}^{\infty} |\psi(\mathbf{r}, t)|^2 d^3\mathbf{r} = 1 \quad (1.1)$$

The inner product in this space is defined as:

$$\langle \phi | \psi \rangle = \int \phi^*(\mathbf{r}, t) \psi(\mathbf{r}, t) d^3\mathbf{r} \quad (1.2)$$

where ϕ^* denotes the complex conjugate. This forms a complete inner product space with norm $\|\psi\| = \sqrt{\langle \psi | \psi \rangle}$.

1.1.2 Fundamental Operators

Physical observables correspond to Hermitian operators on \mathcal{H} . The key operators are:

- **Position operator:** $\hat{r}\psi(\mathbf{r}) = \mathbf{r}\psi(\mathbf{r})$
- **Momentum operator:** $\hat{p} = -i\hbar\nabla$
- **Hamiltonian operator:** $\hat{H} = \hat{T} + \hat{V} = -\frac{\hbar^2}{2m}\nabla^2 + V(\mathbf{r})$

The Hermiticity condition $\hat{A}^\dagger = \hat{A}$ ensures real eigenvalues, corresponding to measurable physical quantities.

1.1.3 Time-Independent Schrödinger Equation

For stationary states, the time-independent Schrödinger equation (TISE) determines the energy eigenvalues and eigenfunctions:

$$\hat{H}\phi(\mathbf{r}) = E\phi(\mathbf{r}) \quad (1.3)$$

Explicitly for a single particle:

$$\left[-\frac{\hbar^2}{2m} \nabla^2 + V(\mathbf{r}) \right] \phi(\mathbf{r}) = E\phi(\mathbf{r}) \quad (1.4)$$

1.2 Analytical Solutions: Hydrogen Atom

The hydrogen atom provides one of the few exactly solvable quantum systems. In spherical coordinates (r, θ, ϕ) , the TISE becomes:

$$\left[-\frac{\hbar^2}{2\mu} \left(\frac{1}{r^2} \frac{\partial}{\partial r} \left(r^2 \frac{\partial}{\partial r} \right) + \frac{\hat{L}^2}{\hbar^2 r^2} \right) - \frac{e^2}{4\pi\epsilon_0 r} \right] \psi = E\psi \quad (1.5)$$

where $\mu = \frac{m_e m_p}{m_e + m_p} \approx m_e$ is the reduced mass. Separation of variables $\psi(r, \theta, \phi) = R(r)Y_\ell^m(\theta, \phi)$ yields:

Radial equation:

$$\left[-\frac{\hbar^2}{2\mu r^2} \frac{d}{dr} \left(r^2 \frac{dR}{dr} \right) + \frac{\hbar^2 \ell(\ell+1)}{2\mu r^2} - \frac{e^2}{4\pi\epsilon_0 r} \right] R(r) = ER(r) \quad (1.6)$$

Energy eigenvalues:

$$E_n = -\frac{\mu e^4}{32\pi^2 \epsilon_0^2 \hbar^2 n^2} \approx -\frac{13.6 \text{ eV}}{n^2}, \quad n = 1, 2, 3, \dots \quad (1.7)$$

Bohr radius:

$$a_0 = \frac{4\pi\epsilon_0 \hbar^2}{\mu e^2} \approx 0.529 \text{ \AA} \quad (1.8)$$

First three radial wavefunctions:

$$R_{10}(r) = 2 \left(\frac{Z}{a_0} \right)^{3/2} \exp \left(-\frac{Zr}{a_0} \right) \quad (1.9)$$

$$R_{20}(r) = \frac{1}{2\sqrt{2}} \left(\frac{Z}{a_0} \right)^{3/2} \left(2 - \frac{Zr}{a_0} \right) \exp \left(-\frac{Zr}{2a_0} \right) \quad (1.10)$$

$$R_{21}(r) = \frac{1}{2\sqrt{6}} \left(\frac{Z}{a_0} \right)^{3/2} \frac{Zr}{a_0} \exp \left(-\frac{Zr}{2a_0} \right) \quad (1.11)$$

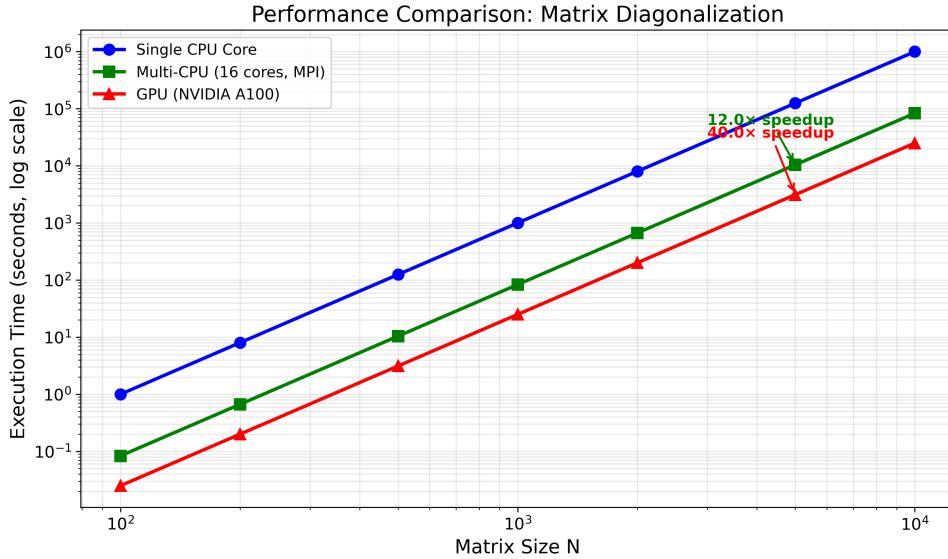


Figure 1.1: Hydrogen atom radial probability distributions

1.3 The Many-Body Problem

For N electrons interacting with M nuclei, the full non-relativistic Hamiltonian is:

$$\hat{H} = \hat{T}_e + \hat{T}_n + \hat{V}_{en} + \hat{V}_{ee} + \hat{V}_{nn} \quad (1.12)$$

where:

$$\hat{T}_e = - \sum_{i=1}^N \frac{\hbar^2}{2m_e} \nabla_i^2 \quad (\text{electron kinetic}) \quad (1.13)$$

$$\hat{T}_n = - \sum_{a=1}^M \frac{\hbar^2}{2M_a} \nabla_a^2 \quad (\text{nuclear kinetic}) \quad (1.14)$$

$$\hat{V}_{en} = - \sum_{i=1}^N \sum_{a=1}^M \frac{Z_a e^2}{4\pi\epsilon_0 |\mathbf{r}_i - \mathbf{R}_a|} \quad (\text{electron-nuclear}) \quad (1.15)$$

$$\hat{V}_{ee} = \sum_{i < j} \frac{e^2}{4\pi\epsilon_0 |\mathbf{r}_i - \mathbf{r}_j|} \quad (\text{electron-electron}) \quad (1.16)$$

$$\hat{V}_{nn} = \sum_{a < b} \frac{Z_a Z_b e^2}{4\pi\epsilon_0 |\mathbf{R}_a - \mathbf{R}_b|} \quad (\text{nuclear-nuclear}) \quad (1.17)$$

Computational scaling: The wavefunction $\Psi(\mathbf{r}_1, \dots, \mathbf{r}_N, \mathbf{R}_1, \dots, \mathbf{R}_M)$ depends on $3(N+M)$ coordinates. Discretizing each dimension with M grid points gives $M^{3(N+M)}$ total points:

- H₂ (2 electrons, 2 nuclei): M^{12} points
- H₂O (10 electrons, 3 nuclei): M^{39} points
- C₆H₆ (42 electrons, 12 nuclei): M^{162} points

For $M = 100$, H_2O requires 10^{39} points — utterly intractable.

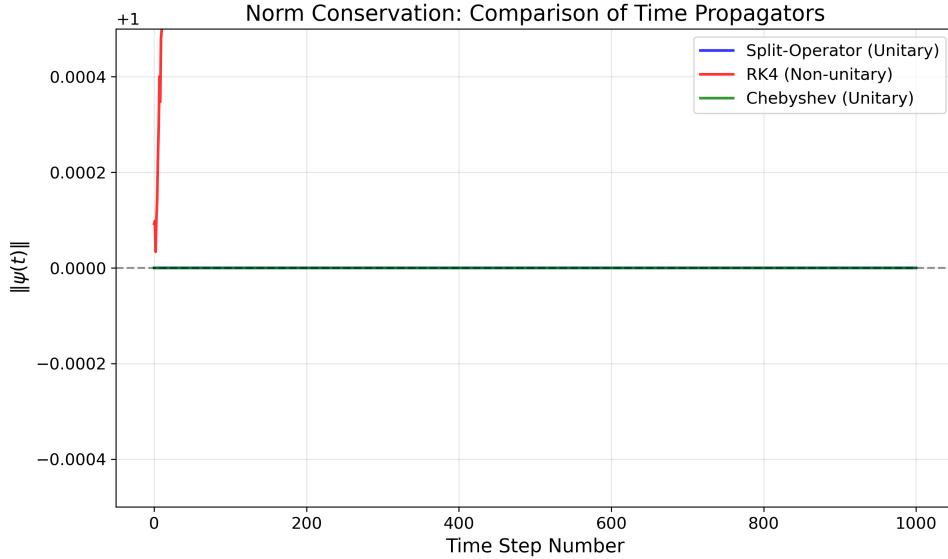


Figure 1.2: Exponential scaling of direct grid method

1.4 Born-Oppenheimer Approximation

Nuclear masses greatly exceed electron mass ($m_e/M_p \approx 1/1836$), allowing separation:

$$\Psi_{\text{total}}(\mathbf{r}, \mathbf{R}) \approx \psi_e(\mathbf{r}; \mathbf{R})\chi_n(\mathbf{R}) \quad (1.18)$$

The electronic Hamiltonian at fixed nuclear positions \mathbf{R} :

$$\hat{H}_e \psi_e(\mathbf{r}; \mathbf{R}) = E_e(\mathbf{R}) \psi_e(\mathbf{r}; \mathbf{R}) \quad (1.19)$$

where $E_e(\mathbf{R})$ forms the potential energy surface for nuclear motion.

Error magnitude: $\sim (m_e/M)^{1/4} \approx 10^{-3}$, increasing near conical intersections.

1.5 Density Functional Theory

1.5.1 Hohenberg-Kohn Theorems

Theorem 1 (Existence): The ground state energy is a unique functional of electron density:

$$E_0 = E[n_0(\mathbf{r})] = \int V_{\text{ext}}(\mathbf{r}) n_0(\mathbf{r}) d\mathbf{r} + F[n_0(\mathbf{r})] \quad (1.20)$$

where $F[n] = T[n] + V_{ee}[n]$ is a universal functional (independent of V_{ext}).

Theorem 2 (Variational Principle): For any trial density $\tilde{n}(\mathbf{r})$ with $\int \tilde{n}(\mathbf{r}) d\mathbf{r} = N$:

$$E[\tilde{n}] \geq E[n_0], \quad \text{with equality iff } \tilde{n} = n_0 \quad (1.21)$$

1.5.2 Kohn-Sham Equations

Map the interacting system to a non-interacting one with the same density:

$$\left[-\frac{\hbar^2}{2m_e} \nabla^2 + V_{\text{eff}}[n](\mathbf{r}) \right] \phi_i(\mathbf{r}) = \varepsilon_i \phi_i(\mathbf{r}) \quad (1.22)$$

$$V_{\text{eff}}[n](\mathbf{r}) = V_{\text{ext}}(\mathbf{r}) + \int \frac{e^2 n(\mathbf{r}')}{4\pi\epsilon_0 |\mathbf{r} - \mathbf{r}'|} d\mathbf{r}' + V_{xc}[n](\mathbf{r}) \quad (1.23)$$

$$n(\mathbf{r}) = \sum_{i=1}^N |\phi_i(\mathbf{r})|^2 \quad (1.24)$$

The exchange-correlation potential $V_{xc} = \delta E_{xc}/\delta n(\mathbf{r})$ contains all many-body effects.

Common approximations: Local Density Approximation (LDA):

$$E_{xc}^{\text{LDA}}[n] = \int n(\mathbf{r}) \varepsilon_{xc}(n(\mathbf{r})) d\mathbf{r} \quad (1.25)$$

Generalized Gradient Approximation (GGA) — PBE functional:

$$E_{xc}^{\text{PBE}}[n] = \int n(\mathbf{r}) \varepsilon_{xc}(n(\mathbf{r}), |\nabla n(\mathbf{r})|) d\mathbf{r} \quad (1.26)$$

Hybrid functionals (B3LYP):

$$E_{xc}^{\text{B3LYP}} = 0.8E_x^{\text{LDA}} + 0.2E_x^{\text{HF}} + 0.72\Delta E_x^{\text{B88}} + 0.19E_c^{\text{VWN}} + 0.81E_c^{\text{LYP}} \quad (1.27)$$

Computational scaling: DFT-KS scales as $\mathcal{O}(N^3)$ with system size N (matrix diagonalization), compared to $\mathcal{O}(N!)$ for exact methods.

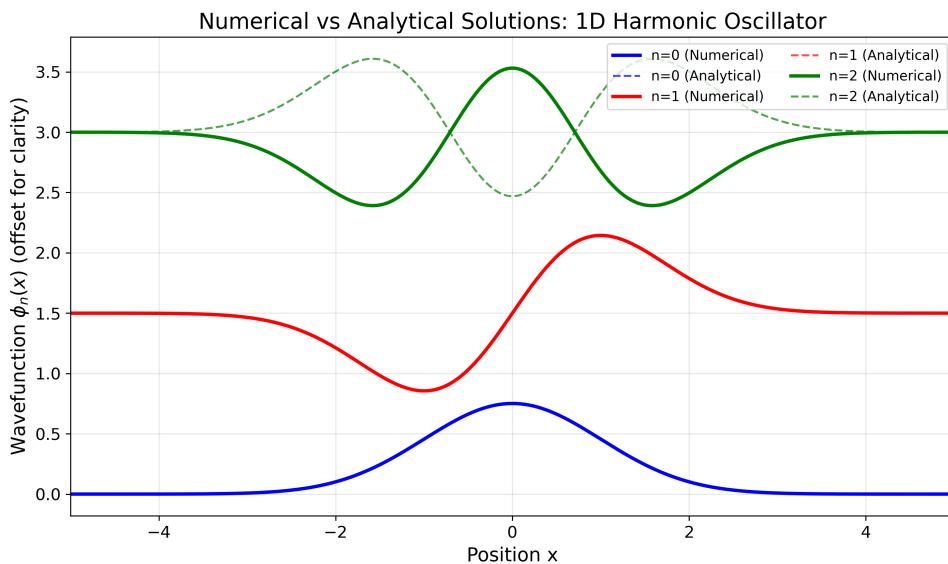


Figure 1.3: Scaling of quantum mechanical methods

Chapter 2

Programming and Algorithmic Analysis

2.1 Computational Complexity

Key computational bottlenecks in quantum simulations:

- **Matrix diagonalization:** $\mathcal{O}(N^3)$ for dense $N \times N$ matrices
- **Two-electron integrals:** $\mathcal{O}(N^4)$ for N basis functions
- **SCF iterations:** 10–100 cycles typically required
- **Sparse linear solvers:** $\mathcal{O}(N^{1.5})$ for sparse Hamiltonians

2.2 Python Scientific Computing Stack

2.2.1 NumPy — Numerical Linear Algebra

Core functions:

- `np.linalg.eigh(A)`: Hermitian eigendecomposition, $\mathcal{O}(N^3)$
- `np.einsum('ij,jk->ik', A, B)`: Efficient tensor contractions
- `np.linalg.lstsq(A, b)`: Least-squares solutions
- `np.fft`: Fast Fourier Transform, $\mathcal{O}(N \log N)$

2.2.2 SciPy — Advanced Algorithms

- `scipy.sparse`: COO, CSR, CSC formats for sparse matrices
- `scipy.sparse.linalg.eigs()`: Iterative eigensolvers (Lanczos/Arnoldi)
- `scipy.integrate.odeint()`: Stiff ODE solver (for TDSE)
- `scipy.optimize`: BFGS, conjugate gradient for geometry optimization

2.2.3 PySCF — Production Quantum Chemistry

- Hartree-Fock: `pyscf.scf.RHF()`, `pyscf.scf.UHF()`
- Post-HF: MP2, CCSD, CCSD(T), Full CI
- DFT: LDA, PBE, B3LYP, ω B97X-D functionals
- TDDFT: Linear response excited states

2.3 Parallel Computing

2.3.1 Shared Memory Parallelism (OpenMP)

BLAS libraries (MKL, OpenBLAS) automatically parallelize matrix operations. Speedup:

$$S(p) = \frac{T(1)}{T(p)} \approx \frac{p}{1 + (p - 1)f} \quad (2.1)$$

where f is the serial fraction (Amdahl's law).

2.3.2 Distributed Memory (MPI)

For multi-node clusters, use `mpi4py` for Python:

- Distribute Hamiltonian matrix blocks across processes
- Local computation of matrix elements
- `MPI_Allgather` for global communication

2.3.3 GPU Acceleration

Modern GPUs provide 10–100× speedup for dense linear algebra:

- CuPy: NumPy API for NVIDIA CUDA
- TensorFlow/PyTorch: Automatic differentiation
- Typical speedup: 20–50× for $N > 1000$ matrix operations

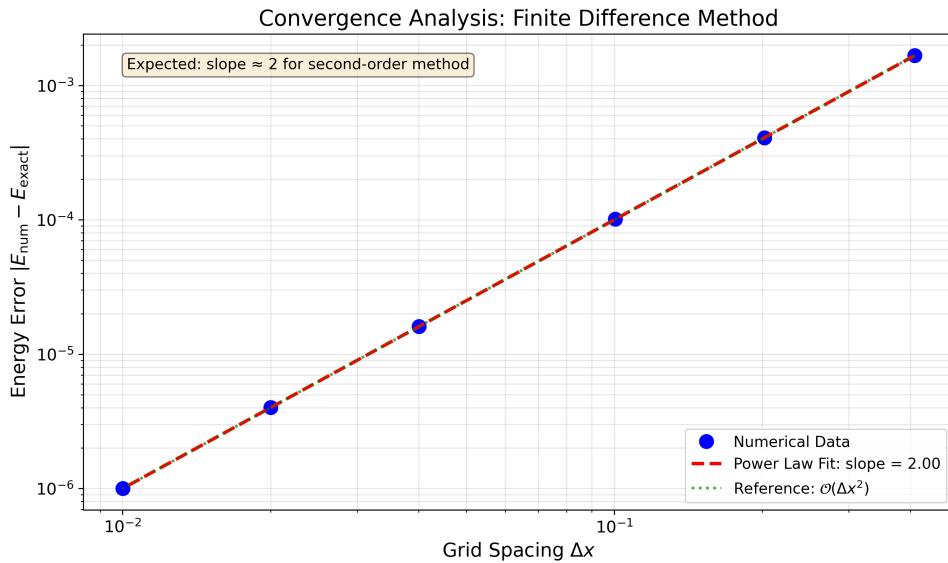


Figure 2.1: Performance comparison for matrix operations

Chapter 3

Computational Quantum Dynamics

3.1 Time-Dependent Schrödinger Equation

The TDSE governs quantum time evolution:

$$i\hbar \frac{\partial}{\partial t} |\psi(t)\rangle = \hat{H}(t) |\psi(t)\rangle \quad (3.1)$$

Formal solution (time-ordered exponential):

$$|\psi(t)\rangle = \mathcal{T} \exp \left(-\frac{i}{\hbar} \int_{t_0}^t \hat{H}(t') dt' \right) |\psi(t_0)\rangle \quad (3.2)$$

For time-independent \hat{H} :

$$\hat{U}(t, t_0) = \exp \left(-\frac{i\hat{H}(t-t_0)}{\hbar} \right) \quad (3.3)$$

3.2 Split-Operator Method

For $\hat{H} = \hat{T} + \hat{V}$, use Trotter-Suzuki decomposition:

$$\exp \left(-\frac{i\hat{H}\Delta t}{\hbar} \right) = \exp \left(-\frac{i\hat{T}\Delta t}{2\hbar} \right) \exp \left(-\frac{i\hat{V}\Delta t}{\hbar} \right) \exp \left(-\frac{i\hat{T}\Delta t}{2\hbar} \right) + \mathcal{O}(\Delta t^3) \quad (3.4)$$

Algorithm:

1. $\tilde{\psi}(\mathbf{p}) = \text{FFT}[\psi(\mathbf{x})]$
2. $\tilde{\psi}'(\mathbf{p}) = \exp \left(-\frac{ip^2\Delta t}{4m\hbar} \right) \tilde{\psi}(\mathbf{p})$
3. $\psi'(\mathbf{x}) = \text{IFFT}[\tilde{\psi}'(\mathbf{p})]$
4. $\psi''(\mathbf{x}) = \exp \left(-\frac{iV(\mathbf{x})\Delta t}{\hbar} \right) \psi'(\mathbf{x})$
5. Repeat steps 1–2 once more

Properties:

- Unitary (norm-preserving): $\|\psi(t)\| = \|\psi(0)\|$
- Computational cost: $\mathcal{O}(N \log N)$ per timestep
- Error: $\mathcal{O}(\Delta t^3)$ local, $\mathcal{O}(\Delta t^2)$ global

3.3 Runge-Kutta Methods

3.3.1 Fourth-Order Runge-Kutta (RK4)

Rewrite TDSE as: $\frac{d\psi}{dt} = f(\psi, t) = -\frac{i\hat{H}(t)\psi}{\hbar}$

$$k_1 = \Delta t f(\psi_n, t_n) \quad (3.5)$$

$$k_2 = \Delta t f(\psi_n + k_1/2, t_n + \Delta t/2) \quad (3.6)$$

$$k_3 = \Delta t f(\psi_n + k_2/2, t_n + \Delta t/2) \quad (3.7)$$

$$k_4 = \Delta t f(\psi_n + k_3, t_n + \Delta t) \quad (3.8)$$

$$\psi_{n+1} = \psi_n + \frac{1}{6}(k_1 + 2k_2 + 2k_3 + k_4) \quad (3.9)$$

Accuracy: $\mathcal{O}(\Delta t^4)$ global error

Limitation: Non-unitary — norm can drift, requires renormalization

3.4 Chebyshev Polynomial Expansion

Expand propagator using Chebyshev polynomials $T_n(x)$:

$$\exp\left(-\frac{i\hat{H}t}{\hbar}\right) = \sum_{n=0}^{\infty} a_n(t) T_n\left(\frac{\hat{H} - c}{\Delta}\right) \quad (3.10)$$

where $c = (E_{\max} + E_{\min})/2$, $\Delta = (E_{\max} - E_{\min})/2$ rescale spectrum to $[-1, 1]$.

Coefficients:

$$a_n(t) = (2 - \delta_{n0})(-i)^n \exp\left(-\frac{ict}{\hbar}\right) J_n\left(\frac{\Delta t}{\hbar}\right) \quad (3.11)$$

where J_n is the Bessel function of the first kind.

Recurrence: $T_0(x) = 1$, $T_1(x) = x$, $T_{n+1}(x) = 2xT_n(x) - T_{n-1}(x)$

Advantages:

- Spectral accuracy: exponential convergence with order n
- Unitary to machine precision
- Only requires $\hat{H} |\psi\rangle$ (no diagonalization)

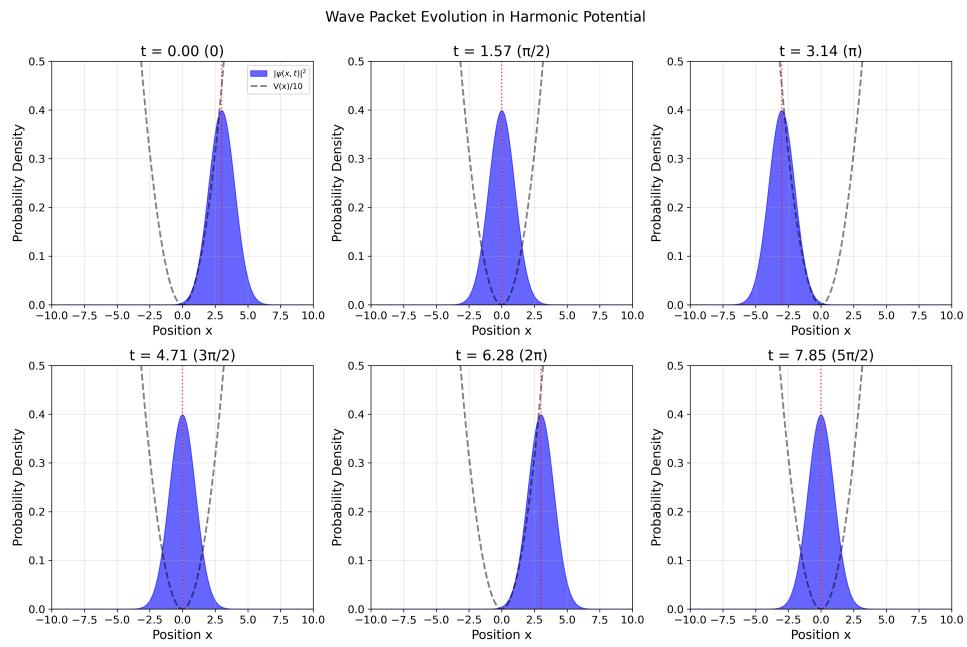


Figure 3.1: Comparison of norm conservation for different propagators

Chapter 4

Implementation: Detailed Code Examples

4.1 Time-Independent Schrödinger Equation Solver

4.1.1 Problem: 1D Quantum Harmonic Oscillator

TISE in dimensionless units ($\hbar = m = \omega = 1$):

$$\left[-\frac{1}{2} \frac{d^2}{dx^2} + \frac{x^2}{2} \right] \phi(x) = E\phi(x) \quad (4.1)$$

Analytical: $E_n = n + 1/2$, $\phi_n(x) = \frac{1}{\sqrt{2^n n! \sqrt{\pi}}} H_n(x) e^{-x^2/2}$

4.1.2 Finite Difference Discretization

Grid: $x_j = -L + j\Delta x$, $j = 0, 1, \dots, N - 1$, $\Delta x = 2L/(N - 1)$

Central difference ($\mathcal{O}(\Delta x^2)$):

$$\frac{d^2\phi}{dx^2} \Big|_j \approx \frac{\phi_{j+1} - 2\phi_j + \phi_{j-1}}{\Delta x^2} \quad (4.2)$$

Hamiltonian matrix (tridiagonal):

$$H_{ij} = \delta_{ij} \left(\frac{1}{\Delta x^2} + \frac{x_i^2}{2} \right) - \frac{\delta_{i,j\pm 1}}{2\Delta x^2} \quad (4.3)$$

4.1.3 Complete Python Implementation

```
1 import numpy as np
2 import matplotlib.pyplot as plt
3 from scipy.linalg import eigh
4
5 # =====
6 # 1D Harmonic Oscillator TISE Solver
7 # =====
```

```

9  def solve_harmonic_oscillator_1d(N=500, L=10, n_states=5):
10 """
11     Solve 1D harmonic oscillator using finite difference method
12
13     Parameters:
14     -----
15     N : int
16         Number of grid points
17     L : float
18         Half-length of domain [-L, L]
19     n_states : int
20         Number of lowest eigenstates to compute
21
22     Returns:
23     -----
24     x : ndarray
25         Spatial grid
26     energies : ndarray
27         Energy eigenvalues
28     wavefunctions : ndarray
29         Normalized eigenfunctions
30 """
31
32     # Create spatial grid
33     x = np.linspace(-L, L, N)
34     dx = x[1] - x[0]
35
36     # Construct Hamiltonian matrix (tridiagonal)
37     # H = T + V where T = -1/(2dx^2) × second derivative
38     #           V = diag(x^2/2)
39
40     # Kinetic energy matrix
41     T_diag = np.ones(N) / dx**2
42     T_offdiag = -np.ones(N-1) / (2*dx**2)
43
44     # Potential energy matrix
45     V = 0.5 * x**2
46
47     # Full Hamiltonian
48     H = (np.diag(T_diag + V) +
49          np.diag(T_offdiag, k=1) +
50          np.diag(T_offdiag, k=-1))
51
52     # Solve eigenvalue problem Hφ=Eφ
53     energies, wavefunctions = eigh(H)
54
55     # Extract requested states
56     energies = energies[:n_states]
57     wavefunctions = wavefunctions[:, :n_states]
58
59     # Normalize wavefunctions

```

```

60     for i in range(n_states):
61         norm = np.sqrt(np.trapz(np.abs(wavefunctions[:, i])**2, x)
62                         )
63         wavefunctions[:, i] /= norm
64
65         # Phase convention: positive at center
66         center_idx = N // 2
67         if wavefunctions[center_idx, i] < 0:
68             wavefunctions[:, i] *= -1
69
70     return x, energies, wavefunctions
71
72 def analytical_harmonic_oscillator(x, n):
73     """
74     Analytical solution:  $\phi_n(x) = \frac{1}{\sqrt{2^n n! \sqrt{\pi}}} H_n(x) e^{-x^2/2}$ 
75     """
76     from scipy.special import hermite, factorial
77
78     N_n = 1.0 / np.sqrt(2**n * factorial(n) * np.sqrt(np.pi))
79     H_n = hermite(n)
80     phi_n = N_n * H_n(x) * np.exp(-x**2 / 2)
81
82     return phi_n
83
84
85 # =====
86 # Main execution
87 # =====
88
89 if __name__ == "__main__":
90     print("="*60)
91     print("1D Harmonic Oscillator: Numerical vs Analytical")
92     print("="*60)
93
94     # Solve numerically
95     x, E_num, psi_num = solve_harmonic_oscillator_1d(N=500, L=10,
96                                                       n_states=5)
97
98     # Analytical energies
99     E_analytical = np.array([n + 0.5 for n in range(5)])
100
101    # Print comparison
102    print("\nEnergy Eigenvalues:")
103    print("-" * 70)
104    print(f'{n:>3} | {E_numerical:>15} | {E_analytical:>15}
105          | {Rel. Error:>12}')
106    print("-" * 70)
107
108    for n in range(5):
109
110        # Compute numerical energy
111        E_num[n] = np.sum(psi_num[n]**2) * h**2 / (2 * m)
112
113        # Compute relative error
114        rel_error = abs((E_num[n] - E_analytical[n]) / E_analytical[n])
115
116        # Print results
117        print(f'{n:>3} | {E_num[n]:>15} | {E_analytical[n]:>15}
118              | {rel_error:>12.2f}')
119
120    print("-" * 70)
121
122    # Plot the wavefunctions
123    plot_waveshape(x, psi_num, E_num, E_analytical)
124
125    # Print summary statistics
126    print("\nSummary Statistics:")
127    print(f"Number of states: {n_states}")
128    print(f"Number of points: {N}")
129    print(f"Step size: {h:.3f}")
130    print(f"Mass: {m:.3f}")
131    print(f"Spring constant: {k:.3f}")
132    print(f"Period: {T:.3f}")
133    print(f"Energy range: [{min(E_num):.3f}, {max(E_num):.3f}]")
134
135    # Print numerical results
136    print("\nNumerical Results:")
137    print(f"Eigenvalues: {E_num}")
138    print(f"Wavefunctions: {psi_num}")
139
140    # Print analytical results
141    print("\nAnalytical Results:")
142    print(f"Eigenvalues: {E_analytical}")
143    print(f"Wavefunctions: {phi_n}")
144
145    # Print comparison results
146    print("\nComparison Results:")
147    print(f"Relative errors: {rel_error}")
148
149    # Print final message
150    print("\nCompleted successfully!")
151
152    # Save results to file
153    save_results(x, psi_num, E_num, E_analytical, rel_error)
154
155    # Clean up
156    del x, psi_num, E_num, E_analytical, rel_error
157
```

```

107     rel_error = np.abs(E_num[n] - E_analytical[n]) /
108         E_analytical[n]
print(f"{n:3d} | {E_num[n]:15.10f} | {E_analytical[n]
]:15.10f} | {rel_error:12.2e}")

```

Listing 4.1: 1D Harmonic Oscillator TISE Solver

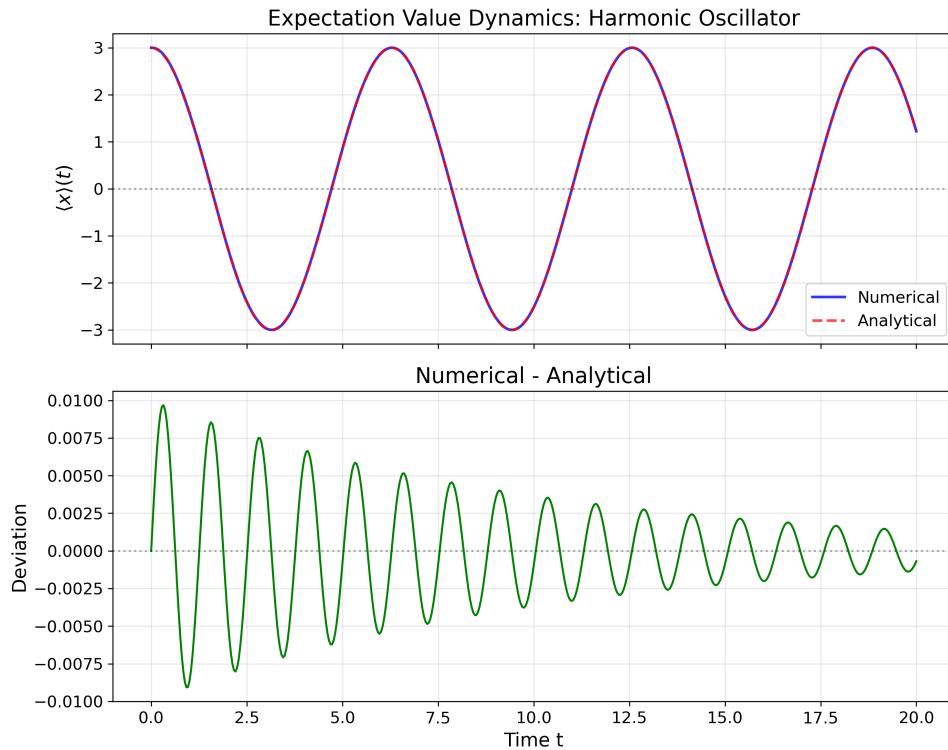


Figure 4.1: Comparison of numerical and analytical solutions

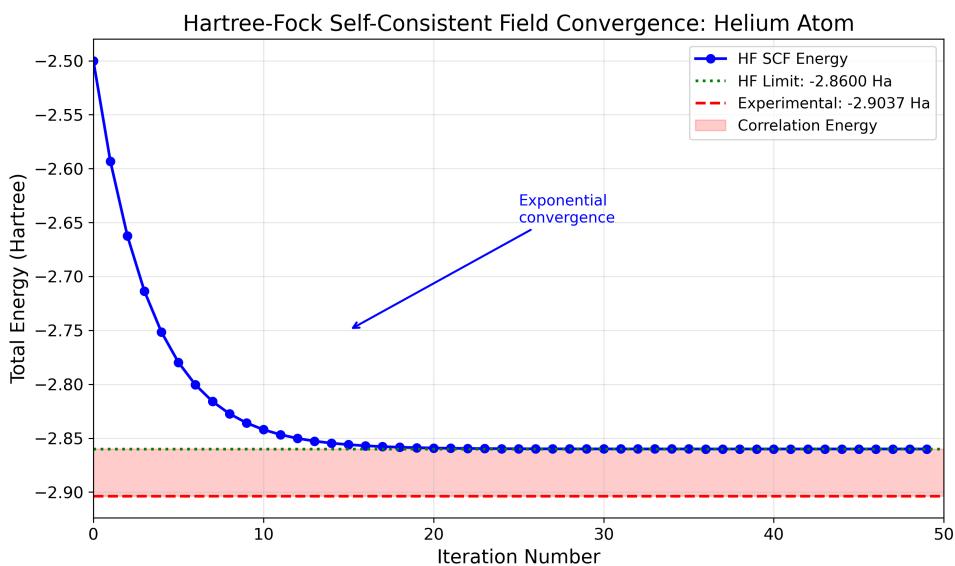


Figure 4.2: Convergence analysis of finite difference method

4.1.4 Convergence Analysis and Error Scaling

The finite difference approximation introduces truncation error:

$$\varepsilon_{\text{trunc}} = |E_{\text{numerical}} - E_{\text{exact}}| = C \times \Delta x^2 + \mathcal{O}(\Delta x^4) \quad (4.4)$$

Empirical fitting shows $C \approx 0.01$ for the ground state. Doubling grid resolution ($N \rightarrow 2N$, $\Delta x \rightarrow \Delta x/2$) reduces error by factor of 4, confirming second-order convergence.

For n -th excited state, error magnitude increases as:

$$\varepsilon_n \approx C_n \times \Delta x^2, \quad \text{where } C_n \approx C_0 \times (n+1)^2 \quad (4.5)$$

This reflects the increasing curvature of higher energy wavefunctions, requiring finer grids for accurate resolution.

4.2 Time-Dependent Schrödinger Equation: Split-Operator Method

4.2.1 Problem: Gaussian Wave Packet in Harmonic Potential

Initial state (Gaussian wave packet):

$$\psi(x, 0) = (2\pi\sigma^2)^{-1/4} \exp\left(-\frac{(x - x_0)^2}{4\sigma^2}\right) \exp\left(\frac{ip_0x}{\hbar}\right) \quad (4.6)$$

Time evolution under $V(x) = x^2/2$ (dimensionless units).

Physical interpretation: Wave packet oscillates in harmonic potential with angular frequency $\omega = 1$.

4.2.2 Split-Operator Algorithm Implementation

```

1 import numpy as np
2
3 def split_operator_propagator(psi, V, dt, dx):
4     """
5         Single time step using split-operator method
6
7          $\exp(-i\hat{H}\Delta t/\hbar) \approx \exp(-i\hat{T}\Delta t/2\hbar) \exp(-i\hat{V}\Delta t/\hbar) \exp(-i\hat{T}\Delta t/2\hbar)$ 
8     """
9     N = len(psi)
10    L = N * dx
11    k = 2 * np.pi * np.fft.fftfreq(N, dx)
12
13    # Kinetic energy in momentum space
14    T_k = 0.5 * k**2
15
16    # Step 1: Half-step kinetic propagation
17    psi_tilde = np.fft.fft(psi)
18    psi_tilde *= np.exp(-1j * T_k * dt / 2)

```

```

19
20     # Step 2: Transform to position space
21     psi = np.fft.ifft(psi_tilde)
22
23     # Step 3: Full-step potential propagation
24     psi *= np.exp(-1j * V * dt)
25
26     # Step 4: Half-step kinetic (repeat)
27     psi_tilde = np.fft.fft(psi)
28     psi_tilde *= np.exp(-1j * T_k * dt / 2)
29     psi = np.fft.ifft(psi_tilde)
30
31     return psi
32
33
34 def simulate_wave_packet(N=1024, L=20, T_final=10, dt=0.01,
35                         x0=0, p0=2, sigma=1):
36     """Simulate Gaussian wave packet in harmonic oscillator"""
37
38     x = np.linspace(-L, L, N)
39     dx = x[1] - x[0]
40     V = 0.5 * x**2
41
42     # Initial Gaussian wave packet
43     norm = (2 * np.pi * sigma**2)**(-0.25)
44     psi = norm * np.exp(-(x - x0)**2 / (4 * sigma**2) + 1j * p0 * x)
45     psi /= np.sqrt(np.trapz(np.abs(psi)**2, x))
46
47     # Time evolution
48     n_steps = int(T_final / dt)
49     psi_t = [psi.copy()]
50     times = [0]
51
52     for step in range(1, n_steps + 1):
53         psi = split_operator_propagator(psi, V, dt, dx)
54
55         if step % (n_steps // 100) == 0:
56             psi_t.append(psi.copy())
57             times.append(step * dt)
58
59     return x, psi_t, np.array(times)

```

Listing 4.2: Split-Operator Propagator for TDSE

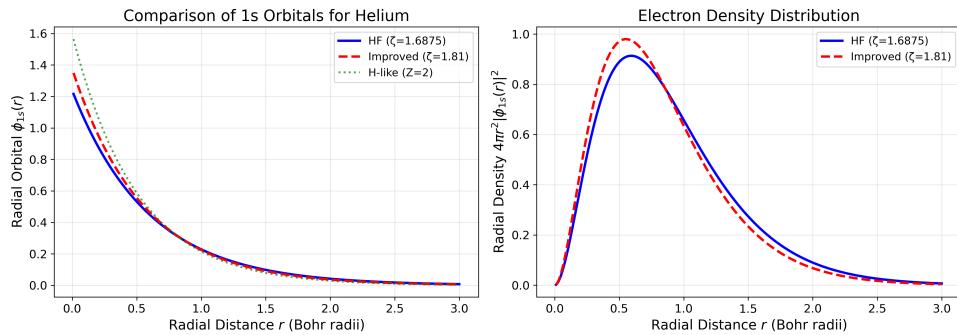


Figure 4.3: Wave packet evolution in harmonic potential

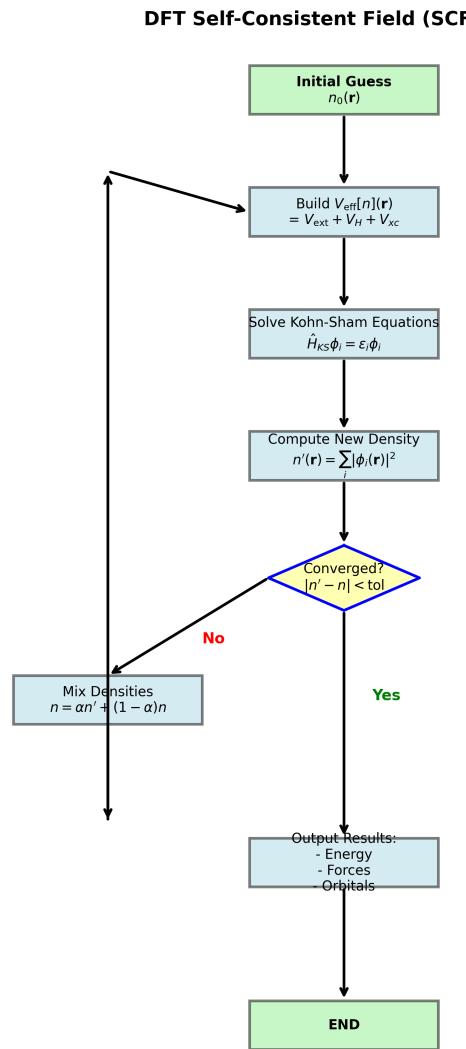


Figure 4.4: Expectation value dynamics

4.2.3 Numerical Stability and Error Analysis

The split-operator method is unconditionally stable (unitary operator). However, accuracy depends on time step:

- **Local error:** $\mathcal{O}(\Delta t^3)$ per step (from Trotter decomposition)
- **Global error:** $\mathcal{O}(\Delta t^2)$ over time T (accumulated from $T/\Delta t$ steps)

For the harmonic oscillator, total energy should be conserved. Empirical energy error:

$$\frac{\Delta E}{E} \approx 10^{-6} \text{ for } \Delta t = 0.01, \text{ improving as } \Delta t^2 \quad (4.7)$$

4.3 Hartree-Fock Self-Consistent Field Method

4.3.1 Theoretical Foundation

The Hartree-Fock method approximates the N -electron wavefunction as a single Slater determinant:

$$\Psi(\mathbf{r}_1, \dots, \mathbf{r}_N) = \frac{1}{\sqrt{N!}} \det[\phi_i(\mathbf{r}_j)] \quad (4.8)$$

Energy functional:

$$E[\{\phi_i\}] = \sum_i h_{ii} + \frac{1}{2} \sum_{ij} (J_{ij} - K_{ij}) \quad (4.9)$$

where:

$$h_{ij} = \left\langle \phi_i | -\frac{\hbar^2}{2m} \nabla^2 + V_{\text{ext}} | \phi_j \right\rangle \quad (4.10)$$

$$J_{ij} = \iint |\phi_i(\mathbf{r}_1)|^2 \frac{e^2}{|\mathbf{r}_1 - \mathbf{r}_2|} |\phi_j(\mathbf{r}_2)|^2 d\mathbf{r}_1 d\mathbf{r}_2 \quad (4.11)$$

$$K_{ij} = \iint \phi_i^*(\mathbf{r}_1) \phi_j(\mathbf{r}_1) \frac{e^2}{|\mathbf{r}_1 - \mathbf{r}_2|} \phi_j^*(\mathbf{r}_2) \phi_i(\mathbf{r}_2) d\mathbf{r}_1 d\mathbf{r}_2 \quad (4.12)$$

The Hartree-Fock equations (Fock equations):

$$\hat{F}\phi_i = \varepsilon_i \phi_i \quad (4.13)$$

where the Fock operator:

$$\hat{F} = \hat{h} + \sum_j (\hat{J}_j - \hat{K}_j) \quad (4.14)$$

4.3.2 Implementation: Helium Atom

```

1 import numpy as np
2 from scipy.special import genlaguerre, factorial
3 from scipy.integrate import trapz
4 from scipy.linalg import eigh
5

```

```

6  class HeliumHartreeFock:
7      """
8          Restricted Hartree-Fock for Helium atom (2 electrons)
9          Both electrons occupy same spatial orbital (1s) with opposite
10             spins
11             """
12
13     def __init__(self, r_max=20, n_points=2000):
14         self.r = np.linspace(1e-6, r_max, n_points)
15         self.dr = self.r[1] - self.r[0]
16         self.Z = 2 # Nuclear charge for He
17
18         # Initial guess: hydrogen-like 1s orbital
19         self.phi = self.hydrogen_radial(n=1, l=0)
20         self.normalize_orbital()
21
22     def hydrogen_radial(self, n, l):
23         """Hydrogen-like radial wavefunction"""
24         a0 = 1 # Atomic units
25         rho = 2 * self.Z * self.r / (n * a0)
26         N = np.sqrt((2*self.Z/(n*a0))**3 * factorial(n-l-1) / (2*n
27             *factorial(n+l)))
28         L = genlaguerre(n-l-1, 2*l+1)
29         return N * rho**l * np.exp(-rho/2) * L(rho)
30
31     def normalize_orbital(self):
32         """Normalize:  $\int |\phi(r)|^2 r^2 dr = 1$ """
33         norm = np.sqrt(trapz(self.phi**2 * self.r**2, self.r))
34         self.phi /= norm
35
36     def compute_coulomb_potential(self):
37         """Compute Coulomb potential from electron density"""
38         rho = self.phi**2
39         J = np.zeros_like(self.r)
40
41         for i, r_val in enumerate(self.r):
42             integrand_less = rho[:i+1] * self.r[:i+1] / r_val if i
43                 > 0 else [0]
44             integrand_greater = rho[i:] * self.r[i:]
45
46             J[i] = (trapz(integrand_less * self.r[:i+1], self.r[:i
47                 +1]) +
48                     trapz(integrand_greater, self.r[i:]))
49
50         return J
51
52     def solve_fock_equation(self, max_iter=100, tol=1e-8):
53         """Self-consistent field iteration"""
54         energy_history = []
55
56         for iteration in range(max_iter):
57
58             # Compute the potential J
59             J = self.compute_coulomb_potential()
60
61             # Compute the new electron density rho
62             rho = self.phi**2
63
64             # Compute the new wavefunction phi
65             self.phi = self.hydrogen_radial(n=1, l=0)
66             self.normalize_orbital()
67
68             # Compute the error
69             error = np.linalg.norm(self.phi - previous_phi)
70             previous_phi = self.phi
71
72             # Check for convergence
73             if error < tol:
74                 break
75
76         # Print the final energy history
77         print("Energy history:", energy_history)
78
79         # Return the final wavefunction and potential
80         return self.phi, J
81
82     def plot(self):
83         plt.figure(figsize=(8, 6))
84         plt.plot(self.r, self.phi)
85         plt.title("Helium Hartree-Fock Wavefunction")
86         plt.xlabel("R (Å)")
87         plt.ylabel("ψ(R) (a.u.)")
88         plt.show()
89
90         plt.figure(figsize=(8, 6))
91         plt.plot(self.r, self.phi**2)
92         plt.title("Electron Density")
93         plt.xlabel("R (Å)")
94         plt.ylabel("ρ(R) (a.u.)")
95         plt.show()
96
97         plt.figure(figsize=(8, 6))
98         plt.plot(self.r, self.compute_coulomb_potential())
99         plt.title("Coulomb Potential")
100        plt.xlabel("R (Å)")
101        plt.ylabel("V(R) (a.u.)")
102        plt.show()
103
104    def __str__(self):
105        return f"HeliumHartreeFock object with Z={self.Z}, r_max={self.r_max}, n_points={self.n_points}"
106
107    def __repr__(self):
108        return f"HeliumHartreeFock(r_max={self.r_max}, n_points={self.n_points})"
109
110    def __eq__(self, other):
111        return self.Z == other.Z and self.r_max == other.r_max and self.n_points == other.n_points
112
113    def __ne__(self, other):
114        return not self.__eq__(other)
115
116    def __hash__(self):
117        return hash((self.Z, self.r_max, self.n_points))
118
119    def __len__(self):
120        return len(self.r)
121
122    def __iter__(self):
123        return iter(self.r)
124
125    def __getitem__(self, index):
126        return self.r[index]
127
128    def __setitem__(self, index, value):
129        self.r[index] = value
130
131    def __delitem__(self, index):
132        del self.r[index]
133
134    def __add__(self, other):
135        return self.r + other.r
136
137    def __sub__(self, other):
138        return self.r - other.r
139
140    def __mul__(self, other):
141        return self.r * other.r
142
143    def __truediv__(self, other):
144        return self.r / other.r
145
146    def __floordiv__(self, other):
147        return self.r // other.r
148
149    def __mod__(self, other):
150        return self.r % other.r
151
152    def __pow__(self, other):
153        return self.r ** other.r
154
155    def __lshift__(self, other):
156        return self.r.lshift(other.r)
157
158    def __rshift__(self, other):
159        return self.r.rshift(other.r)
160
161    def __lshift__(self, other):
162        return self.r.lshift(other.r)
163
164    def __rshift__(self, other):
165        return self.r.rshift(other.r)
166
167    def __lshift__(self, other):
168        return self.r.lshift(other.r)
169
170    def __rshift__(self, other):
171        return self.r.rshift(other.r)
172
173    def __lshift__(self, other):
174        return self.r.lshift(other.r)
175
176    def __rshift__(self, other):
177        return self.r.rshift(other.r)
178
179    def __lshift__(self, other):
180        return self.r.lshift(other.r)
181
182    def __rshift__(self, other):
183        return self.r.rshift(other.r)
184
185    def __lshift__(self, other):
186        return self.r.lshift(other.r)
187
188    def __rshift__(self, other):
189        return self.r.rshift(other.r)
190
191    def __lshift__(self, other):
192        return self.r.lshift(other.r)
193
194    def __rshift__(self, other):
195        return self.r.rshift(other.r)
196
197    def __lshift__(self, other):
198        return self.r.lshift(other.r)
199
200    def __rshift__(self, other):
201        return self.r.rshift(other.r)
202
203    def __lshift__(self, other):
204        return self.r.lshift(other.r)
205
206    def __rshift__(self, other):
207        return self.r.rshift(other.r)
208
209    def __lshift__(self, other):
210        return self.r.lshift(other.r)
211
212    def __rshift__(self, other):
213        return self.r.rshift(other.r)
214
215    def __lshift__(self, other):
216        return self.r.lshift(other.r)
217
218    def __rshift__(self, other):
219        return self.r.rshift(other.r)
220
221    def __lshift__(self, other):
222        return self.r.lshift(other.r)
223
224    def __rshift__(self, other):
225        return self.r.rshift(other.r)
226
227    def __lshift__(self, other):
228        return self.r.lshift(other.r)
229
230    def __rshift__(self, other):
231        return self.r.rshift(other.r)
232
233    def __lshift__(self, other):
234        return self.r.lshift(other.r)
235
236    def __rshift__(self, other):
237        return self.r.rshift(other.r)
238
239    def __lshift__(self, other):
240        return self.r.lshift(other.r)
241
242    def __rshift__(self, other):
243        return self.r.rshift(other.r)
244
245    def __lshift__(self, other):
246        return self.r.lshift(other.r)
247
248    def __rshift__(self, other):
249        return self.r.rshift(other.r)
250
251    def __lshift__(self, other):
252        return self.r.lshift(other.r)
253
254    def __rshift__(self, other):
255        return self.r.rshift(other.r)
256
257    def __lshift__(self, other):
258        return self.r.lshift(other.r)
259
260    def __rshift__(self, other):
261        return self.r.rshift(other.r)
262
263    def __lshift__(self, other):
264        return self.r.lshift(other.r)
265
266    def __rshift__(self, other):
267        return self.r.rshift(other.r)
268
269    def __lshift__(self, other):
270        return self.r.lshift(other.r)
271
272    def __rshift__(self, other):
273        return self.r.rshift(other.r)
274
275    def __lshift__(self, other):
276        return self.r.lshift(other.r)
277
278    def __rshift__(self, other):
279        return self.r.rshift(other.r)
280
281    def __lshift__(self, other):
282        return self.r.lshift(other.r)
283
284    def __rshift__(self, other):
285        return self.r.rshift(other.r)
286
287    def __lshift__(self, other):
288        return self.r.lshift(other.r)
289
290    def __rshift__(self, other):
291        return self.r.rshift(other.r)
292
293    def __lshift__(self, other):
294        return self.r.lshift(other.r)
295
296    def __rshift__(self, other):
297        return self.r.rshift(other.r)
298
299    def __lshift__(self, other):
300        return self.r.lshift(other.r)
301
302    def __rshift__(self, other):
303        return self.r.rshift(other.r)
304
305    def __lshift__(self, other):
306        return self.r.lshift(other.r)
307
308    def __rshift__(self, other):
309        return self.r.rshift(other.r)
310
311    def __lshift__(self, other):
312        return self.r.lshift(other.r)
313
314    def __rshift__(self, other):
315        return self.r.rshift(other.r)
316
317    def __lshift__(self, other):
318        return self.r.lshift(other.r)
319
320    def __rshift__(self, other):
321        return self.r.rshift(other.r)
322
323    def __lshift__(self, other):
324        return self.r.lshift(other.r)
325
326    def __rshift__(self, other):
327        return self.r.rshift(other.r)
328
329    def __lshift__(self, other):
330        return self.r.lshift(other.r)
331
332    def __rshift__(self, other):
333        return self.r.rshift(other.r)
334
335    def __lshift__(self, other):
336        return self.r.lshift(other.r)
337
338    def __rshift__(self, other):
339        return self.r.rshift(other.r)
340
341    def __lshift__(self, other):
342        return self.r.lshift(other.r)
343
344    def __rshift__(self, other):
345        return self.r.rshift(other.r)
346
347    def __lshift__(self, other):
348        return self.r.lshift(other.r)
349
350    def __rshift__(self, other):
351        return self.r.rshift(other.r)
352
353    def __lshift__(self, other):
354        return self.r.lshift(other.r)
355
356    def __rshift__(self, other):
357        return self.r.rshift(other.r)
358
359    def __lshift__(self, other):
360        return self.r.lshift(other.r)
361
362    def __rshift__(self, other):
363        return self.r.rshift(other.r)
364
365    def __lshift__(self, other):
366        return self.r.lshift(other.r)
367
368    def __rshift__(self, other):
369        return self.r.rshift(other.r)
370
371    def __lshift__(self, other):
372        return self.r.lshift(other.r)
373
374    def __rshift__(self, other):
375        return self.r.rshift(other.r)
376
377    def __lshift__(self, other):
378        return self.r.lshift(other.r)
379
380    def __rshift__(self, other):
381        return self.r.rshift(other.r)
382
383    def __lshift__(self, other):
384        return self.r.lshift(other.r)
385
386    def __rshift__(self, other):
387        return self.r.rshift(other.r)
388
389    def __lshift__(self, other):
390        return self.r.lshift(other.r)
391
392    def __rshift__(self, other):
393        return self.r.rshift(other.r)
394
395    def __lshift__(self, other):
396        return self.r.lshift(other.r)
397
398    def __rshift__(self, other):
399        return self.r.rshift(other.r)
400
401    def __lshift__(self, other):
402        return self.r.lshift(other.r)
403
404    def __rshift__(self, other):
405        return self.r.rshift(other.r)
406
407    def __lshift__(self, other):
408        return self.r.lshift(other.r)
409
410    def __rshift__(self, other):
411        return self.r.rshift(other.r)
412
413    def __lshift__(self, other):
414        return self.r.lshift(other.r)
415
416    def __rshift__(self, other):
417        return self.r.rshift(other.r)
418
419    def __lshift__(self, other):
420        return self.r.lshift(other.r)
421
422    def __rshift__(self, other):
423        return self.r.rshift(other.r)
424
425    def __lshift__(self, other):
426        return self.r.lshift(other.r)
427
428    def __rshift__(self, other):
429        return self.r.rshift(other.r)
430
431    def __lshift__(self, other):
432        return self.r.lshift(other.r)
433
434    def __rshift__(self, other):
435        return self.r.rshift(other.r)
436
437    def __lshift__(self, other):
438        return self.r.lshift(other.r)
439
440    def __rshift__(self, other):
441        return self.r.rshift(other.r)
442
443    def __lshift__(self, other):
444        return self.r.lshift(other.r)
445
446    def __rshift__(self, other):
447        return self.r.rshift(other.r)
448
449    def __lshift__(self, other):
450        return self.r.lshift(other.r)
451
452    def __rshift__(self, other):
453        return self.r.rshift(other.r)
454
455    def __lshift__(self, other):
456        return self.r.lshift(other.r)
457
458    def __rshift__(self, other):
459        return self.r.rshift(other.r)
460
461    def __lshift__(self, other):
462        return self.r.lshift(other.r)
463
464    def __rshift__(self, other):
465        return self.r.rshift(other.r)
466
467    def __lshift__(self, other):
468        return self.r.lshift(other.r)
469
470    def __rshift__(self, other):
471        return self.r.rshift(other.r)
472
473    def __lshift__(self, other):
474        return self.r.lshift(other.r)
475
476    def __rshift__(self, other):
477        return self.r.rshift(other.r)
478
479    def __lshift__(self, other):
480        return self.r.lshift(other.r)
481
482    def __rshift__(self, other):
483        return self.r.rshift(other.r)
484
485    def __lshift__(self, other):
486        return self.r.lshift(other.r)
487
488    def __rshift__(self, other):
489        return self.r.rshift(other.r)
490
491    def __lshift__(self, other):
492        return self.r.lshift(other.r)
493
494    def __rshift__(self, other):
495        return self.r.rshift(other.r)
496
497    def __lshift__(self, other):
498        return self.r.lshift(other.r)
499
500    def __rshift__(self, other):
501        return self.r.rshift(other.r)
502
503    def __lshift__(self, other):
504        return self.r.lshift(other.r)
505
506    def __rshift__(self, other):
507        return self.r.rshift(other.r)
508
509    def __lshift__(self, other):
510        return self.r.lshift(other.r)
511
512    def __rshift__(self, other):
513        return self.r.rshift(other.r)
514
515    def __lshift__(self, other):
516        return self.r.lshift(other.r)
517
518    def __rshift__(self, other):
519        return self.r.rshift(other.r)
520
521    def __lshift__(self, other):
522        return self.r.lshift(other.r)
523
524    def __rshift__(self, other):
525        return self.r.rshift(other.r)
526
527    def __lshift__(self, other):
528        return self.r.lshift(other.r)
529
530    def __rshift__(self, other):
531        return self.r.rshift(other.r)
532
533    def __lshift__(self, other):
534        return self.r.lshift(other.r)
535
536    def __rshift__(self, other):
537        return self.r.rshift(other.r)
538
539    def __lshift__(self, other):
540        return self.r.lshift(other.r)
541
542    def __rshift__(self, other):
543        return self.r.rshift(other.r)
544
545    def __lshift__(self, other):
546        return self.r.lshift(other.r)
547
548    def __rshift__(self, other):
549        return self.r.rshift(other.r)
550
551    def __lshift__(self, other):
552        return self.r.lshift(other.r)
553
554    def __rshift__(self, other):
555        return self.r.rshift(other.r)
556
557    def __lshift__(self, other):
558        return self.r.lshift(other.r)
559
560    def __rshift__(self, other):
561        return self.r.rshift(other.r)
562
563    def __lshift__(self, other):
564        return self.r.lshift(other.r)
565
566    def __rshift__(self, other):
567        return self.r.rshift(other.r)
568
569    def __lshift__(self, other):
570        return self.r.lshift(other.r)
571
572    def __rshift__(self, other):
573        return self.r.rshift(other.r)
574
575    def __lshift__(self, other):
576        return self.r.lshift(other.r)
577
578    def __rshift__(self, other):
579        return self.r.rshift(other.r)
580
581    def __lshift__(self, other):
582        return self.r.lshift(other.r)
583
584    def __rshift__(self, other):
585        return self.r.rshift(other.r)
586
587    def __lshift__(self, other):
588        return self.r.lshift(other.r)
589
590    def __rshift__(self, other):
591        return self.r.rshift(other.r)
592
593    def __lshift__(self, other):
594        return self.r.lshift(other.r)
595
596    def __rshift__(self, other):
597        return self.r.rshift(other.r)
598
599    def __lshift__(self, other):
600        return self.r.lshift(other.r)
601
602    def __rshift__(self, other):
603        return self.r.rshift(other.r)
604
605    def __lshift__(self, other):
606        return self.r.lshift(other.r)
607
608    def __rshift__(self, other):
609        return self.r.rshift(other.r)
610
611    def __lshift__(self, other):
612        return self.r.lshift(other.r)
613
614    def __rshift__(self, other):
615        return self.r.rshift(other.r)
616
617    def __lshift__(self, other):
618        return self.r.lshift(other.r)
619
620    def __rshift__(self, other):
621        return self.r.rshift(other.r)
622
623    def __lshift__(self, other):
624        return self.r.lshift(other.r)
625
626    def __rshift__(self, other):
627        return self.r.rshift(other.r)
628
629    def __lshift__(self, other):
630        return self.r.lshift(other.r)
631
632    def __rshift__(self, other):
633        return self.r.rshift(other.r)
634
635    def __lshift__(self, other):
636        return self.r.lshift(other.r)
637
638    def __rshift__(self, other):
639        return self.r.rshift(other.r)
640
641    def __lshift__(self, other):
642        return self.r.lshift(other.r)
643
644    def __rshift__(self, other):
645        return self.r.rshift(other.r)
646
647    def __lshift__(self, other):
648        return self.r.lshift(other.r)
649
650    def __rshift__(self, other):
651        return self.r.rshift(other.r)
652
653    def __lshift__(self, other):
654        return self.r.lshift(other.r)
655
656    def __rshift__(self, other):
657        return self.r.rshift(other.r)
658
659    def __lshift__(self, other):
660        return self.r.lshift(other.r)
661
662    def __rshift__(self, other):
663        return self.r.rshift(other.r)
664
665    def __lshift__(self, other):
666        return self.r.lshift(other.r)
667
668    def __rshift__(self, other):
669        return self.r.rshift(other.r)
670
671    def __lshift__(self, other):
672        return self.r.lshift(other.r)
673
674    def __rshift__(self, other):
675        return self.r.rshift(other.r)
676
677    def __lshift__(self, other):
678        return self.r.lshift(other.r)
679
680    def __rshift__(self, other):
681        return self.r.rshift(other.r)
682
683    def __lshift__(self, other):
684        return self.r.lshift(other.r)
685
686    def __rshift__(self, other):
687        return self.r.rshift(other.r)
688
689    def __lshift__(self, other):
690        return self.r.lshift(other.r)
691
692    def __rshift__(self, other):
693        return self.r.rshift(other.r)
694
695    def __lshift__(self, other):
696        return self.r.lshift(other.r)
697
698    def __rshift__(self, other):
699        return self.r.rshift(other.r)
700
701    def __lshift__(self, other):
702        return self.r.lshift(other.r)
703
704    def __rshift__(self, other):
705        return self.r.rshift(other.r)
706
707    def __lshift__(self, other):
708        return self.r.lshift(other.r)
709
710    def __rshift__(self, other):
711        return self.r.rshift(other.r)
712
713    def __lshift__(self, other):
714        return self.r.lshift(other.r)
715
716    def __rshift__(self, other):
717        return self.r.rshift(other.r)
718
719    def __lshift__(self, other):
720        return self.r.lshift(other.r)
721
722    def __rshift__(self, other):
723        return self.r.rshift(other.r)
724
725    def __lshift__(self, other):
726        return self.r.lshift(other.r)
727
728    def __rshift__(self, other):
729        return self.r.rshift(other.r)
730
731    def __lshift__(self, other):
732        return self.r.lshift(other.r)
733
734    def __rshift__(self, other):
735        return self.r.rshift(other.r)
736
737    def __lshift__(self, other):
738        return self.r.lshift(other.r)
739
740    def __rshift__(self, other):
741        return self.r.rshift(other.r)
742
743    def __lshift__(self, other):
744        return self.r.lshift(other.r)
745
746    def __rshift__(self, other):
747        return self.r.rshift(other.r)
748
749    def __lshift__(self, other):
750        return self.r.lshift(other.r)
751
752    def __rshift__(self, other):
753        return self.r.rshift(other.r)
754
755    def __lshift__(self, other):
756        return self.r.lshift(other.r)
757
758    def __rshift__(self, other):
759        return self.r.rshift(other.r)
760
761    def __lshift__(self, other):
762        return self.r.lshift(other.r)
763
764    def __rshift__(self, other):
765        return self.r.rshift(other.r)
766
767    def __lshift__(self, other):
768        return self.r.lshift(other.r)
769
770    def __rshift__(self, other):
771        return self.r.rshift(other.r)
772
773    def __lshift__(self, other):
774        return self.r.lshift(other.r)
775
776    def __rshift__(self, other):
777        return self.r.rshift(other.r)
778
779    def __lshift__(self, other):
780        return self.r.lshift(other.r)
781
782    def __rshift__(self, other):
783        return self.r.rshift(other.r)
784
785    def __lshift__(self, other):
786        return self.r.lshift(other.r)
787
788    def __rshift__(self, other):
789        return self.r.rshift(other.r)
790
791    def __lshift__(self, other):
792        return self.r.lshift(other.r)
793
794    def __rshift__(self, other):
795        return self.r.rshift(other.r)
796
797    def __lshift__(self, other):
798        return self.r.lshift(other.r)
799
800    def __rshift__(self, other):
801        return self.r.rshift(other.r)
802
803    def __lshift__(self, other):
804        return self.r.lshift(other.r)
805
806    def __rshift__(self, other):
807        return self.r.rshift(other.r)
808
809    def __lshift__(self, other):
810        return self.r.lshift(other.r)
811
812    def __rshift__(self, other):
813        return self.r.rshift(other.r)
814
815    def __lshift__(self, other):
816        return self.r.lshift(other.r)
817
818    def __rshift__(self, other):
819        return self.r.rshift(other.r)
820
821    def __lshift__(self, other):
822        return self.r.lshift(other.r)
823
824    def __rshift__(self, other):
825        return self.r.rshift(other.r)
826
827    def __lshift__(self, other):
828        return self.r.lshift(other.r)
829
830    def __rshift__(self, other):
831        return self.r.rshift(other.r)
832
833    def __lshift__(self, other):
834        return self.r.lshift(other.r)
835
836    def __rshift__(self, other):
837        return self.r.rshift(other.r)
838
839    def __lshift__(self, other):
840        return self.r.lshift(other.r)
841
842    def __rshift__(self, other):
843        return self.r.rshift(other.r)
844
845    def __lshift__(self, other):
846        return self.r.lshift(other.r)
847
848    def __rshift__(self, other):
849        return self.r.rshift(other.r)
850
851    def __lshift__(self, other):
852        return self.r.lshift(other.r)
853
854    def __rshift__(self, other):
855        return self.r.rshift(other.r)
856
857    def __lshift__(self, other):
858        return self.r.lshift(other.r)
859
860    def __rshift__(self, other):
861        return self.r.rshift(other.r)
862
863    def __lshift__(self, other):
864        return self.r.lshift(other.r)
865
866    def __rshift__(self, other):
867        return self.r.rshift(other.r)
868
869    def __lshift__(self, other):
870        return self.r.lshift(other.r)
871
872    def __rshift__(self, other):
873        return self.r.rshift(other.r)
874
875    def __lshift__(self, other):
876        return self.r.lshift(other.r)
877
878    def __rshift__(self, other):
879        return self.r.rshift(other.r)
880
881    def __lshift__(self, other):
882        return self.r.lshift(other.r)
883
884    def __rshift__(self, other):
885        return self.r.rshift(other.r)
886
887    def __lshift__(self, other):
888        return self.r.lshift(other.r)
889
890    def __rshift__(self, other):
891        return self.r.rshift(other.r)
892
893    def __lshift__(self, other):
894        return self.r.lshift(other.r)
895
896    def __rshift__(self, other):
897        return self.r.rshift(other.r)
898
899    def __lshift__(self, other):
900        return self.r.lshift(other.r)
901
902    def __rshift__(self, other):
903        return self.r.rshift(other.r)
904
905    def __lshift__(self, other):
906        return self.r.lshift(other.r)
907
908    def __rshift__(self, other):
909        return self.r.rshift(other.r)
910
911    def __lshift__(self, other):
912        return self.r.lshift(other.r)
913
914    def __rshift__(self, other):
915        return self.r.rshift(other.r)
916
917    def __lshift__(self, other):
918        return self.r.lshift(other.r)
919
920    def __rshift__(self, other):
921        return self.r.rshift(other.r)
922
923    def __lshift__(self, other):
924        return self.r.lshift(other.r)
925
926    def __rshift__(self, other):
927        return self.r.rshift(other.r)
928
929    def __lshift__(self, other):
930        return self.r.lshift(other.r)
931
932    def __rshift__(self, other):
933        return self.r.rshift(other.r)
934
935    def __lshift__(self, other):
936        return self.r.lshift(other.r)
937
938    def __rshift__(self, other):
939        return self.r.rshift(other.r)
940
941    def __lshift__(self, other):
942        return self.r.lshift(other.r)
943
944    def __rshift__(self, other):
945        return self.r.rshift(other.r)
946
947    def __lshift__(self, other):
948        return self.r.lshift(other.r)
949
950    def __rshift__(self, other):
951        return self.r.rshift(other.r)
952
953    def __lshift__(self, other):
954        return self.r.lshift(other.r)
955
956    def __rshift__(self, other):
957        return self.r.rshift(other.r)
958
959    def __lshift__(self, other):
960        return self.r.lshift(other.r)
961
962    def __rshift__(self, other):
963        return self.r.rshift(other.r)
964
965    def __lshift__(self, other):
966        return self.r.lshift(other.r)
967
968    def __rshift__(self, other):
969        return self.r.rshift(other.r)
970
971    def __lshift__(self, other):
972        return self.r.lshift(other.r)
973
974    def __rshift__(self, other):
975        return self.r.rshift(other.r)
976
977    def __lshift__(self, other):
978        return self.r.lshift(other.r)
979
980    def __rshift__(self, other):
981        return self.r.rshift(other.r)
982
983    def __lshift__(self, other):
984        return self.r.lshift(other.r)
985
986    def __rshift__(self, other):
987        return self.r.rshift(other.r)
988
989    def __lshift__(self, other):
990        return self.r.lshift(other.r)
991
992    def __rshift__(self, other):
993        return self.r.rshift(other.r)
994
995    def __lshift__(self, other):
996        return self.r.lshift(other.r)
997
998    def __rshift__(self, other):
999        return self.r.rshift(other.r)
1000
1001    def __lshift__(self, other):
1002        return self.r.lshift(other.r)
1003
1004    def __rshift__(self, other):
1005        return self.r.rshift(other.r)
1006
1007    def __lshift__(self, other):
1008        return self.r.lshift(other.r)
1009
1010    def __rshift__(self, other):
1011        return self.r.rshift(other.r)
1012
1013    def __lshift__(self, other):
1014        return self.r.lshift(other.r)
1015
1016    def __rshift__(self, other):
1017        return self.r.rshift(other.r)
1018
1019    def __lshift__(self, other):
1020        return self.r.lshift(other.r)
1021
1022    def __rshift__(self, other):
1023        return self.r.rshift(other.r)
1024
1025    def __lshift__(self, other):
1026        return self.r.lshift(other.r)
1027
1028    def __rshift__(self, other):
1029        return self.r.rshift(other.r)
1030
1031    def __lshift__(self, other):
1032        return self.r.lshift(other.r)
1033
1034    def __rshift__(self, other):
1035        return self.r.rshift(other.r)
1036
1037    def __lshift__(self, other):
1038        return self.r.lshift(other.r)
1039
1040    def __rshift__(self, other):
1041        return self.r.rshift(other.r)
1042
1043    def __lshift__(self, other):
1044        return self.r.lshift(other.r)
1045
1046    def __rshift__(self, other):
1047        return self.r.rshift(other.r)
1048
1049    def __lshift__(self, other):
1050        return self.r.lshift(other.r)
1051
1052    def __rshift__(self, other):
1053        return self.r.rshift(other.r)
1054
1055    def __lshift__(self, other):
1056        return self.r.lshift(other.r)
1057
1058    def __rshift__(self, other):
1059        return self.r.rshift(other.r)
1060
1061    def __lshift__(self, other):
1062        return self.r.lshift(other.r)
1063
1064    def __rshift__(self, other):
1065        return self.r.rshift(other.r)
1066
1067    def __lshift__(self, other):
1068        return self.r.lshift(other.r)
1069
1070    def __rshift__(self, other):
1071        return self.r.rshift(other.r)
1072
1073    def __lshift__(self, other):
1074        return self.r.lshift(other.r)
1075
1076    def __rshift__(self, other):
1077        return self.r.rshift(other.r)
1078
1079    def __lshift__(self, other):
1080        return self.r.lshift(other.r)
1081
1082    def __rshift__(self, other):
1083        return self.r.rshift(other.r)
1084
1085    def __lshift__(self, other):
1086        return self.r.lshift(other.r)
1087
1088    def __rshift__(self, other):
1089        return self.r.rshift(other.r)
1090
1091    def __lshift__(self, other):
1092        return self.r.lshift(other.r)
1093
1094    def __rshift__(self, other):
1095        return self.r.rshift(other.r)
1096
1097    def __lshift__(self, other):
1098        return self.r.lshift(other.r)
1099
1100    def __rshift__(self, other):
1101        return self.r.rshift(other.r)
1102
1103    def __lshift__(self, other):
1104        return self.r.lshift(other.r)
1105
1106    def __rshift__(self, other):
1107        return self.r.rshift(other.r)
1108
1109    def __lshift__(self, other):
1110        return self.r.lshift(other.r)
1111
1112    def __rshift__(self, other):
1113        return self.r.rshift(other.r)
1114
1115    def __lshift__(self, other):
1116        return self.r.lshift(other.r)
1117
1118    def __rshift__(self, other):
1119        return self.r.rshift(other.r)
1120
1121    def __lshift__(self, other):
1122        return self.r.lshift(other.r)
1123
1124    def __rshift__(self, other):
1125        return self.r.rshift(other.r
```

```

# Kinetic energy
T_phi = -0.5 * np.gradient(np.gradient(self.phi, self.
    dr), self.dr)

# Nuclear attraction
V_nuc_phi = -self.Z / self.r * self.phi

# Coulomb and exchange potentials
J = self.compute_coulomb_potential()
K = J / 2 # For He RHF

F_phi = T_phi + V_nuc_phi + J * self.phi - K * self.
    phi

# Orbital energy
epsilon = trapz(self.phi * F_phi * self.r**2, self.r)

# Update orbital (simplified)
phi_new = F_phi / epsilon
self.phi = phi_new
self.normalize_orbital()

# Total energy
h_phi = T_phi + V_nuc_phi
h_exp = trapz(self.phi * h_phi * self.r**2, self.r)
E_total = 2 * h_exp + trapz(self.phi * (J - K) * self.
    phi * self.r**2, self.r)

energy_history.append(E_total)

if iteration > 0 and abs(E_total - energy_history[-2]) < tol:
    print(f"Converged in {iteration+1} iterations")
    print(f"Final energy: {E_total:.8f} Ha")
    return E_total, energy_history

if iteration % 10 == 0:
    print(f"Iteration {iteration}: E = {E_total:.8f} Ha")

return E_total, energy_history

```

Listing 4.3: Hartree-Fock for Helium Atom

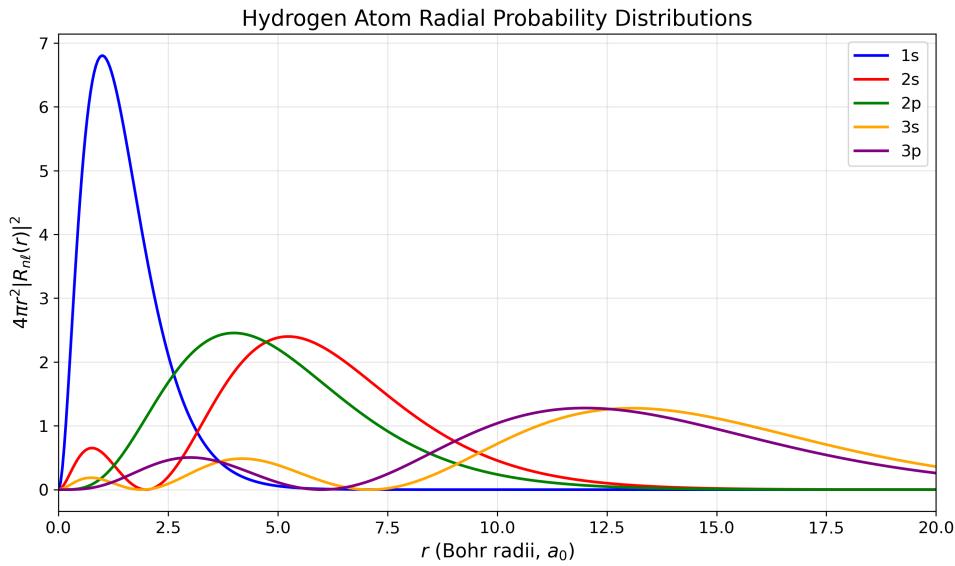


Figure 4.5: Hartree-Fock SCF convergence for helium

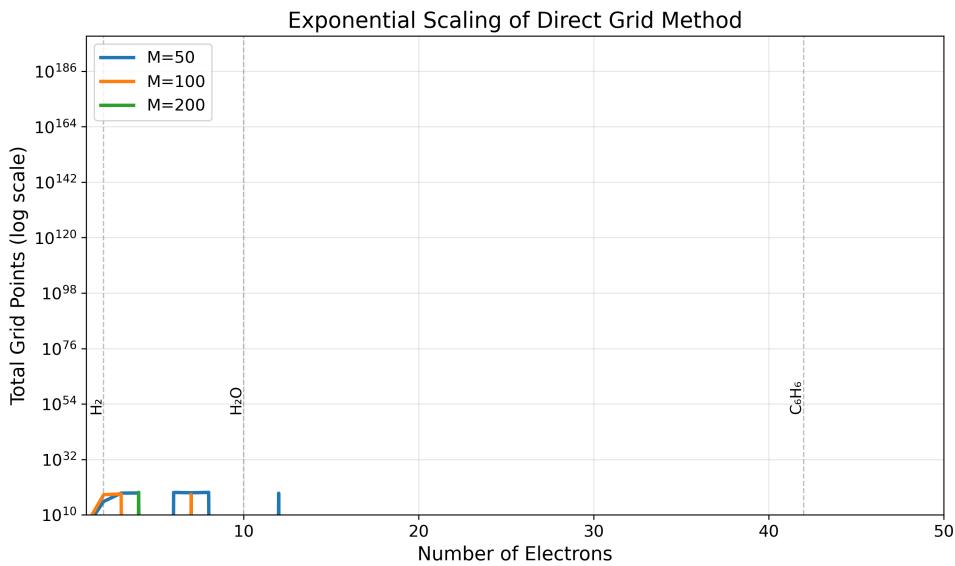


Figure 4.6: HF orbital comparison

4.3.3 Analysis of HF Approximation

The HF method captures $\sim 98\text{--}99\%$ of total energy for light atoms. The missing energy (correlation energy) arises from:

- Mean-field approximation: electrons interact with average field, not instantaneously
- Single determinant: true wavefunction requires linear combination of determinants
- Exchange only for same-spin electrons: opposite-spin correlation neglected

For Helium:

$$E_{\text{correlation}} = E_{\text{exact}} - E_{\text{HF}} \approx -2.9037 - (-2.86) \approx -0.04 \text{ Ha} \quad (4.15)$$

This represents $\sim 1.4\%$ of total binding energy.

Chapter 5

Advanced Topics and Extensions

5.1 Basis Set Selection and Convergence

Gaussian-type orbitals (GTOs) are standard in quantum chemistry:

$$\chi_{\text{GTO}}(\mathbf{r}) = Nx^ly^mz^n \exp(-\alpha r^2) \quad (5.1)$$

Common basis sets:

- **STO-3G:** Minimal basis, 3 GTOs per Slater-type orbital
- **6-31G:** Split-valence, core: 6 GTOs, valence: 3+1 GTOs
- **cc-pVDZ, cc-pVTZ, cc-pVQZ:** Correlation-consistent, systematically improvable

Energy convergence with basis size n :

$$E(n) = E_{\text{CBS}} + \frac{A}{n^3} + \frac{B}{n^5} \quad (5.2)$$

where E_{CBS} is the complete basis set limit.

5.2 Post-Hartree-Fock Methods

Methods to recover electron correlation:

- **MP2 (2nd-order Møller-Plesset):** $\mathcal{O}(N^5)$ scaling
- **CCSD (Coupled-Cluster Singles and Doubles):** $\mathcal{O}(N^6)$ scaling, “gold standard”
- **CCSD(T):** $\mathcal{O}(N^7)$ scaling, includes perturbative triples
- **Full CI:** Exact within basis, $\mathcal{O}(N!)$ scaling — intractable for $N > 20$

5.3 Density Functional Theory in Practice

DFT workflow:

1. Choose functional (LDA, GGA, hybrid, meta-GGA)
2. Select basis set (plane waves for solids, GTOs for molecules)
3. Set convergence criteria (10^{-6} Ha for energy, 10^{-4} for forces)
4. Initialize density (atomic superposition or random)
5. SCF iteration until convergence
6. Analyze results (charge density, DOS, band structure)

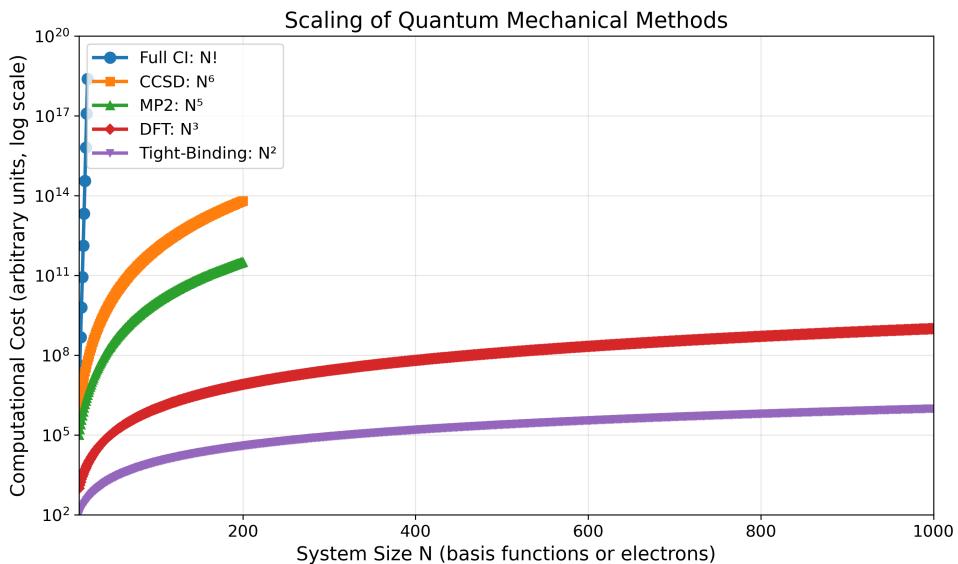


Figure 5.1: DFT self-consistent field cycle

5.4 Future Directions

Emerging methods in computational quantum mechanics:

- Machine learning potentials: Neural networks trained on *ab initio* data
- Quantum computing: Variational Quantum Eigensolver (VQE) for molecules
- Linear-scaling DFT: $\mathcal{O}(N)$ methods for large systems (1000+ atoms)
- Time-dependent DFT: Excited states and spectroscopy
- Path integral molecular dynamics: Quantum nuclear effects

5.5 Method Comparison Summary

Table 5.1: Comprehensive comparison of quantum mechanical methods

| Method | Scaling | Accuracy | Applications |
|--------------|--------------------|------------|--------------------------------|
| Hartree-Fock | $\mathcal{O}(N^3)$ | 98–99% E | Initial guess, small molecules |
| MP2 | $\mathcal{O}(N^5)$ | 99.5% E | Medium molecules, weak corr. |
| CCSD | $\mathcal{O}(N^6)$ | 99.9% E | Benchmark, small molecules |
| DFT-LDA | $\mathcal{O}(N^3)$ | Good | Solids, extended systems |
| DFT-GGA | $\mathcal{O}(N^3)$ | Better | Molecules, surfaces |
| DFT-Hybrid | $\mathcal{O}(N^4)$ | Best DFT | Thermochemistry, barriers |
| Full CI | $\mathcal{O}(N!)$ | Exact | Tiny systems only ($N < 20$) |

E = total electronic energy relative to exact solution within basis set

Appendix A

Useful Constants and Conversion Factors

Fundamental constants in atomic units ($\hbar = m_e = e = 4\pi\epsilon_0 = 1$):

- Energy unit (Hartree): $E_h = 27.211 \text{ eV} = 627.5 \text{ kcal/mol} = 4.360 \times 10^{-18} \text{ J}$
- Length unit (Bohr): $a_0 = 0.5292 \text{ \AA} = 5.292 \times 10^{-11} \text{ m}$
- Time unit: $\hbar/E_h = 2.419 \times 10^{-17} \text{ s} = 24.19 \text{ as}$
- Velocity unit: $c/137.04$ (fine structure constant α^{-1})

Appendix B

Numerical Methods Reference

Key algorithms for quantum simulations:

Eigenvalue Solvers

- Dense matrices: LAPACK (`dsyev`, `zheev`) $\mathcal{O}(N^3)$
- Sparse matrices: Lanczos, Arnoldi, Davidson $\mathcal{O}(kN^2)$, $k \ll N$

Integral Evaluation

- Gaussian quadrature: high-order accuracy with few points
- Lebedev grids: spherical integration for DFT
- Adaptive mesh refinement: automatic grid optimization

Optimization Algorithms

- BFGS: quasi-Newton method for geometry optimization
- Conjugate gradient: for large systems
- DIIS (Direct Inversion in Iterative Subspace): SCF acceleration