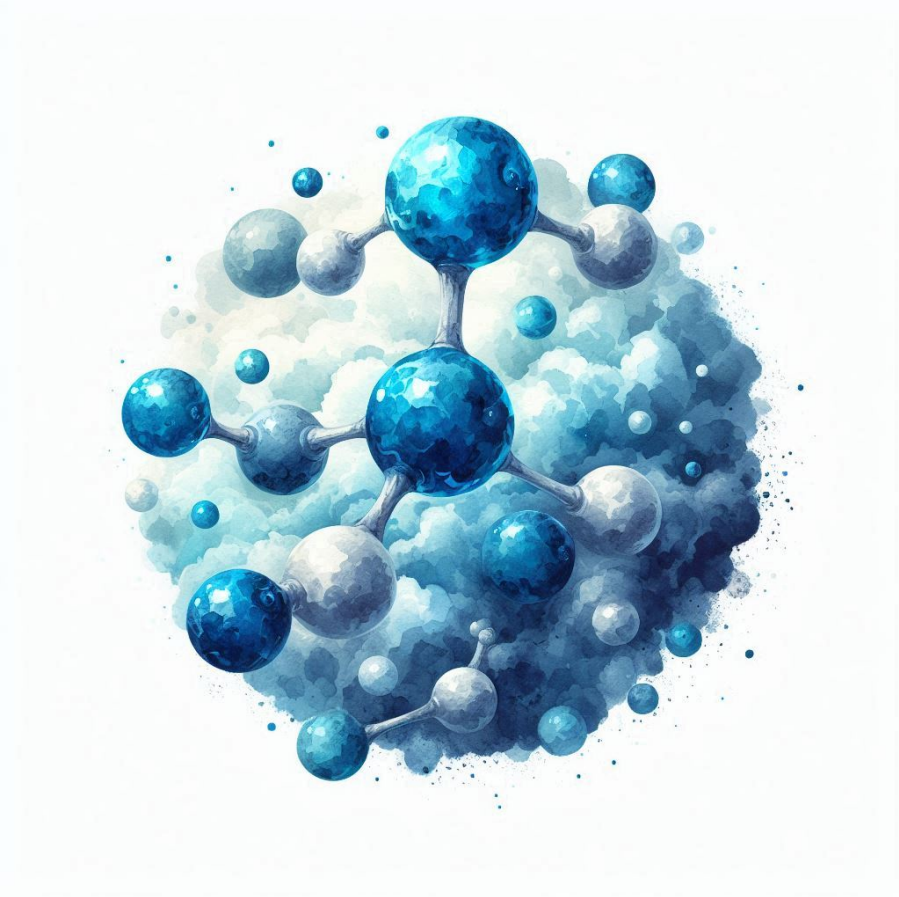
Quantum Mechanics in Chemistry: Hands on



Jiří Janoš, Tomáš Jíra

Created in LaTeX

Version from October 9, 2024

Contents

Introduction

The quantum mechanics is gradually becoming one of the pillars of chemistry, penetrating in all its branches. University libraries are full of textbooks explaining basic phenomena in quantum mechanics, introducing simple examples with analytic solutions such as harmonic potential, rectangular well and others. It fair to say that all students of quantum chemistry have seen Hartree–Fock equations or the time-dependent Schrödinger equation. Many of them have also solved these equation or are doing so routinely using quantum chemistry codes available to the community. However, only a small fraction of the current students has ever implemented any of these equations or even know how to proceed with their implementation. The authors were typical examples of such students after finishing their Masters degrees.

We believe that the reason for that is a abundance of well-optimized and ready-to-use codes containing standard methods of quantum chemistry. There is, indeed, no need to reinvent the wheel. Nevertheless, it is worth to look at the equations and their implementation at least one since it provides valuable insights and forces one to truly understand what they are doing. Therefore, we prepared this short *Quantum Mechanics: Hands on* material which introduces the main techniques of quantum chemistry and guide students to their own simple implementation. The text contains prepared exercises and their solutions in a form of Python scripts suitable even for programming beginners.

The text is split into two parts. The *Electronic structure theory* part will deal with the basic methods for solving electronic structure of atoms and molecules such as the Hartree–Fock method, MP2 method, configuration interaction and Hückel method. The *Time-dependent quantum mechanics* part will delve into numerical methods for solving the time-dependent Schrödinger equation in real and imaginary time, Bohmian dynamics and others.

Brief introduction to Python

```
import numpy as np
```

Calling functions from a Python library, e.g. number π, is done as follows:

```
pi = np.pi
```

of if we want to take absolute value of a number

```
number = np.abs(-1)
```

Part I.

Electronic structure theory

1. Hartree–Fock Method

The Hartree–Fock (HF) method stands as a cornerstone in quantum chemistry, offering a systematic approach to solve the electronic structure problem in molecules. This computational technique strives to determine the optimal wave function for a given molecular system, providing insights into the distribution of electrons and their energies.

1.1. Theoretical Background

Ultimately, we are interested in solving the Schrödinger equation in the form

$$\hat{\mathbf{H}}\,|\Psi\rangle = E\,|\Psi\rangle\,, \tag{1.1}$$

where $\hat{H}$ is the molecular Hamiltonian operator, $|\Psi\rangle$ is the molecular wave function, and E is the total energy of the system. The HF method aims to approximate the wave function $|\Psi\rangle$ by a single Slater determinant, which we will write in the form

$$|\Psi\rangle = |\chi_1\chi_2\cdots\chi_N\rangle\,, \tag{1.2}$$

where χ_i represents a Molecular Spinorbital (MS) and N is the total number of electrons. The HF method seeks to optimize these molecular orbitals to minimize the total energy of the system, providing a reliable estimate of the electronic structure. However, in the Restricted Hartree–Fock (RHF) method, we impose a constraint on the electron spin, and instead of spin orbitals, we work with spatial orbitals, which allows us to write the slater determinant in terms of spatial orbitals in the form

$$|\Psi\rangle = |\Phi_1\Phi_2\cdots\Phi_{N/2}\rangle\,, \tag{1.3}$$

where Φ_i represents a molecular spatial orbital. We can see, that for the RHF method, we need to have an even number of electrons in the system. In practical calculations, it is convenient to expand the molecular orbitals (spin or spatial) in terms of basis functions ϕ (usually sums of Gaussian functions) and work with the expansion coefficients. If we assume that the wavefunction is a single Slater determinant, our molecular orbitals are expanded in terms of basis functions and optimize the energy of such determinant, we arrive at the Roothaan equations in the form

$$\mathbf{FC} = \mathbf{SC}\square, \tag{1.4}$$

where **F** is the Fock matrix (defined later), **C** is a matrix of orbital coefficients, **S** is the overlap matrix (also defined later), and □ represents orbital energies.

1.2. Implementation of the Restricted Hartree–Fock Method

Let's begin by defining the core Hamiltonian, also known as the one-electron Hamiltonian. The core Hamiltonian represents a part of the full Hamiltonian that excludes electron-electron repulsion. In index notation, it is expressed as

$$H_{\mu\nu}^{core} = T_{\mu\nu} + V_{\mu\nu} \tag{1.5}$$

where μ and ν are indices of basis functions, $T_{\mu\nu}$ is a kinetic energy matrix element and $V_{\mu\nu}$ is a potential energy matrix element. These matrix elements are given as

$$T_{\mu\nu} = \langle \phi_\mu | \hat{T} | \phi_\nu \rangle \tag{1.6}$$
$$V_{\mu\nu} = \langle \phi_\mu | \hat{V} | \phi_\nu \rangle \tag{1.7}$$

are usually calculated using analytical expressions. Additionally, using analytical expressions, we can calculate the overlap integrals

$$S_{\mu\nu} = \langle \phi_\mu | \phi_\nu \rangle \tag{1.8}$$

and the two-electron Coulomb repulsion integrals

$$J_{\mu\nu\kappa\lambda} = \langle \phi_\mu \phi_\mu | \hat{J} | \phi_\kappa \phi_\lambda \rangle, \tag{1.9}$$

which play crucial roles in the HF calculation.[1] The HF method revolves around solving the Roothaan equations 1.4 iteratively. The Fock matrix is defined as

$$F_{\mu\nu} = H_{\mu\nu}^{core} + D_{\kappa\lambda}(J_{\mu\nu\kappa\lambda} - \frac{1}{2}J_{\mu\lambda\kappa\nu}) \tag{1.10}$$

depends on the unknown density matrix **D**. This iterative process is carried out through a Self-Consistent Field (SCF) method. In each iteration, a guess for the density matrix is made, and the Roothaan equations are solved. The density matrix is then updated as

$$D_{\mu\nu} = 2C_{\mu i}C_{\nu i} \tag{1.11}$$

and the total energy of the system

$$E = \frac{1}{2}D_{\mu\nu}(H_{\mu\nu}^{core} + F_{\mu\nu}) + E_{nuc} \tag{1.12}$$

is then calculated using the core Hamiltonian and the Fock matrix. The important thing to note is that the Fock matrix depends on the density matrix, which is updated in each iteration. The initial guess for the density matrix is often set to zero. After the density matrix and the total energy are converged, the SCF procedure is terminated, and the optimized molecular orbitals are obtained. To get the final energy of the system, we need to add the nuclear repulsion energy of the form

$$E_{nuc} = \sum_A \sum_{B<A} \frac{Z_A Z_B}{R_{AB}}, \tag{1.13}$$

where Z_A is the nuclear charge of atom A, and R_{AB} is the distance between atoms A and B.

1.2.1. Gradient of the Restricted Hartree–Fock Method

If we perform the calculation as described above and get the density matrix **D** we can evaluate the nuclear energy gradient as

$$\frac{\partial E}{\partial X_{A,i}} = D_{\mu\nu}\frac{\partial H^{core}_{\mu\nu}}{\partial X_{A,i}} + 2D_{\mu\nu}D_{\kappa\lambda}\frac{\partial J_{\mu\nu\kappa\lambda}}{\partial X_{A,i}} - 2W_{\mu\nu}\frac{\partial S_{\mu\nu}}{\partial X_{A,i}} \tag{1.14}$$

where i is the index of the coordinate and where **W** is energy weighed density matrix defined as

$$W_{\mu\nu} = 2C_{\mu i}C_{\nu i}\varepsilon_i \tag{1.15}$$

1.3. Integral Transforms to the Basis of Molecular Spinorbitals

To perform most of the post-Hartree–Fock (post-HF) calculations, we usually need to transform the integrals to the MS basis. We will describe it here and refer to it in the post-HF methods sections. We will also present the post-HF methods using the integrals in the MS basis (and its antisymmetrized form in case of the Coulomb integrals), since it is more general.

All the integrals defined in the equations 1.5, 1.8, and 1.9 and even the Fock matrix in the equation 1.10 are defined in the basis of atomic orbitals. To transform these integrals to the MS basis, we need to use the coefficient matrix **C** obtained from the solution of the Roothaan equations 1.4. The coefficient matrix **C**, which is obtained from the RHF calculation, is calculated in the spatial molecular orbital basis. The first step is to expand the coefficient matrix **C** to the MS basis. This can be done mathematically using the tiling matrix $\mathbf{P}_{n\times 2n}$, defined as

$$\mathbf{P} = \begin{pmatrix} e_1 & e_1 & e_2 & e_2 & \dots & e_n & e_n \end{pmatrix}, \tag{1.16}$$

where e_i represents the i-th column of the identity matrix $\mathbf{I}_n$ and the matrices $\mathbf{M}_{n\times 2n}$ and $\mathbf{N}_{n\times 2n}$ with elements given by

$$M_{ij} = 1 - j \bmod 2, N_{ij} = j \bmod 2. \tag{1.17}$$

The coefficient matrix **C** in the MS basis can be then expressed as

$$\mathbf{C}^{MS} = \begin{pmatrix} \mathbf{CP} \\ \mathbf{CP} \end{pmatrix} \odot \begin{pmatrix} \mathbf{M} \\ \mathbf{N} \end{pmatrix}, \tag{1.18}$$

where $\odot$ denotes the Hadamard product. This transformed matrix $\mathbf{C}^{MS}$ is then used to transform the Coulomb integrals **J** to the MS basis as

$$J^{MS}_{pqrs} = C^{MS}_{\mu p}C^{MS}_{\nu q}(\mathbf{I}_2 \otimes_K (\mathbf{I}_2 \otimes_K \mathbf{J})^{(4,3,2,1)})_{\mu\nu\kappa\lambda}C^{MS}_{\kappa r}C^{MS}_{\lambda s}, \tag{1.19}$$

where the superscript $(4,3,2,1)$ denotes the axes transposition and $\otimes_K$ is the Kronecker product. This notation accounts for the spin modifications and ensures that the transformations adhere to quantum mechanical principles. We also define the antisymmetrized Coulomb integrals in physicists' notation as

$$\langle pq||rs\rangle = (J^{MS}_{pqrs} - J^{MS}_{psrq})^{(1,3,2,4)}. \tag{1.20}$$

For the transformation of the one-electron integrals such as the core Hamiltonian, the overlap matrix and also the Fock matrix, we use the formula

$$A_{pq}^{MS} = C_{\mu p}^{MS}(\mathbf{I}_2 \otimes_K \mathbf{A})_{\mu\nu} C_{\nu q}^{MS}, \tag{1.21}$$

where $\mathbf{A}$ is an arbitrary one-electron integral. Since a lot of the post-HF methods also use differences of orbital energies in the denominator, it is practical to define the tensors

$$\Delta\varepsilon_i^a = \frac{1}{\varepsilon_i - \varepsilon_a} \tag{1.22}$$

$$\Delta\varepsilon_{ij}^{ab} = \frac{1}{\varepsilon_i + \varepsilon_j - \varepsilon_a - \varepsilon_b} \tag{1.23}$$

$$\Delta\varepsilon_{ijk}^{abc} = \frac{1}{\varepsilon_i + \varepsilon_j + \varepsilon_k - \varepsilon_a - \varepsilon_b - \varepsilon_c}, \tag{1.24}$$

where a, b, c are virtual orbitals and i, j, k are occupied orbitals. These tensors make the code more readable, easier to understand and also more efficient.

1.4. Hartree–Fock Method and Integral Transform Code Example

This section provides code snippets for the HF method and the integral transforms to the MS basis. The code snippets are written in Python and use the NumPy package for numerical calculations. The exercises are designed to help you understand the implementation of the HF method and the transformation of integrals to the MS basis. The solutions are provided to guide you through the implementation process and ensure that you can verify your results.

1.4.1. Exercise

The exercise codes are designed to be self-contained and can be run in any Python environment. They contain placeholders that you need to fill in to complete the implementation. The exercises assume that you have defined the `atoms`, `coords`, `S`, `H`, and `J` variables, which represent the list of atomic numbers, atomic coordinates, overlap matrix, core Hamiltonian, and Coulomb integral tensor, respectively. These variables can be generaly obtained from a .xyz molecule file and the output of a quantum chemistry software. If you want to just get to the coding part, you can save the molecule.xyz, S_AO.mat, H_AO.mat, and J_AO.mat files to the same directory as the exercise codes and and load the variables using the Listing below. The `ATOM` variable is a dictionary that maps the atomic symbols to atomic numbers.

```
# get the atomic numbers and coordinates of all atoms
atoms = np.array([ATOM[line.split()[0]] for line in open("molecule.xyz").readlines()[2:]], dtype=
    int)
coords = np.array([line.split()[1:] for line in open("molecule.xyz").readlines()[2:]], dtype=float)

# convert to bohrs
coords *= 1.8897261254578281

# load the integrals from the files
H, S = np.loadtxt("H_AO.mat", skiprows=1), np.loadtxt("S_AO.mat", skiprows=1); J = np.loadtxt("J_AO
    .mat", skiprows=1).reshape(4 * [S.shape[1]])
```

With all the variables defined, you can proceed to the HF exercise in the Listing 1.1 below.

```
"""
Here are defined some of the necessary variables. The variable "E_HF" stores the Hartree-Fock
    energy, while "E_HF_P" keeps track of the previous iteration's energy to monitor convergence.
    The "thresh" defines the convergence criteria for the calculation. The variables "nocc" and "
    nbf" represent the number of occupied orbitals and the number of basis functions, respectively.
     Initially, "E_HF" is set to zero and "E_HF_P" to one to trigger the start of the Self-
    Consistent Field (SCF) loop. Although you can rename these variables, it is important to note
    that certain sections of the code are tailored to these specific names.
"""
E_HF, E_HF_P, nocc, nbf, thresh = 0, 1, sum(atoms) // 2, S.shape[0], 1e-8

"""
These lines set up key components for our HF calculations. We initialize the density matrix as a
    zero matrix, and the coefficients start as an empty array. Although the coefficient matrix is
    computed within the while loop, it's defined outside to allow for its use in subsequent
    calculations, such as the MP energy computation. Similarly, the exchange tensor is accurately
    calculated here by transposing the Coulomb tensor. The "eps" vector, which contains the orbital
     energies, is also defined at this stage to facilitate access throughout the script. This setup
     ensures that all necessary variables are ready for iterative processing and further
    calculations beyond the SCF loop.
"""
K, F, D, C, eps = J.transpose(0, 3, 2, 1), np.zeros_like(S), np.zeros_like(S), np.zeros_like(S), np
    .array(nbf * [0])

"""
This while loop is the SCF loop. Please fill it so it calculates the Fock matrix, solves the Fock
    equations, builds the density matrix from the coefficients and calculates the energy. You can
    use all the variables defined above and all the functions in numpy package. The recommended
    functions are np.einsum and np.linalg.eigh. Part of the calculation will probably be
    calculation of the inverse square root of a matrix. The numpy package does not conatin a
    function for this. You can find a library that can do that or you can do it manually. The
    manual calculation is, of course, preferred.
"""
while abs(E_HF - E_HF_P) > thresh:
    break

"""
In the followng block of code, please calculate the nuclear-nuclear repulsion energy. You should
    use only the atoms and coords variables. The code can be as short as two lines. The result
    should be stored in the "VNN" variable.
"""
VNN = 0

# print the results
print("RHF ENERGY: {:.8f}".format(E_HF + VNN))
```

Listing 1.1: HF method exercise code.

Now is a good time to check your results with the provided solution. If you are satisfied with the results, you can proceed to the next exercise in the Listing 1.2 below, which involves transforming the integrals to the molecular spinorbital basis.

```
"""
To perform most of the post-HF calculations, we need to transform the Coulomb integrals to the
    molecular spinorbital basis, so if you don't plan to calculate any post-HF methods, you can end
     the eercise here. The restricted MP2 calculation could be done using the Coulomb integral in
    MO basis, but for the sake of subsequent calculations, we enforce here the integrals in the MS
    basis. The first thing you will need for the transform is the coefficient matrix in the
    molecular spinorbital basis. To perform this transform using the mathematical formulation
    presented in the materials, the first step is to form the tiling matrix "P" which will be used
    to duplicate columns of a general matrix. Please define it here.
"""
P = np.zeros((nbf, 2 * nbf))

```

```
"""
Now, please define the spin masks "M" and "N". These masks will be used to zero out spinorbitals, that should be empty.
"""
M, N = np.zeros((nbf, 2 * nbf)), np.zeros((nbf, 2 * nbf))

"""
With the tiling matrix and spin masks defined, please transform the coefficient matrix into the molecular spinorbital basis. The resulting matrix should be stored in the "Cms" variable.
"""
Cms = np.zeros(2 * np.array(C.shape))

"""
For some of the post-HF calculations, we will also need the Hamiltonian and Fock matrix in the molecular spinorbital basis. Please transform it and store it in the "Hms" and "Fms" variable. If you don't plan to calculate the CCSD method, you can skip the transformation of the Fock matrix, as it is not needed for the MP2 and CI calculations.
"""
Hms, Fms = np.zeros(2 * np.array(H.shape)), np.zeros(2 * np.array(H.shape))

"""
With the coefficient matrix in the molecular spinorbital basis available, we can proceed to transform the Coulomb integrals. It is important to note that the transformed integrals will contain twice as many elements along each axis compared to their counterparts in the atomic orbital (AO) basis. This increase is due to the representation of both spin states in the molecular spinorbital basis.
"""
Jms = np.zeros(2 * np.array(J.shape))

"""
The post-HF calculations also require the antisymmetrized two-electron integrals in the molecular spinorbital basis. These integrals are essential for the MP2 and CC calculations. Please define the "Jmsa" tensor as the antisymmetrized two-electron integrals in the molecular spinorbital basis.
"""
Jmsa = np.zeros(2 * np.array(J.shape))

"""
As mentioned in the materials, it is also practical to define the tensors of reciprocal orbital energy differences in the molecular spinorbital basis. These tensors are essential for the MP2 and CC calculations. Please define the "Emss", "Emsd" and "Emst" tensors as tensors of single, double and triple excitation energies, respectively. The configuration interaction will not need these tensors, so you can skip this step if you don't plan to program the CI method. The MP methods will require only the "Emsd" tensor, while the CC method will need both tensors.
"""
Emss, Emsd = np.array([]), np.array([])
```

Listing 1.2: Integral transform exercise code.

If you successfully completed the exercise, you can compare your results with the provided solution. If you are satisfied with the results, you are now set for the post-HF methods exercises.

1.4.2. Solution

The solutions provided below are complete implementations of the HF method (Listing 1.3) and the integral transforms (Listing 1.4) to the MS basis. The solutions are written in Python and use the NumPy package for numerical calculations. The solutions are designed to guide you through the implementation process and ensure that you can verify your results.

```
# define energies, number of occupied and virtual orbitals and the number of basis functions
```

```
E_HF, E_HF_P, VNN, nocc, nvirt, nbf = 0, 1, 0, sum(atoms) // 2, S.shape[0] - sum(atoms) // 2, S.shape[0]

# define some matrices and tensors
K, F, D, C, eps = J.transpose(0, 3, 2, 1), np.zeros_like(S), np.zeros_like(S), np.zeros_like(S), np.array(nbf * [0])

# calculate the X matrix which is the inverse of the square root of the overlap matrix
SEP = np.linalg.eigh(S); X = SEP[1] @ np.diag(1 / np.sqrt(SEP[0])) @ SEP[1].T

# the scf loop
while abs(E_HF - E_HF_P) > args.threshold:

    # build the Fock matrix
    F = H + np.einsum("ijkl,ij->kl", J - 0.5 * K, D, optimize=True)

    # solve the Fock equations
    eps, C = np.linalg.eigh(X @ F @ X); C = X @ C

    # build the density from coefficients
    D = 2 * np.einsum("ij,kj->ik", C[:, :nocc], C[:, :nocc])

    # save the previous energy and calculate the current electron energy
    E_HF_P, E_HF = E_HF, 0.5 * np.einsum("ij,ij", D, H + F, optimize=True)

# calculate nuclear-nuclear repulsion
for i, j in it.product(range(len(atoms)), range(len(atoms))):
    VNN += 0.5 * atoms[i] * atoms[j] / np.linalg.norm(coords[i, :] - coords[j, :]) if i != j else 0

# print the results
print("    RHF ENERGY: {:.8f}".format(E_HF + VNN))
```

Listing 1.3: HF method exercise code solution.

```
# define the occ and virt spinorbital slices shorthand
o, v = slice(0, 2 * nocc), slice(2 * nocc, 2 * nbf)

# define the tiling matrix for the Jmsa coefficients and energy placeholders
P = np.array([np.eye(nbf)[:, i // 2] for i in range(2 * nbf)]).T

# define the spin masks
M = np.repeat([1 - np.arange(2 * nbf, dtype=int) % 2], nbf, axis=0)
N = np.repeat([    np.arange(2 * nbf, dtype=int) % 2], nbf, axis=0)

# tile the coefficient matrix, apply the spin mask and tile the orbital energies
Cms, epsms = np.block([[C @ P], [C @ P]]) * np.block([[M], [N]]), np.repeat(eps, 2)

# transform the core Hamiltonian and Fock matrix to the molecular spinorbital basis
Hms = np.einsum("ip,ij,jq->pq", Cms, np.kron(np.eye(2), H), Cms, optimize=True)
Fms = np.einsum("ip,ij,jq->pq", Cms, np.kron(np.eye(2), F), Cms, optimize=True)

# transform the coulomb integrals to the MS basis in chemist's notation
Jms = np.einsum("ip,jq,ijkl,kr,ls->pqrs", Cms, Cms, np.kron(np.eye(2), np.kron(np.eye(2), J).T), Cms, Cms, optimize=True);

# antisymmetrized two-electron integrals in physicist's notation
Jmsa = (Jms - Jms.swapaxes(1, 3)).transpose(0, 2, 1, 3)

# create the tensors of reciprocal differences of orbital energies in MS basis used in post-HF methods
Emss, Emsd = 1 / (epsms[o].reshape(-1) - epsms[v].reshape(-1, 1)), 1 / (epsms[o].reshape(-1) + epsms[o].reshape(-1, 1) - epsms[v].reshape(-1, 1, 1) - epsms[v].reshape(-1, 1, 1, 1))
```

Listing 1.4: Integral transform exercise code solution.

2. Møller–Plesset Perturbation Theory

Møller–Plesset Perturbation Theory (MPPT) is a quantum mechanical method used to improve the accuracy of electronic structure calculations within the framework of HF theory. It involves treating electron-electron correlation effects as a perturbation to the reference HF wave function. The method is named after its developers, physicists C. Møller and M. S. Plesset. By systematically including higher-order corrections, MPPT provides more accurate predictions of molecular properties compared to the initial HF approximation.

2.1. Theory of the Perturbative Approach

As for the HF method, we start with the Schrödinger equation in the form

$$\hat{\mathbf{H}} |\Psi\rangle = E |\Psi\rangle , \tag{2.1}$$

where $\hat{\mathbf{H}}$ is the molecular Hamiltonian operator, $|\Psi\rangle$ is the molecular wave function, and E is the total energy of the system. In the Møller–Plesset perturbation theory we write the Hamiltonian operator as

$$\hat{\mathbf{H}} = \hat{\mathbf{H}}^{(0)} + \lambda \hat{\mathbf{H}}^{'} , \tag{2.2}$$

where $\hat{\mathbf{H}}^{(0)}$ is the Hamiltonian used in the HF method (the electrons are moving in the mean field), λ is a parameter between 0 and 1, and $\hat{\mathbf{H}}^{'}$ is the perturbation operator representing the missing electron-electron interactions. We can then expand the wavefunction $|\Psi\rangle$ and total energy E in a power series of λ as

$$|\Psi\rangle = |\Psi^{(0)}\rangle + \lambda |\Psi^{(1)}\rangle + \lambda^2 |\Psi^{(2)}\rangle + ... \tag{2.3}$$

$$E = E^{(0)} + \lambda E^{(1)} + \lambda^2 E^{(2)} + ... \tag{2.4}$$

and ask, how how does the total energy change with the included terms. After some algebra, we can show that the first order correction to the total energy is zero, the second order correction is given by

$$E_{corr}^{MP2} = \sum_{s>0} \frac{H_{0s}^{'} H_{s0}^{'}}{E_0 - E_s} , \tag{2.5}$$

where s runs over all doubly excited determinants, $H_{0s}^{'}$ is the matrix element of the perturbation operator between the HF determinant and the doubly excited determinant, and E_0 and E_s are the energies of the reference and doubly excited determinants, respectively.[2, 3] We could express all higher-order corrections in a similar way, using only the matrix elements of the perturbation operator and the energies of the determinants. For practical calculations, we apply Slater-Condon rules to evaluate the matrix elements and use the orbital energies obtained from the HF calculation. The expressions for calculation are summarised below.

2.2. Implementation of 2nd and 3rd Order Corrections

Having the antisymmetrized Coulomb integrals in the MS basis and physicists' notation defined in section 1.3, we can now proceed with the calculation of the correlation energy. We wil use the convention, that the indices i, j, k, and l run over occupied spinorbitals, while the indices a, b, c, and d run over virtual spinorbitals. The 2nd order and 3rd

$$E_{corr}^{MP2} = \frac{1}{4} \sum_{ijab} \frac{\langle ab||ij\rangle \langle ij||ab\rangle}{\varepsilon_i + \varepsilon_j - \varepsilon_a - \varepsilon_b} \tag{2.6}$$

The 3rd order correlation energy:

$$\begin{aligned} E_{corr}^{MP3} = & \frac{1}{8} \sum_{ijab} \frac{\langle ab||ij\rangle \langle cd||ab\rangle \langle ij||cd\rangle}{(\varepsilon_i + \varepsilon_j - \varepsilon_a - \varepsilon_b)(\varepsilon_i + \varepsilon_j - \varepsilon_c - \varepsilon_d)} + \\ & + \frac{1}{8} \sum_{ijab} \frac{\langle ab||ij\rangle \langle ij||kl\rangle \langle kl||ab\rangle}{(\varepsilon_i + \varepsilon_j - \varepsilon_a - \varepsilon_b)(\varepsilon_k + \varepsilon_l - \varepsilon_a - \varepsilon_b)} + \\ & + \sum_{ijab} \frac{\langle ab||ij\rangle \langle cj||kb\rangle \langle ik||ac\rangle}{(\varepsilon_i + \varepsilon_j - \varepsilon_a - \varepsilon_b)(\varepsilon_i + \varepsilon_k - \varepsilon_a - \varepsilon_c)} \end{aligned} \tag{2.7}$$

To calculate the 4th order correction, we would need to write 39 terms, which is not practical. Higher-order corrections are usually not programmed this way, instead, the diagrammatic approach is used.[3–5]

2.3. 2nd and 3rd Order Corrections Code Example

In a similar fashion to the HF method, we can implement the Møller–Plesset perturbation theory in Python. The code below first proposes a self-contained exercise to calculate the Møller–Plesset Perturbation Theory of 2nd Order (MP2) and Møller–Plesset Perturbation Theory of 3rd Order (MP3) correlation energies. The solution is then provided in the following code snippet. You should append the code after your HF implementation, since the MP2 and MP3 methods are built on top of the HF method so should have already completed the HF calculations in section 1.4.

2.3.1. Exercise

In the exercise, you are expected to calculate the MP2 and MP3 correlation energies. The exercise is provided in the Listing 2.1 below.

```
"""
Since we have everything we need for the MP calculations, we can now calculate the MP2 correlation
    energy. The result should be stored in the "E_MP2" variable.
"""
E_MP2 = 0

"""
Let's not stop here. We can calculate MP3 correlation energy as well. Please calculate it and store
     it in the "E_MP3" variable.
"""
E_MP3 = 0

# print the results
```

```
print("MP2 ENERGY: {:.8f}".format(E_HF + E_MP2 +       + VNN))
print("MP3 ENERGY: {:.8f}".format(E_HF + E_MP2 + E_MP3 + VNN))
```

Listing 2.1: MP2 and MP3 exercise code.

2.3.2. Solution

The solutions provided in the Listing 2.2 below are complete implementations of the MP2 and MP3 correlation energies. The code should be appended after the HF implementation. The code calculates the MP2 and MP3 correlation energies and prints the results.

```
# energy containers
E_MP2, E_MP3 = 0, 0

# calculate the MP2 correlation energy
if args.mp2 or args.mp3:
    E_MP2 += 0.25 * np.einsum("abij,ijab,abij", Jmsa[v, v, o, o], Jmsa[o, o, v, v], Emsd, optimize=True)
    print("    MP2 ENERGY: {:.8f}".format(E_HF + E_MP2 + VNN))

# calculate the MP3 correlation energy
if args.mp3:
    E_MP3 += 0.125 * np.einsum("abij,cdab,ijcd,abij,cdij", Jmsa[v, v, o, o], Jmsa[v, v, v, v], Jmsa[o, o, v, v], Emsd, Emsd, optimize=True)
    E_MP3 += 0.125 * np.einsum("abij,ijkl,klab,abij,abkl", Jmsa[v, v, o, o], Jmsa[o, o, o, o], Jmsa[o, o, v, v], Emsd, Emsd, optimize=True)
    E_MP3 += 1.000 * np.einsum("abij,cjkb,ikac,abij,acik", Jmsa[v, v, o, o], Jmsa[v, o, o, v], Jmsa[o, o, v, v], Emsd, Emsd, optimize=True)
    print("    MP3 ENERGY: {:.8f}".format(E_HF + E_MP2 + E_MP3 + VNN))
```

Listing 2.2: MP2 and MP3 exercise code solution.

3. Configuration Interaction

Configuration Interaction (CI) is a post-HF, utilizing a linear variational approach to address the nonrelativistic Schrödinger equation under the Born–Oppenheimer approximation for multi-electron quantum systems. CI mathematically represents the wave function as a linear combination of Slater determinants. The term "configuration" refers to different ways electrons can occupy orbitals, while "interaction" denotes the mixing of these electronic configurations or states. CI computations, however, are resource-intensive, requiring significant CPU time and memory, limiting their application to smaller molecular systems. While Full Configuration Interaction (FCI) considers all possible electronic configurations, making it computationally prohibitive for larger systems, truncated versions like Configuration Interaction Singles and Doubles (CISD) or Configuration Interaction Singles, Doubles and Triples (CISDT) are more feasible and commonly employed in quantum chemistry studies.

3.1. Theoretical Background of General Configuration Interaction

The idea is quite simple, using the convention, that the indices i, j, k, and l run over occupied spinorbitals and the indices a, b, c, and d run over virtual spinorbitals. The CI wavefunction is written as

$$|\Psi\rangle = c_0 |\Psi_0\rangle + \left(\frac{1}{1!}\right)^2 c_i^a |\Psi_i^a\rangle + \left(\frac{1}{2!}\right)^2 c_{ij}^{ab} |\Psi_{ij}^{ab}\rangle + \left(\frac{1}{3!}\right)^2 c_{ijk}^{abc} |\Psi_{ijk}^{abc}\rangle + \ldots \tag{3.1}$$

and we would like to know the coefficients c that minimize the energy. To do that, we simply construct the hamiltonian matrix in the basis of excited determinants and diagonalize it. The CI Hamiltonian matrix $\mathbf{H}^{CI}$ is constructed as

$$\mathbf{H}^{CI} = \begin{bmatrix} \langle\Psi_0|\hat{H}|\Psi_0\rangle & \langle\Psi_0|\hat{H}|\Psi_i^a\rangle & \langle\Psi_0|\hat{H}|\Psi_{ij}^{ab}\rangle & \ldots \\ \langle\Psi_i^a|\hat{H}|\Psi_0\rangle & \langle\Psi_i^a|\hat{H}|\Psi_i^a\rangle & \langle\Psi_i^a|\hat{H}|\Psi_{ij}^{ab}\rangle & \ldots \\ \langle\Psi_{ia}^{jb}|\hat{H}|\Psi_0\rangle & \langle\Psi_{ia}^{jb}|\hat{H}|\Psi_1\rangle & \langle\Psi_{ia}^{jb}|\hat{H}|\Psi_{ia}^{jb}\rangle & \ldots \\ \vdots & \vdots & \vdots & \ddots \end{bmatrix} \tag{3.2}$$

and solving the eigenvalue problem

$$\mathbf{H}^{CI}\mathbf{C}^{CI} = \mathbf{C}^{CI}\square^{CI} \tag{3.3}$$

where $\mathbf{C}^{CI}$ is a matrix of coefficients and $\square^{CI}$ is a diagonal matrix of eigenvalues. The lowest eigenvalue corresponds to the ground state energy, while the eigenvector corresponding to the lowest eigenvalue gives the coefficients that minimize the energy. The matrix elements of the CI Hamiltonian are calculated using the Slater-Condon rules in the form

$$\mathbf{H}_{ij}^{CI} = \begin{cases} \sum_k H_{kk}^{core,MS} + \frac{1}{2}\sum_k \sum_l \langle kl||kl\rangle & D_i = D_j \\ H_{pr}^{core,MS} + \sum_k \langle pk||lk\rangle & D_i = \{\cdots p\cdots\} \wedge D_j = \{\cdots r\cdots\} \\ \langle pq||rs\rangle & D_i = \{\cdots p\cdots q\cdots\} \wedge D_j = \{\cdots r\cdots s\cdots\} \\ 0 & \text{otherwise,} \end{cases} \tag{3.4}$$

where D_i and D_j are Slater determinants, $\mathbf{H}^{core,MS}$ is the core Hamiltonian in the basis of molecular spinorbitals, and $\langle pk||lk\rangle$ are the antisymmetrized Coulomb repulsion integrals in the basis of molecular spinorbitals and physicists' notation. The sums extend over all spinorbitals common between the two determinants. All the integrals in the MS basis are already explained in section 1.3. Keep in mind, that to apply the Slater-Condon rules, the determinants must be aligned, and the sign of the matrix elements must be adjusted accordingly, based on the number of permutations needed to align the determinants.

The problem with CI is that it is not size-extensive, meaning that the energy does not scale linearly with the number of electrons. This is because the CI wavefunction is not size-consistent, and the energy of a system is not the sum of the energies of its parts. This is a significant drawback of CI, as it limits its application to small systems.

3.2. Full Configuration Interaction Implementation

Let's consider the FCI method, which considers all possible electronic configurations within a given basis set. The FCI method provides the most accurate description of the electronic structure, but its computational cost grows exponentially with the number of electrons and basis functions, making it infeasible for large systems.

The only thing that is needed, besides the general CI equations, are the determinants. For simplicity, we will include singlet and triplet states. The number of these determinants N_D can be for FCI calculated using the binomial coefficients

$$N_D = \binom{n}{k} \tag{3.5}$$

assuming k is the total number of electrons, and n is the total number of spinorbitals. Each determinant is formed by permuting the electrons between spinorbitals. For practical representation, it's useful to describe determinants as arrays of numbers, where each number corresponds to the index of an occupied orbitals. For example, the ground state determinant for a system with 6 electrons and 10 spinorbitals can be represented as $\{0,1,2,3,4,5\}$, whereas the determinant $\{0,1,2,3,4,6\}$ represents an excited state with one electron excited from orbital 5 to orbital 6. Using the determinants, the CI Hamiltonian matrix 3.2 can be constructed, and the eigenvalue problem 3.3 can be solved to obtain the ground and excited state energies.

3.3. Full Configuration Interaction Implementation Code Example

This section provides a simple example of a FCI implementation. The code snippets are, as the previous coding examples, written in Python and use the NumPy library for numerical operations. The example demonstrates the generation of determinants, the construction of the CI Hamiltonian matrix, and the solution of the eigenvalue

problem to obtain the ground state energy. The code is designed for educational purposes and may require modifications for practical applications, such as the inclusion of truncation schemes for larger systems. You are expected to have the HF calculations in section 1.4 completed before proceeding with the CI implementation, as the CI method builds on the HF method.

3.3.1. Exercise

In the exercise, you are expected to calculate the FCI energy for a simple system. The exercise is provided in the Listing 3.1 below.

```
"""
Since we already calculated the necessary integrals in the MS basis, we can proceed. The next step
    involves generating determinants. We will store these in a simple list, with each determinant
    represented by an array of numbers, where each number corresponds to an occupied spinorbital.
    Since we are programming for Full Configuration Interaction (FCI), we aim to generate all
    possible determinants. However, should we decide to implement methods like CIS, CID, or CISD,
    we could easily limit the number of excitations. It is important to remember that for all CI
    methods, the rest of the code remains unchanged. The only difference lies in the determinants
    used. Don't overcomplicate this. Generating all possible determinants can be efficiently
    achieved using a simple list comprehension. I recommend employing the combinations function
    from the itertools package to facilitate this task.
"""
dets = list()

"""
Now, for your convenince, I define here the CI Hamiltonian.
"""
Hci = np.zeros([len(dets), len(dets)])

"""
Before we begin constructing the Hamiltonian, I recommend defining the Slater-Condon rules. Let's
    consider that the input for these functions will be an array of spinorbitals, segmented into
    unique and common ones. A practical approach might be to arrange this 1D array with all unique
    spinorbitals at the front, followed by the common spinorbitals. This arrangement allows you to
    easily determine the number of unique spinorbitals based on the rule being applied, meaning you
     will always know how many entries at the beginning of the array are unique spinorbitals. While
     you can develop your own method for managing this array, I will proceed under the assumption
    that the Slater-Condon rules we use will take a single array of spinorbitals and return an
    unsigned matrix element. The sign of this element will be corrected later in the script. For
    simplicity and flexibility, I'll define these rules using lambda functions, but you're welcome
    to expand them into full functions if you prefer.
"""
slater0 = lambda so: 0
slater1 = lambda so: 0
slater2 = lambda so: 0

"""
We can now proceed to filling the CI Hamiltonian. The loop is simple.
"""
for i in range(Hci.shape[0]):
    for j in range(Hci.shape[1]):

        """
        The challenging part of this process is aligning the determinants. In this step, I transfer
     the contents of the j-th determinant into the "aligned" determinant. It's important not to
    alter the j-th determinant directly within its original place, as doing so could disrupt the
    computation of other matrix elements. Instead, we carry out the next steps on the determinant
    now contained in the "aligned" variable. Additionally, the element sign is defined at this
    stage. You probably want to leave this unchanged.
        """
        aligned, sign = dets[j].copy(), 1
```

```
28
29          """
30          Now it's your turn. Please adjust the "aligned" determinant to match the i-th determinant
        as closely as possible. By "align", I mean you should execute a series of spinorbital swaps to
        minimize the differences between the "aligned" and the i-th determinant. It's also important to
         monitor the number of swaps you make, as each swap affects the sign of the determinant, hence
        the reason for the "sign" variable defined earlier. This task is not straightforward, so don't
        hesitate to reach out to the authors if you need guidance.
31          """
32          aligned = aligned
33
34          """
35          After aligning, we end up with two matched determinants: "aligned" and "dets[i]". At this
        point, we can apply the Slater-Condon rules. I suggested earlier that the input for these rules
         should be an array combining both unique and common spinorbitals. You can prepare this array
        now. However, if you've designed your Slater-Condon rules to directly accept the determinants
        instead, you can skip this preparatory step.
36          """
37          so = list()
38
39          """
40          Now, you'll need to assign the matrix element. Start by determining the number of
        differences between the two determinants. Based on this number, apply the corresponding Slater-
        Condon rule. Don't forget to multiply the result by the sign to account for any changes due to
        swaps made during the alignment of the determinants.
41          """
42          H[i, j] = 0
43
44  """
45  You can finally solve the eigenvalue problem. Please, assign the correlation energy to the "E_FCI"
        variable.
46  """
47  E_FCI = 0
```

Listing 3.1: CI exercise code.

3.3.2. Solution

The solution to the exercise is provided in the Listing 3.2 below. It includes the generation of determinants, the construction of the CI Hamiltonian matrix, and the solution of the eigenvalue problem to obtain the ground state energy.

```
1  # generate the determiants
2  dets = [np.array(det) for det in it.combinations(range(2 * nbf), 2 * nocc)]
3
4  # define the CI Hamiltonian
5  Hci = np.zeros([len(dets), len(dets)])
6
7  # define the Slater-Condon rules, "so" is an array of unique and common spinorbitals [unique,
       common]
8  slater0 = lambda so: sum([Hms[m, m] for m in so]) + sum([0.5 * Jmsa[m, n, m, n] for m, n in it.
       product(so, so)])
9  slater1 = lambda so: Hms[so[0], so[1]] + sum([Jmsa[so[0], m, so[1], m] for m in so[2:]])
10 slater2 = lambda so: Jmsa[so[0], so[1], so[2], so[3]]
11
12 # filling of the CI Hamiltonian
13 for i in range(0, Hci.shape[0]):
14     for j in range(i, Hci.shape[1]):
15
16         # aligned determinant and the sign
17         aligned, sign = dets[j].copy(), 1
```

```
        # align the determinant "j" to "i" and calculate the sign
        for k in (k for k in range(len(aligned)) if aligned[k] != dets[i][k]):
            while len(l := np.where(dets[i] == aligned[k])[0]) and l[0] != k:
                aligned[[k, l[0]]] = aligned[[l[0], k]]; sign *= -1

        # find the unique and common spinorbitals
        so = np.block([
            np.array([aligned[k] for k in range(len(aligned)) if aligned[k] not in dets[i]]),
            np.array([dets[i][k] for k in range(len(dets[j])) if dets[i][k] not in aligned]),
            np.array([aligned[k] for k in range(len(aligned)) if aligned[k] in dets[i]])
        ]).astype(int)

        # apply the Slater-Condon rules and multiply by the sign
        if ((aligned - dets[i]) != 0).sum() == 0: Hci[i, j] = slater0(so) * sign
        if ((aligned - dets[i]) != 0).sum() == 1: Hci[i, j] = slater1(so) * sign
        if ((aligned - dets[i]) != 0).sum() == 2: Hci[i, j] = slater2(so) * sign

        # fill the lower triangle
        Hci[j, i] = Hci[i, j]

# solve the eigensystem and assign energy
eci, Cci = np.linalg.eigh(Hci); E_FCI = eci[0] - E_HF
```

Listing 3.2: CI exercise code solution.

4. Coupled Cluster Theory

Coupled Cluster (CC) theory is a post-HF method used in quantum chemistry to achieve highly accurate solutions to the electronic Schrödinger equation, particularly for ground states and certain excited states. It improves upon HF by incorporating electron correlation effects through a systematic inclusion of excitations (singles, doubles, triples, etc.) from a reference wavefunction, usually the HF wavefunction. The method uses an exponential ansatz to account for these excitations, leading to a size-consistent and size-extensive approach, making it one of the most accurate methods available for small to medium-sized molecular systems.

Within CC theory, specific truncations are often applied to manage computational cost. The Coupled Cluster Doubles (CCD) method considers only double excitations, capturing electron correlation more effectively than simpler methods like HF but at a lower computational expense than higher-level methods. Coupled Cluster Singles and Doubles (CCSD) extends this approach by including both single and double excitations, offering greater accuracy, particularly for systems where single excitations play a significant role. CCSD is widely used due to its balance between accuracy and computational feasibility, making it a reliable choice for many chemical systems. In the below equations, we will again use the convention that the indices i, j, k, and l run over occupied spinorbitals, while the indices a, b, c, and d run over virtual spinorbitals.

4.1. Coupled Cluster Formalism

In the CC formalism, we write the total wavefunction in an exponential form as

$$|\Psi\rangle = e^{\hat{\mathbf{T}}} |\Psi_0\rangle \tag{4.1}$$

where $|\Psi_0\rangle$ is the reference wavefunction, usually the HF wavefunction, and $\hat{T}$ is the cluster operator that generates excitations from the reference wavefunction. The cluster operator is defined as

$$\hat{\mathbf{T}} = \hat{\mathbf{T}}_1 + \hat{\mathbf{T}}_2 + \hat{\mathbf{T}}_3 + \ldots \tag{4.2}$$

where $\hat{\mathbf{T}}_1$ generates single excitations, $\hat{\mathbf{T}}_2$ generates double excitations, and so on. For example

$$\hat{\mathbf{T}}_1 |\Psi_0\rangle = \left(\frac{1}{1!}\right)^2 t_i^a |\Psi_i^a\rangle \,, \tag{4.3}$$

where t_i^a are the single excitation amplitudes. These amplitudes are just expansion coefficients that determine the contribution of each excitation to the total wavefunction. In the context of configuration interaction, we denoted these coefficients as c_i^a. Now that we have the total wavefunction, we want to solve the Schrödinger equation

$$\hat{\mathbf{H}} |\Psi\rangle = E |\Psi\rangle \tag{4.4}$$

where $\hat{H}$ is the molecular Hamiltonian operator, E is the total energy of the system, and $|\Psi\rangle$ is the total wavefunction. In the CC theory, we usually rewrite the Schrödinger equation in the exponential form as

$$e^{-\hat{\mathbf{T}}}\hat{\mathbf{H}}e^{\hat{\mathbf{T}}}\,|\Psi_0\rangle = E\,|\Psi_0\rangle \tag{4.5}$$

because we can then express the CC energy as

$$E = \langle\Psi_0|e^{-\hat{\mathbf{T}}}\hat{\mathbf{H}}e^{\hat{\mathbf{T}}}|\Psi_0\rangle\,, \tag{4.6}$$

taking advantage of the exponential form of the wavefunction. We could then proceed to express the total energy for various CC methods like CCD and CCSD, but the equations would be quite lengthy. Instead, we will leave the theory here and proceed to the actual calculations. One thing to keep in mind is that the CC equations are nonlinear and require iterative solution methods to obtain the final amplitudes. The initial guess for the amplitudes is often set to zero, and the equations are solved iteratively until convergence is achieved.

4.2. Implementation of Truncated Coupled Cluster Methods

The derivation of the equations that are actually used to perform the calculations is quite lengthy and involves a lot of algebra. We will not go into the details here, but we will provide the final expressions for the CCD and CCSD methods.[6] The CCD and CCSD methods are the most commonly used CC methods, and they are often used as benchmarks for other methods. All we need for the evaluation of the expressions below are the Coulomb integrals in the MS basis and physicists' notation, Fock matrix in the MS basis and the orbital energies obtained from the HF calculation. All these transformations are already explained in section 1.3 in the HF section. The expressions for the CCD can be written as

$$E_{\text{CCD}} = \frac{1}{4}\,\langle ij||ab\rangle\, t_{ij}^{ab} \tag{4.7}$$

where the double excitation amplitudes t_{ij}^{ab} are determined by solving the CCD amplitude equation. The CCD amplitude equations are given by

$$\begin{aligned} t_{ij}^{ab} = \langle ab||ij\rangle &+ \frac{1}{2}\,\langle ab||cd\rangle\, t_{cd}^{ij} + \frac{1}{2}\,\langle kl||ij\rangle\, t_{ab}^{kl} + \hat{P}_{(a/b)}\hat{P}_{(i/j)}\,\langle ak||ic\rangle\, t_{cb}^{ij} - \\ &- \frac{1}{2}\hat{P}_{(a/b)}\,\langle kl||cd\rangle\, t_{ac}^{ij} t_{bd}^{kl} - \frac{1}{2}\hat{P}_{(i/j)}\,\langle kl||cd\rangle\, t_{ab}^{ik} t_{cd}^{jl} + \\ &+ \frac{1}{4}\,\langle kl||cd\rangle\, t_{cd}^{ij} t_{ab}^{kl} + \hat{P}_{(i/j)}\,\langle kl||cd\rangle\, t_{ac}^{ik} t_{bd}^{jl} \end{aligned} \tag{4.8}$$

where $\hat{P}_{(a/b)}$ and $\hat{P}_{(i/j)}$ are permutation operators that ensure the correct antisymmetry of the amplitudes. The CCSD energy expression is given by

$$E_{\text{CCSD}} = F_{ia}^{MS} t_a^i + \frac{1}{4}\,\langle ij||ab\rangle\, t_{ij}^{ab} + \frac{1}{2}\,\langle ij||ab\rangle\, t_i^a t_b^j \tag{4.9}$$

where the single and double excitation amplitudes t_a^i and t_{ij}^{ab} are determined by solving the CCSD amplitude equations. To simplify the notation a little bit, we define the the $\mathscr{F}$ and $\mathscr{W}$ intermediates as

$$\mathscr{F}_{ae} = (1-\delta_{ae})\,F_{ae} - \frac{1}{2}\sum_{m} F_{me}t^{a}_{m} + \sum_{mf} t^{f}_{m}\,\langle ma||fe\rangle - \frac{1}{2}\sum_{mnf}\tilde{\tau}^{af}_{mn}\,\langle mn||ef\rangle \tag{4.10}$$

$$\mathscr{F}_{mi} = (1-\delta_{mi})\,F_{mi} + \frac{1}{2}\sum_{e} F_{me}t^{e}_{i} + \sum_{en} t^{e}_{n}\,\langle mn||ie\rangle + \frac{1}{2}\sum_{nef}\tilde{\tau}^{ef}_{in}\,\langle mn||ef\rangle \tag{4.11}$$

$$\mathscr{F}_{me} = F_{me} + \sum_{nf} t^{f}_{n}\,\langle mn||ef\rangle \tag{4.12}$$

$$\mathscr{W}_{mnij} = \langle mn||ij\rangle + \hat{P}_{(i/j)}\sum_{e} t^{e}_{j}\,\langle mn||ie\rangle + \frac{1}{4}\sum_{ef}\tau^{ef}_{ij}\,\langle mn||ef\rangle \tag{4.13}$$

$$\mathscr{W}_{abef} = \langle ab||ef\rangle - \hat{P}_{(a/b)}\sum_{e} t^{b}_{m}\,\langle am||ef\rangle + \frac{1}{4}\sum_{mn}\tau^{ab}_{mn}\,\langle mn||ef\rangle \tag{4.14}$$

$$\mathscr{W}_{mbej} = \langle mb||ej\rangle + \sum_{f} t^{f}_{j}\,\langle mb||ef\rangle - \sum_{n} t^{b}_{n}\,\langle mn||ej\rangle - \tag{4.15}$$

$$- \sum_{nf}\left(\frac{1}{2}t^{fb}_{jn} + t^{f}_{j}t^{b}_{n}\right)\langle mn||ef\rangle \tag{4.16}$$

and two-particle excitation operators as

$$\tilde{\tau}^{ab}_{ij} = t^{ab}_{ij} + \frac{1}{2}\left(t^{a}_{i}t^{b}_{j} - t^{b}_{i}t^{a}_{j}\right) \tag{4.17}$$

$$\tau^{ab}_{ij} = t^{ab}_{ij} + t^{a}_{i}t^{b}_{j} - t^{b}_{i}t^{a}_{j} \tag{4.18}$$

The CCSD single excitations amplitude equations are then given by

$$t^{a}_{i} = F^{MS}_{ai} + \sum_{e} t^{e}_{i}\mathscr{F}_{ae} - \sum_{m} t^{a}_{m}\mathscr{F}_{mi} \sum_{me} t^{ae}_{im}\mathscr{F}_{me} - \sum_{nf} t^{f}_{n}\,\langle na||if\rangle - $$

$$- \frac{1}{2}\sum_{mef} t^{ef}_{im}\,\langle ma||ef\rangle - \frac{1}{2}\sum_{men} t^{ae}_{mn}\,\langle nm||ei\rangle \tag{4.19}$$

and the CCSD double excitations amplitude equations are given by

$$t^{ab}_{ij} = \langle ab||ij\rangle + \hat{P}_{(a/b)}\sum_{e} t^{ae}_{ij}\left(\mathscr{F}_{be} - \frac{1}{2}\sum_{m} t^{b}_{m}\mathscr{F}_{ae}\right) -$$

$$- \hat{P}_{(i/j)}\sum_{m} t^{ab}_{im}\left(\mathscr{F}_{mi} + \frac{1}{2}\sum_{e} t^{e}_{j}\mathscr{F}_{me}\right) + \frac{1}{2}\sum_{mn}\tau^{ab}_{mn}\mathscr{W}_{mnij} +$$

$$+ \frac{1}{2}\sum_{ef}\tau^{ef}_{ij}\mathscr{W}_{abef} + \hat{P}_{(i/j)}\hat{P}_{(a/b)}\sum_{me}\left(t^{ae}_{im}\mathscr{W}_{mbej} - t^{e}_{i}t^{a}_{m}\,\langle mb||ej\rangle\right) +$$

$$+ \hat{P}_{(i/j)}\sum_{e} t^{e}_{i}\,\langle ab||ej\rangle - \hat{P}_{(a/b)}\sum_{m} t^{a}_{m}\,\langle mb||ij\rangle \tag{4.20}$$

The CCSD amplitude equations are, again, nonlinear and require iterative solution methods to obtain the final amplitudes. The initial guess for the amplitudes is often set to zero, and the equations are solved iteratively until convergence is achieved.

4.3. Coupled Cluster Singles and Doubles Code Example

If you have completed the HF implementation in section 1.4, you can now proceed with the implementation of the CCSD methods. The code below first proposes a self-contained exercise to calculate the CCSD correlation energies. The solution is then provided in the following code snippet. You should append the code after your HF implementation, since the CCSD method is built on top of the HF method.

4.3.1. Exercise

In the exercise, you are expected to calculate the CCSD correlation energy. The exercise is provided in the Listing 4.1 below.

```
"""
We also have everything we need for the CC calculations. In this exercise, we will calculate the
    CCSD energy. Since the calculation will be iterative, I define here the CCSD energy as zero,
    the "E_CCSD_P" variable will be used to monitor convergence.
"""
E_CCSD, E_CCSD_P = 0, 1

"""
The first step of the calculation is to define the "t1" and "t2" amplitudes. These arrays can be
    initialized as zero arrays with the appropriate dimensions. I will leave this task to you.
"""
t1, t2 = np.array([]), np.array([])

"""
Now for the more complicated part. The CCSD calculation is iterative, and the convergence criterion
     is set by the "thresh" variable. The while loop should be filled with the appropriate
    calculations. The calculation of the "t1" and "t2" amplitudes is the most challenging part of
    the CCSD calculation. After convergence, the "E_CCSD" variable should store the final CCSD
    energy.
"""
while abs(E_CCSD - E_CCSD_P) > thresh:
    break

# print the CCSD energy
print("CCSD ENERGY: {:.8f}".format(E_HF + E_CCSD + VNN))
```

Listing 4.1: CCSD exercise code.

4.3.2. Solution

The solutions provided in the Listing 4.2 below are complete implementations of the CCD and CCSD correlation energies. The code should be appended after the HF implementation. The code calculates the CCD and CCSD correlation energies and prints the results. The variable `args` is an argument parser that allows you to choose which method you want to calculate. Since the variable is not defined in this snippet (but is defined in the complete code), you can ignore it for now and remove the conditionals.

```
# energy containers for all the CC methods
E_CCD, E_CCD_P, E_CCSD, E_CCSD_P = 0, 1, 0, 1

# initialize the first guess for the t-amplitudes as zeros
t1, t2 = np.zeros((2 * nvirt, 2 * nocc)), np.zeros((2 * nvirt, 2 * nvirt, 2 * nocc, 2 * nocc))

# CCD loop
if args.ccd:
    while abs(E_CCD - E_CCD_P) > args.threshold:

```

```
        # collect all the distinct LCCD terms
        lccd1 = 0.5 * np.einsum("abcd,cdij->abij", Jmsa[v, v, v, v], t2, optimize=True)
        lccd2 = 0.5 * np.einsum("klij,abkl->abij", Jmsa[o, o, o, o], t2, optimize=True)
        lccd3 =       np.einsum("akic,bcjk->abij", Jmsa[v, o, o, v], t2, optimize=True)

        # apply the permuation operator and add it to the corresponding LCCD terms
        lccd3 = lccd3 + lccd3.transpose(1, 0, 3, 2) - lccd3.transpose(1, 0, 2, 3) - lccd3.transpose(0, 1, 3, 2)

        # collect all the distinct first CCD terms
        ccd1 = -0.50 * np.einsum("klcd,acij,bdkl->abij", Jmsa[o, o, v, v], t2, t2, optimize=True)
        ccd2 = -0.50 * np.einsum("klcd,abik,cdjl->abij", Jmsa[o, o, v, v], t2, t2, optimize=True)
        ccd3 =  0.25 * np.einsum("klcd,cdij,abkl->abij", Jmsa[o, o, v, v], t2, t2, optimize=True)
        ccd4 =         np.einsum("klcd,acik,bdjl->abij", Jmsa[o, o, v, v], t2, t2, optimize=True)

        # apply the permuation operator and add it to the corresponding CCD terms
        ccd1, ccd2, ccd4 = ccd1 - ccd1.transpose(1, 0, 2, 3), ccd2 - ccd2.transpose(0, 1, 3, 2), ccd4 - ccd4.transpose(0, 1, 3, 2)

        # update the t-amplitudes
        t2 = Emsd * (Jmsa[v, v, o, o] + lccd1 + lccd2 + lccd3 + ccd1 + ccd2 + ccd3 + ccd4)

        # evaluate the energy
        E_CCD_P, E_CCD = E_CCD, 0.25 * np.einsum("ijab,abij", Jmsa[o, o, v, v], t2, optimize=True)

    # print the CCD energy
    print("    CCD ENERGY: {:.8f}".format(E_HF + E_CCD + VNN))

# CCSD loop
if args.ccsd:
    while abs(E_CCSD - E_CCSD_P) > args.threshold:

        # calculate the effective two-particle excitation operators
        ttau = t2 + 0.5 * np.einsum("ai,bj->abij", t1, t1, optimize=True) - 0.5 * np.einsum("ai,bj->abij", t1, t1, optimize=True).swapaxes(2, 3)
        tau  = t2 +       np.einsum("ai,bj->abij", t1, t1, optimize=True) -       np.einsum("ai,bj->abij", t1, t1, optimize=True).swapaxes(2, 3)

        # calculate the 2D two-particle intermediates
        Fae = (1 - np.eye(2 * nvirt)) * Fms[v, v] - 0.5 * np.einsum("me,am->ae",     Fms[o, v],
        t1,   optimize=True)                                                    +       np.einsum("mafe,fm->ae",   Jmsa[o, v, v, v], t1,   optimize=True)
              - 0.5 * np.einsum("mnef,afmn->ae", Jmsa[o, o, v, v], ttau, optimize=True)
        Fmi = (1 - np.eye(2 * nocc )) * Fms[o, o] + 0.5 * np.einsum("me,ei->mi",     Fms[o, v],
        t1,   optimize=True)                                                    +       np.einsum("mnie,en->mi",   Jmsa[o, o, o, v], t1,   optimize=True)
              + 0.5 * np.einsum("mnef,efin->mi", Jmsa[o, o, v, v], ttau, optimize=True)
        Fme =                           Fms[o, v] +       np.einsum("mnef,fn->me",   Jmsa[o, o, v, v], t1,   optimize=True)

        # define some complementary variables used in the following expressions
        Fmea =            np.einsum("bm,me->be",   t1, Fme, optimize=True)
        Fmeb =            np.einsum("ej,me->mj",   t1, Fme, optimize=True)
        t12  = 0.5 * t2 + np.einsum("fj,bn->fbjn", t1, t1,  optimize=True)

        # define the permutation arguments for all terms the W intermediates
        P1 = np.einsum("ej,mnie->mnij", t1, Jmsa[o, o, o, v], optimize=True)
        P2 = np.einsum("bm,amef->abef", t1, Jmsa[v, o, v, v], optimize=True)

        # calculate the 4D two-particle intermediates
        Wmnij = Jmsa[o, o, o, o] + 0.25 * np.einsum("efij,mnef->mnij", tau, Jmsa[o, o, v, v], optimize=True) + P1 - P1.swapaxes(2, 3)
```

```
        Wabef = Jmsa[v, v, v, v] + 0.25 * np.einsum("abmn,mnef->abef", tau, Jmsa[o, o, v, v],
    optimize=True) - P2 + P2.swapaxes(0, 1)
        Wmbej = Jmsa[o, v, v, o] +        np.einsum("fj,mbef->mbej",   t1,  Jmsa[o, v, v, v],
    optimize=True)                                  -        np.einsum("bn,mnej->mbej",   t1,  Jmsa
    [o, o, v, o], optimize=True)                                 -        np.einsum("fbjn,mnef->
    mbej", t12, Jmsa[o, o, v, v], optimize=True)

        # define the right hand side of the T1 and T2 amplitude equations
        rhs_T1, rhs_T2 = Fms[v, o].copy(), Jmsa[v, v, o, o].copy()

        # calculate the right hand side of the CCSD equation for T1
        rhs_T1 +=       np.einsum("ei,ae->ai",     t1, Fae,              optimize=True)
        rhs_T1 -=       np.einsum("am,mi->ai",     t1, Fmi,              optimize=True)
        rhs_T1 +=       np.einsum("aeim,me->ai",   t2, Fme,              optimize=True)
        rhs_T1 -=       np.einsum("fn,naif->ai",   t1, Jmsa[o, v, o, v], optimize=True)
        rhs_T1 -= 0.5 * np.einsum("efim,maef->ai", t2, Jmsa[o, v, v, v], optimize=True)
        rhs_T1 -= 0.5 * np.einsum("aemn,nmei->ai", t2, Jmsa[o, o, v, o], optimize=True)

        # define the permutation arguments for all terms in the equation for T2
        P1  = np.einsum("aeij,be->abij",    t2,     Fae - 0.5 * Fmea, optimize=True)
        P2  = np.einsum("abim,mj->abij",    t2,     Fmi + 0.5 * Fmeb, optimize=True)
        P3  = np.einsum("aeim,mbej->abij",  t2,     Wmbej,            optimize=True)
        P3 -= np.einsum("ei,am,mbej->abij", t1, t1, Jmsa[o, v, v, o], optimize=True)
        P4  = np.einsum("ei,abej->abij",    t1,     Jmsa[v, v, v, o], optimize=True)
        P5  = np.einsum("am,mbij->abij",    t1,     Jmsa[o, v, o, o], optimize=True)

        # calculate the right hand side of the CCSD equation for T2
        rhs_T2 += 0.5 * np.einsum("abmn,mnij->abij", tau, Wmnij, optimize=True)
        rhs_T2 += 0.5 * np.einsum("efij,abef->abij", tau, Wabef, optimize=True)
        rhs_T2 += P1.transpose(0, 1, 2, 3) - P1.transpose(1, 0, 2, 3)
        rhs_T2 -= P2.transpose(0, 1, 2, 3) - P2.transpose(0, 1, 3, 2)
        rhs_T2 += P3.transpose(0, 1, 2, 3) - P3.transpose(0, 1, 3, 2)
        rhs_T2 -= P3.transpose(1, 0, 2, 3) - P3.transpose(1, 0, 3, 2)
        rhs_T2 += P4.transpose(0, 1, 2, 3) - P4.transpose(0, 1, 3, 2)
        rhs_T2 -= P5.transpose(0, 1, 2, 3) - P5.transpose(1, 0, 2, 3)

        # Update T1 and T2 amplitudes
        t1, t2 = rhs_T1 * Emss, rhs_T2 * Emsd

        # evaluate the energy
        E_CCSD_P, E_CCSD = E_CCSD, np.einsum("ia,ai", Fms[o, v], t1) + 0.25 * np.einsum("ijab,abij"
    , Jmsa[o, o, v, v], t2) + 0.5 * np.einsum("ijab,ai,bj", Jmsa[o, o, v, v], t1, t1)

    # print the CCSD energy
    print("   CCSD ENERGY: {:.8f}".format(E_HF + E_CCSD + VNN))
```

Listing 4.2: CCD and CCSD method exercise code solution.

Part II.

Time-dependent quantum mechanics

5. Quantum dynamics in real time

Up to this point, we have taken the time-independent perspective on quantum mechanics, focusing on methods for calculating stationary states of electrons in molecules. In this chapter, we shift to time-dependent quantum mechanics and look at the time evolution of quantum systems. We will focus on simple low-dimensional models and introduce a simple numerical scheme suitable for wave function propagation. The power of simple low-dimensional models resides in the conceptual understanding of time evolution in quantum mechanics: they allow us to examine dynamics such as interference of two wave packets, tunnelling through energy barriers, or reflection of the wave packet on boundaries.

5.1. Theoretical background

The time evolution in quantum mechanics is governed by the time-dependent Schrödinger equation (TDSE),

$$i\hbar\frac{\mathrm{d}\psi(x,t)}{\mathrm{d}t} = \hat{H}(x)\psi(x,t)\,, \tag{5.1}$$

where i is the imaginary unit, $\hbar$ is the reduced Planck constant, ψ is the wave function, $\hat{H}$ is the Hamiltonian, t is time and x represents the position of a quantum particle, such as an electron, proton, or any other particle. The solution of the TDSE can be in general written as

$$\psi(x,t) = \hat{U}(t)\psi(x,0)\,. \tag{5.2}$$

The operator $\hat{U}$ is called a *propagator* or *time evolution operator*. Application of the propagator to a wave function is equivalent to evolving the wave function from time 0 to time t. The knowledge of the evolution operator is, therefore, elementary for a description of dynamics in quantum systems.

The propagator is easy to find if the Hamiltonian $\hat{H}$ does not depend on time. Then, we can formally write a solution of equation (5.1) in the following form:

$$\psi(x,t) = \mathrm{e}^{-\frac{i}{\hbar}\hat{H}(x)t}\psi(x,0)\,. \tag{5.3}$$

Notice that the result resembles the separation of variables for differential equations, only for a equation with operators. Considering the Hamiltonian to be time-independent restricts the validity of equation (5.3) only for systems without any time-dependent interaction such as oscillating electromagnetic field (radiation), collisions between particles, etc. The solution for time-dependent Hamiltonian is somewhat more complicated and will be omitted here.

Comparing equations (5.3) and (5.4), we see that the evolution operator has a simple form of an exponential of the Hamiltonian,

$$\hat{U}(t) = \mathrm{e}^{-\frac{i}{\hbar}\hat{H}(x)t}\,. \tag{5.4}$$

Formally, we now have a propagator containing all information about the time evolution of the system and we can propagate the wave function to an arbitrary time t. While this works well on paper, analytical evaluation of the propagator is in practice not a straightforward task for most Hamiltonians. The hindrance lies in the evaluation of the exponential in equation (5.4).

Let us briefly discuss why analytic evaluation of the exponential of Hamiltonian is a cumbersome task. The exponential of an operator is defined through its Taylor series as

$$e^{\hat{A}} = \sum_{l=0}^{\infty} \frac{\hat{A}^l}{l!}, \tag{5.5}$$

where $\hat{A}$ is and arbitrary operator. In our case, $\hat{A} = -\frac{i}{\hbar}\hat{H}(x)t$ which leads to the following relation:

$$\hat{U}(t) = \sum_{l=0}^{\infty} \frac{\hat{H}^l}{l!} \left(-\frac{i}{\hbar}t\right)^l. \tag{5.6}$$

Considering the Hamiltonian in the standard form,

$$\hat{H}(x) = \hat{T} + \hat{V}(x) = -\frac{\hbar^2}{2m}\frac{\mathrm{d}^2}{\mathrm{d}x^2} + V(x), \tag{5.7}$$

where $\hat{T}$ represents the kinetic energy operator, $\hat{V}$ stands for the potential energy operator and m is the particle mass, the propagator becomes:

$$\hat{U}(t) = \sum_{l=0}^{\infty} \frac{(\hat{T} + \hat{V})^l}{l!} \left(-\frac{i}{\hbar}t\right)^l. \tag{5.8}$$

Calculating this propagator requires computing an infinite series of powers of the Hamiltonian, further complicated by the fact that $\hat{T}$ and $\hat{V}$ do not commute and the exponential cannot be factorized. This makes direct evaluation infeasible for most of the potential.

Thus, analytical solutions of the TDSE are used only in model systems, usually when the complete basis of the respective Hilbert space is known. Instead, most time-dependent problems are solved numerically where arbitrary Hamiltonians are possible.

5.2. Numerical solution

Systems of chemical interest span, in general, an infinite Hilbert space. In other words, an infinite number of orthogonal basis functions are required to fully represent the wave function of such a system. However, any computer can only handle a finite basis set, since computer memory is finite, and often even quite limited. Thus, in any numerical implementation of quantum mechanics, we work in a truncated basis set. This basis truncation is the first and fundamental source of error for any computer simulation. There are other sources of error due to the numeric schemes employed, e.g. trapezoid rule for integration, or due to a finite time step in our propagation. While these sources of error can be eliminated by improving our numeric schemes or the time step, the truncation error will be always present and dependent on how much we truncated our basis.

In this text, we will represent the system on a grid of discrete points, a natural representation for a computer. These grid points (we can think of them as a set of Dirac δ-distributions) form our truncated basis set commonly referred to as a *pseudospectral* basis. In the following, we will explain how to work with such a representation, including how to evaluate the action of an operator and calculate the expectation value. In the end, the flow of the text will lead us to the evolution operator $\hat{U}(t)$ and techniques for efficient propagation of the wave function.

5.2.1. Representing wave function on a grid: the *pseudospectral* basis

As stated above we will work in a truncated *pseudospectral* basis and represent the system with a set of N discrete points. Since we work as usual in chemistry in the position representation, we start with discretizing the position coordinate x and replacing it by a set of points:

$$x \to \{x_0, x_1, x_2, \dots, x_N\}. \tag{5.9}$$

It is convenient for numerical applications to make the grid equidistant, meaning that the distance between any two neighbouring grid points, $\Delta x = x_i - x_{i-1}$, remains constant.

Next, we express the wave function ψ on this grid by evaluating it at each grid point x_i, giving:

$$\psi(x,t) \to \{\psi(x_0,t), \psi(x_1,t), \psi(x_2,t), \dots, \psi(x_N,t)\} = \{\psi_{0,t}, \psi_{1,t}, \psi_{2,t}, \dots, \psi_{N,t}\}\,. \tag{5.10}$$

We have now two indexes of ψ: the grid point index i and time index t. Further in the text, we will drop the index t to simplify the notation unless ψ in two different times will appear in the equation.

Having the wave function defined on a grid, the next step is to compute its norm. In the position representation, the norm is an integral of $\psi^*\psi$ over the position x, which can be numerically realized for example by the trapezoidal rule:

$$\langle\psi(x,t)|\psi(x,t)\rangle = \int_{-\infty}^{\infty} \psi^*(x,t)\psi(x,t)\,\mathrm{d}x = \sum_{j=1}^{N} \frac{\psi^*_{j-1}\psi_{j-1} + \psi^*_j\psi_j}{2}\Delta x\,. \tag{5.11}$$

While the trapezoidal rule is a simple and commonly used method, more accurate numerical integration schemes can be employed to reduce integration errors, though these may come at the cost of increased computational effort.

With the wave function defined, our attention now turns to operators: how to represent them and how to apply them. We will start with operators dependent only on the position x since they are simple to represent on the grid. Take the potential energy operator $V(x)$ which can be represented on the grid similarly to the wave function as

$$V(x) \to \{V(x_0), V(x_1), V(x_2), \dots, V(x_N)\} = \{V_0, V_1, V_2, \dots, V_N\}\,. \tag{5.12}$$

Now, applying the potential energy operator to the wave function reduces to a simple multiplication of the discretized operator and wave function as

$$\hat{V}\psi \to \{V_0\psi_0, V_1\psi_1, V_2\psi_2, \dots, V_N\psi_N\}\,. \tag{5.13}$$

The expectation value of the potential energy in analogy with the norm using the trapezoid rule:

$$\langle\psi(x,t)|\hat{V}|\psi(x,t)\rangle = \int_{-\infty}^{\infty} \psi^*(x,t)V(x)\psi(x,t)\,\mathrm{d}x = \sum_{j=1}^{N} \frac{V_{j-1}\psi^*_{j-1}\psi_{j-1} + V_j\psi^*_j\psi_j}{2}\Delta x \tag{5.14}$$

Any operator that depends only on the position x can be represented on the grid similarly to $\hat{V}$ and the expectation values are then evaluated in the same manner as above. However, operators that depend on momentum $\hat{p}$ require a more complex treatment, as they involve derivatives, which cannot be directly represented on the grid in the same straightforward manner.

5.2.2. Applying the kinetic energy operator: the Fourier method

To evaluate the action of the kinetic energy operator, or any other operator depending on momentum, it is convenient to switch to a basis where the action of momentum operator $\hat{p}$ is a simple multiplication. Such basis is the basis of the free particle states e^{-ikx}, the so-called k-space or momentum space since $p = \hbar k$. The wave function represented in the k-space can be obtained by Fourier transform (FT) $\mathscr{F}$ as

$$\psi(k,t) = \mathscr{F}[\psi(x,t)] = \int_{-\infty}^{\infty} \psi(x,t)\mathrm{e}^{-ikx}\,\mathrm{d}x\,. \tag{5.15}$$

Performing the FT on a computer is a routine task and allows us to easily transfer between the position representation $\psi(x,t)$ and momentum representation $\psi(k,t)$.

The action of the kinetic operator on the wave function is cumbersome to evaluate numerically in the position space due to the derivatives, but it is simple in the k-space using the FT:

$$\mathscr{F}[\hat{T}\psi(x,t)] = \mathscr{F}\left[-\frac{\hbar^2}{2m}\frac{\mathrm{d}^2\psi(x,t)}{\mathrm{d}x^2}\right] = -\frac{\hbar^2}{2m}\int_{-\infty}^{\infty}\frac{\mathrm{d}^2\psi(x,t)}{\mathrm{d}x^2}\mathrm{e}^{-ikx}\,\mathrm{d}x \overset{p.p.}{=} -\frac{\hbar^2}{2m}ik\int_{-\infty}^{\infty}\frac{\mathrm{d}\psi(x,t)}{\mathrm{d}x}\mathrm{e}^{-ikx}\,\mathrm{d}x$$
$$\overset{p.p.}{=} \frac{\hbar^2}{2m}k^2\int_{-\infty}^{\infty}\psi(x,t)\mathrm{e}^{-ikx}\,\mathrm{d}x = \frac{\hbar^2k^2}{2m}\psi(k,t)\,. \tag{5.16}$$

where $p.p.$ signifies integration *per partes*. The result of $\hat{T}\psi$ in the k-space is very straightforward, just multiply $\psi(k,t)$ by the kinetic energy $\frac{\hbar^2k^2}{2m}$.

However, we are interested in the action of $\hat{T}$ in the position space since we work on our x-grid. This is easily achieved by evaluating the action of $\hat{T}$ in the k-space, as we showed in Eq. (5.16), and then transforming the result back to the position representation with the inverse Fourier transform (IFT) $\mathscr{F}^{-1}$ as

$$\hat{T}\psi(x,t) = \mathscr{F}^{-1}\left[\frac{\hbar^2k^2}{2m}\psi(k,t)\right]. \tag{5.17}$$

Once again, IFT is a routine operation performed on computers.

Let us now summarize the evaluation of $\hat{T}\psi(x,t)$. First, the wave function $\psi(x,t)$ has to be FT to the momentum representation $\psi(k,t)$. Then, the action of $\hat{T}$ on $\psi(k,t)$ is evaluated by simply multiplying it by $\frac{\hbar^2k^2}{2m}$. Finally, the IFT is applied to $\frac{\hbar^2k^2}{2m}\psi(k,t)$ transforming it back to the position space. The whole procedure is depicted in the following scheme:

$$\psi(x,t) \overset{FT}{\longrightarrow} \psi(k,t) \longrightarrow \frac{\hbar^2k^2}{2m}\psi(k,t) \overset{IFT}{\longrightarrow} \hat{T}\psi(x,t)\,. \tag{5.18}$$

5.2.3. The propagator and the Split-operator technique

As we discussed in section 5.1, evaluating the propagator is strenuous due to its exponential structure and the non-commutativity of the potential and kinetic operators. The non-commutativity is the major issue for us as it prevents us from factorizing the propagator as

$$\mathrm{e}^{-\frac{i}{\hbar}\hat{H}(x)\Delta t} \neq \mathrm{e}^{-\frac{i}{\hbar}V(x)t}\mathrm{e}^{-\frac{i}{\hbar}\hat{T}(x)t}\,. \tag{5.19}$$

If we could factorize the propagator as above, we could evaluate the exponential of $\hat{V}$ (in the position representation) and the exponential of $\hat{T}$ (in the momentum representation) separately. However, the non-commutativity of $\hat{V}$ and $\hat{T}$ does not allow for such factorization.

A way to deal with this problem is to split the propagator into a series of n short-time propagator

$$\hat{U}(t) = \mathrm{e}^{-\frac{i}{\hbar}\hat{H}(x)t} = \mathrm{e}^{-\frac{i}{\hbar}\hat{H}(x)\Delta t}\mathrm{e}^{-\frac{i}{\hbar}\hat{H}(x)\Delta t}\cdots\mathrm{e}^{-\frac{i}{\hbar}\hat{H}(x)\Delta t}, \tag{5.20}$$

where Δt is a fixed time step. Now, each individual short-time propagator acts on the wave function and causes a small alteration (understand short-time propagation). This operator can be approximated in the following way

$$\mathrm{e}^{-\frac{i}{\hbar}\hat{H}(x)\Delta t} = \mathrm{e}^{-\frac{i}{\hbar}V(x)\Delta t/2}\mathrm{e}^{-\frac{i}{\hbar}\hat{T}(x)\Delta t}\mathrm{e}^{-\frac{i}{\hbar}\hat{V}(x)\Delta t/2} + \mathscr{O}(\Delta t^3)\,, \tag{5.21}$$

where we symmetrically factorized the exponential of $\hat{H}$ into separate exponentials of $\hat{V}$ and $\hat{T}$. The factorization here is possible only if the time step Δt is short since the error behaves as Δt^3.

These exponentials now need to be evaluated. While the exponential of the potential energy is simply evaluated on the grid and multiplied by the wave function, the exponential of the kinetic energy operator must be evaluated using the Fourier method introduced in the previous subsection as

$$\psi(x,t) \overset{FT}{\longrightarrow} \psi(k,t) \longrightarrow \mathrm{e}^{\frac{i}{\hbar}\frac{\hbar^2k^2}{2m}t}\psi(k,t) \overset{IFT}{\longrightarrow} \mathrm{e}^{-\frac{i}{\hbar}\hat{T}t}\psi(x,t)\,. \tag{5.22}$$

The whole propagation is then done in a series of short time steps Δt. At each time step, the wave function is first propagated by a half step in potential energy, then a full step in kinetic energy and finally again a half step in potential energy.

5.3. Code

The numerical procedure for quantum dynamics described above can be readily implemented in Python or another suitable programming language. Let us start with a few short notes regarding a Python implementation. We will exploit the `numpy` library (imported as `np`). The Fourier transform can be conveniently performed as

```
np.fft.fft(psi, norm="ortho")
```

with `psi` being the wave function. Setting `norm="ortho"` employs the orthogonal norm. While we could use a different norm, we need to make sure we use the same norm for the IFT to ensure unitarity. The corresponding k-space is generated with

```
2*np.pi*np.fft.fftfreq(ngrid, d=dx)
```

where we supply the number of grid points (`ngrid`) and the distance between two neighbouring points (`dx`). The inverse Fourier transform can be calculated similarly:

```
f = np.fft.ifft(psi_k, norm="ortho")
```

The integration can be achieved with the trapezoidal rule in Python by selecting

```
np.trapz(y=np.conjugate(psi)*psi, x=x)
```

The trapezoidal rule is used here to compute the normalization of the wave function or to calculate expectation values by integrating over space.

Since the wave function is a complex function, we will also need to work with complex numbers in Python. Python works with `j` as the complex unit. The corresponding data type is called `complex`. Thus, creating an initial wave function can look like

```
psi = (2*alpha/np.pi)**0.25*np.exp(-alpha*(x-x0)**2+1j*p0*(x-x0), dtype=complex)
```

Follows the incomplete code for quantum dynamics with empty gaps where the students are supposed to define operators and propagate the wave function according to the method description above. The code is prepared in atomic units where $\hbar = 1$ a.u. and requires all the input parameters also in atomic units. The span of the x-axis, defined by `xmin` and `xmax`, should be large enough to contain the entire wave packet during propagation. The wave packet should never reach the boundaries defined by `xmin` and `xmax`. The number of grid points should be sufficiently big to represent the wave function. A few hundred to a few thousand grid points should suffice for simple dynamics. The time step `dt` should be significantly smaller than the time scale on which the wave packet evolves, which depends on its mass and potential. Always make sure the energy and norm are conserved during dynamics as they are indicators of a spurious propagation. Decreasing `dt` and increasing `ngrid` should always improve both energy and norm conservation.

```
"""Exercise for chapter Quantum dynamics: real-time quantum dynamics."""

# import necessary libraries
import matplotlib.pyplot as plt  # plotting
import numpy as np  # numerical python

# parameters of the simulation
ngrid = 500  # number of grid points
xmin, xmax = -15, 15  # minimum and maximum of x
```

```
simtime = 100.0  # simulation time in atomic time units
dt = 0.5  # time step in atomic time units
m = 1.0  # mass in atomic units

# physical constants in atomic units
hbar =  # fill in

# generate equidistant x grid from xmin to xmax
x =  # fill in (we recommend using function linspace from numpy)

# generate momentum grid for the discrete Fourier transform
dx =  # fill in distance between two neighbouring (x_{i+1}-x_{i})
k = 2*np.pi*np.fft.fftfreq(ngrid, d=dx)

# generate potential V
V =  # fill in

# generate kinetic energy T in momentum space
T =  # fill in

# generate V propagator with time step dt/2
expV =  # fill in

# generate T propagator in momentum space with time step dt
expT =  # fill in

# initiate wave function; be careful, it must be a complex numpy array
psi =  # fill in

# initiate online plotting to follow the wave function on the fly
plt.ion()
ymin, ymax = np.min(V), np.max(V) # plotting range of the vertical axis

# propagation
t = 0  # set initial time to 0
print("\nLaunching quantum dynamics.\n----------------------------\n")
while t < simtime:  # loop until simulation time is reached
    # propagate half-step in V
    # fill in

    # propagate full step in T
    # first Fourier transform to momentum space
    psi_k = np.fft.fft(psi, norm="ortho")
    # apply expT in momentum space
    # fill in
    # finally inverse Fourier transform back to coordinate space
    psi = np.fft.ifft(psi_k, norm="ortho")

    # propagate half-step in V for the second time
    # fill in

    # calculate new time after propagation
    t += dt

    # calculate norm
    norm =  # fill in

    # check that norm is conserved
    # fill in, choose a reasonable threshold, e.g. 0.00001

    # calculate expectation value of energy
    # potential energy <V>
    energyV =  # fill in
```

```
    # kinetic energy <T>, T operator will be again applied in the momentum space
    energyT =  # fill in
    # total energy <E>
    energy =  # fill in

    # print simulation data
    print(f"--Time: {t:.2f} a.t.u.")
    print(f"  <psi|psi> = {norm:.4f}, <E> = {energy:.4f}, <V> = {energyV:.4f}, <T> = {energyT:.4f}"
    )

    # plot
    plt.cla()  # clean figure
    plt.fill_between(x, np.abs(psi) + energy, energy, alpha=0.2, color='black', label=r"$|\Psi|$")
     # plot |wf|
    plt.plot(x, np.real(psi) + energy, linestyle='--', label=r"$Re[\Psi]$")  # plot Re(wf)
    plt.plot(x, np.imag(psi) + energy, linestyle='--', label=r"$Im[\Psi]$")  # plot Im(wf)
    plt.plot(x, V, color='black')  # plot potential
    plt.title(f"$E = $ {energy:.4f} a.u.")
    plt.xlabel("$x$ (a.u.)")
    plt.ylabel("$\Psi$")
    # set new ymin, ymax
    max_wf = np.max(np.abs(psi))
    ymin, ymax = min([ymin, -max_wf+energy]), max([ymax, max_wf+energy])
    plt.ylim(ymin, ymax)
    plt.legend(frameon=False)
    plt.pause(interval=0.01)  # update plot and wait given interval

# close online plotting
plt.pause(2.0)  # wait 2s before closing the window
plt.ioff()
plt.close()
```

Listing 5.1: Incomplete code for quantum dynamics in real time.

5.4. Applications

Exercise: Squeezed and coherent states of harmonic oscillator

The harmonic oscillator and Gaussian wave packet are some of the most important objects in quantum mechanics. While the harmonic oscillator is used as the first approximation to many bound Hamiltonians, the Gaussian wave packet is a commonly used basis function in quantum dynamics. There is a special relationship between them: any initial Gaussian wave function, $\psi(x,0)$, propagated in quadratic potential remains Gaussian throughout its evolution. In other words, a Gaussian wave packet evolving in a harmonic oscillator will retain its Gaussian form, with only its position, momentum, and width changing over time.

Gaussian wave packets evolving in harmonic oscillators can be classified into two groups: *coherent* and *squeezed* states. The *coherent* states are characterized by a constant width of the Gaussian during the dynamics, meaning the wave packet preserves its shape while its position and momentum vary. Contrarily, if the Gaussian spreads and contracts (or contracts and spreads) during the dynamics, it is referred to as a *squeezed* state.

In this exercise, we will explore the properties of *coherent* and *squeezed* states. We define the initial Gaussian wave packet as

$$\psi(x,0) = \left(\frac{2\alpha_0}{\pi}\right)^{-\frac{1}{4}} \mathrm{e}^{-\alpha_t(x-x_0)^2+\frac{i}{\hbar}p_0(x-x_0)}, \tag{5.23}$$

where x_0 and p_0 refer to the initial mean position and momentum of the Gaussian, and α_0 is the initial Gaussian width. The potential for harmonic oscillator takes the following form:

$$V = \frac{1}{2} m\omega^2 x^2 \,. \tag{5.24}$$

The aim is to determine the Gaussian width α_0 corresponding to a *coherent* state by trying different values of α_0.

Assignment: Set up the harmonic oscillator from Eq. (5.24) and initial Gaussian wave packet from Eq (5.23) with the following parameters: $m = 20$ a.u., $\omega = 2$ a.u., $x_0 = 0$ a.u. and $p_0 = 20$ a.u. Set up a trial α_0 (Hint: try α_0 as a factor of $\frac{m\omega}{\hbar}$). Propagate the wave packet for several oscillation and visualize it. If the wave packet contracts in the center of the potential, increase α_0. If the wave packet spreads, decrease α_0 to approach a coherent state. Iterate that procedure until you find the value of α_0 corresponding to the *coherent* state.

Exercise: Ehrenfest's theorems in harmonic oscillator

Ehrenfest's theorems provide a connection between classical and quantum dynamics. They represent equations of motion for expectation values of position $\langle x \rangle$ and momenta $\langle p \rangle$. For a harmonic oscillator, Ehrenfest's theorems take the following form:

$$\begin{aligned} \frac{\mathrm{d}\langle x \rangle}{\mathrm{d}t} &= \frac{\langle p \rangle}{m} \\ \frac{\mathrm{d}\langle p \rangle}{\mathrm{d}t} &= -\frac{\mathrm{d}V(\langle x \rangle)}{\mathrm{d}\langle x \rangle} \,. \end{aligned}$$

These equations are equivalent to the classical Newton's equations of motion, only for expectation values of position $\langle x \rangle$ and momenta $\langle p \rangle$ instead of classical x and p. This exercise aims to verify that the relations above hold true for harmonic oscillator and $\langle x \rangle$ and $\langle p \rangle$ follow a classical trajectory. Equally, one can consider a anharmonic potential, e.g. Morse potential, and show that the relations above are not exact.

Assignment: Choose an arbitrary initial wave function and harmonic potential. Propagate the wave function for several oscillations and calculate the expectation values of position $\langle x \rangle$ and momenta $\langle p \rangle$ over time. Compare the calculated values with a classical trajectory with the same initial conditions. The classical trajectory can be obtained by solving Newton's equations of motion. Repeat the same procedure for an anharmonic Morse potential and show that the relation above does not hold. Note that you will probably need to calculate the classical trajectory numerically, for example by a Verlet method.

Exercise: Tunnelling in a double-well potential

Tunnelling is an ubiquitous phenomenon in quantum world which is difficult to grasp from classical perspective. It plays an important role in chemistry where light atoms, such as protons, can tunnel through energy barriers causing energetically forbidden reactions. This exercise focuses at tunnelling through a double-well potential,

$$V = \frac{1}{2} m\omega^2 (x^4 - x^2) \,.$$

The wave function will be initiated at the edge of the left well, which will send it to the right towards the barrier. Two scenarios are possible. First, the energy of the wave packet is below the barrier. Then, no density should be observed at the other well from classical perspective. Thus, any density observed in the right well is due to quantum tunnelling. In the second scenario, the wave packet energy is above the barrier. Classical particles would then transfer to the other well without loss of density, yet this is again not true for the quantum world. The wave packet partially reflects on the barrier and some density remains in the original well.

To assess the amount of density transferred to the right well, we devise a coefficient κ defined as

$$\kappa(t) = \int_0^\infty \psi^*(x,t)\psi(x,t)\,\mathrm{d}x\,.$$

The value of κ will tell us how big portion of the density is located in the right well, i.e. about the tunnelling efficiency in the first scenario and reflective strength in the second scenario.

Assignment: Prepare a double well potential with parameters $m = 20$ a.u. and $\omega = 2$ a.u. First, initialize the dynamics with a wave function in form of Eq. (5.23) with parameters $x_0 = -0.90$ a.u., $p_0 = 0$ a.u. and $\alpha_0 = 2m\omega$ a.u. (the energy of the wave packet should be below the barrier). Second, initialize the dynamics with parameters $x_0 = -0.95$ a.u., $p_0 = 8$ a.u. and $\alpha_0 = 2m\omega$ a.u. (the energy should be above the barrier). Calculate the parameter κ for both scenarios and compare them with your expectations base on classical physics. Analyse the dynamics also visually by plotting the density in time. Con you identify interferences of the reflected and tunnelled wave packets?

Exercise: Heisenberg's uncertainity relations

The uncertainty principle developed by Werner Heisenberg is one of the corner stones of quantum mechanics. It states that

$$\Delta x \Delta p \geq \frac{\hbar}{2}\,,$$

where

$$\Delta x = \langle x^2\rangle - \langle x\rangle^2 = \langle\psi(x,t)|\,x^2\,|\psi(x,t)\rangle - \langle\psi(x,t)|\,x\,|\psi(x,t)\rangle^2$$
$$\Delta p = \langle p^2\rangle - \langle p\rangle^2 = \langle\psi(x,t)|\,\hat{p}^2\,|\psi(x,t)\rangle - \langle\psi(x,t)|\,\hat{p}\,|\psi(x,t)\rangle^2\,.$$

However, the uncertainty principle states only the lower boundary for $\Delta x\Delta p$ and does not tell us if it remains constant, rises or oscillates throughout the time evolution. The goal of this exercise is to empirically estimate whether the product $\Delta x\Delta p$ remains constant in time or not. If not, how does it behave?

Assignment: Evolve an arbitrary initial Gaussian wave packet in a harmonic oscillator and estimate the product $\Delta x\Delta p$ in time during the dynamics. Try the same also for a Morse potential and a double-well potential. Estimate how the product $\Delta x\Delta p$ behaves and if the behaviour differs between the potentials.

6. Wave packet spectrum and correlation function

7. Imaginary-time dynamics and stationary states

8. Nonadiabatic quantum dynamics

9. Bohmian dynamics

Solutions

Quantum dynamics

```
"""Solution for chapter Quantum dynamics: real-time quantum dynamics."""

# import necessary libraries
import matplotlib.pyplot as plt  # plotting
import numpy as np  # numerical python

# parameters of the simulation
ngrid = 500  # number of grid points
xmin, xmax = -15, 15  # minimum and maximum of x
simtime = 100.0  # simulation time in atomic time units
dt = 0.5  # time step in atomic time units
m = 1.0  # mass in atomic units

# physical constants in atomic units
hbar = 1.0

# generate equidistant x grid from xmin to xmax
x = np.linspace(xmin, xmax, ngrid)

# generate momentum grid for the discrete Fourier transform
dx = (xmax - xmin)/(ngrid - 1)
k = 2*np.pi*np.fft.fftfreq(ngrid, d=dx)

# generate potential V
V = 0.005*x**2

# generate kinetic energy T in momentum space
T = hbar**2*k**2/2/m

# generate V propagator with time step dt/2
expV = np.exp(-1j*V*dt/2)

# generate T propagator in momentum space with time step dt
expT = np.exp(-1j*T*dt)

# initiate wave function; be careful, it must be a complex numpy array
alpha, x0, p0 = 0.1, 0.5, 0.0
psi = (2*alpha/np.pi)**0.25*np.exp(-alpha*(x - x0)**2 + 1j*p0*(x - x0), dtype=complex)

# initiate online plotting to follow the wave function on the fly
plt.ion()
ymin, ymax = np.min(V), np.max(V)  # plotting range of the vertical axis

# propagation
t = 0  # set initial time to 0
print("\nLaunching quantum dynamics.\n--------------------------\n")
while t < simtime:  # loop until simulation time is reached
    # propagate half-step in V
    psi *= expV

    # propagate full step in T
```

```
    # first Fourier transform to momentum space
    psi_k = np.fft.fft(psi, norm="ortho")
    # apply expT in momentum space
    psi_k *= expT
    # finally inverse Fourier transform back to coordinate space
    psi = np.fft.ifft(psi_k, norm="ortho")

    # propagate half-step in V for the second time
    psi *= expV

    # calculate new time after propagation
    t += dt

    # calculate norm
    norm = np.real(np.trapz(y=np.conjugate(psi)*psi, x=x))

    # check that norm is conserved
    if np.abs(norm - 1) > 1e-5:
        print(f"ERROR: Norm ({norm:.9f}) is not conserved!")
        exit(1)

    # calculate expectation value of energy
    # potential energy <V>
    energyV = np.real(np.trapz(y=np.conjugate(psi)*V*psi, x=x))/norm
    # kinetic energy <T>, T operator will be again applied in the momentum space
    psi_k = np.fft.fft(psi, norm="ortho")
    psi_t = np.fft.ifft(T*psi_k, norm="ortho")
    energyT = np.real(np.trapz(y=np.conjugate(psi)*psi_t, x=x))/norm
    # total energy <E>
    energy = energyV + energyT

    # print simulation data
    print(f"--Time: {t:.2f} a.t.u.")
    print(f"  <psi|psi> = {norm:.4f}, <E> = {energy:.4f}, <V> = {energyV:.4f}, <T> = {energyT:.4f}"
    )

    # plot
    plt.cla()  # clean figure
    plt.fill_between(x, np.abs(psi) + energy, energy, alpha=0.2, color='black', label=r"$|\Psi|$")
     # plot |wf|
    plt.plot(x, np.real(psi) + energy, linestyle='--', label=r"$Re[\Psi]$")  # plot Re(wf)
    plt.plot(x, np.imag(psi) + energy, linestyle='--', label=r"$Im[\Psi]$")  # plot Im(wf)
    plt.plot(x, V, color='black')  # plot potential
    plt.title(f"$E = $ {energy:.4f} a.u.")
    plt.xlabel("$x$ (a.u.)")
    plt.ylabel("$\Psi$")
    # set new ymin, ymax
    max_wf = np.max(np.abs(psi))
    ymin, ymax = min([ymin, -max_wf + energy]), max([ymax, max_wf + energy])
    plt.ylim(ymin, ymax)
    plt.legend(frameon=False)
    plt.pause(interval=0.01)  # update plot and wait given interval

# close online plotting
plt.pause(2.0)  # wait 2s before closing the window
plt.ioff()
plt.close()
```

Listing 9.1: Solution: quantum dynamics in real-time

Literature

I. Electronic structure

- Szabo, A.; Ostlund, N. S. *Modern Quantum Chemistry : Introduction to Advanced Electronic Structure Theory*; Dover Publications: Mineola, N.Y., 1989.

II. Time-dependent quantum mechanics

- Tannor, D. J. *Introduction to Quantum Mechanics : A Time-Dependent Perspective*; University Science Books: Sausalito, Calif., 2007.
- Ratner, M. A.; Schatz, G. C. *Introduction to Quantum Mechanics in Chemistry*; Prentice Hall: Upper Saddle River, NJ, 2001.
- Schinke, R. *Photodissociation Dynamics: Spectroscopy and Fragmentation of Small Polyatomic Molecules*; Cambridge University Press: Cambridge [England], 1993.

Acknoledgement

?

List of acronyms

CC Coupled Cluster. 17, 18

CCD Coupled Cluster Doubles. 17, 18, 20, 22, 42

CCSD Coupled Cluster Singles and Doubles. 17, 18, 19, 20, 22, 42

CI Configuration Interaction. 12, 13, 14, 15, 16, 42

CISD Configuration Interaction Singles and Doubles. 12

CISDT Configuration Interaction Singles, Doubles and Triples. 12

FCI Full Configuration Interaction. 12, 13, 14

FT Fourier transform. 26, 27, 28

HF Hartree–Fock. 2, 3, 5, 6, 7, 8, 9, 10, 11, 14, 17, 18, 20, 42

IFT inverse Fourier transform. 27, 28

MP2 Møller–Plesset Perturbation Theory of 2nd Order. 10, 11, 42

MP3 Møller–Plesset Perturbation Theory of 3rd Order. 10, 11, 42

MPPT Møller–Plesset Perturbation Theory. 9

MS Molecular Spinorbital. 2, 4, 5, 7

post-HF post-Hartree–Fock. 4, 5, 12, 17

RHF Restricted Hartree–Fock. 2, 4

SCF Self-Consistent Field. 3

TDSE time-dependent Schrödinger equation. 24, 25

List of codes