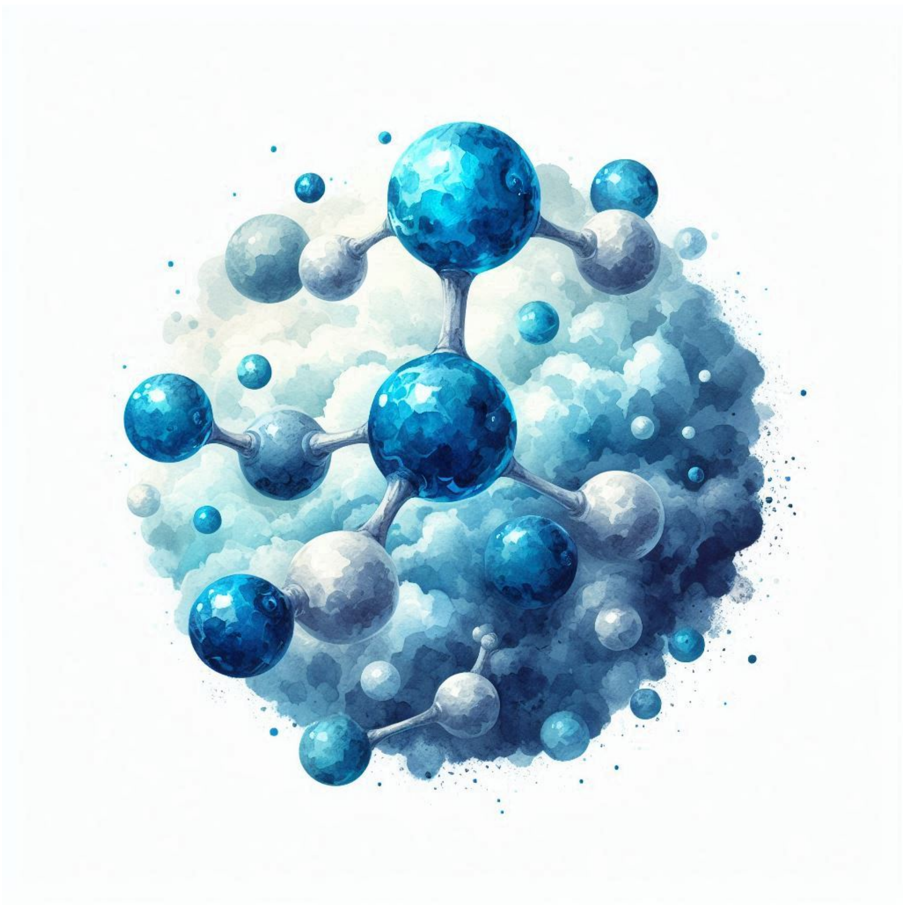
Quantum Mechanics in Chemistry: Hands on



Jiří Janoš, Tomáš Jíra

Created in LaTeX
© Prague 2024
Version from October 22, 2025

Contents

Introduction

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 a harmonic potential, rectangular well and others. It is 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 equations or are doing so routinely using quantum chemistry codes available to the community. However, only a small fraction of the current students have 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 an 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 looking at the equations and their implementation at least once since it provides valuable insights and forces one to truly understand what they are doing. Therefore, we prepared this short *Quantum Mechanics in Chemistry: 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 the 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

Python is a modern, general-purpose programming language widely used for scientific computing and numerical simulations. While it is not as fast as compiled languages like C++ or Fortran, its user-friendliness and vast ecosystem of libraries make it a popular choice, especially for beginners. In this text, we use Python to introduce numerical methods because it requires minimal prior programming experience and is easily understandable even for nonexperts. We assume a basic understanding of programming concepts such as conditions, loops and variables from the reader and discuss only more advanced concepts necessary for finishing the codes. If Python is new to the reader, we recommend exploring online tutorials like learnpython.org or taking beginner-friendly courses like Python Full Course for Beginners. AI tools like ChatGPT can also be valuable resources for generating code snippets and explaining unknown parts of codes. Below, we introduce the essentials needed to complete coding exercises, focusing on libraries, numerical computations, and data visualization.

Downloading and Importing Libraries

One of Python's greatest strengths is its extensive library ecosystem. Libraries can be easily installed using the `pip` package manager. For example, to install the NumPy library, type the following command in a terminal:

```
$ pip install numpy
```

To use a library in your Python code, import it at the beginning of your script. Here, we import the NumPy library and give it the alias `np` for convenience:

```
import numpy as np
```

The alias is then used to access the library and its functions. A whole list of functions of a given library can always be found online or printed with the `dir()` function:

```
print(dir(np))
```

Functions within a library can then be accessed using dot notation. For instance, the absolute value function in NumPy can be used as follows:

```
abs_value = np.abs(a)
```

We first specified the NumPy library using its alias `np`, the used the dot to access its functions and from those functions called the `abs` function. You can also refer to the library documentation for more details on its functions and capabilities.

Basic Python Syntax and Concepts

Before diving into numerical computations, let us revisit some basic Python concepts.

Data in Python are stored in variables. Python supports various data types like integers, floats, strings, and booleans. Here are some examples:

```
x = 10          # Integer
pi = 3.14159    # Float
name = ”Python” # String
is_fun = True   # Boolean
```

Use the `type()` function to check the type of a variable:

```
print(type(x))  # Output: <class ’int’>
```

Control Structures Python uses control structures like `if`, `for`, and `while` for decision-making and iteration:

```
# If-else statement
if x > 5:
    print(”x is greater than 5”)
else:
    print(”x is 5 or less”)

# For loop
for i in range(5):
    print(i)  # Outputs 0, 1, 2, 3, 4

# While loop
count = 0
while count < 5:
    print(count)
    count += 1
```

Functions Functions allow you to reuse code. Define a function using the `def` keyword:

```
def square(x):
    return x ** 2

# Call the function
result = square(4)  # Output: 16
```

NumPy

NumPy, short for *Numerical Python*, is a powerful library for numerical computations. It simplifies the implementation of mathematical operations, making it ideal for working with vectors, matrices, and arrays. Below, we demonstrate key features of NumPy.

Creating Arrays Arrays are the primary data structure in NumPy and can be created from Python lists:

```
# Creating a 1D array (vector)
vector = np.array([1, 2, 3, 4, 5, 6])

# Creating a 2D array (matrix)
matrix = np.array([[1, 2, 3], [4, 5, 6]])
```

You can create arrays with predefined values using helper functions like `np.zeros`, `np.ones`, and `np.arange`:

```
zeros = np.zeros(5)        # Array of five zeros
ones = np.ones((2, 3))     # 2x3 array of ones
sequence = np.arange(0, 10, 2)  # Array: [0, 2, 4, 6, 8]
```

Element-wise Operations Operations like addition, subtraction, multiplication, and division are performed element-wise on arrays:

```
c = vector * 2    # Multiply each element by 2
b = vector + 10   # Add 10 to each element
```

Array operations can also involve multiple arrays of the same size:

```
a = np.array([1, 2, 3])
b = np.array([4, 5, 6])

# Add corresponding elements of two arrays
sum_array = a + b
```

Dot and Cross Products To compute the *dot product* or *cross product* of two arrays, use the respective NumPy functions:

```
# Compute dot product
dot_product = np.dot(a, b)  # Result: 1*4 + 2*5 + 3*6 = 32

# Compute cross product
cross_product = np.cross(a, b)  # Result: [-3, 6, -3]
```

Mathematical Functions NumPy provides a wide range of mathematical functions, such as `sin`, `cos`, `exp`, and `sqrt`. These functions operate element-wise on arrays:

```
x = np.array([1, 2, 3])
square = x ** 2           # Square each element
sqrt_values = np.sqrt(x) # Compute square root of each element
```

Matplotlib

Matplotlib is a comprehensive library for creating static, animated, and interactive plots in Python. It is widely used for visualizing data.

Plotting a Simple Dataset Here is an example of how to create line and scatter plots:

```
import matplotlib.pyplot as plt

# Example data
a = np.array([1, 2, 3])
b = np.array([4, 5, 6])

# Plot a line
plt.plot(a, b, color='blue', linestyle='dashed', label='Line')

# Plot points
plt.scatter(a, b, color='red', label='Points')

# Add legend and show plot
plt.legend()
plt.show()
```

This script creates a plot with both a dashed line and red scatter points.

Customizing Plots You can enhance your plots by adding titles, axis labels, and gridlines:

```
plt.plot(a, b, color='green', label='Line')
plt.title("Example Plot")
plt.xlabel("X-axis")
plt.ylabel("Y-axis")
plt.grid(True)
plt.legend()
plt.show()
```

Complex Numbers

Python natively supports complex numbers. The imaginary unit is represented as `j`. You can create complex numbers as follows:

```
c = 1 + 2j
```

Python automatically recognizes `c` as a complex number. NumPy extends support for complex arithmetic, offering functions like `np.real` and `np.imag` to extract the real and imaginary parts, respectively:

```
real_part = np.real(c)  # Extract real part (1)
imag_part = np.imag(c)  # Extract imaginary part (2)
```

Part I.

Electronic structure theory

This part provides an educational exploration into the computational techniques fundamental to understanding molecular electronic structure in quantum chemistry. Beginning with the Hartree–Fock (HF) method, the text introduces this foundational approach for determining molecular orbitals and electronic energies by approximating the interactions of electrons through a mean-field approximation. The HF method forms the basis for subsequent methods and is presented with a practical coding exercise that guides readers in implementing and calculating HF energies in Python.

Moving beyond HF, the text delves into Møller–Plesset Perturbation Theory (MPPT), which improves HF predictions by introducing corrections for electron correlation through a perturbative approach. This section includes exercises on calculating second- and third-order corrections, allowing readers to enhance their understanding of how electron interactions can be more accurately incorporated. Configuration Interaction (CI) theory is then presented as an approach for representing the molecular wavefunction as a combination of electron configurations. Here, readers learn the theoretical basis of CI and engage with practical examples focused on constructing the CI Hamiltonian matrix and solving for molecular energies, particularly emphasizing Full Configuration Interaction (FCI) for high accuracy in small systems.

The part culminates with a discussion on the Coupled Cluster (CC) theory, a highly accurate and computationally efficient method for capturing electron correlation effects, often used for small to medium-sized systems. By introducing truncations such as Coupled Cluster Doubles (CCD) and Coupled Cluster Singles and Doubles (CCSD), the text demonstrates how electron correlation can be systematically included while balancing computational cost. The CC section provides iterative coding exercises for calculating correlation energies, rounding out the document's comprehensive approach to electronic structure methods in computational quantum chemistry. Through this blend of theory, mathematical formulations, and hands-on coding exercises, the document serves as an invaluable resource for building a strong foundational understanding of electronic structure methods.

1. Hückel theory

Hückel theory (HT) is one of the first and probably also the simplest methods to solve the electronic structure of molecules. It ranks among semiempirical quantum chemical methods as it contains empirical parameters parametrized to experiments or high-level calculations. The strength of HT lies in its simplicity because it allows to solve the secular equations using a pen-and-paper approach, which makes it excellent for education purposes. The downside of HT is, on the contrary, its applicability to only frontier molecular orbitals. The most famous applications of HT are unsaturated polyens such as buta-1,3-diene or benzene. Although these small molecules can be solved with only a pen and paper, HT is applicable also to larger systems such as molecular rotors or porphyrins. In this Chapter, we will use HT as a warm-up for the next more advanced electronic structure chapters and show how to calculate HT for an arbitrary conjugated planar system.

1.1 Theoretical background

The HT is a one-particle theory (mean-field such as Hartree–Fock) and considers the electronic Hamiltonian to be approximated as a sum of effective one-particle hamiltonians $\hat{h}_{i,\text{eff}}$:

$$\hat{H}_{\text{HT}} = \sum \hat{h}_{i,\text{eff}} \, . \tag{1.1}$$

The one-particle Hamiltonian is defined with an effective potential as

$$\hat{h}_{i,\text{eff}} = -\frac{1}{2}\nabla_i^2 - \sum_j \frac{Z_j}{|\vec{r}_i - \vec{R}_j|} + V(\vec{r}_{i,\text{eff}}) \, , \tag{1.2}$$

where j is the index of the nuclei, Z_j is the charge of a nucleus and $\vec{r}_i$ and $\vec{R}_j$ are the positions of the electrons and nuclei respectively.

The electronic wave function in HT is expressed as a linear combination of atomic orbitals ϕ_ν,

$$\psi = \sum_\nu c_\nu \phi_\nu \, , \tag{1.3}$$

where the index ν runs over all the basis functions. The spirit of the HT lies in the selection of the basis, which comprises only a selected set of the frontier atomic orbitals. For describing conjugated systems, only the p orbitals perpendicular to the conjugation plane are taken. This truncation is the core limitation of the theory as it cannot go beyond the selected set of frontier orbitals but it is also what greatly simplifies the working equations. Applying the variational principle to the wave function leads to secular equations, written in a matrix as

$$\mathbf{H}\vec{c} = \varepsilon \mathbf{S}\vec{c} , \tag{1.4}$$

where ε is the energy corresponding to the molecular orbital, $\mathbf{H}$ is the Hamiltoninan matrix with elements defined as expectation vlues of the one-electron Hamiltonian and the basis functions

$$H_{\nu\mu} = \langle \phi_\nu | \hat{h}_{i,\text{eff}} | \phi_\mu \rangle \, , \tag{1.5}$$

$\mathbf{S}$ is the overlap matrix,

$$S_{\nu\mu} = \langle \phi_\nu | \phi_\mu \rangle \, , \tag{1.6}$$

and $\vec{c}$ is the vector of the expansion coefficients.

The second series of approximations in HT (the first is the truncation of the basis set) is on the elements of the Hamiltonian and overlap matrices:

- The basis set is considered orthonormal. Thus, the overlap matrix is unitary, that is, $S_{\nu\mu} = \delta_{\nu\mu}$.
- The diagonal terms of the Hamiltonian matrix are equal to a predefined parameter, $H_{\nu\nu} = \alpha$.
- If the basis functions ϕ_ν and ϕ_μ reside in neighboring atoms, that is, atoms linked by a chemical bond, the Hamiltonian matrix element is set to another predefined parameter, $H_{\nu\mu} = \beta$.
- If the basis functions ϕ_ν and ϕ_μ reside in atoms that are not linked by a chemical bond, the Hamiltonian matrix element is zero, $H_{\nu\mu} = 0$.

The parameters α and β are usually parameterized from experiments. The value of α is $-11{,}4\,\mathrm{eV}$ which is the carbon ionization energy. The parameter β can acquire quite different values based on the parametrization. Using spectroscopic data gives $\beta = -3{,}48\,\mathrm{eV}$, while thermochemistry yields $\beta = -0{,}78\,\mathrm{eV}$. The parameterization depends on the purpose of our calculations. If we intend to calculate energy gaps between HOMO and LUMO orbitals, we will rather take the value obtained from spectroscopy and vice versa.

Taking all the approximations on the matrix elements together, the final working equations of the Hückel theory are

$$\mathbf{H}\vec{c} = \varepsilon\vec{c}, \tag{1.7}$$

We can recognize this as an eigenvalue equation which is solve by diagonalization of the Hamiltonian matrix. The energies ε are the eigenvalues and the expansion coefficients $\vec{c}$ are their corresponding eigenvectors. The Hamiltonian matrix has the parameter α on the diagonal and the parameter β at the place corresponding to the neighboring atoms.

To illustrate the form of $\mathbf{H}$ on a simple example, we will use the famous textbook case: buta-1,3-diene. If we label the atoms as $C_1 = C_2 - C_3 = C_4$, the Hamiltonian matrix reads

$$\mathbf{H} = \begin{pmatrix} \alpha & \beta & 0 & 0 \\ \beta & \alpha & \beta & 0 \\ 0 & \beta & \alpha & \beta \\ 0 & 0 & \beta & \alpha \end{pmatrix} \tag{1.8}$$

It is a common trick to make a substitution $\varepsilon = \alpha - \beta x$ and divide the both sides of the equation by β which leads to

$$\begin{pmatrix} 0 & 1 & 0 & 0 \\ 1 & 0 & 1 & 0 \\ 0 & 1 & 0 & 1 \\ 0 & 0 & 1 & 0 \end{pmatrix} \vec{c} = x\vec{c}. \tag{1.9}$$

The new matrix is then diagonalized with the eigenvalues corresponding to x and the eigenvectors still being the same expansion coefficients. The energies are recovered from $\varepsilon = \alpha - \beta x$. The main advantage is that this procedure does not require specifying α and β a priori. We calculate the energy levels in terms of parameters α and β for any system, and then we can try to parameterize them. Construction of the new matrix is even simpler than constructing the Hamiltonian matrix. The diagonal contains all zeros and all the elements corresponding to the neighbouring atoms are 1 instead of β.

1.2 Numerical implementation

The numerical implementation of Hückel theory is very simple and requires only diagonalization of either the Hamiltonian matrix or the matrix using 1 instead of β and 0 instead of α. The diagonalization can be performed with NumPy's routines such as `np.linalg.eig(H)` which provides both the eigenvalues and eigenvectors:

```
eigenvalues, eigenvectors = np.linalg.eigh(H)
```

where the eigenvectors are the columns of the eigenvector matrix, see the NumPy manual. Since the matrix is Hermitian, algorithms tailored for such matrices like `np.linalg.eigh(H)` can be used.

The implementation of HT is simple and we leave the coding fully to the student and provide only one possible author solution at the end of the book.

1.3 Applications

Exercise: Benchmarking the code on buta-1,3-diene

The Hückel theory for buta-1,3-diene yields the following orbital energies

$$\begin{aligned}
\varepsilon_1 &= \alpha + 1.62\beta\,, \\
\varepsilon_2 &= \alpha + 0.62\beta\,, \\
\varepsilon_3 &= \alpha - 0.62\beta\,, \\
\varepsilon_4 &= \alpha - 1.62\beta\,,
\end{aligned}$$

and wave functions

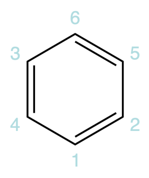
$$\begin{aligned}
\psi_1 &= 0.37\phi_1 + 0.60\phi_2 + 0.60\phi_3 + 0.37\phi_4\,, \\
\psi_2 &= 0.60\phi_1 + 0.37\phi_2 - 0.37\phi_3 - 0.60\phi_4\,, \\
\psi_3 &= 0.60\phi_1 - 0.37\phi_2 - 0.37\phi_3 + 0.60\phi_4\,, \\
\psi_4 &= 0.37\phi_1 - 0.60\phi_2 + 0.60\phi_3 - 0.37\phi_4\,.
\end{aligned}$$

In this Exercise, we will use this analytic result to benchmark the implemented code for HT.

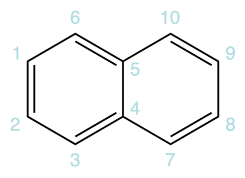
Assignment: Calculate the orbital energies ε and expansion coefficients $\vec{c}$, and compare them with the analytic solution to confirm a correct implementation of the Hückel theory. Additionaly, use the parameters $\alpha = -11{,}4\,\mathrm{eV}$ and $\beta = -3{,}48\,\mathrm{eV}$ to predict the excitation energy of buta-1,3-diene.

Exercise: Excitation energies of benzene, naphthalene and phenanthrene

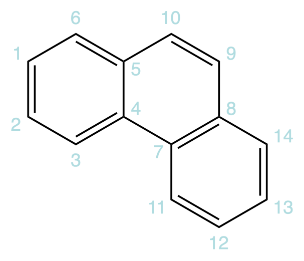
The Hückel theory can be applied effectively to conjugated systems of aromatic rings such as benzene, naphthalene or phenanthrene. Frontier molecular orbitals in these systems are composed of the π bonding and antibonding orbitals which can be well captured by HT. In this Exercise, we will try to predict the structure of the conjugated system of aromatic compounds and their excitation energies.



benzene



naphthalene

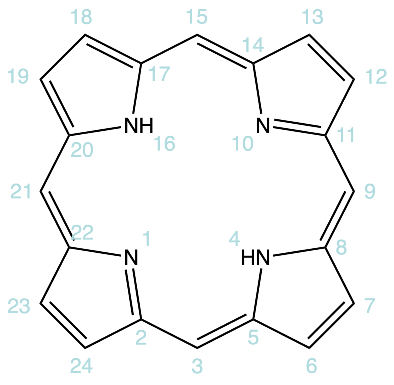


phenanthrene

Assignment: Calculate the orbital energies ε and expansion coefficients $\vec{c}$ for benzene, naphthalene and phenanthrene. Try to plot the orbitals and compare them with literature or Hartree–Fock calculations. Then, calculate the energy gap between the HOMO and LUMO orbitals using $\alpha = -11{,}4\,\mathrm{eV}$ and $\beta = -3{,}48\,\mathrm{eV}$ and predict excitation energies. Do they compare to experimental values? Do you see the same trends?

Exercise: Excitation energy of porphyrin and its HOMO and LUMO

Porphyrin is one of the building blocks of life and is often used in nature to capture photons. Its conjugated system of double bonds is responsible for a shift of the absorption spectrum to the visible region. Both HOMO and LUMO are mainly composed of p orbitals on carbons and nitrogen. In this Exercise, we will test the predictability of HT on bare porphyrin.



porphyrin

Assignment: Calculate the orbital energies ε and expansion coefficients $\vec{c}$ for porphyrin. Plot the orbitals and compare them with literature or Hartree–Fock calculations. Then, calculate the energy gap between HOMO and LUMO orbitals using $\alpha = -11{,}4\,\mathrm{eV}$ and $\beta = -3{,}48\,\mathrm{eV}$ and predict excitation energies. Do they compare to experimental values?

2. Hartree–Fock Method

The HF method is a foundational approach in quantum chemistry, aimed at solving the electronic structure problem in molecules by determining an optimal wavefunction. This method simplifies the complex interactions of electrons through a mean-field approximation, where each electron moves in an average field created by all others. This allows for the use of a single set of orbitals, leading to the construction of the Fock operator and iterative solutions to one-electron equations.

However, HF has notable limitations. Its reliance on a single-determinant wavefunction means it struggles to account for electron correlation. This leads to inaccuracies in energy predictions, particularly for systems with strong electron interactions, such as transition metal complexes or molecules with delocalized electrons.

2.1 Theoretical Background

Our primary objective is to solve the Schrödinger equation in the form

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

where $\hat{\mathbf{H}}$ denotes the molecular Hamiltonian operator, $\left|\Psi\right\rangle$ represents the molecular wavefunction, and E is the total energy of the system. The HF approximates the total wavefunction $\left|\Psi\right\rangle$ as a single Slater determinant, expressed as

$$\left|\Psi\right\rangle = \left|\chi_1 \chi_2 \cdots \chi_N\right\rangle \tag{2.2}$$

where χ_i denotes a spin orbital, and N is the total number of electrons. The goal of the HF method is to optimize these orbitals in order to minimize the system's total energy, thereby providing a reliable estimate of the electronic structure.

In the Restricted Hartree–Fock (RHF) method, we impose a constraint on electron spin, which allows us to work with spatial orbitals instead of spin orbitals. This reformulation expresses the Slater determinant in terms of spatial orbitals as

$$\left|\Psi\right\rangle = \left|\Phi_1 \Phi_2 \cdots \Phi_{N/2}\right\rangle \tag{2.3}$$

where Φ_i represents a spatial orbital. Notably, the RHF method requires an even number of electrons to satisfy spin-pairing. In practice, atomic orbitals (whether spin or spatial) are typically expanded in a set of basis functions $\{\phi_i\}$, which are often Gaussian functions, allowing for convenient computation with expansion coefficients. After performing some algebraic manipulations, the HF method leads to the Roothaan equations, which are expressed as

$$\mathbf{FC} = \mathbf{SC}\varepsilon \tag{2.4}$$

where $\mathbf{F}$ is the Fock matrix, $\mathbf{C}$ is the matrix of orbital coefficients, $\mathbf{S}$ is the overlap matrix, and ε represents the orbital energies. These matrices will be defined in detail later.

2.2 Implementation of the Restricted Hartree–Fock Method

To begin, we define the core Hamiltonian, also known as the one-electron Hamiltonian. This component of the full Hamiltonian excludes electron-electron repulsion and is expressed in index notation as

$$H_{\mu\nu}^{\text{core}} = T_{\mu\nu} + V_{\mu\nu} \tag{2.5}$$

where μ and ν are indices of the basis functions. Here, $T_{\mu\nu}$ represents a kinetic energy matrix element, while $V_{\mu\nu}$ denotes a potential energy matrix element. These matrix elements are defined as

$$T_{\mu\nu} = \langle\phi_\mu|\hat{T}|\phi_\nu\rangle \tag{2.6}$$

$$V_{\mu\nu} = \langle\phi_\mu|\hat{V}|\phi_\nu\rangle \tag{2.7}$$

Additionally, the overlap integrals, which describe the extent of overlap between basis functions, are given by

$$S_{\mu\nu} = \langle\phi_\mu|\phi_\nu\rangle \tag{2.8}$$

and another essential component are the two-electron repulsion integrals, defined as

$$J_{\mu\nu\kappa\lambda} = \langle\phi_\mu\phi_\mu|\hat{J}|\phi_\kappa\phi_\lambda\rangle \tag{2.9}$$

which play crucial roles in the HF calculation. All of these integrals over (Gausssian) basis functions are usually calculated using analytical expressions.[1] The solution to the Roothaan equations (2.4) requires an iterative procedure, since the Fock matrix defined as

$$F_{\mu\nu} = H_{\mu\nu}^{\text{core}} + D_{\kappa\lambda}\left(J_{\mu\nu\kappa\lambda} - \frac{1}{2}J_{\mu\lambda\kappa\nu}\right) \tag{2.10}$$

depends on the unknown density matrix $\mathbf{D}$, defined as

$$D_{\mu\nu} = 2C_{\mu i}C_{\nu i} \tag{2.11}$$

This iterative procedure is executed through the Self-Consistent Field (SCF) method. An initial guess is made for the density matrix $\mathbf{D}$ (usually zero), the Roothaan equations (2.4) are solved and the density matix is updated using the equation (2.11). The total energy of the system is then calculated using the core Hamiltonian and the Fock matrix as

$$E = \frac{1}{2}D_{\mu\nu}(H_{\mu\nu}^{\text{core}} + F_{\mu\nu}) + E_{\text{nuc}} \tag{2.12}$$

After convergence of both the density matrix and total energy, the process concludes, yielding the optimized molecular orbitals. The total energy of the system also includes the nuclear repulsion energy, which is given by

$$E_{\text{nuc}} = \sum_A \sum_{B<A} \frac{Z_A Z_B}{R_{AB}} \tag{2.13}$$

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

2.2.1 Direct Inversion in the Iterative Subspace

In the HF method, convergence of the density matrix and energy can be significantly accelerated by employing the Direct Inversion in the Iterative Subspace (DIIS) technique. DIIS achieves this by storing Fock matrices from previous iterations and constructing an optimized linear combination that minimizes the current iteration's error. This approach is especially valuable in HF calculations for large systems, where convergence issues are more common and challenging to resolve.

We start by defining the error vector of i-th iteration $\mathbf{e}_i$ as

$$\mathbf{e}_i = \mathbf{S}_i\mathbf{D}_i\mathbf{F}_i - \mathbf{F}_i\mathbf{D}_i\mathbf{S}_i \tag{2.14}$$

Our goal is to transform the Fock matrix $\mathbf{F}_i$ as

$$\mathbf{F}_i = \sum_{j=i-(L+1)}^{i-1} c_j\mathbf{F}_j \tag{2.15}$$

where c_j are the coefficients that minimize the error matrix $\mathbf{e}_i$ and L is the number of Fock matrices we store called the subspace size. To calculate the coefficients c_j, we solve the set linear equations

$$\begin{bmatrix} \mathbf{e}_1\cdot\mathbf{e}_1 & \dots & \mathbf{e}_1\cdot\mathbf{e}_{L+1} & 1 \\ \vdots & \ddots & \vdots & \vdots \\ \mathbf{e}_{L+1}\cdot\mathbf{e}_1 & \dots & \mathbf{e}_{L+1}\cdot\mathbf{e}_{L+1} & 1 \\ 1 & \dots & 1 & 0 \end{bmatrix} \begin{bmatrix} c_1 \\ \vdots \\ c_{L+1} \\ \lambda \end{bmatrix} = \begin{bmatrix} 0 \\ \vdots \\ 0 \\ 1 \end{bmatrix} \tag{2.16}$$

After solving the linear equations, we use the coefficients c_j to construct the new Fock matrix $\mathbf{F}_i$ according to the equation (2.15) and proceed as usual with the HF calculation.

2.2.2 Gradient of the Restricted Hartree–Fock Method

If we perform the calculation as described above and get the density matrix $\mathbf{D}$ we can evaluate the nuclear energy gradient as[2]

(2.17)

where A is the index of an atom, i is the index of the coordinate, $\{\phi_A\}$ is the set of all basis functions located at atom A and $\mathbf{W}$ is energy weighed density matrix defined as

$$W_{\mu\nu} = 2C_{\mu i}C_{\nu i}\varepsilon_i \tag{2.18}$$

Keep in mind that the indices κ and λ in the gradient equation (2.17) are summed over all basis functions.

2.3 Integral Transforms to the Basis of Molecular Spinorbitals

To carry out most post-Hartree–Fock (post-HF) calculations, it is essential to transform the integrals into the Molecular Spinorbital (MS) basis. We will outline this transformation process here and refer to it in subsequent sections on post-HF methods. The post-HF methods in this document will be presented using the integrals in the MS basis (and in its antisymmetrized form for the two-electron integrals) as this approach is more general.

All the integrals defined in the equations (2.5), (2.8), and (2.9), as well as Fock matrix in the equation (2.10) are defined in the basis of atomic orbitals. To transform these integrals to the MS basis, we utilize the coefficient matrix $\mathbf{C}$ obtained from the solution of the Roothaan equations (2.4). This coefficient matrix $\mathbf{C}$ is initially calculated in the spatial molecular orbital basis (in the RHF calculation).

The first step involves expanding the coefficient matrix $\mathbf{C}$ to the MS basis. This transformation can be mathematically expressed using the tiling matrix $\mathbf{P}_{n\times 2n}$, defined as

$$\mathbf{P} = \begin{pmatrix} e_1 & e_1 & e_2 & e_2 & \dots & e_n & e_n \end{pmatrix} \tag{2.19}$$

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

$$M_{ij} = 1 - j \bmod 2, N_{ij} = j \bmod 2 \tag{2.20}$$

The coefficient matrix $\mathbf{C}$ in the MS basis can be then expressed as

$$\mathbf{C}^{\text{MS}} = \begin{pmatrix} \mathbf{CP} \\ \mathbf{CP} \end{pmatrix} \odot \begin{pmatrix} \mathbf{M} \\ \mathbf{N} \end{pmatrix} \tag{2.21}$$

where $\odot$ denotes the Hadamard product. This transformed matrix $\mathbf{C}^{\text{MS}}$ is subsequently used to transform the two-electron integrals $\mathbf{J}$ to the MS basis as

$$J_{pqrs}^{\text{MS}} = C_{\mu p}^{\text{MS}} C_{\nu q}^{\text{MS}} (\mathbf{I}_2 \otimes_K (\mathbf{I}_2 \otimes_K \mathbf{J})^{(4,3,2,1)})_{\mu\nu\kappa\lambda} C_{\kappa r}^{\text{MS}} C_{\lambda s}^{\text{MS}} \tag{2.22}$$

where the superscript $(4,3,2,1)$ denotes the axes transposition and $\otimes_K$ is the Kronecker product. This notation accommodates the spin modifications and ensures adherence to quantum mechanical principles. We also define the antisymmetrized two-electron integrals in physicists' notation as

$$\langle pq||rs\rangle = (J_{pqrs}^{\text{MS}} - J_{psrq}^{\text{MS}})^{(1,3,2,4)} \tag{2.23}$$

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

$$A_{pq}^{\text{MS}} = C_{\mu p}^{\text{MS}} (\mathbf{I}_2 \otimes_K \mathbf{A})_{\mu\nu} C_{\nu q}^{\text{MS}} \tag{2.24}$$

where $\mathbf{A}$ is an arbitrary matrix of one-electron integrals. Since many post-HF methods rely on differences of orbital energies in the denominator, we define the tensors

$$\varepsilon_i^a = \varepsilon_i - \varepsilon_a \tag{2.25}$$
$$\varepsilon_{ij}^{ab} = \varepsilon_i + \varepsilon_j - \varepsilon_a - \varepsilon_b \tag{2.26}$$
$$\varepsilon_{ijk}^{abc} = \varepsilon_i + \varepsilon_j + \varepsilon_k - \varepsilon_a - \varepsilon_b - \varepsilon_c \tag{2.27}$$

for convenience. These tensors enhance code readability and efficiency, making it easier to understand and work with the underlying mathematical framework. Here and also throughout the rest of the document, the indices i, j and k run over occupied orbitals, whereas the indices a, b and c run over virtual orbitals.

2.4 Hartree–Fock Method and Integral Transform Coding Exercise

This section provides Python code snippets for implementing the HF method and transforming integrals to the MS basis. The code uses the NumPy library for efficient numerical computations, and the exercises are designed to build familiarity with the HF method and the MS integral transformations.

Each exercise includes placeholders for you to fill. It is assumed that the variables `atoms`, `coords`, `S`, `H`, and `J` are defined before the exercises. These variables represent the atomic numbers, atomic coordinates, overlap matrix, core Hamiltonian, and two-electron integrals, respectively. These variables can typically be obtained from the output of a quantum chemistry software package. If you would like to focus solely on coding you can save the molecule file, overlap integrals, core Hamiltonian, and two-electron integrals in the STO-3G basis to the same directory as the exercise code and load the variables using the Listing 2.1 below. The `ATOM` variable is a dictionary that maps the atomic symbols to atomic numbers.

```python
# 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]])
```

Listing 2.1: Example loading of molecule and integrals over atomic basis functions into variables used throughout exercises. The snippet expects the molecule and integral files to b present in the same directory as the script.

With all the variables defined, you can proceed to the HF and integral transform exercises in the Listings 2.2 and 2.3 below.

```python
"""
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 2.2: HF method exercise code. The important variables like number of occupied orbitals, convergence threshold and matrix containers are already defined. The student is expected to fill the SCF loop and calculate nuclear repulsion energy from the atomic numbers and coordinates. The total energy is then automatically printed.

```
"""
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 2.3: Integral transform exercise code. The tiling matrices are predefined here with a correct shape, but the student is expected to fill the with the correct expressions. After that, the student should transform the coefficient matrix, core Hamoltonian, Fock matrix and two-electron integrals to the MS basis. Additionally, orbital energy tensor are also expected to be calculated.

Solution to this exercises can be found in Section 8.5 and in Section 8.5.

3. 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.

3.1 Theory of the Perturbative Approach

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

$$\hat{\mathbf{H}} \left|\Psi\right\rangle = E \left|\Psi\right\rangle \tag{3.1}$$

where $\hat{\mathbf{H}}$ is the molecular Hamiltonian operator, $\left|\Psi\right\rangle$ 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

$$\hat{\mathbf{H}} = \hat{\mathbf{H}}^{(0)} + \lambda \hat{\mathbf{H}}^{'} \tag{3.2}$$

where $\hat{\mathbf{H}}^{(0)}$ is the Hamiltonian used in the HF method (representing electrons 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 not included in the HF approximation. We then expand the wavefunction $\left|\Psi\right\rangle$ and total energy E as a power series in λ as

$$\left|\Psi\right\rangle = \left|\Psi^{(0)}\right\rangle + \lambda \left|\Psi^{(1)}\right\rangle + \lambda^2 \left|\Psi^{(2)}\right\rangle + \dots \tag{3.3}$$

$$E = E^{(0)} + \lambda E^{(1)} + \lambda^2 E^{(2)} + \dots \tag{3.4}$$

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

$$E_{\text{corr}}^{\text{MP2}} = \sum_{s>0} \frac{H_{0s}^{'} H_{s0}^{'}}{E_0 - E_s} \tag{3.5}$$

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.[3, 4] 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 Hartree-Fock calculation. The expressions for calculation are summarised below.

3.2 Implementation of 2nd and 3rd Order Corrections

Having the antisymmetrized two-electron integrals in the MS basis and physicists' notation defined in Section 2.3, we can now proceed with the calculation of the correlation energy. The 2nd order correlation energy can be expressed as

$$E_{\text{corr}}^{\text{MP2}} = \frac{1}{4}\sum_{ijab} \frac{\langle ab||ij\rangle \langle ij||ab\rangle}{\varepsilon_{ij}^{ab}} \tag{3.6}$$

and the 3rd order correlation energy as

$$\begin{aligned} E_{\text{corr}}^{\text{MP3}} =& \frac{1}{8}\sum_{ijab} \frac{\langle ab||ij\rangle \langle cd||ab\rangle \langle ij||cd\rangle}{\varepsilon_{ij}^{ab}\varepsilon_{ij}^{cd}} + \\ &+ \frac{1}{8}\sum_{ijab} \frac{\langle ab||ij\rangle \langle ij||kl\rangle \langle kl||ab\rangle}{\varepsilon_{ij}^{ab}\varepsilon_{kl}^{ab}} + \\ &+ \sum_{ijab} \frac{\langle ab||ij\rangle \langle cj||kb\rangle \langle ik||ac\rangle}{\varepsilon_{ij}^{ab}\varepsilon_{ik}^{ac}} \end{aligned} \tag{3.7}$$

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.[4–6]

3.3 2nd and 3rd Order Corrections Code Exercise

Similar to the HF method, Møller–Plesset perturbation theory can also be implemented in Python. The code exercise below provides a self-contained guide to calculating the Møller–Plesset Perturbation Theory of 2nd Order (MP2) and Møller–Plesset Perturbation Theory of 3rd Order (MP3) correlation energies. This exercise is designed to be appended to your existing HF implementation, as the MP2 and MP3 methods build on the results of the HF procedure. You can access the foundational HF method and integral transformation coding exercise in Section 2.4. The exercise is provided in the Listing 3.1 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 3.1: MP2 and MP3 exercise code. The placeholders for the energies are initialized to zero. If you transformed all the necessary integrals in the previous exercise, you should be able to fill the placeholders with correct expressions.

Solution to this exercise can be found in Section 8.5.

4. 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 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.

4.1 Theoretical Background of General Configuration Interaction

In CI theory, we expand the wavefunction $|\Psi\rangle$ in terms of the HF reference determinant and its excited configurations as

$$|\Psi\rangle = c_0 |\Psi_0\rangle + \left(\frac{1}{1!}\right)^2 c_i^a |\Psi_i^a\rangle + \left(\frac{1}{2!}\right)^2 c_{ij}^{ab} |\Psi_{ij}^{ab}\rangle + \left(\frac{1}{3!}\right)^2 c_{ijk}^{abc} |\Psi_{ijk}^{abc}\rangle + \dots \tag{4.1}$$

where we seek the coefficients $\mathbf{c}$ that minimize the energy. To determine these coefficients, we construct and diagonalize the Hamiltonian matrix in the basis of these excited determinants. The CI Hamiltonian matrix $\mathbf{H}^{\text{CI}}$ is represented as

$$\mathbf{H}^{\text{CI}} = \begin{bmatrix} \langle\Psi_0|\hat{H}|\Psi_0\rangle & \langle\Psi_0|\hat{H}|\Psi_i^a\rangle & \langle\Psi_0|\hat{H}|\Psi_{ij}^{ab}\rangle & \dots \\ \langle\Psi_i^a|\hat{H}|\Psi_0\rangle & \langle\Psi_i^a|\hat{H}|\Psi_i^a\rangle & \langle\Psi_i^a|\hat{H}|\Psi_{ij}^{ab}\rangle & \dots \\ \langle\Psi_{ia}^{jb}|\hat{H}|\Psi_0\rangle & \langle\Psi_{ia}^{jb}|\hat{H}|\Psi_1\rangle & \langle\Psi_{ia}^{jb}|\hat{H}|\Psi_{ia}^{jb}\rangle & \dots \\ \vdots & \vdots & \vdots & \ddots \end{bmatrix} \tag{4.2}$$

After the Hamiltonian matrix is constructed, we solve the eigenvalue problem

$$\mathbf{H}^{\text{CI}}\mathbf{C}^{\text{CI}} = \mathbf{C}^{\text{CI}}\boldsymbol{\varepsilon}^{\text{CI}} \tag{4.3}$$

where $\mathbf{C}^{\text{CI}}$ is a matrix of coefficients and $\boldsymbol{\varepsilon}^{\text{CI}}$ is a diagonal matrix of eigenvalues. The lowest eigenvalue gives the ground-state energy, and the corresponding eigenvector provides the coefficients that minimize the energy. The elements of the CI Hamiltonian matrix are computed using the Slater–Condon rules, summarized in one function as

$$\mathbf{H}_{ij}^{\mathrm{CI}} = \begin{cases} \sum_k H_{kk}^{\mathrm{core,MS}} + \frac{1}{2}\sum_k\sum_l \langle kl||kl\rangle & D_i = D_j \\ H_{pr}^{\mathrm{core,MS}} + \sum_k \langle pk||lk\rangle & D_i = \{\cdots p \cdots\} \land D_j = \{\cdots r \cdots\} \\ \langle pq||rs\rangle & D_i = \{\cdots p \cdots q \cdots\} \land D_j = \{\cdots r \cdots s \cdots\} \\ 0 & \text{otherwise} \end{cases} \tag{4.4}$$

where D_i and D_j are Slater determinants, $\mathbf{H}^{\mathrm{core,MS}}$ is the core Hamiltonian in the MS basis, and $\langle pk||lk\rangle$ are the antisymmetrized two-electron integrals in MS basis and physicists' notation. The sums extend over all spinorbitals common between the two determinants. These integrals were previously transformed in Section 2.3. 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.

An important caveat in CI theory is its lack of size-extensivity, which implies that the energy does not scale linearly with the number of electrons. This drawback stems from the fact that the CI wavefunction is not size-consistent, meaning the energy of a combined system is not simply the sum of the energies of its isolated parts. This limitation restricts the application of CI mainly to small molecular systems.

4.2 Full Configuration Interaction Implementation

In FCI, we aim to account for all possible electronic configurations within a chosen basis set, offering the most accurate wavefunction representation for the given basis. Although this method yields highly precise electronic structure information, it is computationally intensive. Its cost scales exponentially with both the number of electrons and basis functions, limiting its feasibility to smaller systems.

The FCI process involves constructing all possible Slater determinants for a system. For simplicity, we'll assume that we want to include both singlet and triplet states in our determinant space. The total number of these determinants N_D can be calculated using binomial coefficients

$$N_D = \binom{n}{k} \tag{4.5}$$

where k is the total number of electrons, and n is the total number of 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 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 (4.2) can be constructed, and the eigenvalue problem (4.3) can be solved to obtain the ground and excited state energies.

4.3 Full Configuration Interaction Code Exercise

The FCI example builds on the HF method and demonstrates how to implement a FCI calculation in Python using NumPy. This exercise focuses on generating determinants, constructing the CI Hamiltonian matrix using Slater–Condon rules, and solving the eigenvalue problem to obtain the ground state energy. The code is designed for educational purposes and is based on prior HF results that can be done in Section 2.4. The exercise is provided in the Listing 4.1 below.

```python
"""
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

        """
        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.
        """
        aligned = aligned

        """
        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.
        """
        so = list()

        """
        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.
        """
        H[i, j] = 0

"""
You can finally solve the eigenvalue problem. Please, assign the correlation energy to the "E_FCI"
    variable.
"""
E_FCI = 0

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

Listing 4.1: CI exercise code. Here, student is expected to calculate the CI ground state energy. The task here is to fill the CI Hamiltonian using the Slater–Condon rules and diagonalize it.

Solution to this exercise can be found in Section 8.5.

5. Coupled Cluster Theory

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 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. 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. ## CC Formalism

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

$$|\Psi\rangle = e^{\hat{\mathbf{T}}} |\Psi_0\rangle \tag{5.1}$$

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

$$\hat{\mathbf{T}} = \hat{\mathbf{T}}_1 + \hat{\mathbf{T}}_2 + \hat{\mathbf{T}}_3 + \dots \tag{5.2}$$

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

$$\hat{\mathbf{T}}_1 |\Psi_0\rangle = \left(\frac{1}{1!}\right)^2 t_i^a |\Psi_i^a\rangle \tag{5.3}$$

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

$$\hat{\mathbf{H}} |\Psi\rangle = E |\Psi\rangle \tag{5.4}$$

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

$$e^{-\hat{\mathbf{T}}} \hat{\mathbf{H}} e^{\hat{\mathbf{T}}} |\Psi_0\rangle = E |\Psi_0\rangle \tag{5.5}$$

because we can then express the CC energy as

$$E = \langle\Psi_0| e^{-\hat{\mathbf{T}}} \hat{\mathbf{H}} e^{\hat{\mathbf{T}}} |\Psi_0\rangle \tag{5.6}$$

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.

5.1 Implementation of Truncated Coupled Cluster Methods

We will not go into the details here, but we will provide the final expressions for the CCD and CCSD methods.[7] 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 two-electron integrals in the MS basis and physicists' notation, Fock matrix in the MS basis and the orbital energy tensors obtained from the HF calculation. All these transformations are already explained in Section 2.3. The expressions for the CCD can be written as

$$E_{\text{CCD}}=\frac{1}{4}\left\langle ij||ab\right\rangle t_{ij}^{ab} \tag{5.7}$$

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

$$\begin{aligned}t_{ij}^{ab}=&\left\langle ab||ij\right\rangle+\frac{1}{2}\left\langle ab||cd\right\rangle t_{cd}^{ij}+\frac{1}{2}\left\langle kl||ij\right\rangle t_{ab}^{kl}+\hat{P}_{(a/b)}\hat{P}_{(i/j)}\left\langle ak||ic\right\rangle t_{cb}^{ij}-\\&-\frac{1}{2}\hat{P}_{(a/b)}\left\langle kl||cd\right\rangle t_{ac}^{ij}t_{bd}^{kl}-\frac{1}{2}\hat{P}_{(i/j)}\left\langle kl||cd\right\rangle t_{ab}^{ik}t_{cd}^{jl}+\\&+\frac{1}{4}\left\langle kl||cd\right\rangle t_{cd}^{ij}t_{ab}^{kl}+\hat{P}_{(i/j)}\left\langle kl||cd\right\rangle t_{ac}^{ik}t_{bd}^{jl}\end{aligned} \tag{5.8}$$

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

$$E_{\text{CCSD}}=F_{ia}^{\text{MS}}t_{a}^{i}+\frac{1}{4}\left\langle ij||ab\right\rangle t_{ij}^{ab}+\frac{1}{2}\left\langle ij||ab\right\rangle t_{i}^{a}t_{b}^{j} \tag{5.9}$$

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

$$\mathscr{F}_{ae}=\left(1-\delta_{ae}\right)F_{ae}-\frac{1}{2}F_{me}t_{m}^{a}+t_{m}^{f}\left\langle ma||fe\right\rangle-\frac{1}{2}\tilde{\tau}_{mn}^{af}\left\langle mn||ef\right\rangle \tag{5.10}$$

$$\mathscr{F}_{mi}=\left(1-\delta_{mi}\right)F_{mi}+\frac{1}{2}F_{me}t_{i}^{e}+t_{n}^{e}\left\langle mn||ie\right\rangle+\frac{1}{2}\tilde{\tau}_{in}^{ef}\left\langle mn||ef\right\rangle \tag{5.11}$$

$$\mathscr{F}_{me}=F_{me}+t_{n}^{f}\left\langle mn||ef\right\rangle \tag{5.12}$$

$$\mathscr{W}_{mnij}=\left\langle mn||ij\right\rangle+\hat{P}_{(i/j)}t_{j}^{e}\left\langle mn||ie\right\rangle+\frac{1}{4}\tau_{ij}^{ef}\left\langle mn||ef\right\rangle \tag{5.13}$$

$$\mathscr{W}_{abef}=\left\langle ab||ef\right\rangle-\hat{P}_{(a/b)}t_{m}^{b}\left\langle am||ef\right\rangle+\frac{1}{4}\tau_{mn}^{ab}\left\langle mn||ef\right\rangle \tag{5.14}$$

$$\mathscr{W}_{mbej}=\left\langle mb||ej\right\rangle+t_{j}^{f}\left\langle mb||ef\right\rangle-t_{n}^{b}\left\langle mn||ej\right\rangle-\left(\frac{1}{2}t_{jn}^{fb}+t_{j}^{f}t_{n}^{b}\right)\left\langle mn||ef\right\rangle \tag{5.15}$$

and two-particle excitation operators as

$$\tilde{\tau}_{ij}^{ab} = t_{ij}^{ab} + \frac{1}{2}\left(t_i^a t_j^b - t_i^b t_j^a\right) \tag{5.16}$$
$$\tau_{ij}^{ab} = t_{ij}^{ab} + t_i^a t_j^b - t_i^b t_j^a \tag{5.17}$$

The CCSD single excitations amplitude equations are then given by

$$\begin{aligned} t_i^a =& F_{ai}^{\mathrm{MS}} + t_i^e \mathscr{F}_{ae} - t_m^a \mathscr{F}_{mi} t_{im}^{ae} \mathscr{F}_{me} - t_n^f \langle na||if\rangle - -\frac{1}{2} t_{im}^{ef} \langle ma||ef\rangle - \\ & - \frac{1}{2} t_{mn}^{ae} \langle nm||ei\rangle \end{aligned} \tag{5.18}$$

and the CCSD double excitations amplitude equations are given by

$$\begin{aligned} t_{ij}^{ab} =& \langle ab||ij\rangle + \hat{P}_{(a/b)} t_{ij}^{ae}\left(\mathscr{F}_{be} - \frac{1}{2} t_m^b \mathscr{F}_{ae}\right) - \hat{P}_{(i/j)} t_{im}^{ab}\left(\mathscr{F}_{mi} + \frac{1}{2} t_j^e \mathscr{F}_{me}\right) + \\ & + \frac{1}{2}\tau_{mn}^{ab} \mathscr{W}_{mnij} + \frac{1}{2}\tau_{ij}^{ef} \mathscr{W}_{abef} + \hat{P}_{(i/j)}\hat{P}_{(a/b)}\left(t_{im}^{ae} \mathscr{W}_{mbej} - t_i^e t_m^a \langle mb||ej\rangle\right) + \\ & + \hat{P}_{(i/j)} t_i^e \langle ab||ej\rangle - \hat{P}_{(a/b)} t_m^a \langle mb||ij\rangle \end{aligned} \tag{5.19}$$

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.

5.2 Coupled Cluster Singles and Doubles Code Exercise

After completing the HF implementation in Section 2.4, you can proceed with coding the CCSD exercise, which builds on the HF results. The exercise is provided in the Listing 5.1 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 5.1: CCSD exercise code. The energy and amplitudes are initialized with default values. The student is expected to fill the loop for the calculation of the excitation amplitudes and ground state energy. After the self-consistency is achieved the result is automatically printed.

Solution to this exercise can be found in Section 8.5.

Part II.

Time-dependent quantum mechanics

6. 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 move 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 numerical scheme suitable for wave function propagation. The power of 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.

6.1 Theoretical background

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

$$i\hbar\frac{\mathrm{d}\psi(x,t)}{\mathrm{d}t} = \hat{H}(x)\psi(x,t)\,, \tag{6.1}$$

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

$$\psi(x,t) = \hat{U}(t)\psi(x,0)\,. \tag{6.2}$$

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, critical 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 (6.1) in the following form:

$$\psi(x,t) = \mathrm{e}^{-\frac{i}{\hbar}\hat{H}(x)t}\psi(x,0)\,. \tag{6.3}$$

Notice that the result resembles the separation of variables for differential equations, only for an equation with operators. Considering the Hamiltonian to be time-independent restricts the validity of equation (6.3) only for systems without any time-dependent interaction, such as oscillating electromagnetic field (radiation), collisions between particles, interaction with the environment, etc. The solution for a time-dependent Hamiltonian is somewhat more complicated and will not be considered here.

Comparing equations (6.3) and (6.2), we see that the evolution operator has a simple form of an exponential of the Hamiltonian,

$$\hat{U}(t) = \mathrm{e}^{-\frac{i}{\hbar}\hat{H}(x)t}\,. \tag{6.4}$$

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 (6.4).

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

$$\mathrm{e}^{\hat{A}} = \sum_{l=0}^{\infty} \frac{\hat{A}^l}{l!}\,, \tag{6.5}$$

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:

$$\hat{U}(t) = \sum_{l=0}^{\infty} \frac{\hat{H}^l}{l!}\left(-\frac{i}{\hbar}t\right)^l\,. \tag{6.6}$$

Considering the Hamiltonian in the standard form,

$$\hat{H}(x) = \hat{T} + \hat{V}(x) = -\frac{\hbar^2}{2m}\frac{\mathrm{d}^2}{\mathrm{d}x^2} + V(x)\,, \tag{6.7}$$

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

$$\hat{U}(t) = \sum_{l=0}^{\infty} \frac{(\hat{T}+\hat{V})^l}{l!}\left(-\frac{i}{\hbar}t\right)^l\,. \tag{6.8}$$

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 potentials.

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.

6.2 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 always be 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.

6.2.1 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:

$$x \rightarrow \{x_0, x_1, x_2, \ldots, x_N\}\,. \tag{6.9}$$

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:

$$\psi(x,t) \to \{\psi(x_0,t), \psi(x_1,t), \psi(x_2,t), \dots, \psi(x_N,t)\} = \{\psi_{0,t}, \psi_{1,t}, \psi_{2,t}, \dots, \psi_{N,t}\} . \tag{6.10}$$

We now have two indices of ψ: the grid point index i and the time index t. Further in the text, we will drop the index t to simplify the notation unless ψ at 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:

$$\langle \psi(x,t)|\psi(x,t)\rangle = \int_{-\infty}^{\infty} \psi^*(x,t)\psi(x,t)\,\mathrm{d}x = \sum_{j=1}^{N} \frac{\psi_{j-1}^*\psi_{j-1} + \psi_j^*\psi_j}{2}\Delta x . \tag{6.11}$$

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

$$V(x) \to \{V(x_0), V(x_1), V(x_2), \dots, V(x_N)\} = \{V_0, V_1, V_2, \dots, V_N\} . \tag{6.12}$$

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

$$\hat{V}\psi \to \{V_0\psi_0, V_1\psi_1, V_2\psi_2, \dots, V_N\psi_N\} . \tag{6.13}$$

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

$$\langle \psi(x,t)|\hat{V}|\psi(x,t)\rangle = \int_{-\infty}^{\infty} \psi^*(x,t)V(x)\psi(x,t)\,\mathrm{d}x = \sum_{j=1}^{N} \frac{V_{j-1}\psi_{j-1}^*\psi_{j-1} + V_j\psi_j^*\psi_j}{2}\Delta x \tag{6.14}$$

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.

6.2.2 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 the momentum operator $\hat{p}$ is a simple multiplication. Such a 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 inverse Fourier transform (IFT) $\mathcal{F}^{-1}$ as

$$\psi(k,t) = \mathcal{F}^{-1}[\psi(x,t)] = \int_{-\infty}^{\infty} \psi(x,t)\mathrm{e}^{ikx}\,\mathrm{d}x . \tag{6.15}$$

Performing the IFT 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 IFT:

$$\mathcal{F}^{-1}[\hat{T}\psi(x,t)] = \mathcal{F}^{-1}\left[-\frac{\hbar^2}{2m}\frac{\mathrm{d}^2\psi(x,t)}{\mathrm{d}x^2}\right] = -\frac{\hbar^2}{2m}\int_{-\infty}^{\infty}\frac{\mathrm{d}^2\psi(x,t)}{\mathrm{d}x^2}\mathrm{e}^{ikx}\,\mathrm{d}x \overset{p.p.}{=} \frac{\hbar^2}{2m}ik\int_{-\infty}^{\infty}\frac{\mathrm{d}\psi(x,t)}{\mathrm{d}x}\mathrm{e}^{ikx}\,\mathrm{d}x$$
$$\overset{p.p.}{=} -i^2\frac{\hbar^2}{2m}k^2\int_{-\infty}^{\infty}\psi(x,t)\mathrm{e}^{ikx}\,\mathrm{d}x = \frac{\hbar^2k^2}{2m}\psi(k,t)\,. \tag{6.16}$$

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. (6.16), and then transforming the result back to the position representation with the Fourier transform (FT) $\mathcal{F}$ as

$$\hat{T}\psi(x,t) = \mathcal{F}\left[\frac{\hbar^2k^2}{2m}\psi(k,t)\right]\,. \tag{6.17}$$

Once again, FT 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 IFT 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 FT 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:

$$\psi(x,t) \xrightarrow{\text{IFT}} \psi(k,t) \longrightarrow \frac{\hbar^2k^2}{2m}\psi(k,t) \xrightarrow{\text{FT}} \hat{T}\psi(x,t)\,. \tag{6.18}$$

Finally, we note that the Fourier method is not the only way to evaluate the action of the kinetic energy operator. The most straightforward way would be to calculate the second derivative numerically using finite differences. However, the finite differences are a local method that converges polynomially, while the Fourier method is global and converges exponentially.

6.2.3 The propagator and the split-operator technique

As we discussed in section 6.1, 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

$$\mathrm{e}^{-\frac{i}{\hbar}\hat{H}(x)\Delta t} \neq \mathrm{e}^{-\frac{i}{\hbar}V(x)t}\mathrm{e}^{-\frac{i}{\hbar}\hat{T}(x)t}\,. \tag{6.19}$$

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 propagators

$$\hat{U}(t) = \mathrm{e}^{-\frac{i}{\hbar}\hat{H}(x)t} = \mathrm{e}^{-\frac{i}{\hbar}\hat{H}(x)\Delta t}\mathrm{e}^{-\frac{i}{\hbar}\hat{H}(x)\Delta t}\cdots\mathrm{e}^{-\frac{i}{\hbar}\hat{H}(x)\Delta t}\,, \tag{6.20}$$

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

$$\mathrm{e}^{-\frac{i}{\hbar}\hat{H}(x)\Delta t} = \mathrm{e}^{-\frac{i}{\hbar}V(x)\Delta t/2}\mathrm{e}^{-\frac{i}{\hbar}\hat{T}(x)\Delta t}\mathrm{e}^{-\frac{i}{\hbar}\hat{V}(x)\Delta t/2} + \mathcal{O}(\Delta t^3)\,, \tag{6.21}$$

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

$$\psi(x,t) \xrightarrow{\text{IFT}} \psi(k,t) \longrightarrow \mathrm{e}^{\frac{i}{\hbar}\frac{\hbar^2 k^2}{2m}t}\psi(k,t) \xrightarrow{\text{FT}} \mathrm{e}^{-\frac{i}{\hbar}\hat{T}t}\psi(x,t)\,. \tag{6.22}$$

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.

6.3 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 and is set for each simulation individually based on the system. 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.

```python
"""Exercise for chapter Quantum dynamics in real time."""

# 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 (necessary to set for each simulation)
simtime = 100.0  # simulation time in atomic time units
dt = 0.2  # 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 the inverse Fourier transform to momentum space
    psi_k = np.fft.ifft(psi, norm="ortho")
    # apply expT in momentum space
    # fill in
    # finally the Fourier transform back to coordinate space
    psi = np.fft.fft(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:.8f}, <E> = {energy:.6f}, <V> = {energyV:.6f}, <T> = {energyT:.6f}”)

    # 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”Time: {t:.2f} a.t.u.\n” + r”$\langle \psi | \psi \rangle$” + f” = {norm:.8f}; $E = $ {energy:.6f} 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=0.0001)  # update plot and wait given interval

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

Listing 6.1: Incomplete code for quantum dynamics in real time.

6.4 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 a 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

$$\psi(x,0) = \left(\frac{2\alpha_0}{\pi}\right)^{\frac{1}{4}} \mathrm{e}^{-\alpha_0(x-x_0)^2+\frac{i}{\hbar}p_0(x-x_0)}\,, \tag{6.23}$$

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 a harmonic oscillator takes the following form:

$$V = \frac{1}{2}m\omega^2x^2\,. \tag{6.24}$$

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. (6.24) and initial Gaussian wave packet from Eq (6.23) 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 oscillations 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:

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

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 for the harmonic oscillator and $\langle x\rangle$ and $\langle p\rangle$ follow a classical trajectory. Equally, one can consider an anharmonic potential, e.g., the 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 the quantum world which is difficult to grasp from a 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 on 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 a 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 the form of Eq. (6.23) 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 based on classical physics. Analyse the dynamics also visually by plotting the density in time. Can you identify the interferences of the reflected and tunnelled wave packets?

Exercise: Heisenberg's uncertainty relations

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

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

where

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

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.

7. Energy spectrum and autocorrelation function

In the previous Chapter, we explored the time evolution of a wave function governed by the evolution operator, $\hat{U}(t)$, and introduced a simple numerical technique for propagating the wave function. This way we can study dynamical processes governed by quantum mechanics and explore intricacies of the quantum world like interference or tunnelling. However, the time evolution of a wave function offers more than just insight into the system's dynamics – it also encodes the energy spectrum of the system. Although counter-intuitive, we can obtain the energy spectrum, a time-independent quantity usually calculated by solving the time-independent Schrödinger equation (TISE), by performing a dynamical simulation. This time-dependent perspective is widely used in theoretical spectroscopy where both the energy spectrum and intensities are calculated from quantum or semiclassical dynamics.

7.1 Theoretical background

To understand how the energy spectrum is encoded in the wave function propagation, we examine the exact solution of the time-dependent Schrödinger equation. Assume we could solve the TISE

$$\hat{H}(x)\phi_k(x) = E_k\phi_k(x) \tag{7.1}$$

and obtain a complete set of eigenfunctions $\phi_k(x)$.[1] This set of eignefunctions forms an orthonormal basis, allowing us to expand any wave function as a linear combination of ϕ_k. Expanding the initial wave function in this basis, we write:

$$\psi(x,0) = \sum_k c_k\phi_k(x)\,, \tag{7.2}$$

where c_k are the expansion coefficients determined as

$$c_k = \langle\phi_k(x)|\psi(x,0)\rangle = \int_{-\infty}^{\infty}\phi_k^*(x)\psi(x,0)\,\mathrm{d}x\,. \tag{7.3}$$

If the initial wave function is normalized, the expansion coefficients further fulfil the following condition

$$\sum_k |c_k|^2 = 1\,. \tag{7.4}$$

The solution of the TDSE, $\psi(x,t)$, can be calculated by applying the propagator $\hat{U}(t)$ to the initial wave function (Eqs. (6.2) and (6.4)):

$$\psi(x,t) = \hat{U}(t)\psi(x,0) = \mathrm{e}^{-\frac{i}{\hbar}\hat{H}(x)t}\psi(x,0) = \sum_k c_k\mathrm{e}^{-\frac{i}{\hbar}\hat{H}(x)t}\phi_k(x)\,. \tag{7.5}$$

Since ϕ_k are eigenfunctions of the Hamiltonian $\hat{H}$, the propagator acts on each ϕ_k as:

$$\mathrm{e}^{-\frac{i}{\hbar}\hat{H}(x)t}\phi_k(x) = \mathrm{e}^{-\frac{i}{\hbar}E_kt}\phi_k(x)\,, \tag{7.6}$$

[1] In practice, obtaining these eigenfunctions is often the most challenging step in this approach, as they cannot usually be computed directly. Therefore, this method is more useful for deriving theoretical insights rather than for practical calculations.

leading to the time-dependent solution:

$$\psi(x,t) = \sum_k c_k \mathrm{e}^{-\frac{i}{\hbar}E_k t}\phi_k(x)\,. \tag{7.7}$$

This expression represents the exact solution of the TDSE in the basis of Hamiltonian eigenstates. For time-independent Hamiltonians (i.e., $\hat{H} \neq \hat{H}(t)$), the coefficients c_k remain constant over time, and the only time-dependent terms are the exponentials $\mathrm{e}^{-\frac{i}{\hbar}E_k t}$, fully governing the evolution.

Using this exact TDSE solution, we can now extract the energy spectrum through the *autocorrelation function*. The autocorrelation function is defined as an overlap of the wave function at time t with the initial wave function at time 0:

$$S(t) = \langle \psi(x,0)|\psi(x,t)\rangle = \int_{-\infty}^{\infty} \psi^*(x,0)\psi(x,t)\,\mathrm{d}x\,. \tag{7.8}$$

It measures the probability amplitude of $\psi(x,t)$ being in the initial state $\psi(x,0)$. Thus, the autocorrelation function is always equal to 1 at time 0. Inserting the exact solution of TDSE (7.7), the autocorrelation function reads

$$\begin{aligned} S(t) &= \int_{-\infty}^{\infty} \sum_l \sum_k c_l^* \phi_l^*(x) c_k \mathrm{e}^{-\frac{i}{\hbar}E_k t}\phi_k(x)\,\mathrm{d}x \\ &= \sum_l \sum_k c_l^* c_k \mathrm{e}^{-\frac{i}{\hbar}E_k t} \int_{-\infty}^{\infty} \phi_l^*(x)\phi_k(x)\,\mathrm{d}x \\ &= \sum_l \sum_k c_l^* c_k \mathrm{e}^{-\frac{i}{\hbar}E_k t}\delta_{kl} \\ &= \sum_k |c_k|^2 \mathrm{e}^{-\frac{i}{\hbar}E_k t}\,. \end{aligned} \tag{7.9}$$

It appears that the autocorrelation function is a sum of the time-dependent exponentials weighted by magnitudes of the coefficients c_k. The energy spectrum σ can then be obtained by applying IFT to the autocorrelation function:

$$\sigma(E) = \int_{-\infty}^{\infty} S(t)\mathrm{e}^{\frac{i}{\hbar}Et}\,\mathrm{d}t = \sum_k |c_k|^2 \int_{-\infty}^{\infty} \mathrm{e}^{\frac{i}{\hbar}(E-E_k)t}\,\mathrm{d}t = \sum_k |c_k|^2 \delta(E-E_k)\,, \tag{7.10}$$

where the Dirac δ-distributions are centred at energies E_k with magnitudes $|c_k|^2$. Thus, the spectrum exhibits peaks at all eigenvalues E_k for which $c_k \neq 0$ in the initial wave function $\psi(x,0)$, see Fig. 7.10.

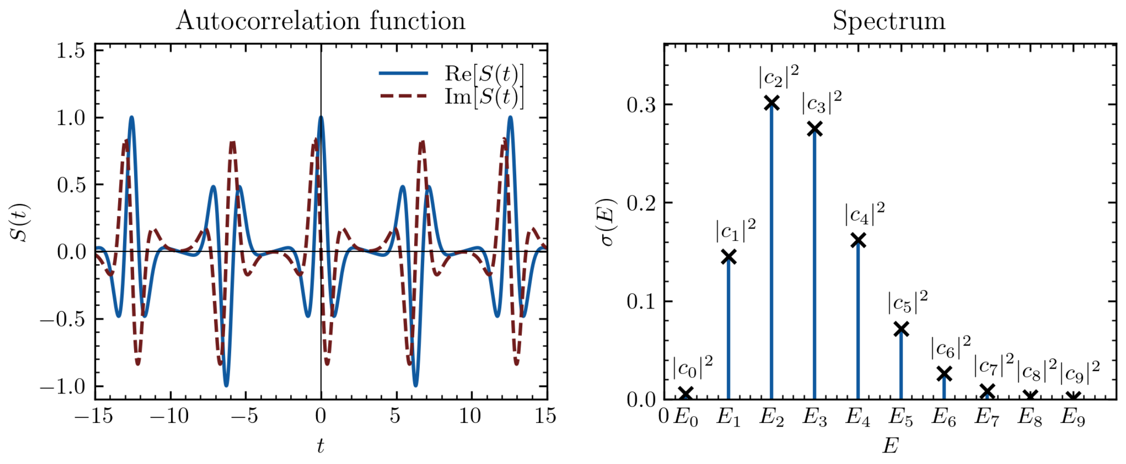


Figure 7.1: An illustrative example of autocorrelation function $S(t)$ of a harmonic oscillator (left) and its corresponding energy spectrum $\sigma(E)$ (right).

Although this analysis is based on the exact TDSE solution and the full set of eigenfunctions ϕ_k, the derived autocorrelation function $S(t)$ and energy spectrum $\sigma(E)$ apply to any wave function evolution. This method provides a powerful tool for calculating the spectrum of any Hamiltonian by time-propagating a wave function, which is often computationally more feasible than directly diagonalizing the Hamiltonian.

7.2 Numerical implementation

The numerical implementation of the outlined procedure for calculating the energy spectrum from the autocorrelation function is straightforward. We need to perform a quantum dynamical simulation as described in Chapter 6 and calculate the overlap between $\psi(x,t)$ and $\psi(x,0)$ at each step. The overlaps form the autocorrelation function $S(t)$ which is then converted into the energy spectrum by the inverse Fourier transform after the simulation. However, the integral in the inverse Fourier transform in Eq. (7.10) runs from time $-\infty$ to ∞ requiring the knowledge of the autocorrelation function from $-\infty$ to ∞, which is usually unavailable since a typical simulation runs only from time 0 to some finite time $t_{\max}$. Thus, we have only truncated information about $S(t)$ in the positive times and lack the information about negative times.

The lack of information about $S(t)$ in negative times can be addressed by exploring the properties of the autocorrelation function.

$$\begin{aligned} S^*(t) &= \langle\psi(x,0)|\psi(x,t)\rangle^* = \langle\psi(x,0)|\mathrm{e}^{-\frac{i}{\hbar}\hat{H}(x)t}|\psi(x,0)\rangle^* = \langle\psi(x,0)|\mathrm{e}^{\frac{i}{\hbar}\hat{H}(x)t}|\psi(x,0)\rangle \\ &= \langle\psi(x,0)|\mathrm{e}^{-\frac{i}{\hbar}\hat{H}(x)(-t)}|\psi(x,0)\rangle = S(-t)\,, \end{aligned} \tag{7.11}$$

where we have complex conjugated $S(t)$ and noticed we can compensate for the complex conjugation by taking the time negative. This leads to a fundamental symmetry of the autocorrelation function,

$$S(-t) = S^*(t)\,, \tag{7.12}$$

which allows us to obtain the autocorrelation in negative times by taking it complex conjugate in the positive times. The symmetry of the autocorrelation function in Eq. (7.12) resolves our problems with the lack of information about negative times and saves us from propagating the wave function also back in time.

Although resolving the negative-time issue, we still have a problem with the truncated autocorrelation function at some time $t_{\max}$ due to a finite propagation of the wave function. Ideally, we want the autocorrelation function to decay to zero at $t_{\max}$. Then, the integral in Eq. (7.12) beyond $t_{\max}$ is zero and we can formally integrate only in limits $[-t_{\max}, t_{\max}]$. Thus, we always want to propagate the wave function until the autocorrelation function decays to zero. A decay of the autocorrelation function is common in natural processes such as photochemistry, luminescence or Auger decay.

Nonetheless, the autocorrelation function does not decay naturally with time in systems we learned to simulate in Chapter 6, since such decay would require coupling to some external bath or a multistate system. The autocorrelation function is then periodic and extends to infinity. In such cases, we need to attenuate the autocorrelation function artificially in order to decay within a feasible simulation time. This attenuation can be achieved by introducing a damping function ξ which attenuates the autocorrelation function such that:

$$\sigma(E) = \int_{-\infty}^{\infty} \xi(t)S(t)\mathrm{e}^{\frac{i}{\hbar}(E)t}\,\mathrm{d}t = \int_{-t_{\max}}^{t_{\max}} \xi(t)S(t)\mathrm{e}^{\frac{i}{\hbar}(E)t}\,. \tag{7.13}$$

Typical damping functions are an exponential,

$$\xi(t) = \mathrm{e}^{-\gamma t}\,, \tag{7.14}$$

or a Gaussian function,

$$\xi(t) = \mathrm{e}^{-\gamma t^2}\,. \tag{7.15}$$

The damping parameter γ governs the strength of the attenuation. The damping has a profound effect on the spectra. Without the damping, the spectrum would be a sum of Dirac δ-distributions, infinitely narrow peaks, with intensity corresponding to $|c_k|^2$, as seen in Figure 7.1. However, the damping introduces broadening of the peaks proportional to the damping factor γ. Thus, applying the damping function artificially broadens our peaks and limits our resolution of energy levels in the system as illustrated in Figure 7.2. It is always crucial to set the damping such that the broadening is smaller than the spacing between the eigenenergies.

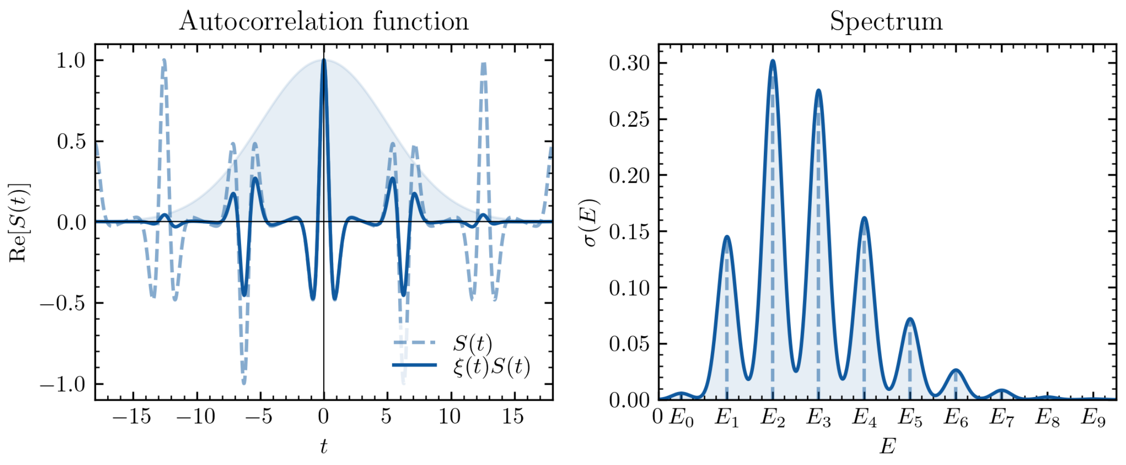


Figure 7.2: The effect of the damping function $\xi(t)$ on the energy spectrum calculated from the autocorrelation function. The damping of the autocorrelation function causes a broadening of the spectral lines but preserves the peak positions. If the damping is caused by a natural effect (fluorescence, internal conversion, Auger decay, etc.), the width of the spectral lines bears information about the lifetime of the damping process. For a very fast damping, the peaks can merge into one blurred spectrum without the possibility to distinguish the energy levels.

This completes our discussion on how to deal with a finite autocorrelation function spanning only the positive-time region. Let us now briefly comment on what would happen if we did not extend the autocorrelation function to negative times and attenuated it. The algorithms for discrete IFT would not prevent us from transforming an autocorrelation function spanning a range between 0 and t_{max}. We would still get a spectrum in the energy domain, yet the spectrum would be plagued with some artefacts. Without deriving it, we just state that in the discrete FT (or IFT), the signal obtained in the region of $[0, t_{\text{max}}]$ is copied to $[t_{\text{max}}, 2t_{\text{max}}]$ and $[-t_{\text{max}}, 0]$ and so on. This, first of all, breaks the symmetry of the autocorrelation function derived in Eq. (7.11) and, secondly, creates discontinuities in $S(t)$. The former causes an incorrect shape of the peaks in the spectrum while the latter causes so-called *ringing* in the spectrum. Both effects decrease the resolution of the spectrum for smaller peaks and may lead to incorrect extraction of eigenenergies from the spectrum. The effect of damping and extending the autocorrelation function is explored in more detail in Figure 7.3.

Finally, we conclude with a brief comment on the resolution of discrete Fourier transform and inverse Fourier transform. Discrete FT takes as input data represented on a grid and creates their image in the energy domain.[2] The energy grid has the same number of grid points as the time grid, but the spacing between energy points is inversely proportional to the total time duration of the grid:

$$\Delta E \propto \frac{\hbar}{t_{\text{max}}} \tag{7.16}$$

Thus, if we want the spectrum with higher resolution (smaller ΔE), we need to extend our simulation time t_{max}. Conversely, the length of the energy grid is inversely proportional to the spacing of the time grid Δt, so a finer simulation time step will extend the energy range of the spectrum. In summary, achieving a higher energy resolution (smaller ΔE) requires extending the total time of the simulation, while widening the energy range requires a finer time step Δt.

7.3 Code

The essential part of calculating the autocorrelation function is again the wave function propagation, for which we have already created a code in Chapter 6. We will reuse the code in this Chapter and modify it to calculate the spec-

[2] Fourier transform naturally provides data in the frequency domain ν, which can be converted to energy domain as $E = 2\pi\hbar\nu$.

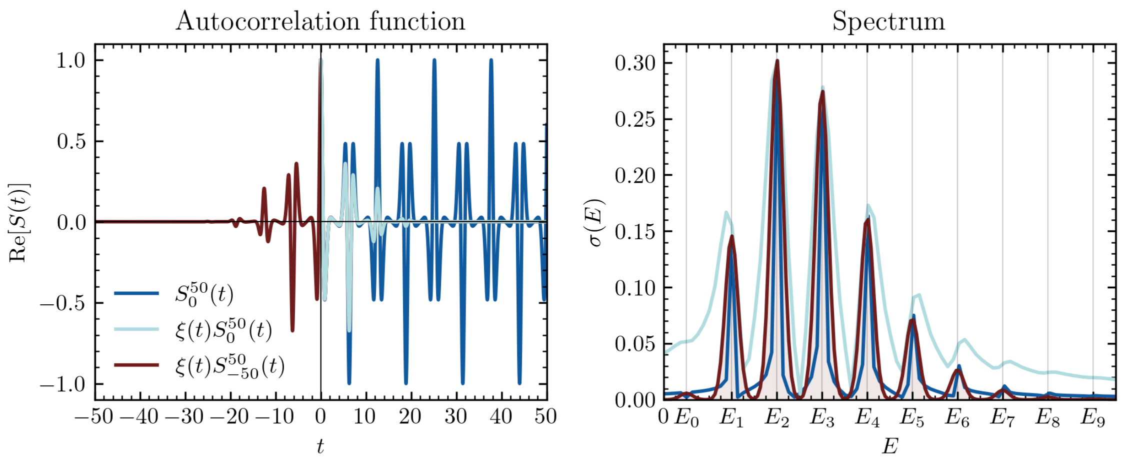


Figure 7.3: The effect of discrete IFT on the spectrum for undamped autocorrelation function in the range $t \in [0, 50]$ (S_0^{50}) as obtained from the simulation, damped autocorrelation function in the range $t \in [0, 50]$ (ξS_0^{50}) and damped autocorrelation function spanning from -50 to 50 (ξS_{-50}^{50}). While for the undamped S_0^{50} the peaks remain at the correct position, the damped ξS_0^{50} has the peak maxima shifted from the exact values. Furthermore, both S_0^{50} and S_0^{50} have asymmetric peak shapes caused by a sharp truncation of S. For both the incorrect line shape decreases the resolution of the spectrum. Notice for example the region around E_0. While we can clearly see a small peak for ξS_{-50}^{50}, there is no peak for the original data S_0^{50} and there is only a shoulder for ξS_0^{50}. The energy E_0 would be then difficult to extract for anything but ξS_{-50}^{50}.

trum from the autocorrelation function. Thus, we will only comment on the additions to the code in the following text, giving more freedom with the implementation to the reader. Still, the reader can find the full exercise code written at our Github page.

First, we need to store the initial wave function to a separate variable and create empty lists for the autocorrelation function and time, where we will append the current values during the propagation. These additions must come in the initialization part of the code prior to the main propagation loop.

```
...
# save the initial wave function for calculating the autocorrelation function
psi0 = psi  # initial wave function
time, autocorr = [], []  # empty lists for appending values of time and autocorrelation function
...
```

Then, we need to calculate the values of the autocorrelation function during the propagation (i.e., in the `while` loop) and store them in our array. The value of the autocorrelation function is the overlap between `psi` and `psi0`, calculated with e.g. the trapezoid rule.

```
while t < simtime:  # loop until simulation time is reached
    ... # wave function propagation
    # calculate the autocorrelation function S(t) = <psi(0)|psi(t)>
    overlap =  # calculate the overlap

    autocorr.append(overlap)  # appending the overlap to our autocorrelation function list
    time.append(t)  # appending t to our time list
    ... # calculation of energies and plotting
```

These two code snippets are the only modifications of the quantum dynamics code. Once the quantum dynamics is finished, we should have the autocorrelation function calculated. Note that the most time-consuming part of the quantum dynamics code is the wave function plotting. We recommend removing the plotting functions, which will significantly speed up the code, since we focus only on the autocorrelation function after the dynamics.

Now, we need to process the autocorrelation function and calculate the energy spectrum. Since we have the data stored as lists, we convert `autocorr` and `time` into NumPy arrays first. Then, we apply the damping function

(7.13) and extend $S(t)$ to negative times using (7.12), both described in Section 7.2. Note that these operations can be interchanges. Finally, we used the discrete IFT on $S(t)$ and calculated the spectrum. All these operations are one or two-line procedures. The major part of the code is the prepared plotting of the results.

```
### autocorrelation function section ###
autocorr = np.array(autocorr)  # converting the autocorrelation function to a numpy array
time = np.array(time)  # converting the time to a numpy array

# apply the damping to the autocorrelation function in form of exp(-kappa*time)
autocorr =  # apply the damping factor

# extend the autocorrelation function to negative times assuming that S(t) = S^*(-t)
time = np.concatenate([-time[::-1], time])  # new time array in range [-t_max, t_max]
autocorr = np.concatenate([np.conjugate(autocorr[::-1]), autocorr])  # new symmetric autocorr in
    range [-t_max, t_max]

# calculate spectrum from autocorrelation function and the frequency axis corresponding to it
spectrum = # fill in the inverse Fourier transform
freq = 2*np.pi*np.fft.fftfreq(len(time), d=dt)

# plot results
fig, axs = plt.subplots(1, 2, figsize=(8, 3), tight_layout=True)

# autocorrelation function
axs[0].plot(time, np.real(autocorr), label=r'$\mathcal{Re}[S(t)]$')
axs[0].plot(time, np.imag(autocorr), label=r'$\mathcal{Im}[S(t)]$')
axs[0].set_xlabel('Time (a.u.)')
axs[0].set_ylabel(r'$S(t)$')
axs[0].set_title('Autocorrelation Function')
axs[0].legend(frameon=False, labelspacing=0)

# spectrum
axs[1].plot(hbar*freq, np.abs(spectrum))
axs[1].set_xlim(0, np.max(hbar*freq[spectrum > np.max(spectrum)/1000]))
axs[1].set_ylim(0)
axs[1].set_xlabel('Energy (a.u.)')
axs[1].set_ylabel(r'$\mathcal{F}^{-1}[S(t)]$')
axs[1].set_title('Spectrum')

# searching for local maxima of the spectrum
print(f"\nMaxima of the spectrum:")
abs_spectrum = np.abs(spectrum)
loc_max_bool = (abs_spectrum[1:-1] > abs_spectrum[:-2]) & (abs_spectrum[1:-1] > abs_spectrum[2:])
loc_max_index = np.where(loc_max_bool)[0] + 1
loc_max_energies = hbar*freq[loc_max_index]
for index, en in enumerate(loc_max_energies):
    intensity = abs_spectrum[loc_max_index[index]]
    print(f" * State {index}: E = {en:.5f} a.u.; I = {intensity:.5e}")
    axs[1].axvline(en, lw=1, color='black', alpha=0.1)
    axs[1].scatter(en, intensity, marker='x', color='black', s=20)

plt.show()
```

7.4 Applications

Exercise: Energy spectrum of a harmonic oscillator

The spectrum of a harmonic oscillator (Eq. (6.24)) is derived in all introductory courses to quantum mechanics and takes a very simple form:

$$E_n^{\mathrm{HO}} = \hbar\omega\left(n + \frac{1}{2}\right) \quad n = 0, 1, 2, \dots .$$

The derivation of the spectrum is a typical example of the Hamiltonian diagonalization, which is the time-independent approach. In this exercise, we will contrast the exact spectrum with the spectrum calculated by the time-dependent approach presented in this Chapter.

Assignment: Set up a harmonic oscillator from Eq. (6.24) and an initial Gaussian wave packet from Eq (6.23) with the following parameters: $m = 1$ a.u., $\omega = 0.1$ a.u., $x_0 = 6$ a.u., $p_0 = 0$ a.u. and $\alpha_0 = 0.3$. Propagate the wave packet for at least 10000 a.u. Don't forget to set a corresponding grid and time step! Then, apply the Gaussian damping function (7.15) with the parameter $\gamma = 0.005$. Do the peak positions correspond to the energies of the harmonic oscillator E_n^{HO}? Try also values of $\gamma = 0.02$, 0.05 and 0.1. Can you still distinguish all the peaks in the spectrum? Try to find the limit of γ where the peaks can be still distinguished and energies clearly estimated.

Exercise: Initial wave function effect on the spectrum

The spectrum calculated from the autocorrelation function depends on the initial wave function, namely on the magnitude of the expansion coefficients c_k. Thus, if $c_k = 0$ for some state, there will be no peak at E_k in the spectrum and we will be blind to this eigenvalue of the Hamiltonian. Since the spectrum depends on the initial wave function through its expansion coefficients c_k, the number and intensity of peaks can be modified by changing the initial wave function for the dynamics. In this exercise, we will explore the effect of the initial wave function on the spectrum.

Assignment: Use the potential from the previous Exercise with the damping parameter $\gamma = 0.005$. Start with the Gaussian function with $x_0 = 0$ a.u., $p_0 = 0$ a.u. and $\alpha_0 = 0.5$, and examine the spectrum. Do you see the full spectrum of the harmonic oscillator? Test what happens if $x_0 = 0.1$ and 0.5 a.u. Can you explain your observation? Hint: think about the symmetry of the potential an the initial guess. Finally, try to optimize the initial wave function to obtain a maximum number of clearly identifiable peaks in the spectrum. Note that you can change also the initial wave function form.

8. Imaginary-time dynamics and stationary states

Postulates of quantum mechanics state, that observable quantities are given by the eigenvalues of the corresponding quantum-mechanical operator. Thus, most quantum mechanics calculations reduce to computing the eigenvalues and eigenfunctions of the operator of interest. The central operator we usually try to find the eigenvalues for is the Hamiltonian $\hat{H}$. The eigenvalues of the Hamiltonian are the energy spectrum that can be measured by, for example, spectroscopic techniques. Part I of this book dealt with diagonalization of the Hamiltonian for atoms and molecules, using techniques of time-independent Schrödinger equation (TISE). This is the standard approach taken in quantum chemistry. In Chapter 7, we have shown that the energy spectrum can be obtained also from the time evolution of a wave function, which may be often more efficient than the full diagonalization of the Hamiltonian. However, this approach based on the autocorrelation function can provide us only with the eigenvalues, not with the eigenfunctions. Yet the eigenfunctions are necessary to calculate observables of other operators and we often need them. In this Chapter, we will show how both the eigenvalues and eigenfunctions can be obtained from time-dependent techniques using propagation in imaginary time. The imaginary-time propagation is less frequent than the autocorrelation approach and much less frequent than the operator diagonalization, but it is still a useful technique.

8.1 Theoretical background

In Chapter 7, we have derived an formula for the time evolution of a wave function expressed in the eigenstates of the Hamiltonian operator (see Section 7.1 and Eq. (7.7)):

$$\psi(x,t) = \sum_k c_k \mathrm{e}^{-\frac{i}{\hbar}E_k t}\phi_k(x)\,, \tag{8.1}$$

where E_k are the eigenvalues of the Hamiltonian (energies) and $\phi_k(x)$ are its eigenfunctions. The coefficients c_k are constant in time since the Hamiltonian is considered time-independent and are set at time 0. This equation was obtained by applying the propagator $\hat{U}(t)$ to the initial wave function $\psi(x,0)$.

First, we will inspect the wave function in Eq. (8.1), starting with its norm:

$$\begin{aligned}\langle\psi(x,t)|\psi(x,t)\rangle &= \sum_k\sum_l c_k^* c_l \mathrm{e}^{\frac{i}{\hbar}E_k t}\mathrm{e}^{-\frac{i}{\hbar}E_l t}\langle\phi_k(x)|\phi_l(x)\rangle = \sum_k\sum_l c_k^* c_l \mathrm{e}^{\frac{i}{\hbar}(E_k-E_l)t}\delta_{kl}\\ &= \sum_k |c_k|^2 \mathrm{e}^{\frac{i}{\hbar}(E_k-E_k)t} = \sum_k |c_k|^2 = 1\,.\end{aligned} \tag{8.2}$$

As expected, the norm of the wave function is preserved in time since c_ks are constant, which is a requirement we have for the wave function in order to conserve the number of particles. The quantity $|c_k|^2$ is the probability of finding state k in a system described by $\psi(x,t)$. In other words, $|c_k|^2$ is the contribution of the k-th state to the wave function. Eq. (8.2) shows that all ϕ_k contribute constantly the same over time and the composition of the wave function does not vary.[1] Where has the time disappeared in Eq. (8.2) such that the different contributions are constant? The time t appears only in the imaginary exponentials $\mathrm{e}^{\frac{i}{\hbar}E_k t}$ which vanished due to the complex conjugation.

[1] We remind the reader that the wave function in Eq. (8.1) was derived with the assumption of a time-independent Hamiltonian. If the Hamiltonian would depend on time (for example by containing a electromagnetic field), the contributions $|c_k|^2$ would vary.

Let us now imagine that the state contributions could vary over time. Could we possibly design a propagator, that would enhance the contribution of one of the states and dump the others? If we could achieve this, applying such a propagator would lead to a pure eigenstate ϕ_k after the propagation, calculating Hamiltonian eigenstates by means of dynamics instead of solving the time-independent Schrödinger equation. Since the time dependence disappears from the norm in Eq. (8.2) due to the complex conjugation, a first natural attempt to create the modified propagator is to replace the complex exponentials with real exponentials. This can be achieved by compounding the imaginary unit i with time t, introducing a so-called imaginary time $\tau = it$. The imaginary time is an elusive concept as we have no experience with it in life or even physics. At this point, we will leave the discussion about the concept of imaginary time aside and we ask the reader to view it from a purely mathematical perspective as a new variable τ introduced into our equations.

Introducing the imaginary time, we can define our new propagator as

$$\hat{U}_{\mathrm{IT}}(\tau) = \mathrm{e}^{-\frac{1}{\hbar}\hat{H}(x)\tau} \,. \tag{8.3}$$

The wave function corresponding to $\hat{U}_{\mathrm{IT}}(\tau)$ is a function of τ instead of t and reads

$$\psi(x,\tau) = \sum_k c_k \mathrm{e}^{-\frac{E_k}{\hbar}\tau}\phi_k(x) \,. \tag{8.4}$$

The form the wave function are analogous to the wave function in Eq. (8.1) and is derived from the propagator the same way. The expansion coefficients are determined from the initial wave function and also fulfil Eq. (7.4). Having the wave function, we can again calculate the norm and inspect the state contributions to it:

$$\langle\psi(x,\tau)|\psi(x,\tau)\rangle = \sum_k\sum_l c_k^* c_l \mathrm{e}^{-\frac{1}{\hbar}(E_k+E_l)\tau}\langle\phi_k(x)|\phi_l(x)\rangle = \sum_k |c_k|^2\mathrm{e}^{-2\frac{E_k}{\hbar}\tau} \neq 1 \,. \tag{8.5}$$

The state contributions are no longer constant but depend exponentially on the imaginary time as $\mathrm{e}^{-2\frac{E_k}{\hbar}\tau}$. Depending on the sign of the state energy E_k, the contribution to the norm decays or increases exponentially. As a result, the norm is not conserved, which means our new propagator is not unitary.

We have now created a propagator which has state contributions varying in time but if we want it to be a useful propagator, we need it to converge to only one of the states. Let us now examine how the state contributions, meaning $|c_k|^2\mathrm{e}^{-2\frac{E_k}{\hbar}\tau}$, behave with respect to each other. Consider two states k and l, where $E_l > E_k$ and $E_l - E_k = \Delta E_{lk} > 0$. The ratio of their contributions to the norm equals to

$$\frac{|c_k|^2\mathrm{e}^{-2\frac{E_k}{\hbar}\tau}}{|c_l|^2\mathrm{e}^{-2\frac{E_l}{\hbar}\tau}} = \mathrm{e}^{-2\frac{E_k-E_l}{\hbar}\tau}\frac{|c_k|^2}{|c_l|^2} = \mathrm{e}^{2\frac{\Delta E_{lk}}{\hbar}\tau}\frac{|c_k|^2}{|c_l|^2} \,. \tag{8.6}$$

and shows that the state of lower energy (k in our case) has exponentially increasing contribution compared to the higher energy state l. Since this holds for an arbitrary pair of states, the contribution of the lowest energy state to the norm and wave function will be exponentially increasing compared to the other states. As a consequence, the wave function $\psi(x,\tau)$ will converge in the limit of $\tau\to\infty$ to the lowest energy eigenstate ϕ_0,

$$\lim_{\tau\to\infty}\psi(x,\tau) = c_0\mathrm{e}^{-\frac{E_0}{\hbar}\tau}\phi_0(x) \,, \tag{8.7}$$

multiplied by $c_0\mathrm{e}^{-\frac{E_0}{\hbar}\tau}$.

As we have discussed before, the wave function is not normalized but nothing prevents us to renormalize at every time τ. We will define a normalized wave function as

$$\tilde{\psi}(x,\tau) = \frac{\psi(x,\tau)}{\sqrt{\langle\psi(x,\tau)|\psi(x,\tau)\rangle}} \,, \tag{8.8}$$

which allows us to write

$$\lim_{\tau\to\infty}\tilde{\psi}(x,\tau) = \phi_0(x) \,. \tag{8.9}$$

The normalized wave function limits in long imaginary times to the lowest-energy eigenstate as we wanted. Thus, our imaginary-time propagator allows us to calculate the ground-state wave function in terms of dynamics instead of the standard diagonalization of the Hamiltonian. The convergence of the imaginary-time dynamics to the lowest-energy eigenstate is demonstrated in Fig. 8.1.

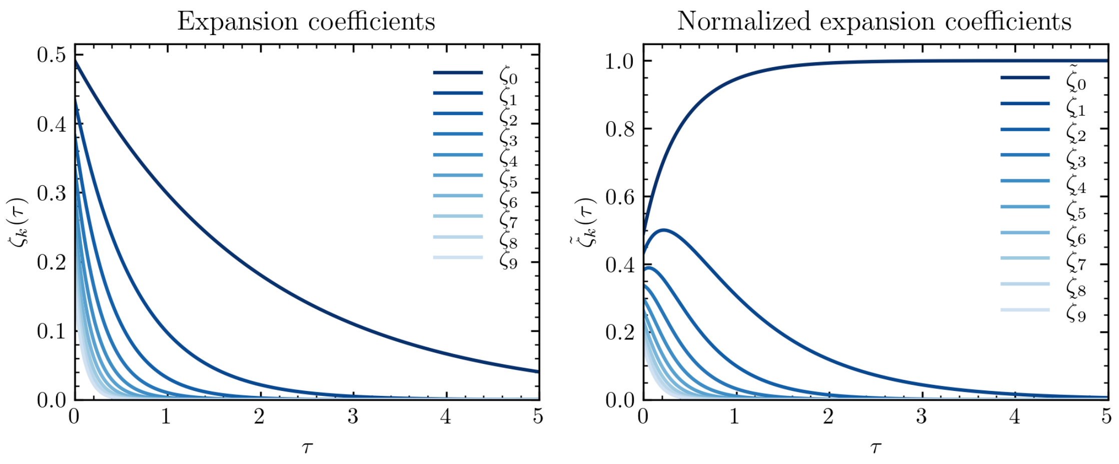


Figure 8.1: The time-dependent expansion coefficients of the wave function $\zeta_k(\tau) = \langle\phi_k|\psi(\tau)\rangle$ (left) and expansion coefficients of its normalized counterpart $\tilde{\zeta}_k(\tau) = \langle\phi_k|\tilde{\psi}(\tau)\rangle$ (right) for a harmonic oscillator. All the coefficients ζ_k decay exponentially, yet some faster and some slower depending on their energy E_k. The normalized expansion coefficients $\tilde{\zeta}_k$ show that the lowest-state contribution is increasing while the other states contribute less in time. The initial increase in $\tilde{\zeta}_1$ is given by the other higer-energy states which decay much faster but once they decay, $\tilde{\zeta}_1$ also starts decreasing.

So far we have shown that the imaginary-time dynamics can help us to optimize the lowest-energy state, but what if we also want higher states? In principle, we can optimize an arbitrary state with imaginary quantum dynamics. The correct statement of what we have derive would rather be that the imaginary-time dynamics converges to the lowest-energy state that contributes to the initial wave function $\psi(x,0)$. If the ground state expansion coefficient c_0 is equal to 0, the ground-state contribution will be zero for all times and we will never acquire the ground state wave function. We can only acquire a wave function of the lowest-energy state contributing to the initial wave function, i.e., with a nonzero expansion coefficient at the beginning of the dynamics.

This realization gives us a recipe to calculate higher-energy states: we need to construct the initial wave function such that it has zero contribution from all the lower-energy states. Thus, we must project out these states from the initial wave function. Taking the example of the first excited state, we need to first run the imaginary time dynamics and optimize ϕ_0. Then, we need to project out this ground state from the initial wave function $\psi(x,0)$, creating a new initial wave function

$$\psi_1(x,0) = \psi(x,0) - c_0\phi_0(x) = \sum_{k=0}^{\infty} c_k\phi_k(x) - c_0\phi_0(x) = \sum_{k=1}^{\infty} c_k\phi_k(x)\,, \tag{8.10}$$

which needs to be renormalized such that $\sum_{k=1}^{\infty}|c_k|^2 = 1$. Applying $\hat{U}_{\mathrm{IT}}$ to $\psi_1(x,0)$ then leads to $\phi_1(x)$ we wanted to optimize:

$$\lim_{\tau\to\infty}\tilde{\psi}_1(x,\tau) = \phi_1(x)\,. \tag{8.11}$$

Hence, if we want to optimize an arbitrary eigenstate, we first need to optimize all the lower-energy states and project them out from the initial wave function. In this manner, we gradually acquire the eigenstates from the bottom.

In the end, let us return and comment on the imaginary time. What is it and how should we image a propagation in imaginary time? The concept of imaginary time seems elusive since no one has ever seen a watch measuring imaginary time. Still, the concept of imaginary time is not new in physics and appears, for example, in general relativity and cosmology, yet the concept is definitely not easy to grasp. The imaginary time often appears in equations as a mathematical 'trick' that allows us to reformulate equations into a better form. Although there are attempts to interpret the imaginary time, they still remain in the realm of speculations. If the imaginary time creates confusion in the reader, we recommend to view it rather as a mathematical construct that allows us to create a new operator with desired properties. The operator serves as a tool to obtain eigenstates and we do not need to attribute it any physical meaning, despite having a clear connection to dynamics

8.2 Numerical implementation

Numerical implementation of the imaginary-time quantum dynamics builds on the real-time quantum dynamics, where the imaginary unit needs to be removed from the exponentials in the propagator. The propagation in time is otherwise the same except for two minor additions to the code.

First, we need to renormalize the propagated wave function $\psi(x,\tau)$ at every time step since we are interested in the normalized $\tilde{\psi}(x,t)$. Normalization is also important to make the calculation numerically stable. We could propagate ψ without normalization and normalize it at the end, yet we would soon run into numerical issues with the exponential decay or increase in the norm. Plus we still need the norm to evaluate the energy at every step. The renormalization is a simple one-line modification after the propagation step.

The second addition to the code concerns the calculation of excited states (states above the ground state). As we have shown in the previous section, the imaginary-time dynamics converge to the lowest state in the expansion of $\psi(x,0)$. Thus, the calculation of an arbitrary state l requires projecting out all states j below l from the initial wave function. Note that renormalization is necessary after the removal of the lower states. Therefore, the renormalization described in the previous paragraph is convenient after this step.

While this projection of the initial wave function would be in theory sufficient only at the beginning, the numerical precision of a computer alter the wave function a bit at every step which can create a tiny contribution of the lower states. These contributions are at the order of a computer precision, i.e., very small, but they can become important at longer dynamics since they are enhanced exponentially. Hence, the projecting out should be done also during the dynamics. The simplest is to project out at every time step but it is possible to optimize the algorithm and project out with some frequency.

We conclude with a remark not related to the implementation. If the eigenstate we seek has $c_k = 0$, we cannot optimize it with imaginary-time dynamics. The initial wave function $\psi(x,0)$ should always be selected such that the coefficient of the desired state k is not zero. This is similar to the previous chapter about the autocorrelation function and we can use the same principles.

8.3 Code

```python
"""Solution for chapter Imaginary-time dynamics and stationary states."""

# 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 = 200.0  # simulation time in atomic time units
dt = 0.2  # time step in atomic time units
m = 1.0  # mass in atomic units
nstates = 10  # number of states to be calculated

# 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 =  # 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
psi0 =  # fill in

# create and empty array for final optimized wave functions and energies
psi_optimized = np.zeros(shape=(nstates, len(x)))
energy_optimized = np.zeros(shape=(nstates))

# 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

print("\nImaginary-time quantum dynamics.\n------------------------------------------\n")
# loop through states
for state in range(nstates):
    # creating wave function for propagation
    psi = psi0

    # first, we need to calculate energy of the initial wave function and store it into energy_old
    variable, so that we
    # can compare at the end of the cycle the new and old energies
    energy_old =  # fill in

    # propagation
    t = 0  # set initial time to 0
    print(f"\nState:  {state + 1:d}")
    while t < simtime:  # loop until simulation time is reached
        # propagate half-step in V
        # fill in

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

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

        # calculate new time after propagation
        t += dt

        # remove contributions from previous states, i.e. remove c_i*psi_optimized, where c_i = <psi_optimized|psi>
        for s in range(state):
        # fill in

        # calculate norm
        norm =  # fill in

        # renormalize the wave function so that the norm is one again
        # fill in

        # 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:.8f}, <E> = {energy:.6f}, <V> = {energyV:.6f}, <T> = {energyT:.6f}”)

        # plot
        plt.cla()  # clean figure
        # plot current wave function
        plt.fill_between(x, psi**2 + energy, energy, alpha=0.2, color=’C1’)  # plot |wf|
        plt.text(np.min(x), energy, f”{energy:.4f}”)
        # plot previously optimized wave functions
        for s in range(state):
            plt.fill_between(x, psi_optimized[s]**2 + energy_optimized[s], energy_optimized[s], alpha=0.1,
                color=’black’)
            plt.text(np.min(x), energy_optimized[s], f”{energy_optimized[s]:.4f}”)

        # plot potential
        plt.plot(x, V, color=’black’)
        # plot title and labels
        plt.title(
            f”State: {state + 1}; Time: {t:.2f} a.t.u.\n” + r”$\langle \psi | \psi \rangle$” + f” = {norm:.8f}; $E = $ {energy:.6f} a.u.”)
        plt.xlabel(”$x$ (a.u.)”)
        plt.ylabel(”$|\Psi|^2$”)

        # set new ymin, ymax
        max_wf = np.max(psi**2)
        ymax = max([ymax, max_wf + energy])
        plt.ylim(ymin, ymax)
        plt.pause(interval=0.0001)  # update plot and wait given interval

        # check energy change between the current and the previous step
        # if |energy - energy_old| < threshold, then the wave function is optimized and break the cycle
        if : # fill in
            break
        else:
            energy_old = energy

    # save the final wave function and energy
    psi_optimized[state] = psi
    energy_optimized[state] = energy
```

```python

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

Listing 8.1: Incomplete code for the imaginary-time quantum dynamics.

8.4 Applications

Exercise: Harmonic oscillator

The spectrum of a harmonic oscillator was already calculated in the previous chapter as Fourier transform of the autocorrelation function. The same spectrum can be also obtained by the imaginary-time quantum dynamics. Furthermore, the imaginary time dynamics also provide wave functions of the eigenstates which are not available from the autocorrelation function. In this Exercise, we will try to calculate the energies and wave functions of the harmonic oscillator and compare them to the exact analytic results, serving as a validation of the code.

Assignment: Set up a harmonic oscillator from Eq. (6.24) and an initial Gaussian wave packet from Eq. (6.23) with the following parameters: $m = 1$ a.u., $\omega = 0.1$ a.u., $x_0 = 6$ a.u., $p_0 = 0$ a.u. and $\alpha_0 = 0.3$. Propagate the wave packet in imaginary time to optimise the lowest ten states of the harmonic oscillator. Compare the energies and wave functions with analytic results.

Exercise: Double-well potential

The double-well potential is one of the simplest model potentials for chemical reactions with an activation barrier. In realistic scenarios, such reactions often exhibit an asymmetry between the two wells, where one minimum is energetically lower than the other. The asymmetric double-well potential can be described by the following expression:

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

where m is the mass, ω is the harmonic frequency, and ξ is the asymmetry parameter. This exercise explores the behaviour of wave functions in such a potential, focusing on the number of quantum states localized in the wells and the extent of tunnelling between them.

Assignment: Prepare a double well potential with parameters $m = 20$ a.u., $\omega = 2$ a.u. and $\xi = 0.5$ a.u. First, initialize the wave function such that it spans both wells and optimize at least fifteen states. How many states are located only in one well? Is there a state with energy below the barrier that tunnels between the wells? Can you describe the difference between states below the activation barrier and above it? Then, initialize the dynamics with a wave function in the form of Eq. (6.23) and parameters $x_0 = -0.90$ a.u., $p_0 = 0$ a.u. and $\alpha_0 = 2m\omega$ a.u., i.e. the initial wave function will be located in the left well. Observe how the wave function first converges to states in the left well before the lower energy contributions from the right well take over. Can you explain the behaviour?

8.5 Diffuse Monte Carlo

The Diffusion Monte Carlo (DMC) method is a numerical approach used to solve the time-independent Schrödinger equation for quantum systems. It is particularly effective for determining the ground state energy and wave function of systems where analytical solutions are not feasible. The method exploits the fact that the time-dependent Schrödinger equation, when reformulated in imaginary time, filters out all but the ground state energy. This transformation is achieved by replacing t with $-i\tau$, converting the Schrödinger equation into:

$$-\frac{\partial\Psi(x,\tau)}{\partial\tau} = \hat{H}\Psi(x,\tau),$$

where $\hat{H}$ is the Hamiltonian of the system. This equation resembles a diffusion equation with a reaction term, and the solution for large imaginary time, $\tau \to \infty$, converges to the ground state:

$$\Psi(x,\tau) \propto \phi_0(x)e^{-E_0\tau}.$$

Here, $\phi_0(x)$ is the ground state wave function, and E_0 is the ground state energy.

The DMC algorithm simulates this process using stochastic techniques. The wave function is represented by a population of ”walkers” in the system’s configuration space. These walkers evolve under two processes: diffusion, governed by the kinetic energy term, and branching, determined by the potential energy. The diffusion step is modeled as a random walk:

$$x_{n+1} = x_n + \sqrt{\Delta\tau}\,\xi,$$

where ξ is a Gaussian random variable with zero mean and unit variance, and $\Delta\tau$ is the time step. The branching step adjusts the population of walkers by replicating or removing them based on their weights:

$$W(x) = \exp\left(-\Delta\tau\left[V(x) - \langle V\rangle\right]\right),$$

where $V(x)$ is the potential energy at position x.

As the simulation progresses, the spatial distribution of walkers converges to the ground state wave function $\phi_0(x)$.

The DMC method has been successfully applied to systems such as the harmonic oscillator, where the potential energy is $V(x) = \frac{1}{2}kx^2$, and the exact ground state energy is $E_0 = \frac{1}{2}\hbar\omega$. For more complex systems, such as the hydrogen molecule, the DMC method provides an accurate numerical approach when analytical solutions are unavailable.

While the DMC method is powerful, it faces challenges, including the sign problem for fermionic systems and computational limitations for large systems. Nevertheless, its ability to accurately determine ground state properties makes it a cornerstone of quantum computational methods.

Solutions

Hückel theory

```python
"""Solution for chapter Hückel theory."""

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

# parameters of the molecule
natoms = 4  # number of atoms in the molecule
neighbors = [[1], [0, 2], [1, 3], [2]]  # neighbors of each atom

# parameters of the Hamiltonian matrix
alpha = -11.4  # alpha parameter (eV)
beta = -3.48  # beta parameter (eV)

# generate the Hamiltonian matrix full of zeros
H = np.zeros(shape=(natoms, natoms), dtype=int)

# fill the Hamiltonian matrix with the ones for the neighbors
for atom in range(natoms):
    for neighbor in neighbors[atom]:
        H[atom, neighbor] = 1

# print the Hamiltonian matrix
print(f"Hamiltonian matrix H: \n {np.array2string(H, separator='  ')}\n")

# diagonalize the Hamiltonian matrix
eigenvalues, eigenvectors = np.linalg.eigh(H)
eigenvectors = eigenvectors.T  # transposition because the eigenvectors are stored in columns and
    we want rows

# NOTE: the eigenvectors are sorted in ascending order, so we need to reverse them
eigenvectors = eigenvectors[::-1,:]

print(np.linalg.eigvalsh(H))

# calculate the energies
energies = alpha - beta*eigenvalues

# print the results for each orbital
for state in range(natoms):
    print(f"* Orbital {state + 1:d}")
    print(f"  Energy = alpha - {eigenvalues[state]:5.2f} * beta = {energies[state]:.2f} eV")
    print("  Psi = " + " + ".join(f"{eigenvectors[state, i]:.4f} * phi_{i + 1:d}" for i in range(
    natoms)))

# plot orbitals, this works onlu for polyenes
fig, ax = plt.subplots(natoms, 1, figsize=(4.0, 2.0*natoms))
for state in range(natoms):
    eigvec = eigenvectors[state]
    ax[state].plot(range(natoms), np.zeros(natoms), color='black', marker='o', ls='-', lw=1)
    ax[state].scatter(range(0, natoms), np.sign(eigvec)*np.ones(natoms)/2, marker='o', color='blue'
    , s=4000*abs(eigvec),
        alpha=abs(eigvec))
    ax[state].scatter(range(0, natoms), -np.sign(eigvec)*np.ones(natoms)/2, marker='o', color='red'
    , s=4000*abs(eigvec),
        alpha=abs(eigvec))
    ax[state].set_ylabel(f"$\psi_{state + 1:d}$")
    ax[state].set_xticks(range(0, natoms), [f"C$_{i + 1:d}$" for i in range(0, natoms)])
    ax[state].set_yticks([])
    ax[state].set_xlim(-0.5, natoms - 0.5)
    ax[state].set_ylim(-1, 1)
    ax[state].tick_params('both', direction='in', which='both', top=True, right=True)

plt.tight_layout()
plt.show()
```

Listing 8.2: Solution: Hückel theory

Hartree–Fock Method

```
# energies, number of occupied and virtual orbitals and the number of basis functions
E_HF, E_HF_P, VNN, nbf, nocc = 0, 1, 0, S.shape[0], sum(atoms) // 2; nvirt = nbf - nocc

# exchange integrals and the guess density matrix
K, D = J.transpose(0, 3, 2, 1), np.zeros((nbf, nbf))

# Fock matrix, coefficient matrix and orbital energies initialized to zero
F, C, eps = np.zeros((nbf, nbf)), np.zeros((nbf, nbf)), np.zeros((nbf))

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

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

Listing 8.3: HF method exercise code solution.

Integral Transform

```python
# 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 chemists' 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 physicists' notation
Jmsa = (Jms - Jms.swapaxes(1, 3)).transpose(0, 2, 1, 3)

# tensor epsilon_i^a
Emss = epsms[o] - epsms[v, None]

# tensor epsilon_ij^ab
Emsd = epsms[o] + epsms[o, None] - epsms[v, None, None] - epsms[v, None, None, None]
```

Listing 8.4: Integral transform exercise code solution.

2nd and 3rd Order Perturbative Corrections

```python
# 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**-1, 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**-1, Emsd**-1,
        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**-1, Emsd**-1,
        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**-1, Emsd**-1,
        optimize=True
    )
    print("    MP3 ENERGY: {:.8f}".format(E_HF + E_MP2 + E_MP3 + VNN))
```

Listing 8.5: MP2 and MP3 exercise code solution.

Full Configuration Interaction

```python
# generate the determinants
dets = [np.array(det) for det in it.combinations(range(2 * nbf), 2 * nocc)]

# define the CI Hamiltonian
Hci = np.zeros([len(dets), len(dets)])

# define the Slater-Condon rules, "so" is an array of unique and common spinorbitals
slater0 = lambda so: (
    sum(np.diag(Hms)[so]) + sum([0.5 * Jmsa[m, n, m, n] for m, n in it.product(so, so)])
)
slater1 = lambda so: (
    Hms[so[0], so[1]] + sum([Jmsa[so[0], m, so[1], m] for m in so[2:]])
)
slater2 = lambda so: (
    Jmsa[so[0], so[1], so[2], so[3]]
)

# filling of the CI Hamiltonian
for i in range(0, Hci.shape[0]):
    for j in range(i, Hci.shape[1]):

        # aligned determinant and the sign
        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(list(map(lambda l: np.array(l), [
            [aligned[k] for k in range(len(aligned)) if aligned[k] not in dets[i]],
            [dets[i][k] for k in range(len(dets[j])) if dets[i][k] not in aligned],
            [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

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

Listing 8.6: CI exercise code solution.

Coupled Cluster Singles and Doubles

```python
# 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 * [2 * nvirt] + 2 * [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 = 1.0 * 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.transpose(1, 0, 3, 2) - lccd3.swapaxes(0, 1) - lccd3.swapaxes(2, 3)

        # collect all the remaining 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 = +1.00 * np.einsum("klcd,acik,bdjl->abij",
            Jmsa[o, o, v, v], t2, t2, optimize=True
        )

        # permutation operators
        ccd1 -= ccd1.swapaxes(0, 1);
        ccd2 -= ccd2.swapaxes(2, 3);
        ccd4 -= ccd4.swapaxes(2, 3)

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

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

    # 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:

        # define taus
        tau, ttau = t2.copy(), t2.copy()

        # add contributions to the tilde tau
        ttau += 0.5 * np.einsum("ai,bj->abij", t1, t1, optimize=True).swapaxes(0, 0)
        ttau -= 0.5 * np.einsum("ai,bj->abij", t1, t1, optimize=True).swapaxes(2, 3)

        # add the contributions to tau
        tau += np.einsum("ai,bj->abij", t1, t1, optimize=True).swapaxes(0, 0)
        tau -= np.einsum("ai,bj->abij", t1, t1, optimize=True).swapaxes(2, 3)

        # define the deltas for Fae and Fmi
        dae, dmi = np.eye(2 * nvirt), np.eye(2 * nocc)

        # define Fae, Fmi and Fme
        Fae, Fmi, Fme = (1 - dae) * Fms[v, v], (1 - dmi) * Fms[o, o], Fms[o, v].copy()

        # add the contributions to Fae
        Fae -= 0.5 * np.einsum("me,am->ae",     Fms[o, v],        t1,   optimize=True)
        Fae += 1.0 * np.einsum("mafe,fm->ae",   Jmsa[o, v, v, v], t1,   optimize=True)
        Fae -= 0.5 * np.einsum("mnef,afmn->ae", Jmsa[o, o, v, v], ttau, optimize=True)

        # add the contributions to Fmi
        Fmi += 0.5 * np.einsum("me,ei->mi",     Fms[o, v],        t1,   optimize=True)
        Fmi += 1.0 * np.einsum("mnie,en->mi",   Jmsa[o, o, o, v], t1,   optimize=True)
        Fmi += 0.5 * np.einsum("mnef,efin->mi", Jmsa[o, o, v, v], ttau, optimize=True)

        # add the contributions to Fme
        Fme += np.einsum("mnef,fn->me", Jmsa[o, o, v, v], t1, optimize=True)

        # define Wmnij, Wabef and Wmbej
        Wmnij = Jmsa[o, o, o, o].copy()
        Wabef = Jmsa[v, v, v, v].copy()
        Wmbej = Jmsa[o, v, v, o].copy()

        # define some complementary variables used in the Wmbej intermediate
        t12  = 0.5 * t2 + np.einsum("fj,bn->fbjn", t1, t1,  optimize=True)

        # add contributions to Wmnij
        Wmnij += 0.25 * np.einsum("efij,mnef->mnij", tau, Jmsa[o, o, v, v], optimize=True)
        Wabef += 0.25 * np.einsum("abmn,mnef->abef", tau, Jmsa[o, o, v, v], optimize=True)
        Wmbej += 1.00 * np.einsum("fj,mbef->mbej",   t1,  Jmsa[o, v, v, v], optimize=True)
        Wmbej -= 1.00 * np.einsum("bn,mnej->mbej",   t1,  Jmsa[o, o, v, o], optimize=True)
        Wmbej -= 1.00 * np.einsum("fbjn,mnef->mbej", t12, Jmsa[o, o, v, v], optimize=True)

        # define the permutation arguments for Wmnij and Wabef and add them
        PWmnij = np.einsum("ej,mnie->mnij", t1, Jmsa[o, o, o, v], optimize=True)
        PWabef = np.einsum("bm,amef->abef", t1, Jmsa[v, o, v, v], optimize=True)

        # add the permutations to Wmnij and Wabef
        Wmnij += PWmnij - PWmnij.swapaxes(2, 3)
        Wabef += PWabef.swapaxes(0, 1) - PWabef

        # 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 += 1.0 * np.einsum("ei,ae->ai",     t1, Fae,              optimize=True)
        rhs_T1 -= 1.0 * np.einsum("am,mi->ai",     t1, Fmi,              optimize=True)
        rhs_T1 += 1.0 * np.einsum("aeim,me->ai",   t2, Fme,              optimize=True)
        rhs_T1 -= 1.0 * 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)

        # contracted F matrices that used in the T2 equations
        Faet = np.einsum("bm,me->be", t1, Fme, optimize=True)
        Fmet = np.einsum("ej,me->mj", t1, Fme, optimize=True)

        # define the permutation arguments for all terms in the equation for T2
        P1  = np.einsum("aeij,be->abij",    t2,     Fae - 0.5 * Faet, optimize=True)
        P2  = np.einsum("abim,mj->abij",    t2,     Fmi + 0.5 * Fmet, 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)

```

```python
        # 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 and save the previous iteration
        t1, t2 = rhs_T1 / Emss, rhs_T2 / Emsd; E_CCSD_P = E_CCSD

        # evaluate the energy
        E_CCSD  = 1.00 * np.einsum("ia,ai",      Fms[o, v],        t1    )
        E_CCSD += 0.25 * np.einsum("ijab,abij",  Jmsa[o, o, v, v], t2    )
        E_CCSD += 0.50 * 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 8.7: CCD and CCSD method exercise code solution.

Quantum dynamics in real time

```python
"""Solution for chapter Quantum dynamics in real time."""

# 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 (necessary to set for each simulation)
simtime = 100.0  # simulation time in atomic time units
dt = 0.2  # 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
```

```python
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 the inverse Fourier transform to momentum space
    psi_k = np.fft.ifft(psi, norm="ortho")
    # apply expT in momentum space
    psi_k *= expT
    # finally the Fourier transform back to coordinate space
    psi = np.fft.fft(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.ifft(psi, norm="ortho")
    psi_t = np.fft.fft(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:.8f}, <E> = {energy:.6f}, <V> = {energyV:.6f}, <T> = {energyT:.6f}"
    )

    # 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"Time: {t:.2f} a.t.u.\n" + r"$\langle \psi | \psi \rangle$" + f" = {norm:.8f}; $E =
    $ {energy:.6f} 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=0.0001)  # update plot and wait given interval

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

Listing 8.8: Solution: quantum dynamics in real-time

Energy spectrum and autocorrelation function

```python
"""Solution for chapter Energy spectrum and autocorrelation function."""

# 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 = -25, 25  # minimum and maximum of x
simtime = 10000.0  # simulation time in atomic time units
dt = 0.2  # time step in atomic time units
m = 1.0  # mass in atomic units
gamma = 1/200  # damping factor for autocorrelation function [exp(-kappa*t)]

# 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 = 0.3, 5.7, 0.0
psi = (2*alpha/np.pi)**0.25*np.exp(-alpha*(x - x0)**2 + 1j*p0*(x - x0), dtype=complex)
psi /= np.sqrt(np.real(np.trapz(y=np.conjugate(psi)*psi, x=x)))

# save the initial wave function for calculating the autocorrelation function
psi0 = psi  # initial wave function
time, autocorr = [], []  # empty lists for appending values of time and autocorrelation function

# propagation
```

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

    # calculate the autocorrelation function S(t) = <psi(0)|psi(t)>
    overlap = np.trapz(y=np.conjugate(psi0)*psi, x=x)

    autocorr.append(overlap)  # appending the overlap to our autocorrelation function list
    time.append(t)  # appending t to our time list

    # print simulation data
    print(f"--Time: {t:.2f} a.t.u.")
    print(f"  <psi|psi> = {norm:.8f}, <E> = {energy:.6f}, <V> = {energyV:.6f}, <T> = {energyT:.6f}"
    )

### autocorrelation function section ###
autocorr = np.array(autocorr) # converting the autocorrelation function to a numpy array
time = np.array(time) # converting the time to a numpy array

# apply the damping to the autocorrelation function in form of exp(-kappa*time)
autocorr *= np.exp(-gamma*time)

# extend the autocorrelation function to negative times assuming that S(t) = S^*(-t)
time = np.concatenate([-time[::-1], time])  # new time array in range [-t_max, t_max]
autocorr = np.concatenate([np.conjugate(autocorr[::-1]), autocorr])  # new symmetric autocorr in
    range [-t_max, t_max]

# calculate spectrum from autocorrelation function and the frequency axis corresponding to it
spectrum = np.fft.ifft(autocorr)
```

```python
freq = 2*np.pi*np.fft.fftfreq(len(time), d=dt)

# plot results
fig, axs = plt.subplots(1, 2, figsize=(8, 3), tight_layout=True)

# autocorrelation function
axs[0].plot(time, np.real(autocorr), label=r'$\mathcal{Re}[S(t)]$')
axs[0].plot(time, np.imag(autocorr), label=r'$\mathcal{Im}[S(t)]$')
axs[0].set_xlabel('Time (a.u.)')
axs[0].set_ylabel(r'$S(t)$')
axs[0].set_title('Autocorrelation Function')
axs[0].legend(frameon=False, labelspacing=0)

# spectrum
axs[1].plot(hbar*freq, np.abs(spectrum))
axs[1].set_xlim(0, np.max(hbar*freq[spectrum > np.max(spectrum)/1000]))
axs[1].set_ylim(0)
axs[1].set_xlabel('Energy (a.u.)')
axs[1].set_ylabel(r'$\mathcal{F}^{-1}[S(t)]$')
axs[1].set_title('Spectrum')

# searching for local maxima of the spectrum
print(f"\nMaxima of the spectrum:")
abs_spectrum = np.abs(spectrum)
loc_max_bool = (abs_spectrum[1:-1] > abs_spectrum[:-2]) & (abs_spectrum[1:-1] > abs_spectrum[2:])
loc_max_index = np.where(loc_max_bool)[0] + 1
loc_max_energies = hbar*freq[loc_max_index]
for index, en in enumerate(loc_max_energies):
    intensity = abs_spectrum[loc_max_index[index]]
    print(f" * State {index}: E = {en:.5f} a.u.; I = {intensity:.5e}")
    axs[1].axvline(en, lw=1, color='black', alpha=0.1)
    axs[1].scatter(en, intensity, marker='x', color='black', s=20)

plt.show()
```

Listing 8.9: Solution: autocorrelation fucntion

Imaginary-time dynamics and stationary states

```python
"""Solution for chapter Imaginary-time dynamics and stationary states."""

# 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 = 200.0  # simulation time in atomic time units
dt = 0.2  # time step in atomic time units
m = 1.0  # mass in atomic units
nstates = 15  # number of states to be calculated
conv_thresh = 1e-8  # energy convergence threshold in a.u.

# 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
```

```python
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(-V*dt/2)

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

# initiate wave function; be careful, it must be a complex numpy array
alpha, x0, p0 = 0.1, 1.5, 0.0
psi0 = (2*alpha/np.pi)**0.25*np.exp(-alpha*(x - x0)**2 + 1j*p0*(x - x0), dtype=complex)

# create and empty array for final optimized wave functions and energies
psi_optimized = np.zeros(shape=(nstates, len(x)))
energy_optimized = np.zeros(shape=(nstates))

# 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

print("\nImaginary-time quantum dynamics.\n-----------------------------------------\n")
# loop through states
for state in range(nstates):
    # creating wave function for propagation
    psi = psi0

    # first, we need to calculate energy of the initial wave function and store it into energy_old
    variable, so that we
    # can compare at the end of the cycle the new and old energies
    # calculating norm for the expectation value calculation
    norm = np.real(np.trapz(y=np.conjugate(psi)*psi, x=x))
    # 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.ifft(psi, norm="ortho")
    psi_t = np.fft.fft(T*psi_k, norm="ortho")
    energyT = np.real(np.trapz(y=np.conjugate(psi)*psi_t, x=x))/norm
    # total energy <E>
    energy_old = energyV + energyT

    # propagation
    t = 0  # set initial time to 0
    print(f"\nState:  {state+1:d}")
    while t < simtime:  # loop until simulation time is reached
        # propagate half-step in V
        psi *= expV

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

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

        # calculate new time after propagation
        t += dt

        # remove contributions from previous states, i.e. remove c_i*psi_optimized, where c_i = <psi_optimized|psi>
        for s in range(state):
            c_i = np.trapz(np.conjugate(psi_optimized[s])*psi, x=x)
            psi -= c_i*psi_optimized[s]

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

        # renormalize the wave function so that the norm is one again
        psi /= np.sqrt(norm)

        # calculate norm for the expectation value calculation
        norm = np.real(np.trapz(y=np.conjugate(psi)*psi, x=x))

        # 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.ifft(psi, norm=”ortho”)
        psi_t = np.fft.fft(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:.8f}, <E> = {energy:.6f}, <V> = {energyV:.6f}, <T> = {energyT:.6f}”)

        # plot
        plt.cla()  # clean figure
        # plot current wave function
        plt.fill_between(x, psi**2 + energy, energy, alpha=0.2, color=’C1’)  # plot |wf|
        plt.text(np.min(x), energy, f”{energy:.4f}”)
        # plot previously optimized wave functions
        for s in range(state):
            plt.fill_between(x, psi_optimized[s]**2 + energy_optimized[s], energy_optimized[s], alpha=0.1, color=’black’)
            plt.text(np.min(x), energy_optimized[s], f”{energy_optimized[s]:.4f}”)

        # plot potential
        plt.plot(x, V, color=’black’)
        # plot title and labels
        plt.title(f”State: {state+1}; Time: {t:.2f} a.t.u.\n” + r”$\langle \psi | \psi \rangle$” +
                  f” = {norm:.8f}; $E = $ {energy:.6f} a.u.”)
        plt.xlabel(”$x$ (a.u.)”)
        plt.ylabel(”$|\Psi|^2$”)

        # set new ymin, ymax
        max_wf = np.max(psi**2)
        ymax = max([ymax, max_wf + energy])
        plt.ylim(ymin, ymax)
        plt.pause(interval=0.0001)  # update plot and wait given interval

        # check energy change between the current and the previous step
        # if |energy - energy_old| < conv threshold, then the wave function is optimized and break the cycle
```

```
        if np.abs(energy - energy_old) < conv_thresh:
            break
        else:
            energy_old = energy

    # save the final wave function and energy
    psi_optimized[state] = psi
    energy_optimized[state] = energy

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

Listing 8.10: Solution: quantum dynamics in imaginary-time

```
    n = 2000;
    nmax = 5000;
    limits = [-20, 20]
    ts = 0.01
    iters = 1000

    def V(x):
        return 0.5 * x**2

    def diffuse(walkers, ts):
        return walkers + np.sqrt(ts) * np.random.normal(size=walkers.shape)

    def branch(walkers, energy, ts):
        W = np.exp(-ts * (energy - np.mean(energy)))
        copies = list(map(lambda w: min(w, 3), (W + np.random.uniform(size=walkers.shape)).astype(
    int)))
        return np.repeat(walkers, copies)

    walkers, energies = np.random.uniform(limits[0], limits[1], size=(n)), []

    for step in range(iters):

        walkers = diffuse(walkers, ts)

        energy = V(walkers); eref = np.mean(energy)

        energies.append(eref)

        walkers = branch(walkers, energy, ts)

        if len(walkers) > nmax:
            walkers = walkers[np.random.choice(len(walkers), nmax, replace=False)]

        print(eref, len(walkers))
```

Listing 8.11: DMC exercise code solution.

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.
- Reiter, S.; Keefer, D.; de Vivie-Riedle, R. *Exact Quantum Dynamics (Wave Packets) in Reduced Dimensionality*; In Quantum Chemistry and Dynamics of Excited States (eds L. González and R. Lindh), pp. 355-381, 2020.

Acknoledgement

We thank Mgr. Ing. Tomáš Ovad for the help in presenting the text on e-learning.

List of acronyms

CC	Coupled Cluster. 2, 20, 21
CCD	Coupled Cluster Doubles. 2, 20, 21, 55, 66
CCSD	Coupled Cluster Singles and Doubles. 2, 20, 21, 22, 23, 55, 66
CI	Configuration Interaction. 2, 16, 17, 19, 52, 66
CISD	Configuration Interaction Singles and Doubles. 16
CISDT	Configuration Interaction Singles, Doubles and Triples. 16
DIIS	Direct Inversion in the Iterative Subspace. 9
FCI	Full Configuration Interaction. 2, 16, 17
FT	Fourier transform. 28, 29, 37, 47
HF	Hartree–Fock. 2, 3, 5, 6, 7, 8, 9, 11, 12, 14, 15, 16, 17, 20, 21, 22, 50, 66
HT	Hückel theory. 3, 4, 5, 6
IFT	inverse Fourier transform. 27, 28, 29, 35, 36, 37, 38, 39
MP2	Møller–Plesset Perturbation Theory of 2nd Order. 15, 52, 66
MP3	Møller–Plesset Perturbation Theory of 3rd Order. 15, 52, 66
MPPT	Møller–Plesset Perturbation Theory. 2, 14
MS	Molecular Spinorbital. 10, 11, 13, 15, 17, 21, 66
post-HF	post-Hartree–Fock. 10, 16, 20
RHF	Restricted Hartree–Fock. 7, 10
SCF	Self-Consistent Field. 8, 12, 66
TDSE	time-dependent Schrödinger equation. 25, 26, 34, 35
TISE	time-independent Schrödinger equation. 34, 41, 42

List of codes