Introduction to quantum dynamics simulation with Python

Shunsuke A. Sato *

August 27, 2025

Contents

*Tohoku University

Preface

This note is provided to support learning numerical methods for investigating the steady states and dynamics of quantum systems. Numerical computation is a powerful analytical tool that uses computers to explore problems that are difficult to solve with paper and pencil alone.

In the first half of the note, we cover the basics of programming with Python and computational physics. We then use one-dimensional quantum systems as a subject to learn numerical methods for analyzing quantum systems, including the time evolution of wave packets and the computation of ground states.

The contents of this note are updated from time to time. For the latest material, please check the most recent version of the note at the URL below.

`https://shunsuke-sato.github.io/page/etc/lecture_notes/LectureNoteForComputationalPhysics_en.pdf`

1 Getting Familiar with Python

In this section, we will familiarize ourselves with programming in Python and prepare to perform numerical computations.

1.1 Running Python

First, let's learn the basic steps of programming with Python. A Python program can be executed by following these steps:

1. Write code in the Python language and save it as a file with the `.py` extension.
2. Execute the file prepared in the previous step using the `python` command.

To learn this procedure, write the following Python code and save it with the filename `hello.py`.

Source code 1: Hello world program

```
1 print('Hello world!!')
```

Here, the `print` statement is a function used to output results. In the above program, the print statement is used to output the string `Hello world!!`. Also, in Python, you can enclose characters in single quotes (') or double quotes (") to treat the enclosed characters as a **string variable**.

Once the above code is ready, try running the Python program by entering and executing the following command in the terminal:

```
$ python hello.py
Hello world!!
```

Then, you should be able to confirm that the string `Hello world!!` is output.

Up to this point, you have understood the basic process of creating and running a Python program. In the next section, we will learn about **variables** and **data types** in Python.

1.2 Variable Types in Python

Not only in Python but in programming in general, **variables** and **variable types** are extremely important concepts. A variable is a storage location for various data used in a program. The kind of data that a variable holds is called its **type**.

- **Variable**: A "named box" that stores data
- **Type**: The "kind of data" inside the box (integer, floating-point number, string, etc.)

To understand variables and types in Python, let's write and run the following program.

Source code 2: Code to check variable types

```
1  a = 2
2  b = 3.0
3  c = 'Tohoku'
4
5  print('a=',a)
6  print('b=',b)
7  print('c=',c)
8
9  print('type(a)=',type(a))
10 print('type(b)=',type(b))
11 print('type(c)=',type(c))
```

Execution Result (example)

```
a= 2
b= 3.0
c= Tohoku
type(a)= <class 'int'>
type(b)= <class 'float'>
type(c)= <class 'str'>
```

In Python, the equals sign (=) represents the assignment operation to a variable. Using this assignment operation, in lines 1–3 of Source Code 2, the values `2, 3.0, 'Tohoku'` are assigned to the variables `a, b, c`, respectively.

In lines 5–7 of Source Code 2, the assigned data are output using the `print` statement. By looking at this output, we can confirm that the correct data have been assigned to the variables.

In lines 9–11 of Source Code 2, the types of the above variables `a, b, c` are checked using the `type` function, and the results are output with the `print` statement. By using `type(a)` in this way, you can check the type of variable `a`.

When executing Source Code 2, you can confirm that the types of variables `a, b, c` are **integer type (int)**, **floating-point type (float)**, and **string type (str)**, respectively. In the above program, the type of each variable is determined at the time the data are assigned, depending on the data type. In this way, Python automatically infers the data type when assignments or similar operations are made. In contrast, in other languages (for example, C or Fortran), it is sometimes necessary to explicitly specify the type of a variable before using it. At first glance, automatic type inference may seem like a very convenient feature, but explicitly specifying types can help reduce errors (bugs) in the source code, so type specification also has its advantages. Moreover, when writing source code in Python, it is also possible to explicitly specify types.

1.3 Basic Mathematical Operations in Python

Many physics simulations are executed by combining fundamental mathematical operations such as the four arithmetic operations and exponentiation. Here, we will learn about the basics of mathematical operations in Python by using Python code that performs such calculations. First, let us look at the Python source code 3.

Source code 3: Code for Arithmetic Operations

```
a = 3.0
b = 5.0

c = a + b
d = a - b
e = a * b
f = a / b
g = a ** b

print('a=',a)
print('b=',b)

print('a+b=',c)
print('a-b=',d)
print('a*b=',e)
print('a/b=',f)
print('a**b=',g)
```

In this program, in the first and second lines, the real numbers `3.0` and `5.0` are assigned to the variables `a` and `b`, respectively.

From lines 4 to 8 of the code, the four arithmetic operations and exponentiation are performed using the prepared variables `a,b`, and the results are assigned to new variables `c,d,e,f,g`. In Python, addition is represented by the symbol `+`, subtraction by `-`, multiplication by `*`, division by `/`, and exponentiation by `**`.

Finally, in lines 14 to 18 of the code, the results of the above arithmetic operations and exponentiation are output using the `print` statement.

Save the above code into a file with the `.py` extension and execute it on the terminal using the `python` command. Then, the instructions written in the code will be executed in order from top to bottom. Check whether the results obtained match the expected results.

2 Numerical Differentiation

In this section, we learn about **numerical differentiation**, which evaluates derivatives of functions using numerical computation.

2.1 Finite Difference Approximation and Forward Difference

Before performing differentiation numerically, let us recall the mathematical definition of a derivative. The derivative of a function $f(x)$ is defined as

$$f'(x) = \lim_{h \to 0} \frac{f(x+h) - f(x)}{h}. \tag{1}$$

Based on this definition, we consider approximating the derivative $f'(x)$ of a function $f(x)$ by using a sufficiently small h as follows:

$$f'(x) \approx \frac{f(x+h) - f(x)}{h}. \tag{2}$$

An approach that approximates derivatives using a finite step width h like this is called a **finite difference approximation**. In particular, an approximation formula such as Eq. (2) is called the **forward difference formula**. Its meaning will become clear by comparing it with the **central difference formula** described in the next subsection.

To check the effectiveness of the finite difference approximation, let us use Eq. (2) to compute the derivative of the function $f(x) = e^x$ at $x = 0$ and compare it with the exact value $f'(0) = e^0 = 1$. A sample source code is shown below.

Source code 4: Evaluation of a derivative using the forward difference formula

```
1  import numpy as np
2
3
4  x = 0.0
5  h = 0.1
6
7  fx = np.exp(x)
8  fxph = np.exp(x+h)
9
10 num_dfdx = (fxph-fx)/h
11 ana_dfdx = np.exp(x)
12
13 error = np.abs(num_dfdx - ana_dfdx)
14
15 # Print results
16 print(f'For␣h␣=␣{h}:')
17 print(f'Numerical␣derivative␣of␣exp({x})␣=␣{num_dfdx}')
18 print(f'Analytical␣derivative␣of␣exp({x})␣=␣{np.exp(x)}')
19 print(f'Error␣=␣{error}')
```

In the first line of the source code, `import numpy as np` imports an extension module called `numpy` for numerical computation. By appending `as np` during the import, we can call `numpy` using the abbreviation `np`. In this example, the module is imported in order to call the exponential function within the `numpy` module.

In the above program, to obtain the derivative at $x = 0$ with a step size $h = 0.1$, lines 4 and 5 set the values using `x = 0.0` and `h = 0.1`.

In lines 7 and 8, the values of the function at x and $x + h$ are computed using `numpy`'s exponential function `np.exp()` and assigned to the variables `fx` and `fxph`. Furthermore, in line 10, the derivative is evaluated using the forward difference formula, Eq. (2). In line 13, the approximate derivative obtained from the above calculations and the difference from the exact derivative are output.

By running the above program, let us verify whether the derivative can be well approximated by the finite difference approximation. Also, by changing the step size h, investigate how the accuracy of the approximation changes.

2.2 Central Difference Formula and Accuracy of Finite Difference Approximation

The finite difference approximation formula, Eq. (2), from the previous section shows that as the step size h becomes smaller, the error also decreases. The magnitude of the error can be estimated using a Taylor expansion as follows:

$$f(x+h) = f(x) + f'(x)h + \mathcal{O}(h^2). \tag{3}$$

Therefore,

$$f'(x) = \frac{f(x+h) - f(x)}{h} + \mathcal{O}(h). \tag{4}$$

Thus, the error of the forward difference formula is $\mathcal{O}(h)$, and in the limit of small h, the error decreases proportionally to h.

Now, let us consider the following two Taylor expansions:

$$f(x+h) = f(x) + f'(x)h + \frac{1}{2}f''(x)h^2 + \mathcal{O}(h^3), \tag{5}$$

$$f(x-h) = f(x) - f'(x)h + \frac{1}{2}f''(x)h^2 + \mathcal{O}(h^3). \tag{6}$$

By taking the difference of these two equations, we obtain the following finite difference formula:

$$f'(x) = \frac{f(x+h) - f(x-h)}{2h} + \mathcal{O}(h^2). \tag{7}$$

This finite difference formula is called the **central difference formula**, which takes differences symmetrically in both the positive and negative directions. The error term is $\mathcal{O}(h^2)$, which means the error is of order h^2.

Here, let us examine how the accuracy of the forward difference formula, Eq. (2), and the central difference formula, Eq. (7), behave with respect to the step size h by actually writing and running a program. Similar to the example above, the following program computes the derivative of the function $f(x) = e^x$ at $x = 0$.

Source code 5: Evaluation of derivative values using forward difference and central difference formulas

```
from matplotlib import pyplot as plt
import numpy as np

# Parameters
x = 0.0  # Point at which derivative is evaluated
step_sizes = np.array([1e-6, 1e-5, 1e-4, 1e-3, 1e-2, 1e-1, 1.0])  # Differentiation step sizes

# Numerical derivatives
forward_diff = (np.exp(x + step_sizes) - np.exp(x)) / step_sizes
central_diff = (np.exp(x + step_sizes) - np.exp(x - step_sizes)) / (2.0 * step_sizes)

# Error evaluation
error_forward = np.abs(forward_diff - np.exp(x))
error_central = np.abs(central_diff - np.exp(x))

# Plotting the data
plt.plot(step_sizes, error_forward, label="Forward Difference", marker='o')
plt.plot(step_sizes, error_central, label="Central Difference", marker='x')
plt.xscale('log')
plt.yscale('log')
plt.xlabel('Step Size (h)')
plt.ylabel('Error')
plt.title('Error in Numerical Differentiation')
plt.legend()
plt.grid(True)

# Saving the plot
plt.savefig("error_derivative.png")
plt.show()
```

Let us now take a look at the above Python program. Line 4 is a comment line beginning with `#`; it does not perform any computation but serves as a reference for humans reading the program. In line 5, the point at which the derivative is computed is specified by `x = 0.0`. Then, various step sizes used in evaluating the derivative via difference methods are defined as a NumPy array.

In line 9, the derivative is numerically evaluated using the forward difference formula for each value in the step size array `step_sizes`. In line 10, the derivative is numerically evaluated using the central difference formula.

Next, in lines 13 and 14, the differences between the numerically evaluated derivatives (using forward and central difference formulas) and the exact derivative value are computed, thereby evaluating the error of each difference formula.

Furthermore, from line 16 onward, the program uses `matplotlib` to plot the errors of the difference formulas obtained from the above calculations. Figure 1 shows the figure generated by this code. Noting that both the horizontal and vertical axes are logarithmic, we can see that the error of the forward difference approximation (Forward difference; blue line) and the error of the central difference approximation (Central difference; orange line) scale proportionally to different powers of h. This behavior is consistent with the earlier discussion of error magnitudes.

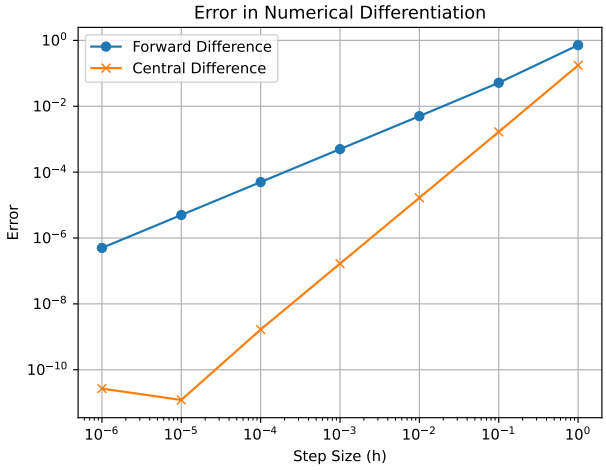

Figure 1: Behavior of the errors in the forward difference and central difference approximations.

2.3 Numerical Differentiation of the Second Derivative

In the previous section, we performed numerical computation of the first derivative using finite difference approximation. Here, we will extend the knowledge gained from the first derivative to learn how to numerically evaluate the second derivative. By applying the Taylor expansion in Eq. (6), we can obtain the following relation:

$$\frac{f(x+h) - 2f(x) + f(x-h)}{h^2} = \frac{d^2 f(x)}{dx} + \mathcal{O}(h^2). \tag{8}$$

Using this equation, the second derivative can be evaluated numerically. Moreover, the error is of order (h^2).

The following Python code numerically computes the second derivative $f''(x)$ of the function $f(x) = \cos(x)$ and outputs the result to the file `cos_derivative.dat`. Try writing a similar code yourself to become familiar with second derivatives and Python programming.

Source code 6: Evaluation of the second derivative using finite difference approximation

```python
import numpy as np
from matplotlib import pyplot as plt

# Constants
dx = 0.1
nx = 128
x_start = -6.3
x_end = 6.3

# Generate x values
x = np.linspace(x_start, x_end, nx)

# Compute function values
fx = np.cos(x)
fx_pdx = np.cos(x + dx)
fx_mdx = np.cos(x - dx)

# Second derivative using central difference
d2fdx2 = (fx_pdx - 2 * fx + fx_mdx) / dx**2

# Plotting the function and its second derivative
plt.plot(x, fx, label="f(x) = cos(x)")
plt.plot(x, d2fdx2, label="f''(x)")

plt.xlabel('x')
plt.ylabel('f(x)')
plt.title('Function and Second Derivative')
plt.legend()
plt.savefig("second_derivative_cos.png")
plt.show()
```

Figure 2 shows a comparison of $f(x) = \cos(x)$ and $f''(x) = -\cos(x)$ obtained from executing the above program.

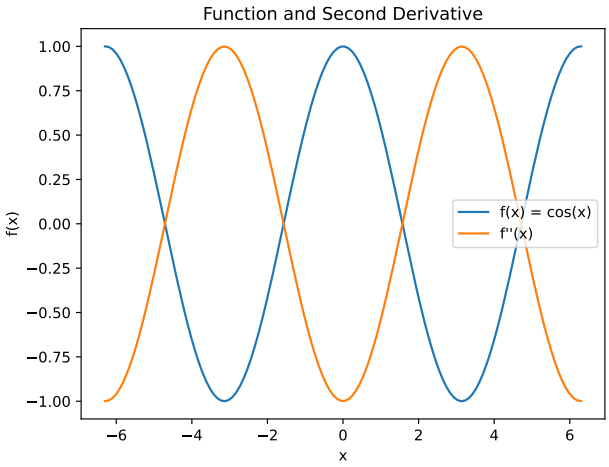

Figure 2: Second order derivative of cos function.

3 Numerical Integration

3.1 Trapezoidal Rule

In this section, we study **numerical integration**, which evaluates the integral of a function using numerical computation. Let us first consider the following one-dimensional integral:

$$S = \int_a^b dx f(x). \tag{9}$$

Here, we divide the integration interval ($a \le x \le b$) into N subdivisions. The width of each subdivision Δx is given by $\Delta x = (b-a)/N$. We can then rewrite the integral S as the sum of integrals over the subdivided intervals as follows:

$$\begin{aligned} S &= \int_a^b dx f(x) = \int_a^{a+\Delta x} dx f(x) + \int_{a+\Delta x}^{a+2\Delta x} dx f(x) + \int_{a+2\Delta x}^{a+3\Delta x} dx f(x) + \cdots + \int_{a+(N-1)\Delta x}^{b} dx f(x) \\ &= \sum_{j=0}^{N-1} \int_{a+j\Delta x}^{a+(j+1)\Delta x} dx f(x). \end{aligned} \tag{10}$$

If the subdivision width Δx is sufficiently small, the integrand $f(x)$ can be approximated by a linear function passing through the two endpoints of each small integration interval. That is, within the small interval ($a + j\Delta x \le x \le a + (j+1)\Delta x$), $f(x)$ can be approximated as

$$f(x) \approx f(a+j\Delta x) + \frac{f(a+(j+1)\Delta x) - f(a+j\Delta x)}{\Delta x} \left(x - a - j\Delta x\right). \tag{11}$$

The integral over the small interval can then be approximated as

$$\int_{a+j\Delta x}^{a+(j+1)\Delta x} dx f(x) \approx \frac{f(a+(j+1)\Delta x) - f(a+j\Delta x)}{2} \Delta x. \tag{12}$$

Thus, Eq. (10) can be approximated as follows:

$$\begin{aligned} S &= \sum_{j=0}^{N-1} \int_{a+j\Delta x}^{a+(j+1)\Delta x} dx \ f(x) \\ &\approx \sum_{j=0}^{N-1} \int_{a+j\Delta x}^{a+(j+1)\Delta x} dx \ \frac{f\left(a+(j+1)\Delta x\right) + f\left(a+j\Delta x\right)}{2} \\ &= \sum_{j=0}^{N-1} \frac{f\left(a+(j+1)\Delta x\right) + f\left(a+j\Delta x\right)}{2} \int_{a+j\Delta x}^{a+(j+1)\Delta x} dx \\ &= \sum_{j=0}^{N-1} \frac{f\left(a+(j+1)\Delta x\right) + f\left(a+j\Delta x\right)}{2} \Delta x \\ &= \frac{f(a)}{2}\Delta x + \left[\sum_{j=1}^{N-1} f(a+j\Delta x)\Delta x\right] + \frac{f(b)}{2}\Delta x. \end{aligned} \tag{13}$$

The approximation formula in Eq. (13) is called the **trapezoidal rule**, which is an approximate formula for the integral S. In the limit of a large number of subdivisions ($N \to \infty$, $\Delta x \to 0$), the trapezoidal rule, Eq. (13) coincides with the exact value of the integral S.

Let us evaluate the following integral numerically using the trapezoidal rule:

$$S = \int_1^2 dx \frac{1}{x} = \log(2) - \log(1) = \log(2) \approx 0.6931471805599453 \tag{14}$$

Source Code 7 is a Python code that numerically evaluates the above integral. To become familiar with numerical integration and Python programming, refer to this sample code and try writing your own code to evaluate the integral.

Source code 7: Example of numerical integration using the trapezoidal rule

```
import numpy as np

a = 1.0
b = 2.0
n = 64

h = (b-a)/n

s = (1.0/a+1.0/b)/2.0
for i in range(1,n):
    x = a + i*h
    s += 1/x

s = s*h

print('num. integral = ',s)
print('log(2)        = ',np.log(2))
```

4 Solution of First-Order Ordinary Differential Equations

Differential equations are extremely important for understanding various phenomena, but except in limited cases, it is difficult to obtain analytic solutions. Even in such cases, it is sometimes possible to solve differential equations numerically. In this section, we learn how to solve first-order ordinary differential equations numerically. Let us first consider the following first-order ordinary differential equation in one variable:

$$\frac{dy(x)}{dx} = f\left(x, y(x)\right). \tag{15}$$

4.1 Euler Method

We begin by learning the most basic method, the Euler method. As in Section 2.1, where we discussed the forward difference formula, Eq. (2), let us recall the definition of the derivative. The derivative of a function $y(x)$ is defined as follows:

$$y'(x) = \lim_{h \to 0} \frac{y(x+h) - y(x)}{h}. \tag{16}$$

If we assume h is sufficiently small, we can approximate the derivative using the forward difference formula as

$$y'(x) \approx \frac{y(x+h) - y(x)}{h}. \tag{17}$$

Rewriting this equation, we obtain

$$y(x+h) \approx y(x) + y'(x)h. \tag{18}$$

Looking at equation (18), we see that the right-hand side depends only on the information of $y(x)$ and $y'(x)$ at x. Using this information, we can approximately evaluate the function value $y(x+h)$ at $x+h$. Therefore, by repeating this procedure recursively, we can evaluate the function value $y(x)$ at any value of x. This numerical method of solving differential equations is called the Euler method.

Below, we describe the procedure of the Euler method in more detail:

1. First, in the differential equation, Eq. (15), set the initial condition $y(x_0) = y_0$.
2. Next, evaluate $f(x_0, y(x_0))$ and obtain the derivative value $y'(x_0)$.
3. Using the evaluated derivative $y'(x_0)$, apply Eq. (18) to obtain the function value $y(x_0 + h)$ at $x_0 + h$.
4. Based on the evaluated $y(x_0+h)$, calculate the function value $f(x_0+h, y(x_0+h))$ and obtain the derivative $y'(x_0 + h)$ at $x_0 + h$.
5. Using the evaluated derivative $y'(x_0+h)$, apply Eq. (18) to obtain the function value $y(x_0 + 2h)$ at $x_0 + 2h$.
6. Repeat the same steps thereafter.

By repeatedly performing this procedure, we can solve the differential equation numerically. As a practice problem in solving differential equations numerically with the Euler method, let us try solving the following differential equation numerically under the initial condition $y(0) = 1$:

$$y'(x) = -y(x). \tag{19}$$

The solution of this differential equation is the exponential function $y(x) = e^{-x}$.

Let us write your own code to solve the differential equation and check how the accuracy of the obtained solution changes when the step size h is varied. An example in Python is shown below:

Source code 8: Solving a differential equation with the Euler method

```
from matplotlib import pyplot as plt
import numpy as np

# compute dy/dx
def dydx(x,y):
    return -y

# compute y(x+h)
def euler_method(y,x,h):
    return y+dydx(x,y)*h


xi = 0.0
xf = 5.0
n = 10

h = (xf-xi)/n

# initial condition
x = xi
y = 1.0


x_j = np.zeros((n+1))
y_euler = np.zeros((n+1))
x_j[0] = x
y_euler[0] = y

for i in range(n):
    x = xi + i*h
    y = euler_method(y_euler[i],x,h)
    x_j[i+1] = x+h
    y_euler[i+1] = y


# plot
plt.plot(x_j, y_euler, label="Euler method")
plt.plot(x_j, np.exp(-x_j), label="Exact", linestyle='dashed')

plt.xlabel('x')
plt.ylabel('y')
plt.legend()
plt.savefig("result_Euler.png")
plt.show()
```

In Source Code 8, a user-defined function is created using `def`. By defining a new function with the `def` statement, it becomes easier to write programs that repeat the same procedure. A function definition can be written as follows:
`def function_name(argument1, argument2, argument3, ···):`
The steps to be carried out by the function are written in an indented block, and finally the function is terminated with a `return` statement, returning control to the main program. At this point, after the `return` statement, the output of the function, known as the function's **return value**, can be specified.

The result obtained by executing Source Code 8 is shown in Fig. 3

4.2 Solution by Heun Method

In the previous section, we discussed Euler's method as the simplest technique. However, Euler's method has difficulty achieving high accuracy in calculations and is not often used in practical applications. Here, we describe **Heun method** as a relatively simple method that offers higher accuracy than Euler's method.

In the previous section, Euler's method was derived based on the forward difference approximation. Here, we consider deriving a numerical solution method based on the central difference formula, Eq. (7), which is more accurate than the forward difference formula. By using the central

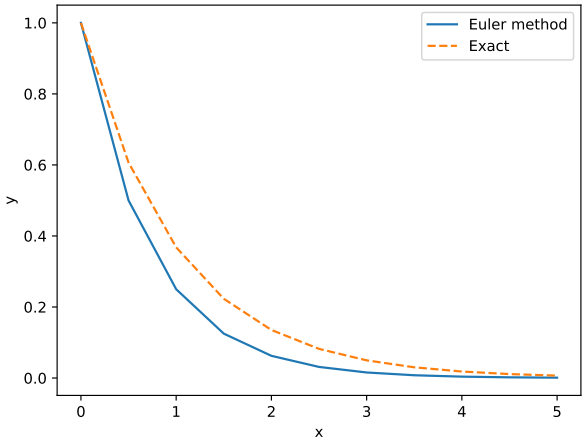

Figure 3: An example of solving a differential equation using the Euler method.

difference formula, we can obtain the following approximation:

$$\frac{dy\left(x+\frac{h}{2}\right)}{dx} \approx \frac{y(x+h)-y(x)}{h}. \tag{20}$$

Furthermore, in the same way as with Euler's method, we can rewrite this as follows:

$$y(x+h) \approx y(x) + \frac{dy\left(x+\frac{h}{2}\right)}{dx}h. \tag{21}$$

Here, by replacing the derivative value $\frac{dy\left(x+\frac{h}{2}\right)}{dx}$ with the average of the derivative values at $x+h$ and x, and further applying the original differential equation, Eq. (15), we obtain the following relation:

$$y(x+h) \approx y(x) + h\frac{f\left(x+h, y(x+h)\right) + f\left(x, y(x)\right)}{2}. \tag{22}$$

This relation provides a more accurate approximation than Euler's method, but it is difficult to use directly in actual calculations. This is because, in order to evaluate the left-hand side $y(x+h)$, information about $y(x+h)$ at $x+h$ is already required when computing the right-hand side. Therefore, to approximate this relation, we adopt the following two-step procedure.

First, in the initial step, we obtain an approximate value of $y(x)$ at $x+h$ using Euler's method:

$$\bar{y}(x+h) = y(x) + hf\left(x, y(x)\right). \tag{23}$$

In the second step, using the approximate value $\bar{y}(x+h)$ obtained in the first step, we evaluate the right-hand side of Eq. (22) and compute $y(x+h)$ as follows:

$$y(x+h) = y(x) + h\frac{f\left(x+h, \bar{y}(x+h)\right) + f\left(x, y(x)\right)}{2}. \tag{24}$$

This two-step procedure using Eq. (23) and Eq. (24) is called Heun's method for solving differential equations. To compare the accuracy of Heun's method with that of Euler's method, let us solve the differential equation, Eq. (19).

Below, we show a Python code example that extends the Euler method sample code, Source Code 8, to include computations by Heun's method.

Source code 9: Solution of a differential equation by Heun's method and Euler's method

```
from matplotlib import pyplot as plt
import numpy as np

# compute dy/dx
def dydx(x,y):
    return -y

# compute y(x+h)
def euler_method(y,x,h):
    return y+dydx(x,y)*h

# compute y(x+h)
def heun_method(y,x,h):
    y_bar = y+dydx(x,y)*h
    return y+0.5*h*(dydx(x,y)+dydx(x+h,y_bar))

xi = 0.0
xf = 5.0
n = 10

h = (xf-xi)/n

# initial condition
x = xi
y = 1.0


x_j = np.zeros((n+1))
y_euler = np.zeros((n+1))
y_heun = np.zeros((n+1))

x_j[0] = x
y_euler[0] = y
y_heun[0] = y

for i in range(n):
    x = xi + i*h
    y_euler[i+1] = euler_method(y_euler[i],x,h)
    y_heun[i+1] = heun_method(y_heun[i],x,h)
    x_j[i+1] = x+h



# plot
plt.plot(x_j, y_euler, label="Euler method")
plt.plot(x_j, y_heun, label="Heun method", linestyle='dashed')
plt.plot(x_j, np.exp(-x_j), label="Exact", linestyle='dotted')

plt.xlabel('x')
plt.ylabel('y')
plt.legend()
plt.savefig("result_Heun.png")
plt.show()
```

Figure 4 shows the result obtained by executing Source Code 9.

4.3 Solution by the Runge-–Kutta Method

Compared with the Euler and Heun methods, the Runge—Kutta method provides a more accurate numerical solution method. In particular, the fourth-order Runge-–Kutta method is widely used for solving differential equations. The fourth-order Runge—Kutta method solves the differential equation

$$\frac{dy(x)}{dx} = f\left(x, y(x)\right) \tag{25}$$

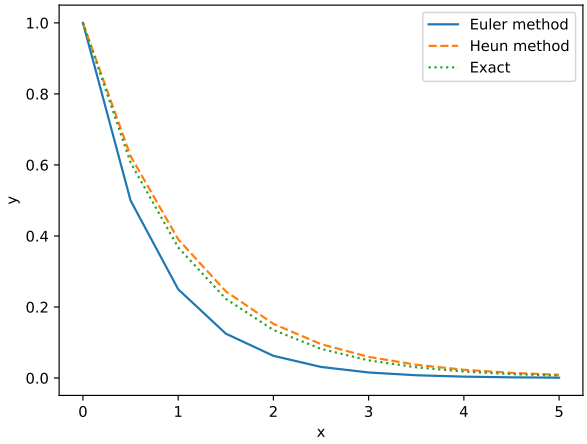

Figure 4: Example of solving a differential equation using Heun's method.

through the following multi-step procedure:

$$k_1 = f\left(x, y(x)\right) \tag{26}$$

$$k_2 = f\left(x + \frac{h}{2}, y(x) + \frac{h}{2}k_1\right) \tag{27}$$

$$k_3 = f\left(x + \frac{h}{2}, y(x) + \frac{h}{2}k_2\right) \tag{28}$$

$$k_4 = f\left(x + h, y(x) + hk_3\right) \tag{29}$$

$$y(x+h) \approx y(x) + h\frac{k_1 + 2k_2 + 2k_3 + k_4}{6}. \tag{30}$$

Let us implement a program to solve the differential equation (19) using the Runge—Kutta method, and examine how the accuracy changes by comparing the results with those of the Euler and Heun methods.

Now editing here!!

5 Solving Second-Order Ordinary Differential Equations

In this section, we learn how to numerically solve second-order ordinary differential equations in a single variable that can be written in the following form.

$$\frac{d^2y(x)}{dx^2} = f\left(x, y(x), \frac{dy(x)}{dx}\right). \tag{31}$$

Second-order ordinary differential equations of this kind include, for example, Newton's equation of motion ($m\frac{d^2x(t)}{dt^2} = F(t)$).

There are several ways to solve second-order ordinary differential equations. Here, by introducing the auxiliary variable $s(x) = \frac{dy(x)}{dx}$, we rewrite Eq. (31) as the following set of two first-order ordinary differential equations:

$$\frac{ds(x)}{dx} = f\left(x, y(x), s(x)\right), \tag{32}$$

$$\frac{dy(x)}{dx} = s(x). \tag{33}$$

Such a two-variable system of first-order ordinary differential equations can be solved numerically using the methods for first-order ordinary differential equations described in Sec. 4.

As an example, let us learn about numerical methods for ordinary differential equations while writing a program that solves the following second-order ordinary differential equation:

$$\frac{d^2x(t)}{dt} = -x(t). \tag{34}$$

Here, we set the initial conditions to $x(0) = 0$, $\dot{x}(0) = 1$.

By introducing the auxiliary variable $v(t) = \frac{dx(t)}{dt}$, we can rewrite Eq. (34) as the following set of differential equations:

$$\frac{dx(t)}{dt} = v(t), \tag{35}$$

$$\frac{dv(t)}{dt} = -x(t). \tag{36}$$

Furthermore, the initial conditions $x(0) = 0, \dot{x}(0) = 1$ can be rewritten as $x(0) = 0, v(0) = 1$. Note that the solution to this differential equation is $x(t) = \sin(t), v(t) = \cos(t)$.

Source Code is Python code that solves the system of first-order differential equations, Eqs. (35) and (36), using the Euler method. Using this source code as a reference, try writing your own code to solve the system (35-36).

Source code 10: Sample code for solving a system of first-order differential equations with the Euler method

```
from matplotlib import pyplot as plt
import numpy as np

def dxdt(v):
    return v

def dvdt(x):
    return -x

def euler_method(x,v,dt):
    x_updated = x + dt*dxdt(v)
    v_updated = v + dt*dvdt(x)
    return x_updated, v_updated
```

```
14
15 ti = 0.0
16 tf = 15.0
17 n = 45
18 dt = (tf - ti)/n
19
20 # Initial conditions
21 x = 1.0
22 v = 0.0
23
24
25 tt = np.zeros(n+1)
26 xt = np.zeros(n+1)
27 vt = np.zeros(n+1)
28
29 tt[0] = ti
30 xt[0] = x
31 vt[0] = v
32
33 for j in range(n):
34     x, v = euler_method(x,v,dt)
35     xt[j+1] = x
36     vt[j+1] = v
37     tt[j+1] = ti + (j+1)*dt
38
39
40 # Plot the results
41 plt.plot(tt, xt, label='x(t)')
42 plt.plot(tt, vt, label='v(t)')
43 plt.plot(tt, np.cos(tt), label='x(t): Exact', linestyle='dashed')
44 plt.plot(tt, -np.sin(tt), label='v(t): Exact', linestyle='dashed')
45
46 plt.xlabel('t')
47 plt.ylabel('x,v')
48 plt.legend()
49 plt.savefig("result_Euler_2nd.png")
50 plt.show()
```

Figure 5 compares the numerical solution obtained by running Source Code 10 with the exact solution. You can observe how the numerical solution obtained by the Euler method deviates significantly from the exact solution as time t progresses.

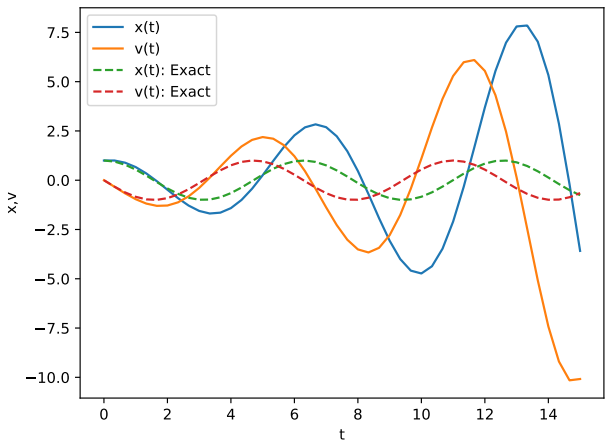

Figure 5: Comparison between the result of solving the system of differential equations, Eq. (35) and Eq. (36), with the Euler method and the exact solution.

As learned in Sec. 4.1, the accuracy of the numerical solution can be improved by using Heun's method or the Runge–Kutta method. Source Code 11 shows an example of code that solves the system, Eqs. (35-36), using Heun's method. Figure 6 shows a comparison between the resulting numerical solution and the exact solution. By using Heun's method, we can achieve higher accuracy

than with the Euler method.

Source code 11: Sample code for solving a system of first-order differential equations with Heun's method

```
from matplotlib import pyplot as plt
import numpy as np

def dxdt(v):
    return v

def dvdt(x):
    return -x

#def euler_method(x,v,dt):
#    x_updated = x + dt*dxdt(v)
#    v_updated = v + dt*dvdt(x)
#    return x_updated, v_updated

def heun_method(x,v,dt):
# first step
    x_bar = x + dt*dxdt(v)
    v_bar = v + dt*dvdt(x)

# second step
    x_updated = x + 0.5*dt*(dxdt(v)+dxdt(v_bar))
    v_updated = v + 0.5*dt*(dvdt(x)+dvdt(x_bar))
    return x_updated, v_updated

ti = 0.0
tf = 15.0
n = 45
dt = (tf - ti)/n

# Initial conditions
x = 1.0
v = 0.0


tt = np.zeros(n+1)
xt = np.zeros(n+1)
vt = np.zeros(n+1)

tt[0] = ti
xt[0] = x
vt[0] = v

for j in range(n):
    x, v = heun_method(x,v,dt)
    xt[j+1] = x
    vt[j+1] = v
    tt[j+1] = ti + (j+1)*dt


# Plot the results
plt.plot(tt, xt, label='x(t)')
plt.plot(tt, vt, label='v(t)')
plt.plot(tt, np.cos(tt), label='x(t): Exact', linestyle='dashed')
plt.plot(tt, -np.sin(tt), label='v(t): Exact', linestyle='dashed')

plt.xlabel('t')
plt.ylabel('x,v')
plt.legend()
plt.savefig("result_Heun_2nd.png")
plt.show()
```

You can implement a program that solves the system of differential equations, Eqs. (35-36), using Heun's method or the Runge–Kutta method and verify the computational accuracy.

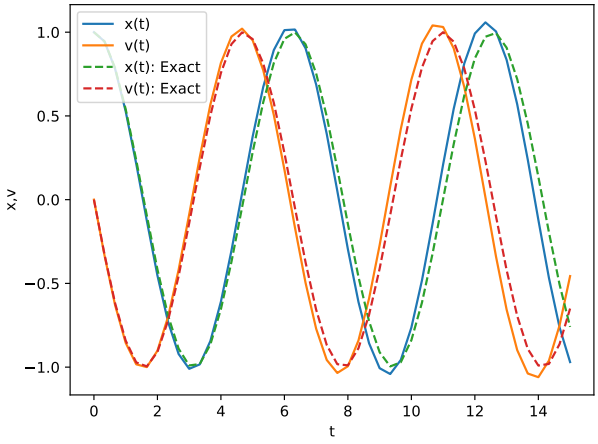

Figure 6: Comparison between the result of solving the system of differential equations (35-36) with Heun's method and the exact solution.

6 Quantum Dynamics Simulation in One Dimension

In this section, we treat the numerical solution of the time-dependent Schrödinger equation to simulate the propagation of a one-dimensional wave packet. The motion of a particle under a time-independent potential $V(x)$ is described by the following Schrödinger equation:

$$i\hbar\frac{\partial}{\partial t}\psi(x,t) = \left[-\frac{\hbar^2}{2m}\frac{\partial^2}{\partial x^2} + V(x)\right]\psi(x,t) = \hat{H}\psi(x,t). \tag{37}$$

Here, the Hamiltonian H is defined as

$$\hat{H} = -\frac{\hbar^2}{2m}\frac{\partial^2}{\partial x^2} + V(x) \tag{38}$$

For the moment, let us impose the boundary conditions $\psi(x = -L/2, t) = \psi(x = L/2, t) = 0$, where L is a sufficiently large value.

We now consider solving this Schrödinger equation under the initial condition at time $t = 0$:

$$\psi(x, t = 0) = e^{-\frac{(x-x_0)^2}{2\sigma_0^2}} e^{ik_0(x-x_0)} \tag{39}$$

Here, x_0 is the central position of the wave packet, k_0 is the central wavenumber, and σ_0 is the width of the wave packet.

6.1 Real-Space Method

To represent the wave function on a computer, let us consider the discretization of spatial coordinates as follows:

$$x \to x_j = -\frac{L}{2} + \Delta x \times (j+1) \qquad (-1 \le j \le N). \tag{40}$$

Here, $\Delta x = L/(N+1)$. The values of the wave function $\psi(x,t)$ at the $(N+1)$ points $x_{-1}, x_0, x_1, \cdots, x_N$ prepared in this way are expressed as:

$$\psi_j(t) = \psi(x_j, t). \tag{41}$$

This method of handling functions at a finite number of points in real space is called the **real-space method**. To become familiar with the description of wave functions using the real-space method, let us plot the wave function given by the initial condition (Eq. (39)). At this time, freely set the values of x_0, k_0, σ_0 and observe how the real and imaginary parts of the wave function change.

Source Code 12 is a sample source code for visualization. Try writing your own code and attempt to visualize the wave function.

`https://github.com/shunsuke-sato/python_qe/blob/develop/note_comp_phys/src/qm_write_init_wf.py`

Source code 12: Visualization of the initial wave function

```
from matplotlib import pyplot as plt
import numpy as np

# Initialize the wavefunction
def initialize_wf(xj, x0, k0, sigma0):
    wf = np.exp(1j*k0*(xj-x0))*np.exp(-0.5*(xj-x0)**2/sigma0**2)
    return wf



# initial wavefunction parameters
x0 = -25.0
```

```
k0 = 0.85
sigma0 = 5.0

# set the coordinate
xmin = -100.0
xmax = 100.0
n = 2500

dx = (xmax-xmin)/(n+1)
xj = np.zeros(n)

for i in range(n):
    xj[i] = xmin + dx*(i+1)


# Initialize the wavefunction
wf = initialize_wf(xj, x0, k0, sigma0)

# Plot the results
plt.plot(xj, np.real(wf), label="Real part")
plt.plot(xj, np.imag(wf), label="Imaginary part")

plt.xlabel('x')
plt.ylabel('$\psi$(x)')
plt.legend()
plt.savefig("init_wf.png")
plt.show()
```

In Source Code 12, the parameters are set as $x_0 = -25$, $k_0 = 0.85$, and $\sigma_0 = 25.0$ for visualizing the wave function. Furthermore, using the real-space method, the range $(-100 \leq x \leq 100)$ is divided into $N = 2500$ grid points for visualization.

By executing Source Code 12, the wave function can be visualized as shown in Fig. 7. Try changing the physical parameters x_0, k_0, σ_0 and the number of divisions N, and observe how the visualized wave function behaves.

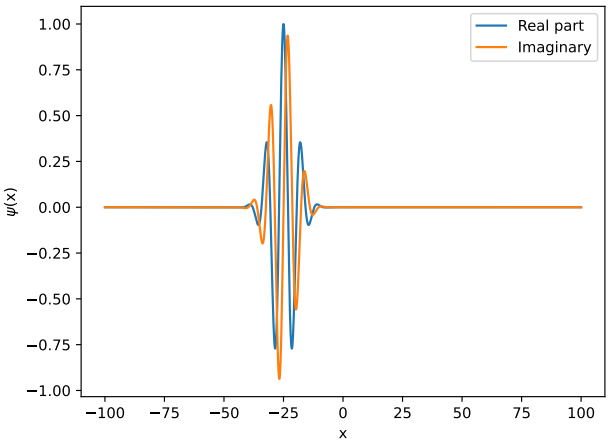

Figure 7: Real and imaginary parts of the initial wave function.

6.2 Real-time method

Next, let us compute the time evolution of a wavefunction defined on the finite set of discrete points x_j prepared by the real-space method above. To do this, we first consider the formal solution of the Schrödinger equation, Eq. (37). Noting that the Hamiltonian in the Schrödinger equation, Eq. (37), is time independent, we can write down its formal solution in the following

form.

$$\psi(x,t) = \exp\left[-\frac{i}{\hbar}\hat{H}t\right]\psi(x,0). \tag{42}$$

Here, the exponential of a general operator A is defined by the following Taylor expansion:

$$\exp\left[\hat{A}\right] = \mathcal{I} + \sum_{n=1}^{\infty}\frac{\hat{A}^n}{n!}. \tag{43}$$

Here $\mathcal{I}$ denotes the identity operator.

If we can evaluate the time-evolution operator $\hat{U}(t,0) = \exp\left[-\frac{i}{\hbar}\hat{H}t\right]$ numerically, then we can compute the time evolution of the wavefunction by numerical calculation. There are several ways to evaluate such operator exponentials; here we adopt a representation in which the time-evolution operator is written as a product of time-evolution operators with a small time step. Namely, we express the time-evolution operator $\hat{U}(t,0)$ as the following product of time-evolution operators:

$$\hat{U}(t,0) = \exp\left[-i\frac{t}{\hbar}\hat{H}\right] = \exp\left[-i\frac{\Delta t}{\hbar}\hat{H}\right]\times\cdots\exp\left[-i\frac{\Delta t}{\hbar}\hat{H}\right] = \left[\exp\left[-i\frac{\Delta t}{\hbar}\hat{H}\right]\right]^{N_t}. \tag{44}$$

Here, the small time step Δt is defined by $\Delta t = t/N_t$.

Using Eq. (44), the time evolution of the wavefunction can be represented as a repetition of evolution over the small time step Δt. Next, let us consider how to evaluate the small time-step propagator, $\exp\left[-i\frac{\Delta t}{\hbar}\hat{H}\right]$, numerically. Noting that the exponential of an operator is defined by Eq. (43), the small time-step propagator can be written in the following Taylor expansion form:

$$\exp\left[-i\frac{\Delta t}{\hbar}\hat{H}\right] = \mathcal{I} + \sum_{n=1}^{\infty}\frac{(-i\Delta t/\hbar)^n}{n!}\hat{H}^n. \tag{45}$$

The expansion of the time-evolution operator in Eq. (45) is exact. If the small time step Δt is sufficiently small, truncating the series in Eq. (45) after a finite number of terms is expected to yield a sufficiently accurate approximation to the small time-step propagator. Therefore, taking N_{exp} to be a finite integer, we approximate the operator as follows:

$$\exp\left[-i\frac{\Delta t}{\hbar}\hat{H}\right] \approx \mathcal{I} + \sum_{n=1}^{N_{\mathrm{exp}}}\frac{(-i\Delta t/\hbar)^n}{n!}\hat{H}^n. \tag{46}$$

Using Eq. (46), we can write down the time-evolution equation for the wavefunction explicitly and obtain the following approximation:

$$\begin{aligned}\psi(x,t+\Delta t) &= \exp\left[-i\frac{\Delta t}{\hbar}\hat{H}\right]\psi(x,t)\\ &\approx \psi(x,t) + \sum_{n=1}^{N_{\mathrm{exp}}}\frac{(-i\Delta t/\hbar)^n}{n!}\hat{H}^n\psi(x,t).\end{aligned} \tag{47}$$

In other words, by applying the Hamiltonian to the wavefunction $\psi(x,t)$ multiple times, multiplying by appropriate constants, and adding the results to the original wavefunction $\psi(x,t)$, we can obtain an approximate wavefunction at the next time step, $\psi(x,t+\Delta t)$. This approximation becomes more accurate as Δt is made smaller, and also as the truncation order N_{exp} is increased. In practice, many applications take $N_{\mathrm{exp}} = 4$ and use a sufficiently small Δt to perform the time propagation.

From here, we will write a code to compute the time evolution of a quantum wave packet using the algorithm described above. Since the time evolution of the wavefunction can be realized by

repeatedly applying the Hamiltonian to the wavefunction many times, let us first write a code that applies the Hamiltonian to a wavefunction. An example is shown in Source Code 13.

https://github.com/shunsuke-sato/python_qe/blob/develop/note_comp_phys/src/qm_ham_write.py

Source code 13: Example code for applying the Hamiltonian to a wavefunction

```
from matplotlib import pyplot as plt
import numpy as np

# Initialize the wavefunction
def initialize_wf(xj, x0, k0, sigma0):
    wf = np.exp(1j*k0*(xj-x0))*np.exp(-0.5*(xj-x0)**2/sigma0**2)
    return wf


# Operate the Hamiltonian to the wavefunction
def ham_wf(wf, vpot, dx):

    n = wf.size
    hwf = np.zeros(n, dtype=complex)

    for i in range(1,n-1):
        hwf[i] = -0.5*(wf[i+1]-2.0*wf[i]+wf[i-1])/(dx**2)

    i = 0
    hwf[i] = -0.5*(wf[i+1]-2.0*wf[i])/(dx**2)
    i = n-1
    hwf[i] = -0.5*(-2.0*wf[i]+wf[i-1])/(dx**2)

    hwf = hwf + vpot*wf

    return hwf


# initial wavefunction parameters
x0 = -25.0
k0 = 0.85
sigma0 = 5.0

# set the coordinate
xmin = -100.0
xmax = 100.0
n = 2500

dx = (xmax-xmin)/(n+1)
xj = np.zeros(n)

for i in range(n):
    xj[i] = xmin + dx*(i+1)


# Initialize the wavefunction
wf = initialize_wf(xj, x0, k0, sigma0)
vpot = np.zeros(n)

hwf = ham_wf(wf, vpot, dx)

# Plot the results
plt.plot(xj, np.real(wf), label="Real part (wf)")
plt.plot(xj, np.imag(wf), label="Imaginary part (wf)")
plt.plot(xj, np.real(hwf), label="Real part (ham wf)")
plt.plot(xj, np.imag(hwf), label="Imaginary part (ham wf)")

plt.xlabel('x')
plt.ylabel('$\psi$(x)')
plt.legend()
plt.savefig("ham_wf.png")
plt.show()
```

Running the above Source Code 13 yields a figure like Figure 8. Here, the real and imaginary parts of the initial wavefunction and of the wavefunction obtained by applying the Hamiltonian to it are shown.

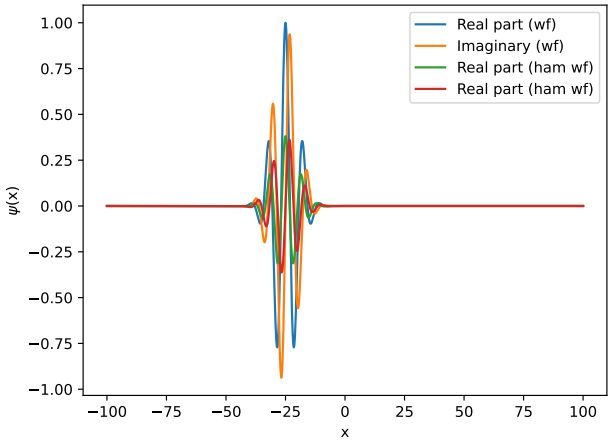

Figure 8: Visualization of the wavefunction with the Hamiltonian applied.

Next, we write code that computes the time evolution of the wavefunction by repeatedly using the small time-step propagation in Eq. (47). Using the function created in the example above (Source Code 12) that applies the Hamiltonian to a wavefunction, we carry out the small time-step propagation of the wavefunction.

Source Code 14 shows an example of code that computes the time evolution of the wavefunction.

`https://github.com/shunsuke-sato/python_qe/blob/develop/note_comp_phys/src/qm_dynamics.py`

Source code 14: Time evolution of a one-dimensional quantum wave packet

```
from matplotlib import pyplot as plt
import numpy as np

# Initialize the wavefunction
def initialize_wf(xj, x0, k0, sigma0):
    wf = np.exp(1j*k0*(xj-x0))*np.exp(-0.5*(xj-x0)**2/sigma0**2)
    return wf


# Operate the Hamiltonian to the wavefunction
def ham_wf(wf, vpot, dx):

    n = wf.size
    hwf = np.zeros(n, dtype=complex)

    for i in range(1,n-1):
        hwf[i] = -0.5*(wf[i+1]-2.0*wf[i]+wf[i-1])/(dx**2)

    i = 0
    hwf[i] = -0.5*(wf[i+1]-2.0*wf[i])/(dx**2)
    i = n-1
    hwf[i] = -0.5*(-2.0*wf[i]+wf[i-1])/(dx**2)

    hwf = hwf + vpot*wf

    return hwf


# Time propagation from t to t+dt
def time_propagation(wf, vpot, dx, dt):

    n = wf.size
    twf = np.zeros(n, dtype=complex)
    hwf = np.zeros(n, dtype=complex)

    twf = wf
```

```
    zfact = 1.0 + 0j
    for iexp in range(1,5):
        zfact = zfact*(-1j*dt)/iexp
        hwf = ham_wf(twf, vpot, dx)
        wf = wf + zfact*hwf
        twf = hwf

    return wf


# initial wavefunction parameters
x0 = -25.0
k0 = 0.85
sigma0 = 5.0

# time propagation parameters
#Tprop = 80.0
Tprop = 8.0
dt = 0.005
nt = int(Tprop/dt)+1

# set the coordinate
xmin = -100.0
xmax = 100.0
n = 2500

dx = (xmax-xmin)/(n+1)
xj = np.zeros(n)

for i in range(n):
    xj[i] = xmin + dx*(i+1)


# Initialize the wavefunction
wf = initialize_wf(xj, x0, k0, sigma0)
vpot = np.zeros(n)


# for loop for the time propagation
for it in range(nt+1):
    wf = time_propagation(wf, vpot, dx, dt)
    print(it, nt)

# Plot the results
plt.plot(xj, np.real(wf), label="Real part (wf)")
plt.plot(xj, np.imag(wf), label="Imaginary part (wf)")


plt.xlabel('x')
plt.ylabel('$\psi$(x)')
plt.legend()
plt.savefig("fin_wf.pdf")
plt.show()
```

By running Source Code 14, starting from the initial wavefunction shown in Figure 7, the time-evolved wavefunction displayed is like that in Figure 9. Based on the behavior of this wavefunction, let us examine the time evolution of a quantum wave packet in free space.

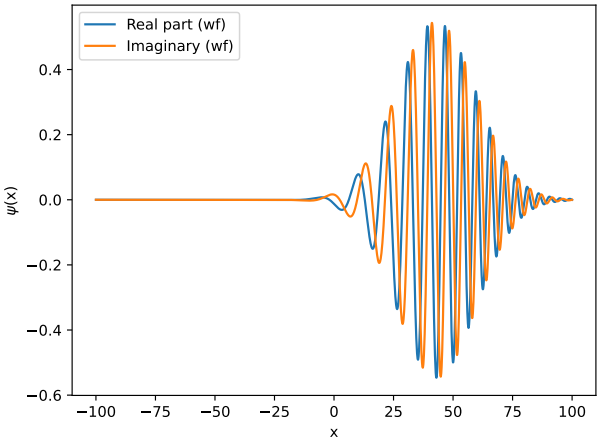

Figure 9: Time-evolved wavefunction.

6.2.1 Speeding up Python code with Numba

The execution of the above code may have taken some time. In general, Python is often slower compared to other languages. However, by applying certain techniques, the execution speed of Python code can be improved. Here, we will accelerate code execution using the JIT (Just in Time) compilation feature of *Numba*.

Using Numba's jit is very simple. At the beginning of the program, add `from numba import jit` to enable the use of Numba's jit. Next, write `@jit(nopytohn=True)` directly above the function you want to speed up. This way, the specified function will be compiled by the jit compiler, allowing the Python code to run faster.

As an example, let us accelerate the quantum wave packet simulation code 14 using Numba's jit. Source Code 15 is an example of code accelerated with Numba. In this code, the function `ham_wf` has been accelerated, and when initializing the complex array, the array type is strictly defined as `np.complex128`, making it possible to accelerate with Numba.

`https://github.com/shunsuke-sato/python_qe/blob/develop/note_comp_phys/src/qm_dynamics_numba.py`

Source code 15: Time evolution of a 1D quantum wave packet (accelerated with Numba)

```
from numba import jit
from matplotlib import pyplot as plt
import numpy as np

# Initialize the wavefunction
def initialize_wf(xj, x0, k0, sigma0):
    wf = np.exp(1j*k0*(xj-x0))*np.exp(-0.5*(xj-x0)**2/sigma0**2)
    return wf


# Operate the Hamiltonian to the wavefunction
@jit(nopython=True)
def ham_wf(wf, vpot, dx):

    n = wf.size
    hwf = np.zeros(n, dtype=np.complex128)

    for i in range(1,n-1):
        hwf[i] = -0.5*(wf[i+1]-2.0*wf[i]+wf[i-1])/(dx**2)

    i = 0
    hwf[i] = -0.5*(wf[i+1]-2.0*wf[i])/(dx**2)
    i = n-1
```

```
    hwf[i] = -0.5*(-2.0*wf[i]+wf[i-1])/(dx**2)

    hwf = hwf + vpot*wf

    return hwf


# Time propagation from t to t+dt
def time_propagation(wf, vpot, dx, dt):

    n = wf.size
    twf = np.zeros(n, dtype=complex)
    hwf = np.zeros(n, dtype=complex)

    twf = wf
    zfact = 1.0 + 0j
    for iexp in range(1,5):
        zfact = zfact*(-1j*dt)/iexp
        hwf = ham_wf(twf, vpot, dx)
        wf = wf + zfact*hwf
        twf = hwf

    return wf


# initial wavefunction parameters
x0 = -25.0
k0 = 0.85
sigma0 = 5.0

# time propagation parameters
#Tprop = 80.0
Tprop = 8.0
dt = 0.005
nt = int(Tprop/dt)+1

# set the coordinate
xmin = -100.0
xmax = 100.0
n = 2500

dx = (xmax-xmin)/(n+1)
xj = np.zeros(n)

for i in range(n):
    xj[i] = xmin + dx*(i+1)


# Initialize the wavefunction
wf = initialize_wf(xj, x0, k0, sigma0)
vpot = np.zeros(n)


# for loop for the time propagation
for it in range(nt+1):
    wf = time_propagation(wf, vpot, dx, dt)
    print(it, nt)

# Plot the results
plt.plot(xj, np.real(wf), label="Real part (wf)")
plt.plot(xj, np.imag(wf), label="Imaginary part (wf)")


plt.xlabel('x')
plt.ylabel('$\psi$(x)')
plt.legend()
plt.savefig("fin_wf.pdf")
plt.show()
```

6.3 Creating a movie of one-dimensional quantum wave packet dynamics

In Source Code 14, only the wave function at the final time of the time-evolution calculation was output as a figure. Here, to visualize how the quantum wave packet evolves from the initial time to the final time, we will learn how to create a movie from the simulation results.

To make a movie from the simulation results, a common method is to prepare many image files, like a flipbook, and then combine these image files into a movie. In Source Code 16, the previous source code 14 is extended so that during the time-evolution calculation, the wave function at each time step is saved. Finally, these wave function plots are generated, combined, and output as a movie.

]https://github.com/shunsuke-sato/python_qe/blob/develop/note_comp_phys/src/qm_dynamics_matplotlib.py

Source code 16: Code for creating a movie of quantum wave packet dynamics

```
from numba import jit
from matplotlib import pyplot as plt
import numpy as np
import matplotlib.animation as animation
from matplotlib.animation import PillowWriter

# Initialize the wavefunction
def initialize_wf(xj, x0, k0, sigma0):
    wf = np.exp(1j*k0*(xj-x0))*np.exp(-0.5*(xj-x0)**2/sigma0**2)
    return wf


# Operate the Hamiltonian to the wavefunction
@jit(nopython=True)
def ham_wf(wf, vpot, dx):

    n = wf.size
    hwf = np.zeros(n, dtype=np.complex128)

    for i in range(1,n-1):
        hwf[i] = -0.5*(wf[i+1]-2.0*wf[i]+wf[i-1])/(dx**2)

    i = 0
    hwf[i] = -0.5*(wf[i+1]-2.0*wf[i])/(dx**2)
    i = n-1
    hwf[i] = -0.5*(-2.0*wf[i]+wf[i-1])/(dx**2)

    hwf = hwf + vpot*wf

    return hwf


# Time propagation from t to t+dt
def time_propagation(wf, vpot, dx, dt):

    n = wf.size
    twf = np.zeros(n, dtype=complex)
    hwf = np.zeros(n, dtype=complex)

    twf = wf
    zfact = 1.0 + 0j
    for iexp in range(1,5):
        zfact = zfact*(-1j*dt)/iexp
        hwf = ham_wf(twf, vpot, dx)
        wf = wf + zfact*hwf
        twf = hwf

    return wf


# initial wavefunction parameters
x0 = -25.0
k0 = 0.85
sigma0 = 5.0

# time propagation parameters
Tprop = 80.0
dt = 0.005
#dt = 0.00905
nt = int(Tprop/dt)+1

# set the coordinate
xmin = -100.0
xmax = 100.0
n = 2500
```

```
66
67  dx = (xmax-xmin)/(n+1)
68  xj = np.zeros(n)
69
70  for i in range(n):
71      xj[i] = xmin + dx*(i+1)
72
73
74  # Initialize the wavefunction
75  wf = initialize_wf(xj, x0, k0, sigma0)
76  vpot = np.zeros(n)
77
78  # For loop for the time propagation
79  wavefunctions = []
80  for it in range(nt+1):
81      if (it % (nt//100) == 0):
82          wavefunctions.append(wf.copy())
83
84      wf = time_propagation(wf, vpot, dx, dt)
85      print(it, nt)
86
87  # Define function to update plot for each frame of the animation
88  def update_plot(frame):
89      plt.cla()
90      plt.xlim([-100, 100])
91      plt.ylim([-1.2, 1.2])
92      plt.plot(xj, np.real(wavefunctions[frame]), label="Real␣part␣of␣$\psi(x)$")
93      plt.plot(xj, np.imag(wavefunctions[frame]), label="Imaginary␣part␣of␣$\psi(x)$")
94      plt.xlabel('$x$')
95      plt.ylabel('$\psi(x)$')
96      plt.legend()
97
98  # Create the animation
99  fig = plt.figure()
100 ani = animation.FuncAnimation(fig, update_plot, frames=len(wavefunctions), interval=50)
101 #ani.save('wavefunction_animation.gif', writer='imagemagick')
102 ani.save('wavefunction_animation.gif', writer='pillow')
```

6.4 Various Dynamics of One-Dimensional Quantum Wave Packets

Using the simulation code for one-dimensional quantum wave-packet dynamics developed up to this point, let us explore the motion of one-dimensional wave packets in various problems.

6.4.1 Tunneling Phenomenon

Here we provide reference information on the dynamics of a one-dimensional wave packet related to tunneling. In the code above, the potential energy $V(x)$ was set to zero, but here we consider the wave-packet dynamics under the following Gaussian potential:

$$V(x) = V_0 e^{-\frac{x^2}{2\sigma_v}}. \tag{48}$$

For example, set $V_0 = 0.735$ a.u. and $\sigma_v = 0.5$ a.u., and use an initial wave function of the form of Eq. (39). With $k_0 = 0.85$ a.u., $x_0 = -25$ a.u., and $\sigma_0 = 5$, run the calculation and create a movie of the wave-packet dynamics. Also vary the potential height V_0 and the potential width σ_v, run simulations, and become familiar with the tunneling phenomenon. You can refer to Source Code 17.

6.4.2 Coherent State in a Harmonic Potential

Here we provide reference information on the dynamics of a quantum wave packet in a harmonic potential. A harmonic potential is a quadratic potential given by

$$V(x) = \frac{K}{2}x^2. \tag{49}$$

Here, K is the spring constant. As a trial, set $K = 1$ a.u. and run the calculation. Use Eq. (39) for the initial wave function. With $k_0 = 0$ a.u., $x_0 = -2$ a.u., and $\sigma_0 = 1$, run the calculation and create a movie of the wave-packet dynamics. Also refer to source code 18.

6.4.3 Anharmonic Potential

Building on Sec. 6.4.2, add an anharmonic term to the potential and examine how the quantum wave-packet dynamics is affected. For example, study the dynamics of a wave packet under the following potential:

$$V(x) = \frac{K}{2}x^2 + 0.01x^4. \tag{50}$$

Also refer to source code 19.

`https://github.com/shunsuke-sato/python_qe/blob/develop/note_comp_phys/src/qm_dynamics_tunnel.py`

Source code 17: Example code for the tunneling phenomenon of a quantum wave packet

```
from numba import jit
from matplotlib import pyplot as plt
import numpy as np
import matplotlib.animation as animation
from matplotlib.animation import PillowWriter

# Initialize the wavefunction
def initialize_wf(xj, x0, k0, sigma0):
    wf = np.exp(1j*k0*(xj-x0))*np.exp(-0.5*(xj-x0)**2/sigma0**2)
    return wf

# Initialize potential
def initialize_vpot(xj):
    v0 = 0.735
    sigma = 0.5
    return v0*np.exp(-0.5*(xj/sigma)**2)

# Operate the Hamiltonian to the wavefunction
@jit(nopython=True)
def ham_wf(wf, vpot, dx):

    n = wf.size
    hwf = np.zeros(n, dtype=np.complex128)

    for i in range(1,n-1):
        hwf[i] = -0.5*(wf[i+1]-2.0*wf[i]+wf[i-1])/(dx**2)

    i = 0
    hwf[i] = -0.5*(wf[i+1]-2.0*wf[i])/(dx**2)
    i = n-1
    hwf[i] = -0.5*(-2.0*wf[i]+wf[i-1])/(dx**2)

    hwf = hwf + vpot*wf

    return hwf


# Time propagation from t to t+dt
def time_propagation(wf, vpot, dx, dt):

    n = wf.size
    twf = np.zeros(n, dtype=complex)
    hwf = np.zeros(n, dtype=complex)

    twf = wf
    zfact = 1.0 + 0j
    for iexp in range(1,5):
        zfact = zfact*(-1j*dt)/iexp
        hwf = ham_wf(twf, vpot, dx)
        wf = wf + zfact*hwf
        twf = hwf

    return wf


# initial wavefunction parameters
x0 = -25.0
k0 = 0.85
sigma0 = 5.0
```

```

# time propagation parameters
Tprop = 80.0
dt = 0.005
#dt = 0.00905
nt = int(Tprop/dt)+1

# set the coordinate
xmin = -100.0
xmax = 100.0
n = 2500

dx = (xmax-xmin)/(n+1)
xj = np.zeros(n)

for i in range(n):
    xj[i] = xmin + dx*(i+1)


# Initialize the wavefunction
wf = initialize_wf(xj, x0, k0, sigma0)
#vpot = np.zeros(n)
vpot = initialize_vpot(xj)

# For loop for the time propagation
wavefunctions = []
for it in range(nt+1):
    if (it % (nt//100) == 0):
        wavefunctions.append(wf.copy())

    wf = time_propagation(wf, vpot, dx, dt)
    print(it, nt)

# Define function to update plot for each frame of the animation
def update_plot(frame):
    plt.cla()
    plt.xlim([-100, 100])
    plt.ylim([-1.2, 1.2])
    plt.plot(xj, np.real(wavefunctions[frame]), label="Real part of $\psi(x)$")
    plt.plot(xj, np.imag(wavefunctions[frame]), label="Imaginary part of $\psi(x)$")
    plt.plot(xj, vpot, label="$V(x)$")
    plt.xlabel('$x$')
    plt.ylabel('$\psi(x)$')
    plt.legend()

# Create the animation
fig = plt.figure()
ani = animation.FuncAnimation(fig, update_plot, frames=len(wavefunctions), interval=50)
#ani.save('wavefunction_animation.gif', writer='imagemagick')
ani.save('wavefunction_animation.gif', writer='pillow')
```

https://github.com/shunsuke-sato/python_qe/blob/develop/note_comp_phys/src/qm_dynamics_harmonic.py

Source code 18: Example code for quantum wave-packet dynamics in a harmonic potential

```
from numba import jit
from matplotlib import pyplot as plt
import numpy as np
import matplotlib.animation as animation
from matplotlib.animation import PillowWriter

# Initialize the wavefunction
def initialize_wf(xj, x0, k0, sigma0):
    wf = np.exp(1j*k0*(xj-x0))*np.exp(-0.5*(xj-x0)**2/sigma0**2)
    return wf

# Initialize potential
def initialize_vpot(xj):
    k0 = 1.0
    return 0.5*k0*xj**2

# Operate the Hamiltonian to the wavefunction
@jit(nopython=True)
def ham_wf(wf, vpot, dx):

```

```
    n = wf.size
    hwf = np.zeros(n, dtype=np.complex128)

    for i in range(1,n-1):
        hwf[i] = -0.5*(wf[i+1]-2.0*wf[i]+wf[i-1])/(dx**2)

    i = 0
    hwf[i] = -0.5*(wf[i+1]-2.0*wf[i])/(dx**2)
    i = n-1
    hwf[i] = -0.5*(-2.0*wf[i]+wf[i-1])/(dx**2)

    hwf = hwf + vpot*wf

    return hwf


# Time propagation from t to t+dt
def time_propagation(wf, vpot, dx, dt):

    n = wf.size
    twf = np.zeros(n, dtype=complex)
    hwf = np.zeros(n, dtype=complex)

    twf = wf
    zfact = 1.0 + 0j
    for iexp in range(1,5):
        zfact = zfact*(-1j*dt)/iexp
        hwf = ham_wf(twf, vpot, dx)
        wf = wf + zfact*hwf
        twf = hwf

    return wf


# initial wavefunction parameters
x0 = -2.0
k0 = 0.00
sigma0 = 1.0

# time propagation parameters
Tprop = 40.0
dt = 0.005
#dt = 0.00905
nt = int(Tprop/dt)+1

# set the coordinate
xmin = -10.0
xmax = 10.0
n = 250

dx = (xmax-xmin)/(n+1)
xj = np.zeros(n)

for i in range(n):
    xj[i] = xmin + dx*(i+1)


# Initialize the wavefunction
wf = initialize_wf(xj, x0, k0, sigma0)
#vpot = np.zeros(n)
vpot = initialize_vpot(xj)

# For loop for the time propagation
wavefunctions = []
for it in range(nt+1):
    if (it % (nt//100) == 0):
        wavefunctions.append(wf.copy())

    wf = time_propagation(wf, vpot, dx, dt)
    print(it, nt)

# Define function to update plot for each frame of the animation
def update_plot(frame):
    plt.cla()
    plt.xlim([-5, 5])
    plt.ylim([-1.2, 5.0])
    plt.plot(xj, np.real(wavefunctions[frame]), label="Real␣part␣of␣$\psi(x)$")
    plt.plot(xj, np.imag(wavefunctions[frame]), label="Imaginary␣part␣of␣$\psi(x)$")
    plt.plot(xj, np.abs(wavefunctions[frame])**2, label="␣$|\psi(x)|^2$␣")
    plt.plot(xj, vpot, label="$V(x)$")
```

```
    plt.xlabel('$x$')
    plt.ylabel('$\psi(x)$')
    plt.legend()

# Create the animation
fig = plt.figure()
ani = animation.FuncAnimation(fig, update_plot, frames=len(wavefunctions), interval=150)
#ani.save('wavefunction_animation.gif', writer='imagemagick')
ani.save('wavefunction_animation.gif', writer='pillow')
```

https://github.com/shunsuke-sato/python_qe/blob/develop/note_comp_phys/src/qm_dynamics_anharmonic.py

Source code 19: Example code for quantum wave-packet dynamics in an anharmonic potential

```
from numba import jit
from matplotlib import pyplot as plt
import numpy as np
import matplotlib.animation as animation
from matplotlib.animation import PillowWriter

# Initialize the wavefunction
def initialize_wf(xj, x0, k0, sigma0):
    wf = np.exp(1j*k0*(xj-x0))*np.exp(-0.5*(xj-x0)**2/sigma0**2)
    return wf

# Initialize potential
def initialize_vpot(xj):
    k0 = 1.0
    return 0.5*k0*xj**2+0.01*xj**4

# Operate the Hamiltonian to the wavefunction
@jit(nopython=True)
def ham_wf(wf, vpot, dx):

    n = wf.size
    hwf = np.zeros(n, dtype=np.complex128)

    for i in range(1,n-1):
        hwf[i] = -0.5*(wf[i+1]-2.0*wf[i]+wf[i-1])/(dx**2)

    i = 0
    hwf[i] = -0.5*(wf[i+1]-2.0*wf[i])/(dx**2)
    i = n-1
    hwf[i] = -0.5*(-2.0*wf[i]+wf[i-1])/(dx**2)

    hwf = hwf + vpot*wf

    return hwf


# Time propagation from t to t+dt
def time_propagation(wf, vpot, dx, dt):

    n = wf.size
    twf = np.zeros(n, dtype=complex)
    hwf = np.zeros(n, dtype=complex)

    twf = wf
    zfact = 1.0 + 0j
    for iexp in range(1,5):
        zfact = zfact*(-1j*dt)/iexp
        hwf = ham_wf(twf, vpot, dx)
        wf = wf + zfact*hwf
        twf = hwf

    return wf


# initial wavefunction parameters
x0 = -2.0
k0 = 0.00
sigma0 = 1.0

# time propagation parameters
Tprop = 40.0
```

```
dt = 0.005
#dt = 0.00905
nt = int(Tprop/dt)+1

# set the coordinate
xmin = -10.0
xmax = 10.0
n = 250

dx = (xmax-xmin)/(n+1)
xj = np.zeros(n)

for i in range(n):
    xj[i] = xmin + dx*(i+1)


# Initialize the wavefunction
wf = initialize_wf(xj, x0, k0, sigma0)
#vpot = np.zeros(n)
vpot = initialize_vpot(xj)

# For loop for the time propagation
wavefunctions = []
for it in range(nt+1):
    if (it % (nt//100) == 0):
        wavefunctions.append(wf.copy())

    wf = time_propagation(wf, vpot, dx, dt)
    print(it, nt)

# Define function to update plot for each frame of the animation
def update_plot(frame):
    plt.cla()
    plt.xlim([-5, 5])
    plt.ylim([-1.2, 5.0])
    plt.plot(xj, np.real(wavefunctions[frame]), label="Real part of $\psi(x)$")
    plt.plot(xj, np.imag(wavefunctions[frame]), label="Imaginary part of $\psi(x)$")
    plt.plot(xj, np.abs(wavefunctions[frame])**2, label=" $|\psi(x)|^2$ ")
    plt.plot(xj, vpot, label="$V(x)$")
    plt.xlabel('$x$')
    plt.ylabel('$\psi(x)$')
    plt.legend()

# Create the animation
fig = plt.figure()
ani = animation.FuncAnimation(fig, update_plot, frames=len(wavefunctions), interval=150)
#ani.save('wavefunction_animation.gif', writer='imagemagick')
ani.save('wavefunction_animation.gif', writer='pillow')
```

6.4.4 Harmonic Potential: Expectation Values of Position and Momentum, and Ehrenfest's Theorem

Here we return to the harmonic potential problem (6.4.2 section) and investigate the time evolution of the expectation values of position and momentum. We also compute the time evolution of the expectation value of the force and verify numerically that Ehrenfest's theorem holds. Below we list the basic quantum-mechanical relations needed.

The time derivative of the expectation value of position is proportional to the expectation value of momentum:

$$\frac{d}{dt}\langle x(t)\rangle = \frac{d}{dt}\int dx\psi^*(x,t)x\psi(x,t) = \int dx\psi^*(x,t)\frac{\left[x,\hat{H}\right]}{i\hbar}\psi(x,t) = \int dx\psi^*(x,t)\frac{\hat{p}_x}{m}\psi(x,t) = \frac{\langle p(t)\rangle}{m}. \tag{51}$$

In general, the time derivative of the expectation value of an operator $\hat{A}$ can be written as

$$\frac{d}{dt}\langle A(t)\rangle = \langle [A,H]\rangle. \tag{52}$$

Furthermore, the second time derivative of the expectation value of position can be evaluated

as follows:

$$\frac{d^2}{dt^2}\langle x(t)\rangle = \frac{1}{m}\frac{d}{dt}\langle p(t)\rangle = \frac{1}{m}\left\langle \frac{\left[p_x, \hat{H}\right]}{i\hbar} \right\rangle = \frac{1}{m}\left\langle -\frac{\partial V(x)}{\partial x} \right\rangle. \tag{53}$$

This equation shows that, in quantum mechanics, Newton's equation of motion holds for expectation values; this is known as Ehrenfest's theorem. To check that Ehrenfest's theorem holds, create your own code like Source Code 20 and compute the expectation values of position, momentum, and force.

`https://github.com/shunsuke-sato/python_qe/blob/develop/note_comp_phys/src/qm_dynamics_harmonic_expectation.py`

Source code 20: Example code for the time evolution of expectation values in quantum wave-packet dynamics in a harmonic potential

```
from numba import jit
from matplotlib import pyplot as plt
import numpy as np
import matplotlib.animation as animation
from matplotlib.animation import PillowWriter

# Initialize the wavefunction
def initialize_wf(xj, x0, k0, sigma0):
    wf = np.exp(1j*k0*(xj-x0))*np.exp(-0.5*(xj-x0)**2/sigma0**2)
    return wf

# Initialize potential
def initialize_vpot(xj):
    k0 = 1.0
    return 0.5*k0*xj**2

# Operate the Hamiltonian to the wavefunction
@jit(nopython=True)
def ham_wf(wf, vpot, dx):

    n = wf.size
    hwf = np.zeros(n, dtype=np.complex128)

    for i in range(1,n-1):
        hwf[i] = -0.5*(wf[i+1]-2.0*wf[i]+wf[i-1])/(dx**2)

    i = 0
    hwf[i] = -0.5*(wf[i+1]-2.0*wf[i])/(dx**2)
    i = n-1
    hwf[i] = -0.5*(-2.0*wf[i]+wf[i-1])/(dx**2)

    hwf = hwf + vpot*wf

    return hwf


# Time propagation from t to t+dt
def time_propagation(wf, vpot, dx, dt):

    n = wf.size
    twf = np.zeros(n, dtype=complex)
    hwf = np.zeros(n, dtype=complex)

    twf = wf
    zfact = 1.0 + 0j
    for iexp in range(1,5):
        zfact = zfact*(-1j*dt)/iexp
        hwf = ham_wf(twf, vpot, dx)
        wf = wf + zfact*hwf
        twf = hwf

    return wf

# Time propagation from t to t+dt
def calc_expectation_values(wf,xj,vpot):

    dx = xj[1]-xj[0]
```

```
    norm = np.sum(np.abs(wf)**2)*dx
    x_exp = np.sum(xj*np.abs(wf)**2)*dx
    x_exp = x_exp/norm

    n = wf.size
    pwf = np.zeros(n, dtype=complex)

    for i in range(1,n-1):
        pwf[i] = -1j*(wf[i+1]-wf[i-1])/(2.0*dx)

    p_exp = np.real(np.sum(np.conjugate(wf)*pwf)*dx)
    p_exp = p_exp/norm


    n = wf.size
    twf = np.zeros(n, dtype=complex)

    for i in range(1,n-1):
        twf[i] = -0.5*(wf[i+1]-2.0*wf[i]+wf[i-1])/(dx**2)


    Ekin = np.real(np.sum(np.conjugate(wf)*twf)*dx)
    Ekin = Ekin/norm

    Epot = np.real(np.sum(np.abs(wf)**2*vpot)*dx)
    Epot = Epot/norm

    return x_exp,p_exp,norm, Ekin, Epot


# initial wavefunction parameters
x0 = -2.0
k0 = 0.00
sigma0 = 1.0

# time propagation parameters
Tprop = 40.0
dt = 0.005
#dt = 0.00905
nt = int(Tprop/dt)+1

# set the coordinate
xmin = -10.0
xmax = 10.0
n = 250

dx = (xmax-xmin)/(n+1)
xj = np.zeros(n)

for i in range(n):
    xj[i] = xmin + dx*(i+1)


# Initialize the wavefunction
wf = initialize_wf(xj, x0, k0, sigma0)
#vpot = np.zeros(n)
vpot = initialize_vpot(xj)

# For expectation values
tt = np.zeros(nt+1)
xt = np.zeros(nt+1)
pt = np.zeros(nt+1)
norm_t = np.zeros(nt+1)
Ekin_t = np.zeros(nt+1)
Epot_t = np.zeros(nt+1)

# For loop for the time propagation
wavefunctions = []
for it in range(nt+1):
    if (it % (nt//100) == 0):
        wavefunctions.append(wf.copy())

    tt[it] = dt*it
    xt[it], pt[it], norm_t[it], Ekin_t[it], Epot_t[it]= calc_expectation_values(wf,xj,vpot)

    wf = time_propagation(wf, vpot, dx, dt)
    print(it, nt)

# Output the expectation value
plt.plot(tt,xt, label="x(t)")
```

```
plt.plot(tt,pt, label="p(t)")
plt.plot(tt,norm_t, label="norm(t)")
plt.xlabel('t')
plt.ylabel('Quantities')
plt.legend()

plt.savefig("expectation_value.pdf")
plt.cla()

plt.plot(tt,Ekin_t, label="Kinetic energy")
plt.plot(tt,Epot_t, label="Potential energy")
plt.plot(tt,Ekin_t+Epot_t, label="Total energy")
plt.xlabel('t')
plt.ylabel('Energy')
plt.legend()

plt.savefig("expectation_value_energy.pdf")

# Define function to update plot for each frame of the animation
def update_plot(frame):
    plt.cla()
    plt.xlim([-5, 5])
    plt.ylim([-1.2, 5.0])
    plt.plot(xj, np.real(wavefunctions[frame]), label="Real part of $\psi(x)$")
    plt.plot(xj, np.imag(wavefunctions[frame]), label="Imaginary part of $\psi(x)$")
    plt.plot(xj, np.abs(wavefunctions[frame])**2, label=" $|\psi(x)|^2$ ")
    plt.plot(xj, vpot, label="$V(x)$")
    plt.xlabel('$x$')
    plt.ylabel('$\psi(x)$')
    plt.legend()

# Create the animation
fig = plt.figure()
ani = animation.FuncAnimation(fig, update_plot, frames=len(wavefunctions), interval=150)
#ani.save('wavefunction_animation.gif', writer='imagemagick')
ani.save('wavefunction_animation.gif', writer='pillow')
```

7 Ground State and Excited State Calculations of One-Dimensional Quantum Systems

In this section, we solve the time-independent Schrödinger equation numerically and investigate the ground state and excited states of one-dimensional quantum systems.

7.1 Review of Linear Algebra

Before performing numerical calculations of the time-independent Schrödinger equation, let us review some fundamental aspects of linear algebra. Here, consider a square matrix A of size $n \times n$ with complex entries. The element in the i-th row and j-th column of matrix A is denoted by a_{ij}.

The transpose of a matrix is the operation of interchanging its rows and columns, and the transpose of matrix A is denoted as A^T. Therefore, for the matrix B defined by $B = A^T$, the element b_{ij} of B and the element of A are related as follows:

$$b_{ij} = a_{ji}. \tag{54}$$

The operation of taking the transpose of a matrix and then taking the complex conjugate is called the Hermitian conjugate, and the Hermitian conjugate of matrix A is denoted as $A^\dagger$. Thus, for the matrix C defined by $C = A^\dagger$, the element c_{ij} of C and the element of A are related as follows:

$$c_{ij} = a^*_{ji}. \tag{55}$$

Moreover, a matrix A for which the Hermitian conjugate $A^\dagger$ equals the original matrix ($A^\dagger = A$) is called a Hermitian matrix. When A is a Hermitian matrix, its eigenvalues are real numbers, and one can find n mutually orthogonal eigenvectors. Let us verify this.

7.1.1 Proof that the Eigenvalues of a Hermitian Matrix are Real

Let λ_i and $\boldsymbol{u}_i$ denote an eigenvalue and the corresponding eigenvector (column vector) of matrix A, respectively. Then the following relation holds:

$$A\boldsymbol{u}_i = \lambda_i \boldsymbol{u}_i \tag{56}$$

Multiplying both sides of Eq. (56) from the left by $\boldsymbol{u}_i^\dagger$, we obtain:

$$\boldsymbol{u}_i^\dagger A\boldsymbol{u}_i = \lambda_i(\boldsymbol{u}_i^\dagger \boldsymbol{u}_i) \tag{57}$$

Next, taking the Hermitian conjugate of Eq. (56), we get:

$$\boldsymbol{u}_i^\dagger A^\dagger = \boldsymbol{u}_i^\dagger A = \lambda_i^* \boldsymbol{u}_i^\dagger \tag{58}$$

Furthermore, multiplying both sides of Eq. (58) from the right by $\boldsymbol{u}_i$, we obtain:

$$\boldsymbol{u}_i^\dagger A\boldsymbol{u}_i = \lambda_i^*(\boldsymbol{u}_i^\dagger \boldsymbol{u}_i) \tag{59}$$

Since the norm of the eigenvector $\boldsymbol{u}_i$, given by $|\boldsymbol{u}_i|^2 = (\boldsymbol{u}_i^\dagger \boldsymbol{u}_i)$, is not zero, comparing Eq. (57) with Eq. (59) shows that $\lambda_i = \lambda_i^*$. In other words, we can confirm that the eigenvalues of a Hermitian matrix are real numbers.

7.1.2 Proof that eigenvectors corresponding to distinct eigenvalues of a Hermitian matrix are orthogonal

Next, for Hermitian matrices, we show that eigenvectors corresponding to distinct eigenvalues are orthogonal. Let λ_i and λ_j be two distinct eigenvalues of a Hermitian matrix, and let $\boldsymbol{u}_i$ and $\boldsymbol{u}_j$ be the eigenvectors corresponding to them, respectively. Then the following equations hold:

$$A\boldsymbol{u}_i = \lambda_i \boldsymbol{u}_i, \tag{60}$$

$$A\boldsymbol{u}_j = \lambda_j \boldsymbol{u}_j. \tag{61}$$

Also, by taking the Hermitian conjugate of Eq. (61), we obtain

$$\boldsymbol{u}_j^\dagger A = \lambda_j \boldsymbol{u}_j^\dagger. \tag{62}$$

Furthermore, multiplying Eq. (60) on the left by $\boldsymbol{u}_j^\dagger$, and multiplying Eq. (62) on the right by $\boldsymbol{u}_i$, we obtain the following relations:

$$\boldsymbol{u}_j^\dagger A\boldsymbol{u}_i = \lambda_i(\boldsymbol{u}_j^\dagger \boldsymbol{u}_i), \tag{63}$$

$$\boldsymbol{u}_j^\dagger A\boldsymbol{u}_i = \lambda_j(\boldsymbol{u}_j^\dagger \boldsymbol{u}_i). \tag{64}$$

Taking the difference between Eq. (63) and Eq. (64) yields

$$(\lambda_i - \lambda_j)(\boldsymbol{u}_j^\dagger \boldsymbol{u}_i) = 0. \tag{65}$$

Since λ_i and λ_j are distinct eigenvalues by assumption, we have $(\lambda_i - \lambda_j) \neq 0$, and hence from Eq. (65) it follows that $(\boldsymbol{u}_j^\dagger \boldsymbol{u}_i) = 0$ must hold. Therefore, eigenvectors corresponding to different eigenvalues of a Hermitian matrix are orthogonal.

7.1.3 On the orthogonality of eigenvectors corresponding to equal eigenvalues of a Hermitian matrix (the case of degenerate eigenvalues)

Next, we discuss the orthogonality of eigenvectors in the case where a Hermitian matrix has degenerate eigenvalues. Here, suppose that the eigenvalue λ of a Hermitian matrix is m-fold degenerate and that m linearly independent eigenvectors $\boldsymbol{u}_1, \boldsymbol{u}_2, \cdots, \boldsymbol{u}_m$ have already been obtained. In Sec. 7.1.2, we showed that eigenvectors corresponding to distinct eigenvalues of a Hermitian matrix are orthogonal; however, the eigenvalues of the eigenvectors considered here $(\boldsymbol{u}_1, \boldsymbol{u}_2, \cdots, \boldsymbol{u}_m)$ are equal, so in general these eigenvectors are not orthogonal. Nevertheless, using the fact that any linear combination of these eigenvectors is again an eigenvector with the same eigenvalue λ, we can construct a new set of orthonormal eigenvectors.

Here we explain how to construct an orthonormal set of eigenvectors using the Gram–Schmidt orthonormalization method. To construct an orthonormal set from the degenerate eigenvectors $(\boldsymbol{u}_1, \boldsymbol{u}_2, \cdots, \boldsymbol{u}_m)$, first normalize the initial eigenvector as follows:

$$\tilde{\boldsymbol{u}}_1 = \frac{1}{\sqrt{(\boldsymbol{u}_1^\dagger \boldsymbol{u}_1)}} \boldsymbol{u}_1. \tag{66}$$

Next, orthogonalize the second eigenvector $\boldsymbol{u}_2$ against the vector $\tilde{\boldsymbol{u}}_1$ and introduce the intermediate vector $\bar{\boldsymbol{u}}_2$ as

$$\bar{\boldsymbol{u}}_2 = \boldsymbol{u}_2 - \tilde{\boldsymbol{u}}_1(\tilde{\boldsymbol{u}}_1^\dagger \boldsymbol{u}_2). \tag{67}$$

By taking the inner product of $\bar{\boldsymbol{u}}_2$ with $\tilde{\boldsymbol{u}}_1$, it is easy to verify that the two vectors are orthogonal. Normalizing this intermediate vector $\bar{\boldsymbol{u}}_2$, we introduce the second orthonormalized eigenvector as

$$\tilde{\boldsymbol{u}}_2 = \frac{1}{\sqrt{(\bar{\boldsymbol{u}}_2^\dagger \bar{\boldsymbol{u}}_2)}} \bar{\boldsymbol{u}}_2. \tag{68}$$

Similarly, orthogonalize the third eigenvector $\boldsymbol{u}_3$ against the vectors $\tilde{\boldsymbol{u}}_1$ and $\tilde{\boldsymbol{u}}_2$, and introduce the intermediate vector $\bar{\boldsymbol{u}}_3$ as

$$\bar{\boldsymbol{u}}_3 = \boldsymbol{u}_3 - \tilde{\boldsymbol{u}}_1(\tilde{\boldsymbol{u}}_1^\dagger \boldsymbol{u}_3) - \tilde{\boldsymbol{u}}_2(\tilde{\boldsymbol{u}}_2^\dagger \boldsymbol{u}_3). \tag{69}$$

By normalizing this intermediate vector $\bar{\boldsymbol{u}}_3$, we introduce the third orthonormalized eigenvector as

$$\tilde{\boldsymbol{u}}_3 = \frac{1}{\sqrt{(\bar{\boldsymbol{u}}_3^\dagger \bar{\boldsymbol{u}}_3)}} \bar{\boldsymbol{u}}_3. \tag{70}$$

Repeating the same procedure, orthogonalize the k-th eigenvector $\boldsymbol{u}_k$ against $\tilde{\boldsymbol{u}}_1, \tilde{\boldsymbol{u}}_2, \cdots, \tilde{\boldsymbol{u}}_{k-1}$ and introduce the intermediate vector $\bar{\boldsymbol{u}}_k$ as

$$\bar{\boldsymbol{u}}_k = \boldsymbol{u}_k - \sum_{i=1}^{k-1} \tilde{\boldsymbol{u}}_i(\tilde{\boldsymbol{u}}_i^\dagger \boldsymbol{u}_k). \tag{71}$$

By normalizing this intermediate vector $\bar{\boldsymbol{u}}_k$, we introduce the k-th orthonormalized eigenvector as

$$\tilde{\boldsymbol{u}}_k = \frac{1}{\sqrt{(\bar{\boldsymbol{u}}_k^\dagger \bar{\boldsymbol{u}}_k)}} \bar{\boldsymbol{u}}_k. \tag{72}$$

By repeating this procedure up to $k = m$, we can construct, from the original set of m eigenvectors $(\boldsymbol{u}_1, \boldsymbol{u}_2, \cdots, \boldsymbol{u}_m)$, an orthonormal set of vectors $(\tilde{\boldsymbol{u}}_1, \tilde{\boldsymbol{u}}_2, \cdots, \tilde{\boldsymbol{u}}_m)$. Moreover, since each newly introduced vector $(\tilde{\boldsymbol{u}}_1, \tilde{\boldsymbol{u}}_2, \cdots, \tilde{\boldsymbol{u}}_m)$ is defined as a linear combination of the original eigenvectors $(\boldsymbol{u}_1, \boldsymbol{u}_2, \cdots, \boldsymbol{u}_m)$, the orthonormal set $(\tilde{\boldsymbol{u}}_1, \tilde{\boldsymbol{u}}_2, \cdots, \tilde{\boldsymbol{u}}_m)$ also consists of eigenvectors belonging to the eigenvalue λ of the matrix A.

7.1.4 A brief summary of properties of eigenvalues and eigenvectors of Hermitian matrices

As seen in Sec. 7.1.2, the eigenvalues of a Hermitian matrix are real. Also, as seen in Sec. 7.1.3 and Sec. 7.1.3, a set of eigenvectors of a Hermitian matrix can be chosen to be an orthonormal set.

7.2 Numerical computation of the diagonalization of a real symmetric matrix

Here, taking a 3×3 real symmetric matrix as an example, we learn Python code for diagonalizing a matrix. Consider the following real symmetric matrix:

$$A = \begin{pmatrix} 0 & 1 & 0 \\ 1 & 0 & 1 \\ 0 & 1 & 0 \end{pmatrix} \tag{73}$$

First, let us find the eigenvalues and eigenvectors of this real symmetric matrix. The eigenvalues λ can be obtained as the solutions of the following equation:

$$|A - \lambda I| = 0. \tag{74}$$

Next, let us write code to solve this eigenvalue problem numerically. The following code uses NumPy's linear algebra functions (`numpy.linalg`) to diagonalize a real symmetric matrix and obtain its eigenvalues and eigenvectors. Here, in particular, we use the diagonalization function specialized for real symmetric (and Hermitian) matrices, `numpy.linalg.eigh`. Read the documentation for NumPy's `numpy.linalg.eigh` ((`https://numpy.org/doc/stable/reference/`

generated/numpy.linalg.eigh.html#numpy.linalg.eigh)) to find out what kind of function it is.

https://github.com/shunsuke-sato/python_qe/blob/develop/note_comp_phys/src/eigen_3x3.py

Source code 21: Example code for diagonalizing a 3×3 real symmetric matrix

```
import numpy as np

matrix = np.array([[0.0, 1.0, 0.0],
                   [1.0, 0.0, 1.0],
                   [0.0, 1.0, 0.0]])

eigenvalues, eigenvectors = np.linalg.eigh(matrix)

print('First eigenvalue  =',eigenvalues[0])
print('Second eigenvalue =',eigenvalues[1])
print('Third eigenvalue  =',eigenvalues[2])
print()

print('First eigenvector')
print(eigenvectors[:,0])
print()

print('Second eigenvector')
print(eigenvectors[:,1])
print()

print('Third eigenvector')
print(eigenvectors[:,2])
print()
```

Run Source Code 21 and compare the analytically obtained eigenvalues and eigenvectors with those obtained by numerical computation.

7.3 Solving the Time-Independent Schrödinger Equation Using the Real-Space Finite Difference Method

7.3.1 Infinite Square Well Potential Problem

Here, we will learn how to solve the time-independent Schrödinger equation using the real-space method introduced in Sec. 6.1. As an example, let us consider the problem of an infinite square well potential:

$$\left[-\frac{\hbar^2}{2m}\frac{\partial^2}{\partial x^2} + V(x)\right] u_n(x) = Eu(x), \tag{75}$$

$$V(x) = \begin{cases} 0 & -\frac{L}{2} \le x \le \frac{L}{2} \\ \infty & \text{otherwise} \end{cases} \tag{76}$$

Here, since the potential $V(x)$ diverges for $|x| > L/2$, the wavefunction $u(x)$ must vanish in that region. That is, the wavefunction $u(x)$ can take finite values only within the region $(-L/2 \le x \le L/2)$. We divide this region into N grid points. For later convenience, let us define the coordinate of the (-1)-th point as $x_{-1} = -L/2$ and that of the N-th point as $x_N = L/2$. Then, the position of the j-th point can be written as

$$x_j = -\frac{L}{2} + \delta x \times (j+1). \tag{77}$$

Here, the grid spacing Δx is defined as

$$\Delta x = \frac{L}{N+1}. \tag{78}$$

Next, let us discretize Eq. (75). Denoting the wavefunction value at the j-th grid point x_j as $u_j = u(x_j)$, and approximating the second derivative with the three-point finite difference formula, we can approximate Eq. (75) by the following system of equations:

$$-\frac{\hbar^2}{2m}\frac{-2u_0+u_1}{\Delta x^2} = Eu_0 \tag{79}$$

$$-\frac{\hbar^2}{2m}\frac{u_0-2u_1+u_2}{\Delta x^2} = Eu_1 \tag{80}$$

$$\vdots$$

$$-\frac{\hbar^2}{2m}\frac{u_{j-1}-2u_j+u_{j+1}}{\Delta x^2} = Eu_j \tag{81}$$

$$\vdots$$

$$-\frac{\hbar^2}{2m}\frac{u_{N-3}-2u_{N-2}+u_{N-1}}{\Delta x^2} = Eu_{N-2} \tag{82}$$

$$-\frac{\hbar^2}{2m}\frac{u_{N-2}-2u_{N-1}}{\Delta x^2} = Eu_{N-1} \tag{83}$$

Here, note that due to the boundary condition ($u(x) = 0$ for $|x| \geq L/2$), we have $u_{-1} = u_N = 0$.

Carefully observing the above set of equations, we see that they can be rewritten as an eigenvalue problem for a matrix:

$$-\frac{\hbar^2}{2m}\frac{1}{\Delta x^2}\begin{pmatrix} -2 & 1 & 0 & \cdots & 0 \\ 1 & -2 & 1 & \ddots & 0 \\ 0 & \ddots & \ddots & \ddots & 0 \\ \vdots & \ddots & 1 & -2 & 1 \\ 0 & 0 & 0 & 1 & -2 \end{pmatrix}\begin{pmatrix} u_0 \\ u_1 \\ \vdots \\ u_{N-2} \\ u_{N-1} \end{pmatrix} = E\begin{pmatrix} u_0 \\ u_1 \\ \vdots \\ u_{N-2} \\ u_{N-1} \end{pmatrix} \tag{84}$$

Thus, the original Schrödinger equation, Eq. (75), can be rewritten, using the finite difference approximation, as the matrix eigenvalue problem in Eq. (84).

Using the program for matrix diagonalization learned in Sec. 7.2 as reference, try implementing your own program to solve the eigenvalue problem represented by Eq. (84) for the infinite square well potential, and obtain its eigenvalues and eigenfunctions. Compare the numerical results with the analytical ones.

For reference, the exact eigenvalues E_n and eigenfunctions $u_n(x)$ of Eq. (75) are:

$$E_n = n^2\frac{\pi^2\hbar^2}{2mL^2} \tag{85}$$

$$u_n(x) = \begin{cases} \sqrt{\frac{2}{L}}\cos\left(\frac{n\pi}{L}x\right) & (n = \text{odd}), (-L/2 < x < L/2) \\ \sqrt{\frac{2}{L}}\sin\left(\frac{n\pi}{L}x\right) & (n = \text{even}), (-L/2 < x < L/2) \\ 0 & (\text{otherwise}) \end{cases} \tag{86}$$

For reference, a sample Python code is provided below:

`https://github.com/shunsuke-sato/python_qe/blob/develop/note_comp_phys/src/qm_grid_infinite_well.py`

Source code 22: Sample code for solving the infinite square well using the real-space finite difference method

```python
import numpy as np
from matplotlib import pyplot as plt

# Constants
mass = 1.0
hbar = 1.0

# Define grid
num_grid = 64
length = 20.0
dx = length / (num_grid + 1)
xj = np.linspace(-length / 2 + dx, length / 2 - dx, num_grid)

# Hamiltonian Matrix
ham_mat = np.zeros((num_grid,num_grid))

for i in range(num_grid):
    for j in range(num_grid):
        if(i == j):
            ham_mat[i,j]=-0.5*hbar**2/mass*(-2.0/dx**2)
        elif(np.abs(i-j) == 1):
            ham_mat[i,j]=-0.5*hbar**2/mass*(1.0/dx**2)

# Calculate eigenvectors and eigenvalues
eigenvalues, eigenvectors = np.linalg.eigh(ham_mat)

# Normalize and check the sign
wf = eigenvectors/np.sqrt(dx)
for i in range(num_grid):
    sign = np.sign(wf[num_grid//2,i])
    if(sign != 0.0):
        wf[:,i] = wf[:,i]*sign


def exact_eigenvalue(n):
    """Calculate exact eigenvalue for particle in a box."""
    return n**2 * np.pi**2 * hbar**2 / (2.0 * mass * length**2)

# Print eigenvalues and errors
for i in range(3):
    print(f"{i}-th eigenvalue = {eigenvalues[i]}")
    print(f"{i}-th eigenvalue Error = {eigenvalues[i] - exact_eigenvalue(i + 1)}")
    print()

# Plotting
plt.figure(figsize=(8,6))
plt.plot(xj, wf[:, 0], label="Ground state (calc.)")
plt.plot(xj, np.sqrt(2.0/length)*np.cos(np.pi*xj/length), label="Ground state (exact.)",
  linestyle='dashed')
plt.plot(xj, wf[:, 1], label="1st excited state")
plt.plot(xj, np.sqrt(2.0/length)*np.sin(2.0*np.pi*xj/length), label="1st excited state (exact.)
  ", linestyle='dashed')
plt.plot(xj, wf[:, 2], label="2nd excited state")
plt.plot(xj, np.sqrt(2.0/length)*np.cos(3.0*np.pi*xj/length), label="2nd excited state (exact.)
  ", linestyle='dashed')

plt.xlim([-length/2.0,length/2.0])
plt.xlabel('x')
plt.ylabel('wave functions')
plt.legend()
plt.savefig('fig_quantum_well_wf.pdf')
plt.show()
```

By running Source Code 22, one can obtain the eigenvalues and wavefunctions of the infinite square well potential. Some comments on the code are given below.

In line 28 of Source Code 22, the command `wf = eigenvectors/np.sqrt(dx)` normalizes the eigenvectors obtained from the diagonalization routine `np.linalg.eigh`. The eigenvectors $\boldsymbol{u}$ returned by `np.linalg.eigh` are normalized according to

$$|\boldsymbol{u}|^2 = \sum_{j=0}^{N-1} |u_j|^2 = 1. \tag{87}$$

However, in typical quantum mechanics problems, it is more convenient to normalize eigen-

functions $u(x)$ such that

$$\int_{-L/2}^{L/2} dx|u(x)|^2 = 1. \tag{88}$$

Recalling that $u(x_j) = u_j$, the normalization condition, Eq. (89), can be rewritten as

$$\int_{-L/2}^{L/2} dx|u(x)|^2 = 1. \tag{89}$$

Thus, to convert an eigenvector originally normalized as in (87) into one normalized as in (**??**), the following transformation is applied:

$$u_j \to \frac{u_j}{\sqrt{\Delta x}}. \tag{90}$$

This renormalization procedure is exactly what is implemented in line 28 of Source Code 22.

Figure 10 shows the wavefunctions output by the program.

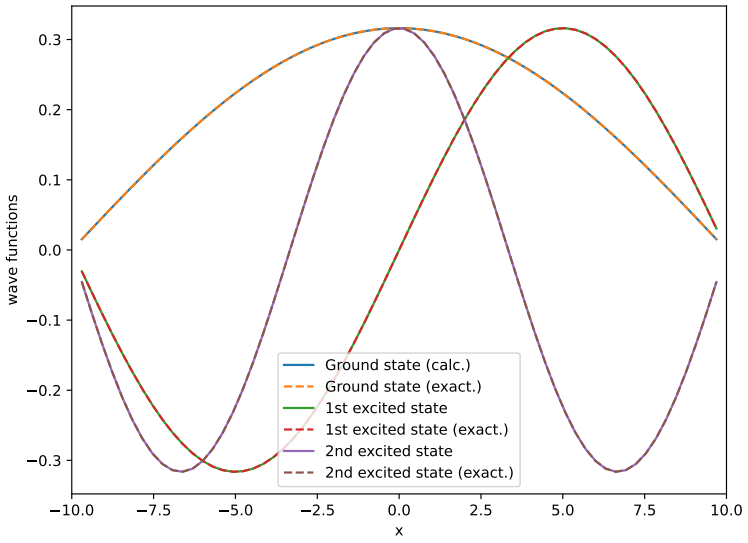

Figure 10: Eigenfunctions of the infinite square well potential.

7.3.2 Ground and Excited States of the 1D Harmonic Oscillator

In Sec. 7.3.1, we worked on the numerical solution of the infinite square well potential. In this section, let us consider the one-dimensional harmonic oscillator problem. The Schrödinger equation for the 1D harmonic oscillator is written as

$$\left[-\frac{\hbar^2}{2m}\frac{d^2}{dx^2} + \frac{k}{2}x^2\right] u(x) = Eu(x). \tag{91}$$

By solving this differential equation under the boundary condition $u(x) \to 0$ as $|x| \to \infty$, we can obtain the energy eigenvalues and eigenstates of the quantum harmonic oscillator. However, it is not easy to handle wavefunctions at infinity numerically. In practice, for numerical eigenstate calculations of quantum systems, instead of imposing boundary conditions at infinity, it is common to impose them at a finite but sufficiently large distance. Specifically, we introduce a sufficiently

large length L and solve Eq. (91) under the boundary condition $u(\pm L/2) = 0$. If L is large enough, the numerical solution approaches the exact solution with vanishing wavefunction at infinity. From another perspective, inspired by Sec. 7.3.1, imposing $u(\pm L/2) = 0$ is equivalent to placing infinitely high potential walls at $x = \pm L/2$. If these walls are placed far from the region of interest, they hardly affect the physical properties of the quantum system.

By following the infinite square well example, discretize the time-independent Schrödinger equation and rewrite it as a matrix diagonalization problem. Then, write your own program to obtain the eigenstates of the harmonic oscillator, and investigate the ground state and excited states. Furthermore, consider the relation between the probability distributions derived from high-energy excited states and the classical probability distribution of position.

For reference, the wavefunctions of the ground, first excited, and second excited states of the harmonic oscillator are:

$$\begin{aligned}
\psi_0(x) &= \left(\frac{m\omega}{\pi\hbar}\right)^{1/4} e^{-\frac{m\omega}{2\hbar}x^2}, \\
\psi_1(x) &= x\sqrt{\frac{2m\omega}{\hbar}}\left(\frac{m\omega}{\pi\hbar}\right)^{1/4} e^{-\frac{m\omega}{2\hbar}x^2}, \\
\psi_2(x) &= \sqrt{2}\left(1 - \frac{2m\omega x^2}{\hbar}\right)\left(\frac{m\omega}{\pi\hbar}\right)^{1/4} e^{-\frac{m\omega}{2\hbar}x^2}.
\end{aligned} \tag{92}$$

`https://github.com/shunsuke-sato/python_qe/blob/develop/note_comp_phys/src/qm_grid_ho.py`

Source code 23: Example code for calculating eigenvalues and eigenfunctions of the 1D harmonic oscillator

```python
import numpy as np
from matplotlib import pyplot as plt

# Constants
mass = 1.0
hbar = 1.0
kconst = 1.0

# Define grid
num_grid = 128
length = 15.0
dx = length / (num_grid + 1)
xj = np.linspace(-length / 2 + dx, length / 2 - dx, num_grid)

# Potential
vpot = 0.5*kconst*xj**2

# Hamiltonian Matrix
ham_mat = np.zeros((num_grid,num_grid))

for i in range(num_grid):
    for j in range(num_grid):
        if(i == j):
            ham_mat[i,j]=-0.5*hbar**2/mass*(-2.0/dx**2) + vpot[i]
        elif(np.abs(i-j) == 1):
            ham_mat[i,j]=-0.5*hbar**2/mass*(1.0/dx**2)


# Calculate eigenvectors and eigenvalues
eigenvalues, eigenvectors = np.linalg.eigh(ham_mat)

# Normalize and check the sign
wf = eigenvectors/np.sqrt(dx)
for i in range(num_grid):
    sign = np.sign(wf[num_grid//2,i])
    if(sign != 0.0):
        wf[:,i] = wf[:,i]*sign


def exact_eigenvalue(n):
    """Calculate exact eigenvalue for quantum harmonic oscillator."""
```

```
    return hbar * np.sqrt(kconst / mass) * (n + 0.5)

# Print eigenvalues and errors
for i in range(3):
    print(f"{i}-th eigenvalue = {eigenvalues[i]}")
    print(f"{i}-th eigenvalue Error = {eigenvalues[i] - exact_eigenvalue(i)}")
    print()


# Plotting
omega = np.sqrt(kconst/mass)

# Ground state plot
plt.figure(figsize=(8, 6))
plt.plot(xj, wf[:, 0], label="Ground state (calc.)")
plt.plot(xj, (mass * omega / (np.pi * hbar))**(1.0 / 4.0) * np.exp(-mass * omega * xj**2 / (2.0 * hbar)),
                  label="Ground state (exact.)", linestyle='dashed')
plt.xlim([-length / 2.0, length / 2.0])
plt.xlabel('x')
plt.ylabel('wave functions')
plt.legend()
plt.savefig('fig_harmonic_oscillator_ground_state.pdf')
plt.show()

# First excited state plot
plt.figure(figsize=(8, 6))
plt.plot(xj, wf[:, 1], label="1st excited state (calc.)")
plt.plot(xj, (mass * omega / (np.pi * hbar))**(1.0 / 4.0) * np.sqrt(2.0 * mass * omega / hbar) * xj * np.exp(-mass * omega * xj**2 / (2.0 * hbar)),
                  label="1st excited state (exact.)", linestyle='dashed')
plt.xlim([-length / 2.0, length / 2.0])
plt.xlabel('x')
plt.ylabel('wave functions')
plt.legend()
plt.savefig('fig_harmonic_oscillator_1st_excited_state.pdf')
plt.show()

# Second excited state plot
plt.figure(figsize=(8, 6))
plt.plot(xj, wf[:, 2], label="2nd excited state (calc.)")
plt.plot(xj, (mass * omega / (np.pi * hbar))**(1.0 / 4.0) * np.sqrt(0.5) * (1.0 - 2.0 * mass * omega * xj**2 / hbar) * np.exp(-mass * omega * xj**2 / (2.0 * hbar)),
                  label="2nd excited state (exact.)", linestyle='dashed')
plt.xlim([-length / 2.0, length / 2.0])
plt.xlabel('x')
plt.ylabel('wave functions')
plt.legend()
plt.savefig('fig_harmonic_oscillator_2nd_excited_state.pdf')
plt.show()
```

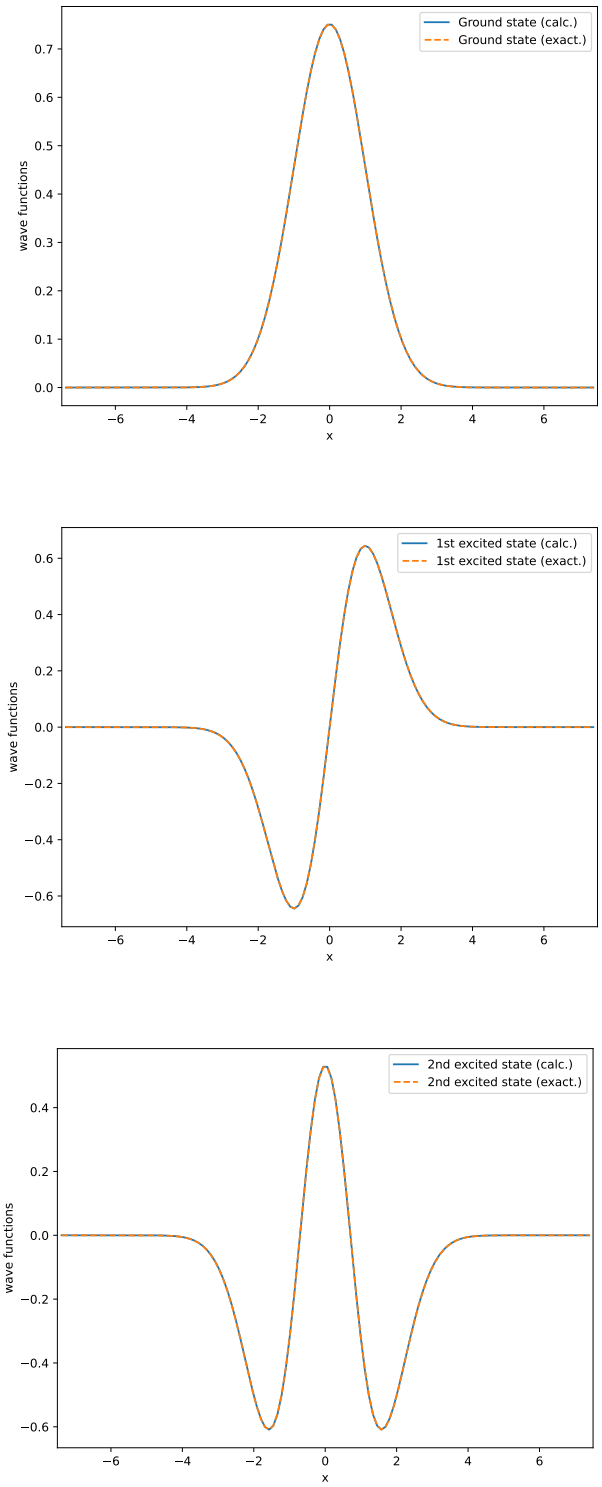

Figure 11: Eigenfunctions of the harmonic oscillator potential.

Next, let us consider the classical probability distribution of a particle. For a classical harmonic oscillator with energy E, suppose we measure the particle's position at random times. Taking into

account that the sum of kinetic and potential energies equals E, the particle can only be found within the interval $(-\sqrt{2E/k} \leq x \leq \sqrt{2E/k})$. Furthermore, when measuring the particle's position at random times within this interval, the probability $P(x)$ of finding the particle in a small interval $[x, x+\delta x]$ is proportional to the inverse of the particle's speed. That is,

$$P(x) \sim \frac{1}{|v|}. \tag{93}$$

Since the total energy E is given, the particle's speed at each position x is expressed as

$$|v| = \sqrt{\frac{2}{m}\left(E - \frac{1}{2}kx^2\right)} \tag{94}$$

Therefore, the probability distribution $P(x)$ of finding the particle at position x for a harmonic oscillator with energy E is given by

$$P(x) = \frac{1}{\pi\sqrt{\frac{2E}{k} - x^2}} \tag{95}$$

Source Code 24 is a modified version of Source Code 23, which includes functionality to compare classical and quantum probability distributions. Figure 12 shows a comparison between the quantum probability distribution of the 64th eigenstate and the corresponding classical distribution.

`https://github.com/shunsuke-sato/python_qe/blob/develop/note_comp_phys/src/qm_grid_ho_high_energy.py`

Source code 24: Example code comparing high-energy states of the 1D harmonic oscillator with the classical probability distribution

```python
import numpy as np
from matplotlib import pyplot as plt

# Constants
mass = 1.0
hbar = 1.0
kconst = 1.0

# Define grid
num_grid = 512
length = 30.0
dx = length / (num_grid + 1)
xj = np.linspace(-length / 2 + dx, length / 2 - dx, num_grid)

# Potential
vpot = 0.5*kconst*xj**2

# Hamiltonian Matrix
ham_mat = np.zeros((num_grid,num_grid))

for i in range(num_grid):
    for j in range(num_grid):
        if(i == j):
            ham_mat[i,j]=-0.5*hbar**2/mass*(-2.0/dx**2) + vpot[i]
        elif(np.abs(i-j) == 1):
            ham_mat[i,j]=-0.5*hbar**2/mass*(1.0/dx**2)


# Calculate eigenvectors and eigenvalues
eigenvalues, eigenvectors = np.linalg.eigh(ham_mat)

# Normalize and check the sign
wf = eigenvectors/np.sqrt(dx)
for i in range(num_grid):
    sign = np.sign(wf[num_grid//2,i])
    if(sign != 0.0):
        wf[:,i] = wf[:,i]*sign
```

```python

def exact_eigenvalue(n):
    """Calculate exact eigenvalue for quantum harmonic oscillator."""
    return hbar * np.sqrt(kconst / mass) * (n + 0.5)

# Print eigenvalues and errors
for i in range(3):
    print(f"{i}-th eigenvalue = {eigenvalues[i]}")
    print(f"{i}-th eigenvalue Error = {eigenvalues[i] - exact_eigenvalue(i)}")
    print()


# Plotting
omega = np.sqrt(kconst/mass)


# Calculate the probability distribution fo a highly-excited state
n_eigen = 64
Ene = eigenvalues[n_eigen]

prob_x = np.zeros(num_grid)
for i in range(num_grid):
    if( 2.0*Ene/kconst-xj[i]**2 < 0):
        prob_x[i]=0.0
    else:
        prob_x[i]=1.0/(np.pi*np.sqrt(2.0*Ene/kconst-xj[i]**2))

plt.figure(figsize=(8,6))

plt.plot(xj, wf[:, n_eigen]**2, label="Quantum distribution")
plt.plot(xj, prob_x, label="Classical distribution")
plt.xlim([-length/2.0,length/2.0])
plt.xlabel('x')
plt.ylabel('$|\psi(x)|^2$')
plt.legend()
plt.savefig('harmonic_oscillator_high_energy_prob.pdf')
plt.show()
```

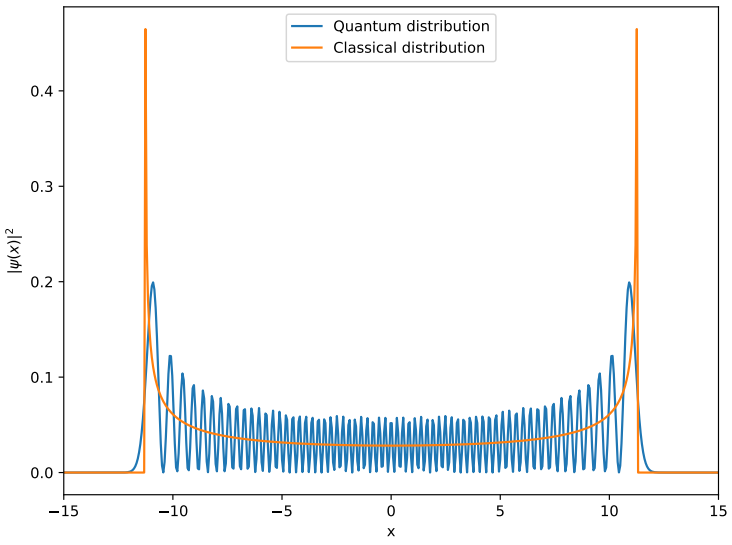

Figure 12: Eigenfunction of the harmonic oscillator potential.

8 Quantum Dynamics under a Time-Dependent Hamiltonian

8.1 Time Evolution under a Time-Dependent Hamiltonian

In Sec. 6, we learned how to numerically solve the time-dependent Schrödinger equation under a time-independent Hamiltonian. In this section, we extend that knowledge and study how to solve the time-dependent Schrödinger equation when the Hamiltonian itself depends on time. Such simulations can be applied, for example, to investigate electron dynamics in atoms under laser fields.

To consider how to numerically solve the time-dependent Schrödinger equation, let us look at the following equation involving a time-dependent Hamiltonian:

$$i\hbar\frac{d}{dt}\psi(x,t) = \hat{H}(t)\psi(x,t). \tag{96}$$

When the Hamiltonian is time-independent ($H(t) = H_0$), the formal solution of the Schrödinger equation can be written as

$$\psi(x,t) = \exp\left[-\frac{i}{\hbar}H_0 t\right]\psi(x,0). \tag{97}$$

However, when the Hamiltonian depends on time, the formal solution of the Schrödinger equation cannot be written in this simple form. To obtain the formal solution in the time-dependent case, we consider an small time-step evolution by advancing time by a small increment Δt. If Δt is taken sufficiently small, the time variation of the Hamiltonian between t and $t + \Delta t$ becomes negligible, allowing us to ignore its time dependence during that interval. Using the formal solution for a time-independent Hamiltonian, the time evolution from t_0 to $t_0 + \Delta t$ can then be expressed as

$$\psi(x, t_0 + \Delta t) = \exp\left[-\frac{i}{\hbar}H(t_0)\Delta t\right]\psi(x,t_0). \tag{98}$$

By repeatedly applying this small time-step evolution, we can obtain the solution of the Schrödinger equation. For example, if we apply this process N times, the wavefunction at time $t = t_N = \Delta t \times N$ is given by

$$\begin{aligned}\psi(x,t_N) =& \exp\left[-\frac{i}{\hbar}H(t_{N-1})\Delta t\right] \times \exp\left[-\frac{i}{\hbar}H(t_{N-2})\Delta t\right] \times \cdots \\ &\times \exp\left[-\frac{i}{\hbar}H(t_1)\Delta t\right] \times \exp\left[-\frac{i}{\hbar}H(t_0)\Delta t\right]\psi(x,t_0),\end{aligned} \tag{99}$$

where we have defined $t_j = t_0 + j \times \Delta t$.

Thus, by repeatedly performing small time-step evolutions using the Hamiltonian at each time step, we can construct the solution to the time-dependent Schrödinger equation. In numerical simulations, this same procedure is employed: Eq. (99) is used to iteratively compute the wavefunction's time evolution.

Before solving the time-dependent Schrödinger equation, let us try to rewrite the formal solution in Eq. (99) into a more compact form. If the Hamiltonian were just a number rather than an operator, the time evolution operator from t_0 to t_N could be written as

$$U(t_N,t_0) = \exp\left[-\frac{i}{\hbar}H(t_{N-1})\Delta t\right] \times \cdots \times \exp\left[-\frac{i}{\hbar}H(t_0)\Delta t\right] \neq \exp\left[-\frac{i}{\hbar}\sum_{j=0}^{N-1}H(t_j)\Delta t\right]. \tag{100}$$

However, since the Hamiltonian is an operator and not just a number, such a simplification as in Eq. (100) is not possible. In fact, expanding both sides in a Taylor series shows that the expansion

of the left-hand side involves products of Hamiltonians at different times (e.g., $H(t_2)H(t_1)H(t_0)$), which always appear ordered such that the Hamiltonian at an earlier time $H(t_i)$ is to the right of any Hamiltonian at a later time $H(t_j)$. On the other hand, the expansion of the right-hand side produces Hamiltonian products in all possible time orders (e.g., $H(t_1)H(t_2)H(t_0)$). Therefore, when Hamiltonians at different times do not commute, Eq. (100) cannot be rewritten in this way.

From another perspective, if we take each product of Hamiltonians appearing in the expansion of the right-hand side of Eq. (100) and rearrange them in chronological order (e.g., $H(t_2)H(t_1)H(t_0)$), we obtain the same result as the expansion of the left-hand side, and equality holds. This rearrangement of operators according to time order is called the **time-ordered product (T-product)**, and it is expressed as

$$\mathcal{T}\left\{H(t_0)H(t_2)H(t_1)\right\} = H(t_2)H(t_1)H(t_0). \tag{101}$$

Using the time-ordered product, the time evolution operator in Eq. (99) can be expressed as

$$\begin{aligned} U(t_N,t_0) =& \exp\left[-\frac{i}{\hbar}H(t_{N-1})\Delta t\right] \times \exp\left[-\frac{i}{\hbar}H(t_{N-2})\Delta t\right] \times \cdots \\ & \times \exp\left[-\frac{i}{\hbar}H(t_1)\Delta t\right] \times \exp\left[-\frac{i}{\hbar}H(t_0)\right] \\ =& \mathcal{T}\left\{\exp\left[-\frac{i}{\hbar}\sum_0^{N-1} H(t_j)\Delta t\right]\right\} \\ =& \mathcal{T}\left\{\exp\left[-\frac{i}{\hbar}\int_{t_0}^{t_N} dt H(t)\right]\right\}. \end{aligned} \tag{102}$$

In the last line, we have assumed that Δt is sufficiently small so that the sum can be replaced by an integral. In this way, even in the case of a time-dependent Hamiltonian, the time evolution operator can be formally expressed using the time-ordered product.

8.2 Dynamics of a Quantum Wavepacket in an Oscillating Harmonic Potential

As an example of a time-evolution calculation using a time-dependent Hamiltonian, let us simulate the dynamics of a quantum wavepacket in a one-dimensional oscillating harmonic potential. The equation to be solved is the time-dependent Schrödinger equation given by

$$i\hbar\frac{\partial}{\partial t}\psi(x,t) = -\frac{\hbar^2}{2m}\frac{\partial^2\psi(x,t)}{\partial x^2} + \frac{m\omega_0^2}{2}\left(x - x_c(t)\right)^2\psi(x,t). \tag{103}$$

Here, $x_c(t)$ is the center of the harmonic potential that moves with time. As the initial condition of this equation, we adopt the ground state of the Hamiltonian at time $t = 0$. For simplicity, from here on we set $m = \hbar = \omega_0 = 1$.

With respect to the natural frequency of this harmonic oscillator ($\omega_0 = 1$), let us perform calculations for the following cases: (a) $x_c(t)$ moves on a sufficiently slow timescale, (b) $x_c(t)$ moves on the same timescale, and (c) $x_c(t)$ moves on a sufficiently fast timescale. Specifically, for the sufficiently slow case (a),

$$x_c(t) = \cos(\Omega t) \tag{104}$$

and let us run the calculation with $\Omega = 0.2\omega_0$. Under the resonance condition (b),

$$x_c(t) = \cos(\Omega t) \tag{105}$$

and let us run the calculation with $\Omega = \omega_0$. Furthermore, for the sufficiently fast case (c),

$$x_c(t) = \sin(\Omega t) \tag{106}$$

and let us run the calculation with $\Omega = 8\omega_0$.

First, let us write code to obtain the ground state of the Hamiltonian of the harmonic oscillator. This ground state will serve as the initial wavefunction for the time evolution. Source Code 25 below shows an example of Python code to obtain the ground state of the harmonic oscillator.

https://github.com/shunsuke-sato/python_qe/blob/develop/note_comp_phys/src/qm_dynamics_td_ham_v1.py

Source code 25: Example code to obtain the ground state of the harmonic oscillator

```
from numba import jit
from matplotlib import pyplot as plt
import numpy as np
import matplotlib.animation as animation
from matplotlib.animation import PillowWriter


# Construct potential
def construct_potential(xj, xc):
    return 0.5*(xj-xc)**2

def calc_ground_state(xj, potential):

    num_grid = xj.size
    dx = xj[1]-xj[0]

    ham = np.zeros((num_grid, num_grid))

    for i in range(num_grid):
        for j in range(num_grid):
            if(i == j):
                ham[i,j] = -0.5*(-2.0/dx**2)+potential[i]
            elif(np.abs(i-j)==1):
                ham[i,j] = -0.5*(1.0/dx**2)

    eigenvalues, eigenvectors = np.linalg.eigh(ham)

    wf = np.zeros(num_grid, dtype=complex)

    wf.real = eigenvectors[:,0]/np.sqrt(dx)

    return wf

# time propagation parameters
Tprop = 40.0
dt = 0.005
#dt = 0.00905
nt = int(Tprop/dt)+1

# set the coordinate
xmin = -10.0
xmax = 10.0
num_grid = 250

xj = np.linspace(xmin, xmax, num_grid)

# set potential
xc = 1.0
potential = construct_potential(xj, xc)

# calculate the ground state
wf = calc_ground_state(xj, potential)



# plot the ground state density, |wf|^2
rho = np.abs(wf)**2

plt.figure(figsize=(8,6))
plt.plot(xj, rho, label="$|\psi(x)|^2$␣(calc.)")
plt.plot(xj, np.exp(-(xj-xc)**2)/np.sqrt(np.pi),
         label="$|\psi(x)|^2$␣(exact)", linestyle='dashed')
plt.plot(xj, 0.5*(xj-xc)**2,
         label="Harmonic␣potential", linestyle='dotted')

plt.xlim([-4.0, 4.0])
```

```
plt.ylim([0.0, 0.8])
plt.xlabel('x')
plt.ylabel('Density, Potential')
plt.legend()
plt.savefig('gs_density.pdf')
plt.show()
```

Figure 13 shows the ground-state density obtained by the above code.

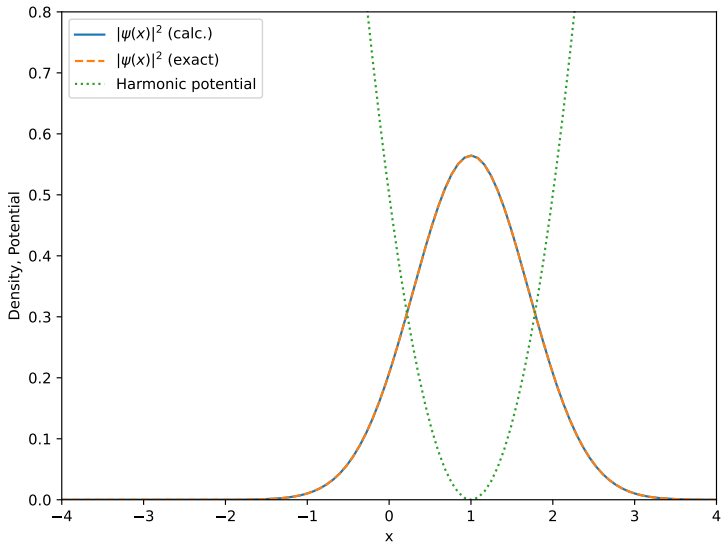

Figure 13: Variational analysis of the harmonic oscillator

Next, using as the initial condition the ground state obtained by the code above, let us write Python code to examine the time evolution of the wavefunction. As expressed in Eq. (99), the time evolution under a time-dependent Hamiltonian can be described by repeatedly applying the infinitesimal time-evolution operator constructed from the Hamiltonian at each time. By evaluating this infinitesimal time-evolution operator using a fourth-order Taylor expansion, let us compute the time evolution of the wavefunction.

$$\begin{aligned}\psi(x, t_{j+1}) &\approx \exp\left[-\frac{i}{\hbar}\Delta t H(t_j)\right]\psi(x, t_j) \\ &\approx \sum_{n=0}^{4}\frac{1}{n!}\left(\frac{-i\Delta t}{\hbar}\right)^n H^n(t_j)\psi(x, t_j). \end{aligned} \tag{107}$$

The source code 26 below provides an example of Python code to investigate the dynamics of a quantum wavepacket moving in an oscillating harmonic potential.

`https://github.com/shunsuke-sato/python_qe/blob/develop/note_comp_phys/src/qm_dynamics_td_ham_v2.py`

Source code 26: Example code to obtain the ground state of the harmonic oscillator

```
from numba import jit
from matplotlib import pyplot as plt
import numpy as np
import matplotlib.animation as animation
from matplotlib.animation import PillowWriter
```

```
6
7
8  # Construct potential
9  def construct_potential(xj, xc):
10     return 0.5*(xj-xc)**2
11
12 def calc_ground_state(xj, potential):
13
14     num_grid = xj.size
15     dx = xj[1]-xj[0]
16
17     ham = np.zeros((num_grid, num_grid))
18
19     for i in range(num_grid):
20         for j in range(num_grid):
21             if(i == j):
22                 ham[i,j] = -0.5*(-2.0/dx**2)+potential[i]
23             elif(np.abs(i-j)==1):
24                 ham[i,j] = -0.5*(1.0/dx**2)
25
26     eigenvalues, eigenvectors = np.linalg.eigh(ham)
27
28     wf = np.zeros(num_grid, dtype=complex)
29
30     wf.real = eigenvectors[:,0]/np.sqrt(dx)
31
32     return wf
33
34 # Operate the Hamiltonian to the wavefunction
35 @jit(nopython=True)
36 def ham_wf(wf, potential, dx):
37
38     n = wf.size
39     hwf = np.zeros(n, dtype=np.complex128)
40
41     for i in range(1,n-1):
42         hwf[i] = -0.5*(wf[i+1]-2.0*wf[i]+wf[i-1])/(dx**2)
43
44     i = 0
45     hwf[i] = -0.5*(wf[i+1]-2.0*wf[i])/(dx**2)
46     i = n-1
47     hwf[i] = -0.5*(-2.0*wf[i]+wf[i-1])/(dx**2)
48
49     hwf = hwf + potential*wf
50
51     return hwf
52
53
54 # Time propagation from t to t+dt
55 def time_propagation(wf, potential, dx, dt):
56
57     n = wf.size
58     twf = np.zeros(n, dtype=complex)
59     hwf = np.zeros(n, dtype=complex)
60
61     twf = wf
62     zfact = 1.0 + 0j
63     for iexp in range(1,5):
64         zfact = zfact*(-1j*dt)/iexp
65         hwf = ham_wf(twf, potential, dx)
66         wf = wf + zfact*hwf
67         twf = hwf
68
69     return wf
70
71
72 # time propagation parameters
73 #omega = 8.0
74 omega = 0.2
75 Tprop = 80.0
76 dt = 0.005
77 nt = int(Tprop/dt)+1
78
79 # set the coordinate
80 xmin = -10.0
81 xmax = 10.0
82 num_grid = 250
83
84 xj = np.linspace(xmin, xmax, num_grid)
85 dx = xj[1] - xj[0]
```

```
86
87  # set potential
88  xc = 1.0
89  potential = construct_potential(xj, xc)
90
91  # calculate the ground state
92  wf = calc_ground_state(xj, potential)
93
94
95  # For loop for the time propagation
96  density_list = []
97  xc_list = []
98  for it in range(nt+1):
99      tt = it*dt
100     xc = np.cos(omega*tt)
101     if(it % (nt//200) == 0):
102         rho = np.abs(wf)**2
103         density_list.append(rho.copy())
104         xc_list.append(xc)
105
106
107     potential = construct_potential(xj, xc)
108
109     wf = time_propagation(wf, potential, dx, dt)
110     print(it, nt)
111
112 # plot the density, |wf|^2
113
114 # Define function to update plot for each frame of the animation
115 def update_plot(frame):
116     plt.cla()
117     plt.xlim([-5.0, 5.0])
118     plt.ylim([0.0, 0.8])
119     xc = xc_list[frame]
120     plt.plot(xj, density_list[frame], label="$|\psi(x)|^2$␣(calc.)")
121     plt.plot(xj, np.exp(-(xj-xc)**2)/np.sqrt(np.pi),
122              label="$|\psi(x)|^2$␣(ref.)", linestyle='dashed')
123     plt.plot(xj, 0.5*(xj-xc)**2,
124          label="Harmonic␣potential", linestyle='dotted')
125
126     plt.xlabel('x')
127     plt.ylabel('Density,␣Potential')
128     plt.legend(loc = 'upper␣right')
129
130
131 # Create the animation
132 fig = plt.figure()
133 ani = animation.FuncAnimation(fig, update_plot, frames=len(density_list), interval=50)
134 #ani.save('wavefunction_animation.gif', writer='imagemagick')
135 ani.save('density_animation.gif', writer='pillow')
```

8.3 Electron dynamics of a one-dimensional hydrogen atom under a laser electric field

As an application of time-propagation calculations using a time-dependent Hamiltonian, let us perform a simulation in which a laser electric field is applied to a one-dimensional hydrogen atom and the dynamics of the driven electron are investigated. As the equation to be solved, we consider the following one-dimensional time-dependent Schrödinger equation.

$$\begin{aligned} i\hbar\frac{\partial}{\partial t}\psi(x,t) &= \left[\hat{H}_0 + \hat{V}(t)\right]\psi(x,t) \\ &= \left[-\frac{\hbar^2}{2m}\frac{\partial^2}{\partial x^2} - \frac{1}{\sqrt{x^2+1}} - eE(t)x\right]\psi(x,t). \end{aligned} \tag{108}$$

Here, the potential, $-1/\sqrt{x^2+1}$, is a one-dimensional potential that mimics the Coulomb potential, with the divergence at the origin removed to avoid computational difficulties. The potential, $-eE(t)x$, is the scalar potential corresponding to a uniform electric field $E(t)$. For convenience, we define the total Hamiltonian by separating it into the unperturbed part $\hat{H}_0$ and the perturbation

$\hat{V}(t)$ as follows.

$$\hat{H}_0 = -\frac{\hbar^2}{2m}\frac{\partial^2}{\partial x^2} - \frac{1}{\sqrt{x^2+1}}, \tag{109}$$

$$\hat{V}(t) = -eE(t)x. \tag{110}$$

In the present calculation, at time $t = 0$ we set the wavefunction $\psi(x, t = 0)$ to be the ground state of the Hamiltonian in the absence of an electric field ($E(t) = 0$), and for $t > 0$ we apply the following oscillating electric field to compute the electron dynamics.

$$E(t) = \begin{cases} E_0 \cos^2\left[\frac{\pi}{T_0}\left(t - \frac{T_0}{2}\right)\right] \sin\left[\omega_0\left(t - \frac{T_0}{2}\right)\right] & 0 < t < T_0 \\ 0 & \text{otherwise} \end{cases} \tag{111}$$

In practical numerical computation, as in Eq. (99), we evaluate the time evolution of the wavefunction by repeatedly applying infinitesimal time steps of size Δt. To facilitate the numerical implementation, we consider approximating this short-time evolution operator as follows.

$$\begin{aligned} \exp\left[-\frac{i}{\hbar}H(t_0)\Delta t\right] &= \exp\left[-\frac{i}{\hbar}\left(\hat{H}_0 + \hat{V}(t_0)\right)\Delta t\right] \\ &= \exp\left[-\frac{i}{\hbar}\hat{V}(t_0)\frac{\Delta t}{2}\right]\exp\left[-\frac{i}{\hbar}\hat{H}_0\Delta t\right]\exp\left[-\frac{i}{\hbar}\hat{V}(t_0)\frac{\Delta t}{2}\right] + \mathcal{O}(\Delta t^3). \end{aligned} \tag{112}$$

This approximation can be verified by Taylor expanding each exponential term. Using this approximation, let us compute the electron dynamics under the electric field given by Equation (111).

`https://github.com/shunsuke-sato/python_qe/blob/develop/note_comp_phys/src/qm_dynamics_hydrogen.py`

Source code 27: Example code for calculating electron dynamics under a laser electric field

```
import numpy as np
import matplotlib.pyplot as plt
import matplotlib.animation as animation
from matplotlib.animation import PillowWriter

# define the potential
def calc_potential(xj):
    vpot = -1.0/np.sqrt(xj**2+1.0)
    return vpot


def calc_static_hamiltonian(num_grid, xj, vpot):

    ham = np.zeros((num_grid, num_grid))
    dx = xj[1]-xj[0]

    for i in range(num_grid):
        for j in range(num_grid):
            if(i == j):
                ham[i,j] = -0.5*(-2.0/dx**2)+vpot[i]
            elif(np.abs(i-j)==1):
                ham[i,j] = -0.5*(1.0/dx**2)


    return ham


def calc_gs_wf(num_grid, ham):

    eigenvalues, eigenvectors = np.linalg.eigh(ham)
    print("gs energy =", eigenvalues[0])

    wf = np.zeros(num_grid, dtype=complex)
    wf.real = eigenvectors[:,0]
```

```
    wf[0] = 0.0
    wf[-1] = 0.0

    return wf


def calc_laser_field(tt):

    omega0 = 0.05
    E0 = 0.1
    tpulse = 10*2.0*np.pi/omega0
    xx = tt-0.5*tpulse

    if(np.abs(xx)/tpulse < 0.5):
        Et = E0*np.cos(np.pi*xx/tpulse)**2*np.sin(omega0*xx)
    else:
        Et = 0.0

    return Et



def ham_wf(wf, vpot, dx):

    num_grid = wf.size
    hwf = np.zeros(num_grid, dtype=complex)

    for i in range(1, num_grid-1):
        hwf[i] = -0.5*(wf[i+1]-2.0*wf[i]+wf[i-1])/(dx**2)

    hwf = hwf + vpot*wf
    return hwf


def time_propagation(xj, wf, vpot, dx, tt, dt):

    Et = calc_laser_field(tt)
    v_Et = -Et*xj

    # apply exp(-0.5*0j*v_Et*dt)
    wf = wf*np.exp(-0.5*1j*v_Et*dt)

    # propagate
    twf = wf


    zfact = 1.0 + 0j
    for iexp in range(1,5):
        zfact = zfact*(-1j*dt)/iexp
        hwf = ham_wf(twf, vpot, dx)
        wf = wf + zfact*hwf
        twf = hwf


    # apply exp(-0.5*0j*v_Et*dt)
    wf = wf*np.exp(-0.5*1j*v_Et*dt)

    return wf

def calc_dipole(xj, wf):

    dx = xj[1]-xj[0]
    dipole = np.sum(np.abs(wf)**2*xj)*dx

    return dipole


# Set the coordinate
xmin = -50.0
xmax = 50.0
num_grid = 500

xj = np.linspace(xmin, xmax, num_grid)
dx = xj[1]-xj[0]

# Time propagation parameters
Tprop = 1300.0 #80.0
dt = 0.05
nt = int(Tprop/dt)+1

```

```
115
116
117
118 # set the potential
119 vpot = calc_potential(xj)
120
121 # set the static Hamiltonian
122 ham = calc_static_hamiltonian(num_grid, xj, vpot)
123
124 # set the initial wavefunction (ground state)
125 wf = calc_gs_wf(num_grid, ham)
126
127 # set output quantities
128 tt_out = np.zeros(nt)
129 Et_out = np.zeros(nt)
130 dipole_out = np.zeros(nt)
131 norm_out = np.zeros(nt)
132
133 # wavefunction array to make a movie
134 wavefunctions = []
135
136 # For loop for the time propagation
137 for it in range(nt):
138     if(it%(nt//100) == 0):
139         print("it=",it,nt)
140         wavefunctions.append(wf.copy())
141
142     tt = dt*it
143
144     # compute outputs
145     tt_out[it] = dt*it
146     Et_out[it] = calc_laser_field(tt)
147     dipole_out[it] = calc_dipole(xj, wf)
148     norm_out[it] = np.sum(np.abs(wf)**2)*dx
149
150     wf = time_propagation(xj, wf, vpot, dx, tt, dt)
151
152
153 plt.figure()
154 plt.plot(tt_out, Et_out, label="E(t)")
155 plt.plot(tt_out, dipole_out, label="d(t)")
156
157 plt.xlabel("t")
158 plt.ylabel("E(t),␣d(t)")
159 plt.legend()
160
161 plt.savefig("dipole_t.png")
162
163
164 # Define function to update plot for each frame of the animation
165 def update_plot(frame):
166     plt.cla()
167     plt.xlim([-20, 20])
168     plt.ylim([0.0, 0.12])
169     plt.plot(xj, np.abs(wavefunctions[frame])**2, label="Density")
170     plt.xlabel('$x$')
171     plt.ylabel('$Density$')
172     plt.legend()
173
174 # Create the animation
175 fig = plt.figure()
176 ani = animation.FuncAnimation(fig, update_plot, frames=len(wavefunctions), interval=50)
177 #ani.save('wavefunction_animation.gif', writer='imagemagick')
178 ani.save('density_animation.gif', writer='pillow')
```

8.4 Absorbing Potential

In the calculation of the previous section, an unphysical phenomenon occurs in which electrons ionized from the atom are reflected at the boundaries of the simulation box. Therefore, to absorb electrons ionized by the laser field, we consider introducing a purely imaginary potential. Let us consider the following complex potential, and impose absorbing boundary conditions such that

the norm of the ionized electrons decreases with time.

$$\hat{V}(t) = \begin{cases} -eE(t)x & |x| < 40 \\ -eE(t)x - i\left(|x| - 40\right) & |x| < 40 \end{cases} \tag{113}$$

By adding this absorbing potential as a perturbation, the ionized electrons are absorbed by the potential, and the influence of unphysical reflections at the simulation-box boundary can be mitigated. In practice, let us run a calculation with the absorbing potential introduced and examine its effect.

`https://github.com/shunsuke-sato/python_qe/blob/develop/note_comp_phys/src/qm_dynamics_hydrogen_abs.py`

Source code 28: Example code for electron dynamics calculation under a laser field

```
import numpy as np
import matplotlib.pyplot as plt
import matplotlib.animation as animation
from matplotlib.animation import PillowWriter

# define the potential
def calc_potential(xj):
    vpot = -1.0/np.sqrt(xj**2+1.0)
    return vpot


def calc_static_hamiltonian(num_grid, xj, vpot):

    ham = np.zeros((num_grid, num_grid))
    dx = xj[1]-xj[0]

    for i in range(num_grid):
        for j in range(num_grid):
            if(i == j):
                ham[i,j] = -0.5*(-2.0/dx**2)+vpot[i]
            elif(np.abs(i-j)==1):
                ham[i,j] = -0.5*(1.0/dx**2)


    return ham


def calc_gs_wf(num_grid, ham):

    eigenvalues, eigenvectors = np.linalg.eigh(ham)
    print("gs energy=", eigenvalues[0])

    wf = np.zeros(num_grid, dtype=complex)
    wf.real = eigenvectors[:,0]
    wf[0] = 0.0
    wf[-1] = 0.0

    return wf


def calc_laser_field(tt):

    omega0 = 0.05
    E0 = 0.1
    tpulse = 10*2.0*np.pi/omega0
    xx = tt-0.5*tpulse

    if(np.abs(xx)/tpulse < 0.5):
        Et = E0*np.cos(np.pi*xx/tpulse)**2*np.sin(omega0*xx)
    else:
        Et = 0.0

    return Et



def ham_wf(wf, vpot, dx):

    num_grid = wf.size
```

```
    hwf = np.zeros(num_grid, dtype=complex)

    for i in range(1, num_grid-1):
        hwf[i] = -0.5*(wf[i+1]-2.0*wf[i]+wf[i-1])/(dx**2)

    hwf = hwf + vpot*wf
    return hwf


def time_propagation(xj, wf, vpot, dx, tt, dt):

    num_grid = xj.size

    Et = calc_laser_field(tt)
    v_Et = -Et*xj

    # set absorbing boundary
    v_abs = np.zeros(num_grid, dtype=complex)
    for i in range(num_grid):
        if(np.abs(xj[i]) > 40.0):
            v_abs[i] = -0.2*1j*(np.abs(xj[i]) - 40.0)



    # apply exp(-0.5*0j*v_Et*dt)
    wf = wf*np.exp(-0.5*1j*(v_Et+v_abs)*dt)

    # propagate
    twf = wf


    zfact = 1.0 + 0j
    for iexp in range(1,5):
        zfact = zfact*(-1j*dt)/iexp
        hwf = ham_wf(twf, vpot, dx)
        wf = wf + zfact*hwf
        twf = hwf


    # apply exp(-0.5*0j*v_Et*dt)
    wf = wf*np.exp(-0.5*1j*(v_Et+v_abs)*dt)

    return wf

def calc_dipole(xj, wf):

    dx = xj[1]-xj[0]
    dipole = np.sum(np.abs(wf)**2*xj)*dx

    return dipole


# Set the coordinate
xmin = -50.0
xmax = 50.0
num_grid = 500

xj = np.linspace(xmin, xmax, num_grid)
dx = xj[1]-xj[0]

# Time propagation parameters
Tprop = 1300.0 #80.0
dt = 0.05
nt = int(Tprop/dt)+1




# set the potential
vpot = calc_potential(xj)

# set the static Hamiltonian
ham = calc_static_hamiltonian(num_grid, xj, vpot)

# set the initial wavefunction (ground state)
wf = calc_gs_wf(num_grid, ham)

# set output quantities
tt_out = np.zeros(nt)
Et_out = np.zeros(nt)
```

```
dipole_out = np.zeros(nt)
norm_out = np.zeros(nt)

# wavefunction array to make a movie
wavefunctions = []

# For loop for the time propagation
for it in range(nt):
    if(it%(nt//100) == 0):
        print("it=",it,nt)
        wavefunctions.append(wf.copy())

    tt = dt*it

    # compute outputs
    tt_out[it] = dt*it
    Et_out[it] = calc_laser_field(tt)
    dipole_out[it] = calc_dipole(xj, wf)
    norm_out[it] = np.sum(np.abs(wf)**2)*dx

    wf = time_propagation(xj, wf, vpot, dx, tt, dt)


plt.figure()
plt.plot(tt_out, Et_out, label="E(t)")
plt.plot(tt_out, dipole_out, label="d(t)")

plt.xlabel("t")
plt.ylabel("E(t), d(t)")
plt.legend()

plt.savefig("dipole_t_abs.png")


# Define function to update plot for each frame of the animation
def update_plot(frame):
    plt.cla()
    plt.xlim([-20, 20])
    plt.ylim([0.0, 0.12])
    plt.plot(xj, np.abs(wavefunctions[frame])**2, label="Density")
    plt.xlabel('$x$')
    plt.ylabel('$Density$')
    plt.legend()

# Create the animation
fig = plt.figure()
ani = animation.FuncAnimation(fig, update_plot, frames=len(wavefunctions), interval=50)
#ani.save('wavefunction_animation.gif', writer='imagemagick')
ani.save('density_animation_abs.gif', writer='pillow')
```

8.5 Analysis of High-Order Harmonic Generation

In the above simulation, we investigated the electron dynamics driven when a high-intensity laser pulse is irradiated onto an atom. As a result of that analysis, we can compute the electric dipole moment $d(t)$ induced by the optical field (in this case equivalent to the expectation value of position) as a function of time. Moreover, since a charged particle radiates electromagnetic waves when undergoing accelerated motion, by examining the frequency components of the acceleration of the charged particle, we can calculate the spectrum of the light emitted from the optically driven electronic system. Specifically, it is convenient to examine the Fourier transform of the dipole moment as follows.

$$\tilde{d}(\omega) = \int_{-\infty}^{\infty} dt e^{i\omega t} d(t). \tag{114}$$

Here, noting that the electron acceleration $a(t)$ is proportional to the second time derivative of the dipole moment, the spectrum of the light emitted from the optically driven electronic system is expressed as follows.

$$I(\omega) \sim \omega^2 \left| \tilde{d}(\omega) \right|^2. \tag{115}$$

Since, in an actual simulation, we cannot treat infinitely long times, we will perform the Fourier transform over a finite time interval instead of Eq. (114).

$$\tilde{d}(\omega) = \int_0^{T_{\rm sim}} dt e^{i\omega t} d(t) W(t). \tag{116}$$

Here, $T_{\rm sim}$ is the simulation time, and the function $W(t)$ is a window function that smoothly goes to zero at $t = T_{\rm sim}$. By introducing such a window function, we can reduce the noise that appears in the Fourier transform over a finite time.

Append the analysis code using the Fourier transform described above to the end of the electron dynamics calculation created up to the previous section, and actually compute the spectrum of the light generated from the optically excited atom.

`https://github.com/shunsuke-sato/python_qe/blob/develop/note_comp_phys/src/qm_dynamics_hydrogen_abs_fft.py`

Source code 29: Example code for electron dynamics under a laser field and analysis of high-order harmonic spectra

```
import numpy as np
import matplotlib.pyplot as plt
import matplotlib.animation as animation
from matplotlib.animation import PillowWriter

# define the potential
def calc_potential(xj):
    vpot = -1.0/np.sqrt(xj**2+1.0)
    return vpot


def calc_static_hamiltonian(num_grid, xj, vpot):

    ham = np.zeros((num_grid, num_grid))
    dx = xj[1]-xj[0]

    for i in range(num_grid):
        for j in range(num_grid):
            if(i == j):
                ham[i,j] = -0.5*(-2.0/dx**2)+vpot[i]
            elif(np.abs(i-j)==1):
                ham[i,j] = -0.5*(1.0/dx**2)


    return ham


def calc_gs_wf(num_grid, ham):

    eigenvalues, eigenvectors = np.linalg.eigh(ham)
    print("gs energy =", eigenvalues[0])

    wf = np.zeros(num_grid, dtype=complex)
    wf.real = eigenvectors[:,0]
    wf[0] = 0.0
    wf[-1] = 0.0

    return wf


def calc_laser_field(tt):

    omega0 = 0.05
    E0 = 0.1
    tpulse = 10*2.0*np.pi/omega0
    xx = tt-0.5*tpulse

    if(np.abs(xx)/tpulse < 0.5):
        Et = E0*np.cos(np.pi*xx/tpulse)**2*np.sin(omega0*xx)
    else:
        Et = 0.0

    return Et
```

```
54
55
56
57  def ham_wf(wf, vpot, dx):
58
59      num_grid = wf.size
60      hwf = np.zeros(num_grid, dtype=complex)
61
62      for i in range(1, num_grid-1):
63          hwf[i] = -0.5*(wf[i+1]-2.0*wf[i]+wf[i-1])/(dx**2)
64
65      hwf = hwf + vpot*wf
66      return hwf
67
68
69  def time_propagation(xj, wf, vpot, dx, tt, dt):
70
71      num_grid = xj.size
72
73      Et = calc_laser_field(tt)
74      v_Et = -Et*xj
75
76      # set absorbing boundary
77      v_abs = np.zeros(num_grid, dtype=complex)
78      for i in range(num_grid):
79          if(np.abs(xj[i]) > 40.0):
80              v_abs[i] = -0.2*1j*(np.abs(xj[i]) - 40.0)
81
82
83
84      # apply exp(-0.5*0j*v_Et*dt)
85      wf = wf*np.exp(-0.5*1j*(v_Et+v_abs)*dt)
86
87      # propagate
88      twf = wf
89
90
91      zfact = 1.0 + 0j
92      for iexp in range(1,5):
93          zfact = zfact*(-1j*dt)/iexp
94          hwf = ham_wf(twf, vpot, dx)
95          wf = wf + zfact*hwf
96          twf = hwf
97
98
99      # apply exp(-0.5*0j*v_Et*dt)
100     wf = wf*np.exp(-0.5*1j*(v_Et+v_abs)*dt)
101
102     return wf
103
104 def calc_dipole(xj, wf):
105
106     dx = xj[1]-xj[0]
107     dipole = np.sum(np.abs(wf)**2*xj)*dx
108
109     return dipole
110
111
112 # Set the coordinate
113 xmin = -50.0
114 xmax = 50.0
115 num_grid = 500
116
117 xj = np.linspace(xmin, xmax, num_grid)
118 dx = xj[1]-xj[0]
119
120 # Time propagation parameters
121 Tprop = 1300.0 #80.0
122 dt = 0.05
123 nt = int(Tprop/dt)+1
124
125
126
127
128 # set the potential
129 vpot = calc_potential(xj)
130
131 # set the static Hamiltonian
132 ham = calc_static_hamiltonian(num_grid, xj, vpot)
133
```

```
# set the initial wavefunction (ground state)
wf = calc_gs_wf(num_grid, ham)

# set output quantities
tt_out = np.zeros(nt)
Et_out = np.zeros(nt)
dipole_out = np.zeros(nt)
norm_out = np.zeros(nt)

# wavefunction array to make a movie
wavefunctions = []

# For loop for the time propagation
for it in range(nt):
    if(it%(nt//100) == 0):
        print("it=",it,nt)
        wavefunctions.append(wf.copy())

    tt = dt*it

    # compute outputs
    tt_out[it] = dt*it
    Et_out[it] = calc_laser_field(tt)
    dipole_out[it] = calc_dipole(xj, wf)
    norm_out[it] = np.sum(np.abs(wf)**2)*dx

    wf = time_propagation(xj, wf, vpot, dx, tt, dt)


plt.figure()
plt.plot(tt_out, Et_out, label="E(t)")
plt.plot(tt_out, dipole_out, label="d(t)")

plt.xlabel("t")
plt.ylabel("E(t), d(t)")
plt.legend()

plt.savefig("dipole_t_abs.png")


# Define function to update plot for each frame of the animation
def update_plot(frame):
    plt.cla()
    plt.xlim([-20, 20])
    plt.ylim([0.0, 0.12])
    plt.plot(xj, np.abs(wavefunctions[frame])**2, label="Density")
    plt.xlabel('$x$')
    plt.ylabel('$Density$')
    plt.legend()

# Create the animation
fig = plt.figure()
ani = animation.FuncAnimation(fig, update_plot, frames=len(wavefunctions), interval=50)
#ani.save('wavefunction_animation.gif', writer='imagemagick')
ani.save('density_animation_abs.gif', writer='pillow')


###  Analysis of HHG ###
def apply_envelope(nt, dt, ft):

    omega0 = 0.05
    tpulse = 10*2.0*np.pi/omega0
    ft_env = np.zeros(nt)

    for it in range(nt):
        tt = it*dt
        xx = tt-0.5*tpulse
        if(np.abs(xx)/tpulse < 0.5):
          ft_env[it] = ft[it]*np.cos(np.pi*xx/tpulse)**2
    return ft_env

# Apply envelope function
Et_env = apply_envelope(nt, dt, Et_out)
dipole_env = apply_envelope(nt, dt, dipole_out)

# Apply Fourier transform
Ew_out = np.fft.fft(Et_env)
spec_Ew = np.abs(Ew_out)**2

dipole_w_out = np.fft.fft(dipole_env)
```

```
spec_dipole = np.abs(dipole_w_out)**2

# Compute the frequency
omega = np.fft.fftfreq(nt, d=dt)*(2.0*np.pi)


# Figure 1
plt.figure()
plt.plot(omega, omega**2*spec_Ew, label="$E(\omega)$")
plt.plot(omega, omega**2*spec_dipole, label="$d(\omega)$")

plt.xlabel("$\omega$")
plt.ylabel("$|E(\omega)|^2$, $|d(\omega)|^2$")

plt.xlim(0,6)
plt.ylim(1e-10,1e4)
plt.yscale("log")
plt.legend()
plt.savefig("hhg_spec_1.png")

# Figure 2
omega0 = 0.05
plt.figure()
plt.plot(omega/omega0, omega**2*spec_Ew, label="$E(\omega)$")
plt.plot(omega/omega0, omega**2*spec_dipole, label="$d(\omega)$")

plt.xlabel("$\omega$/$\omega_0$")
plt.ylabel("$|E(\omega)|^2$, $|d(\omega)|^2$")

plt.xticks(np.arange(1,22,2))
plt.xlim(0,22)
plt.ylim(1e-10,1e4)
plt.yscale("log")
plt.grid(axis='x')
plt.legend()

plt.savefig("hhg_spec_2.png")
```

In high-order harmonic generation that arises when a high-intensity laser such as the one above is irradiated onto an atom, it is known that when the energy of the emitted light exceeds a certain threshold, the intensity drops sharply. This energy is called the cutoff energy, and at the time when high-order harmonic generation was first observed, the mechanism that determines this cutoff energy was a major mystery. Subsequently, Paul Corkum proposed a semiclassical three-step model and clarified that the cutoff energy is given by the following expression.

$$U_{\text{cutoff}} = I_p + 3.17U_p. \tag{117}$$

Here, I_p is the ionization potential, and U_p is a quantity called the ponderomotive energy, representing the average kinetic energy of a charged particle in an oscillating electric field. U_p is defined by the following formula.

$$U_p = \frac{e^2E_0^2}{4m_e\omega_0^2}. \tag{118}$$

To understand this equation, let us consider Newton's equation for an electron in an oscillating electric field.

$$m_e\frac{d}{dt}v(t) = -eE_0\cos(\omega_0 t). \tag{119}$$

Therefore, the electron velocity under an oscillating field is given by

$$v(t) = -\frac{e}{m_e}E_0\sin(\omega_0 t). \tag{120}$$

The time average of the kinetic energy (the average over one period T of the field oscillation) is then given by

$$< E_k >= \frac{m_e}{2}\frac{1}{T}\int_0^T dtv^2(t) = \frac{m_e}{2}\frac{1}{T}\int_0^T dt\frac{e^2}{m_e^2}E_0^2\sin^2(\omega_0 t) = \frac{E_0^2}{4m_e\omega_0^2} = U_p. \tag{121}$$

In the above simulations, we set $E_0 = 0.1$, $\omega_0 = 0.05$, and $e = m_e = 1$, and the ionization potential of the model atom is $I_p = 0.67$. Therefore, the value of the cutoff energy predicted by the three-step model is $U_{\text{cutoff}} = I_p + 3.17U_p \approx 3.84$. Let us compare this value with the spectrum obtained in the calculation above.