Assignment

Exercise sheet 7

Name TN 1: Tim Peinkofer

tim.peinkofer@student.uni-tuebingen.de

Name TN 2: Fabian Kostow

fabian.kostow@student.uni-tuebingen.de

Tutor: Jose Carlos Olvera Meneses

Date: December 11, 2024

Contents

1 Quantum Mechanics Application

1.1 Preparation

For our application, we first need to write a function that fills our array with the analytical function. We will need this later for our integration routine. Let code we will use is the following:

```
def array_filling(h,a,b,n,f, array):

    if n != len(array):
        raise ValueError("Please choose matching values for
                                        calculation") #Compare
                                         the size of the array
                                         and the Number of
                                        data points

    for i in range(n): # Calculate the value for every
                                       datapoint until the upper
                                       bound is reached
        x = a +i*h

        if x > b:
            break

        array[i] = f(x)

    return array

#Testing case, will be deleted in the Application part
def func(x):
    return x

h = 1/100
a= [0]*100
result = array_filling(h,0,1,100,func,a)
print(f"Result array: {result}")
```

We created a test case at the bottom of our code to check the work of our function.

In the next step we want to integrate a function via Newton-Cotes 2nd Order polynomial. The integration via Newton Cotes 2nd works as follows:

The second order Newton Cotes polynomial is defined the following:

$$P_2(x_s) = f_0 + s\triangle f_0 + \frac{s(s-1)}{2}\triangle^2 f_0$$

where $x_s = x_0 + sh$. If we integrate this formula based on our given bounds we get:

$$\int_{x_0}^{x_1} P_2(x_s)\, dx = \frac{h}{3}(f_0 + 4f_1 + f_2)$$

We will now use this equation to calculate the integrals in the sections below. The code looks as follows:

```
import numpy as np


def Newton_cotes(n,x, f):

    h = (x[-1] - x[0]) / (n - 1)  # Calculating the stepsize
                                    h
    sum_integral = 0

    for i in range(0, n - 2, 2):  # Calculating the value for
                                    every odd step
        sum_1 = f[i] + 4 * f[i + 1] + f[i + 2]
        sum_integral += h/3 * sum_1

    return sum_integral


# Testfunction
def func(x):
    return x

n_1 = 1000
n_2 = 2000

x = np.linspace(0, 2, n_1)
f = [func(x_i) for x_i in x]


x_2 = np.linspace(0, 2, n_2)
f_2 = [func(x_i) for x_i in x_2]
```

```
# Integration for two different n to get the error of our
                                integration
result_1 = Newton_cotes(n_1,x, f)
result_2 = Newton_cotes(n_2,x_2, f_2)
err = np.abs(result_1 - result_2)

# Ergebnisse ausgeben
print(f"Result of our Integration: {result_1}")
print(f"Error of our Integration: {err}")
```

1.2 Application and Testing

1.2.1 $f(x) = x + x^3$

First we want to test our code from the section before for the function $f(x) = x + x^3$. We will use the array filling function and Newton Cotes 2nd order for $n = 5$ data points. The bounds are 0 and 1:

```
def func(x): # function need to evaluate
    return x+x**3

def array_filling(h,a,b,n,f, array):

    if n != len(array):
        raise ValueError("Please choose matching values for
                                        calculation") #Compare
                                         the size of the array
                                         and the Number of
                                        data points

    for i in range(n): # Calculate the value for every
                                       datapoint until the upper
                                       bound is reached
        x = a +i*h

        if x > b:
            break

        array[i] = f(x)

    return array


def Newton_cotes(n,x, f):

    h = (x[-1] - x[0]) / (n - 1)  # Calculating the stepsize
                                   h
    sum_integral = 0

    for i in range(0, n - 2, 2):  # Calculating the value for
                                    every odd step
        sum_1 = f[i] + 4 * f[i + 1] + f[i + 2]
        sum_integral += h/3 * sum_1

    return sum_integral
```

```
n = 5
h = 1/4
a = [0]*n
x = [0,1]

array = array_filling(h,0,1,n,func,a)

result = Newton_cotes(5,x,array)
print(f"Result of the Integration: {result}")
```

The result of our calculation is 0.75, which is the exact value of the Integral $\int_0^1 x+x^3 dx$. But why do we get the exact value? As we see in the preparation section before, the error of our calculation was about $\sim 10^{-3}$. The exact value just has to decimals, so the error does not affect our calculation for these two decimals and we get the exact value up to rounding errors.

1.2.2 Wavefunction

In the following step we want integrate the probability of a particle wave function. The equations are the following:

$$\Psi(x,t) = \begin{cases} \frac{1}{\sqrt{L}}[\sin\left(\frac{\pi x}{L}\right) e^{-i\omega_1 t} + \sin\left(\frac{2\pi x}{L}\right) e^{-i\omega_2 t}] & \text{if } 0 < x < L, \\ 0 & \text{otherwise} \end{cases}$$

$$P(x,t) = |\Psi(x,t)|^2$$

We will use the following code for our log(h)-log(E) plots:

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

# Constants
L = 2
w_1 = 3
w_2 = 4.5

# Define the wavefunction
def wave_func(x, t):
    t_1 = 1 / np.sqrt(L) * np.sin(np.pi * x / L) * np.exp(-1j
                                        * w_1 * t)
    t_2 = 1 / np.sqrt(L) * np.sin(2 * np.pi * x / L) * np.exp
                                       (-1j * w_2 * t)
    return t_1 + t_2 if x < L else 0

# Define the probability density
def probability(x, t):
    return np.abs(wave_func(x, t)) ** 2

# Filling an array with the analytical function
def array_filling(h, a, b, func, array, n, t):
    if n != len(array):
        raise ValueError("Please choose matching values for
                                           calculation")  #
                                           Compare the size of
                                           the array and the
                                           number of data points
```

```
    for i in range(n):  # Calculate the value for every data
                                point until the upper
                                bound is reached
        x = a + i * h
        if x > b:
            break
        array[i] = func(x, t)

    return array

# Newton-Cotes function
def Newton_cotes(n, a,b, f):
    h = (b - a) / (n - 1)  # Calculating the step size h
    sum_integral = 0

    for i in range(0, n - 2, 2):  # Calculating the value for
                                    every odd step
        sum_1 = f[i] + 4 * f[i + 1] + f[i + 2]
        sum_integral += h / 3 * sum_1

    return sum_integral

interval_a, interval_b = 3 * L / 4, L  # Bounds
time_steps = [0, 2 * np.pi / (w_2 - w_1)]  # Set time values

for t in time_steps:# Calculate the values for every time

    results = []
    arr_c = [0] * 10000
    h = (interval_b - interval_a) / (10000 - 1)
    check_arr = array_filling(h, interval_a, interval_b,
                                probability, arr_c, 10000,
                                 t) #Calculate comparison
                                values for error
                                calculation
    check_int = Newton_cotes(10000, interval_a, interval_b,
                                check_arr)
    print(f"t = {t}")
    print("\n")

    for n in range(5, 501, 2):  # Calculate values for every
                                odd n
        num_points = n
        h = (interval_b - interval_a) / (num_points - 1)  #
                                    Compute step size h
        arr = [0] * num_points
```

```
        # Fill array with analytical function
        array_filled = array_filling(h, interval_a,
                                     interval_b,
                                     probability, arr, n, t
                                     )

        # Integrate using Newton Cotes
        integral_result = Newton_cotes(n, interval_a,
                                     interval_b,
                                     array_filled)
        error = np.abs(integral_result - check_int)

        results.append((n, np.log(h), np.log(error)))
        print(f"n: {n}, log(h): {np.log(h):.5f}, log(E): {np.
                                     log(error):.5f}")

    print("\n")

    result_array = np.array(results) #Plot results for log(h)
                                    and log(E)
    plt.figure(figsize=(10, 6))
    plt.plot(result_array[:,1], result_array[:,2])
    plt.xlabel("log(h)")
    plt.ylabel("log(E)")
    plt.title(f"Convergence of Newton-Cotes 2nd order for t =
                                    {t:.5f} s")
    plt.grid()
    plt.show()
```

As a result we get the following plots:

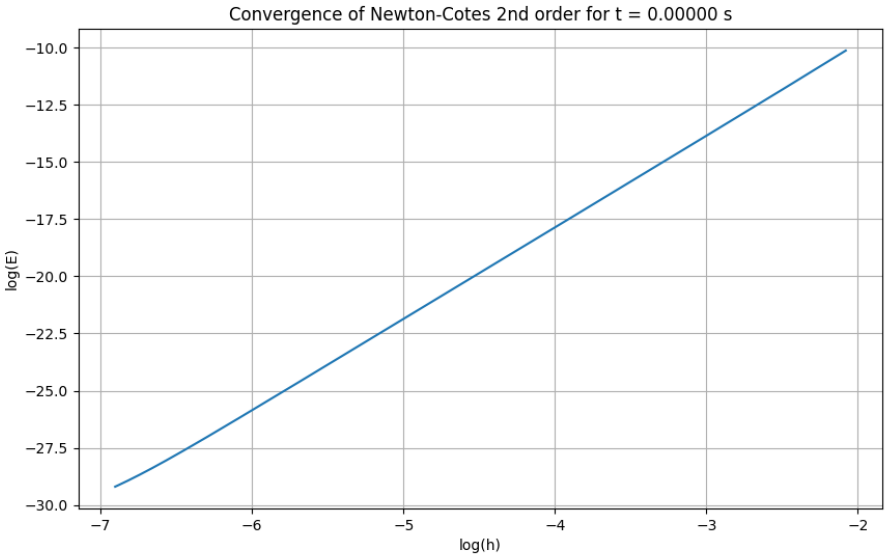

Figure 1: Error plot for $t = 0$

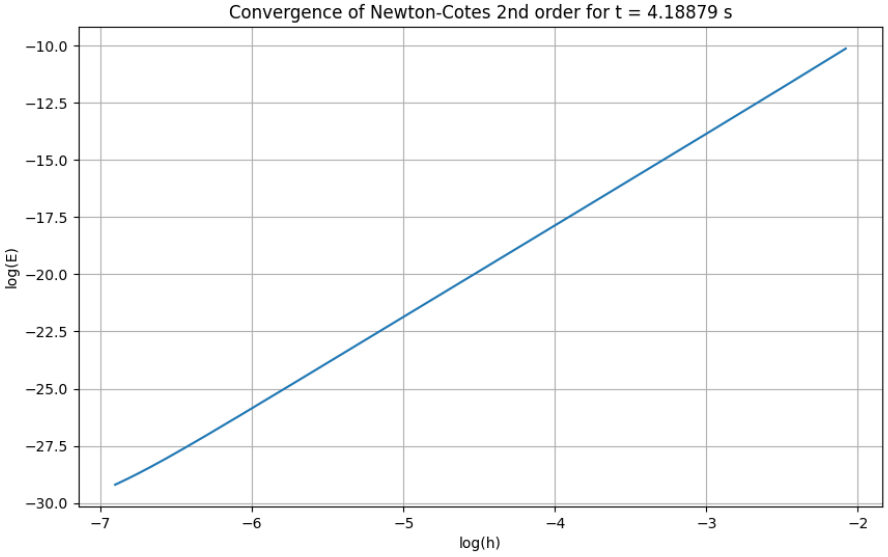

Figure 2: Error plot for $t = \frac{\pi}{\triangle\omega}$

This is the result we expected, because the Error gets decreases if the the step size gets smaller. We got this typical convergence behavior for both plots. As a conclusion we can say that we get the expected convergence rate, which is linear for our log(h)-log(E) diagram.

1.3 Extra Application - 2s state of hydrogen atom

In the extra application task we need to calculate the mean radius of a 2s state hydrogen atom. We mean of a value is normally defined as:

$$\int_0^\infty xp(x)\,dx$$

If we plug in our wave function given in the exercise and substitute our variable, we get:

$$\langle r^n \rangle = 4\pi \int_0^\infty \Psi^*(x,t)\Psi(x,t) \cdot r^{n+2}\,dr \tag{1}$$

We can now use the following code to calculate mean value of r and it's standard deviation:

```
import numpy as np

# Constants
a_0 = 0.0529e-9  # First Bohr radius

# Wave function
def wave_func(r):
    normalization_factor = 1 / (4 * np.sqrt(2 * np.pi * a_0**
                                3))
    psi_value = normalization_factor * (2 - r / a_0) * np.exp
                                (-r / (2 * a_0))
    return np.float64(psi_value)

# Probability function
def prob_density(r, n):
    psi_val = wave_func(r)
    return np.abs(psi_val)**2 * r**n * r**2 # The function on
                                 the excercise sheet was
                                wrong. There must be
                                another r^2
```

```
# Newton Cortes function
def newton_cotes(x, f_vals):

    h = (x[-1] - x[0]) / (len(x) - 1) #Calculate step size
    integral_sum = 0.0


    for i in range(0, len(x) - 2, 2):  # calculate the
                                       integral
        integral_sum += (f_vals[i] + 4 * f_vals[i + 1] +
                                           f_vals[i + 2])

    return (h / 3) * integral_sum



interval_a = 1e-10  # Avoid singularity at r=0
upper_bounds = [10 * a_0, 15 * a_0, 40 * a_0, 60*a_0,1000*a_0
                                  ]  # Define upper bounds, to
                                  get the value for b ->
                                  infinite

results = []
for t in upper_bounds:

    n_points = 10000  # Number of points
    r_values = np.linspace(interval_a, t, n_points)

    integrand_values = prob_density(r_values, 1)  # Calculate
                                       mean radius
    integrand_values_2 = prob_density(r_values, 2)  #
                                       Calculate standard
                                       deviation


    integral_result = 4 * np.pi * newton_cotes(r_values,
                                       integrand_values)
    integral_result_2 = 4 * np.pi * newton_cotes(r_values,
                                       integrand_values_2)
    devitation = np.sqrt(np.sqrt(integral_result_2))
    results.append((t, integral_result, devitation))
    print(f"Upper bound: {t/a_0} * a_0, Integral value: {
                                       integral_result},
                                       Devitation: {devitation}")
```

We have used different upper bounds to simulate that the upper bound goes to infinity. As a result we get:

Upper bound	**Integral value**	**Deviation**
$10 \cdot a_0$	$2.78691 \times 10^{-10}\, m$	$1.75516 \times 10^{-5}\, m$
$15 \cdot a_0$	$3.13302 \times 10^{-10}\, m$	$1.84519 \times 10^{-5}\, m$
$40 \cdot a_0$	$3.15009 \times 10^{-10}\, m$	$1.85106 \times 10^{-5}\, m$
$60 \cdot a_0$	$3.15009 \times 10^{-10}\, m$	$1.85106 \times 10^{-5}\, m$
$1000 \cdot a_0$	$3.15009 \times 10^{-10}\, m$	$1.85106 \times 10^{-5}\, m$

As we see, the integral converges against the mean value $\langle r \rangle = 3.15009 \times 10^{-10}\, m$. The suggested value would be $\langle r \rangle = 6a_0 = 3.17506 \cdot 10^{-10}\, m$. As we see our calculated value is near the real value of $\langle r \rangle$.

2 Mode Decomposition Application

Here we only discuss parts of the code, the full code is on Github in the file ProblemV_IntegrationTest

2.1 Trapezoidal Rule

The first method we want to discuss is the trapezoidal rule, the Python codes witch integrates a function f over the interval [a,b] with known points n given by

```
def Trapezoidal(a,b,n,f):
    x=np.linspace(a,b,n+1)
    m=0
    h=(b-a)/n
    y=f(x)

    for i in range(1,n):
        m += y[i]

    I=h/2*(y[0]+2*m+y[-1])
    return I
```

most of the code is given by writing the following formula in code

$$I = \frac{h}{2} \cdot (y(x_0) + y(x_n) + \sum_{i=1}^{n-1} y(x_i)) + O(h^2)$$

2.2 Simpson 3/8 Rule

The following code gives us the Simpson 3/8 rule

```
def Simpson(a,b,n,f):

    #divide the interval n into two, where n1 is multible of
                                        3
    n1 = (n // 3) * 3
    n2 = n - n1

    x=np.linspace(a,b,n+1)
    m1,m2,m3=0,0,0
    y=f(x)

    #h for the intervall n1
    h1 = (x[n1] - a) / n1
    #h for the intervall n2
    if n2 > 0:
        h2 = (b - x[n1]) / n2
    else:
        h2 =0

    #calculate m1
    for i in range(1,n1,3):
        m1 += y[i]
    #calculate m2
    for i in range(2,n1,3):
        m2 += y[i]
    #calculate m3
    for i in range(3,n1,3):
        m3 += y[i]

    I_Simpson=(3*h1/8)*(y[0]+ 3*m1 + 3*m2 + 2*m3 +y[n1])

    # calculate the rest with trapezoidal rule
    if n2 > 0:
        I_Trapezoid = (h2 / 2) * (y[n1] + y[-1])
    else:
         I_Trapezoid = 0

    return I_Simpson + I_Trapezoid
```

Here is a little more to do, first the core of the code is also given by a relatively simple formula

$$I = \frac{3h}{8}\left[f(x_0) + 3\sum_{i=1,3,\ldots}^{n-2} f(x_i) + 3\sum_{i=2,4,\ldots}^{n-1} f(x_i) + 2\sum_{i=3,6,\ldots}^{n-3} f(x_i) + f(x_n)\right] + O(h^4)$$

where the sums are calculated by $m_{1,2,3}$ in the code.
The problem is that this only works if n is a multiple of 3. To make this code work for all n, we take the largest value n_1 such that n_1 is a multiple of 3 if there are rest terms n_2 we calculate these values by the trapezoidal rule. By calculating the rest values, we have a maximum of two values, so the additional trapezoidal is even easier to calculate in this case.

2.3 Testing

Now we want to test this method on three functions

$$f_1(x) = e^x \cos(x), \quad \text{in } I = [0, \frac{\pi}{2}]$$

$$f_2(x) = e^x, \quad \text{in } I = [-1, 3]$$

$$f_3(x) = \begin{cases} e^{2x}, & x < 0, \\ x - 2\cos(x) + 4, & x \geq 0, \end{cases} \quad \text{in } I = [-1, 1]$$

to calculate the error, therefore the difference between analytical and numerical value, the following code is given.

```
def ErrorTrap (a,b,f,F):
    Error = []
    hs= []
    numerical_value = 0
    Error_Calculation = 0
    for i in range(5,501):
        numerical_value = Trapezoidal(a,b,i,f)
        Error_Calculation = (F(b) - F(a)) - numerical_value
        Error.append(Error_Calculation)
        h= (b-a)/i
        hs.append(h)
    return Error,hs

def ErrorSimpson (a,b,f,F):
    Error = []
    hs= []
    numerical_value = 0
    Error_Calculation = 0
    for i in range(5,501):
        numerical_value = Simpson(a,b,i,f)
        Error_Calculation = (F(b) - F(a)) - numerical_value
        Error.append(Error_Calculation)
        h= (b-a)/i
        hs.append(h)
    return Error,hs
```

the Functions to calculating the error are mostly equal, the only difference is that they call there corresponding function (Simpson or trapezoidal)

To analyze the error for different h we look at it on a logarithmic scale. First, we want to start with the errors from the trapezoidal rule

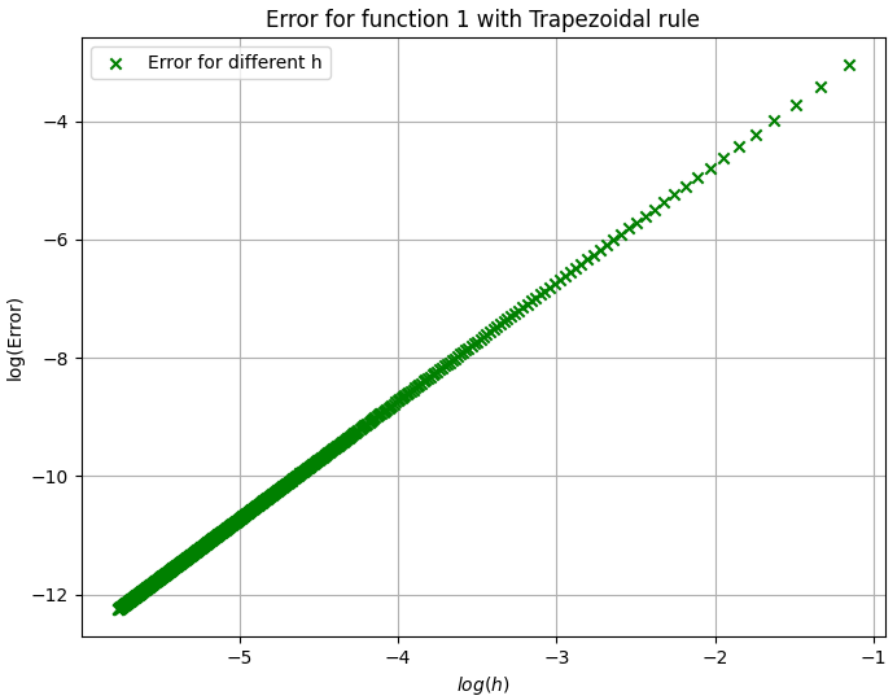

Figure 3: logarithmic Error plot for f_1 with Trapezoidal rule

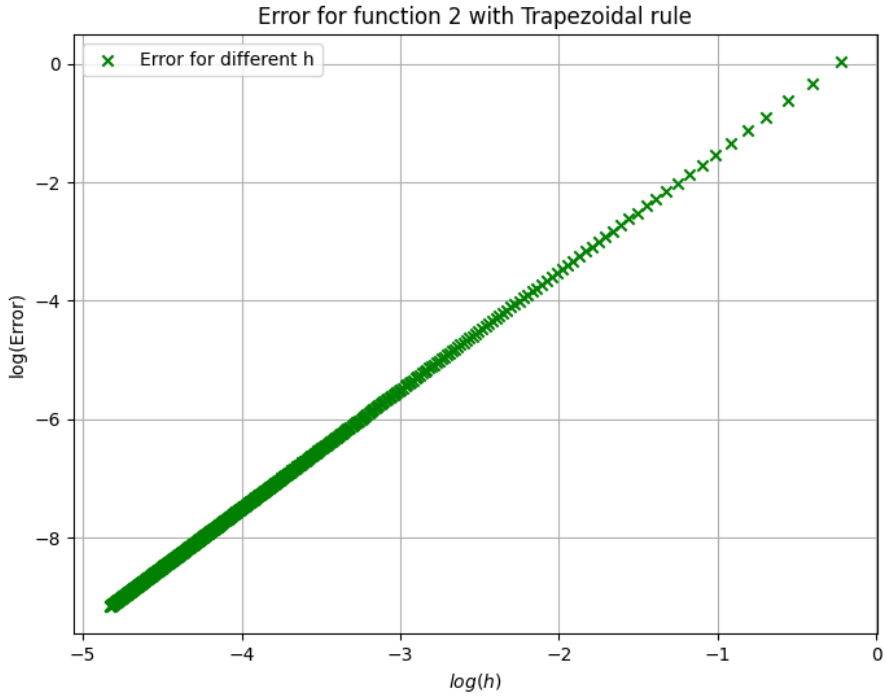

Figure 4: logarithmic plot for f_2 with Trapezoidal rule

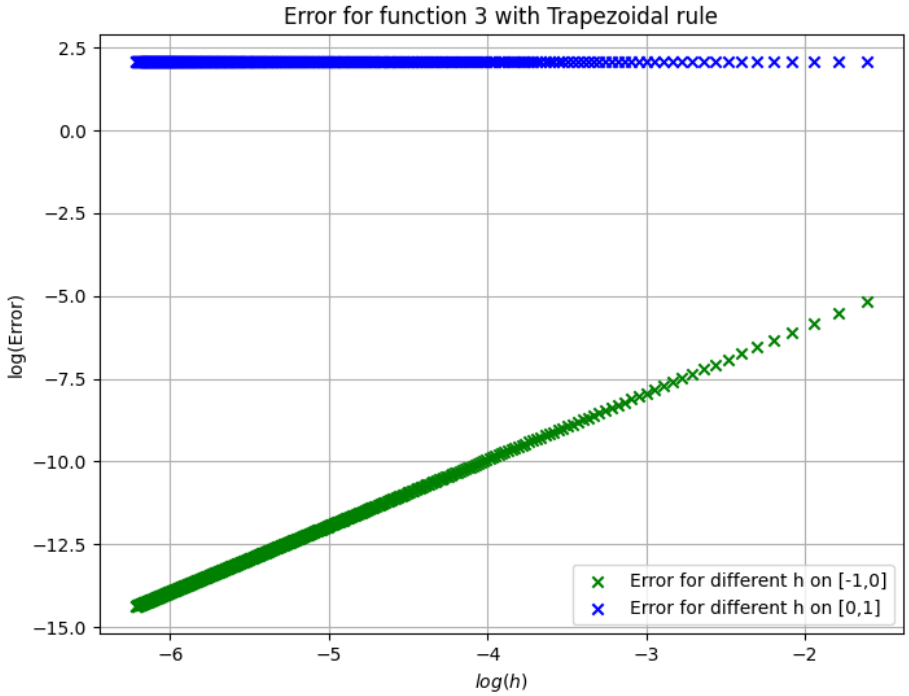

Figure 5: logarithmic plot for f_3 with Trapezoidal rule

We can see that for all "green" graphs the slope is around 2 which represents that the error is quadratic which confirms the assumption.
For the "blue" graph we see that the slope of the error is really small, therefore the method works well for this part of the function.

Now we want to do the same with Simpson Rule.

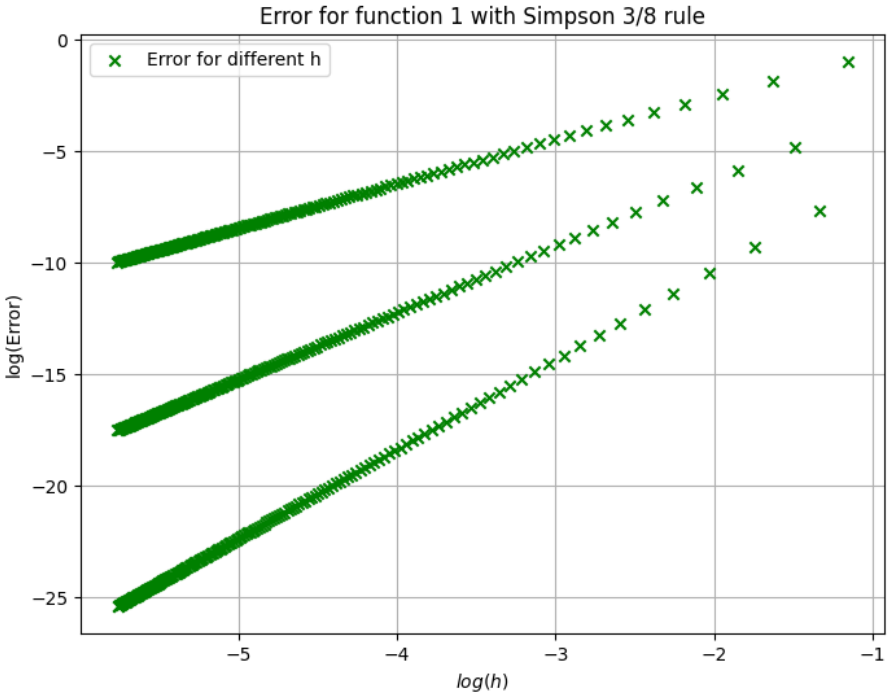

Figure 6: logarithmic Error plot for f_1 with Simpson rule

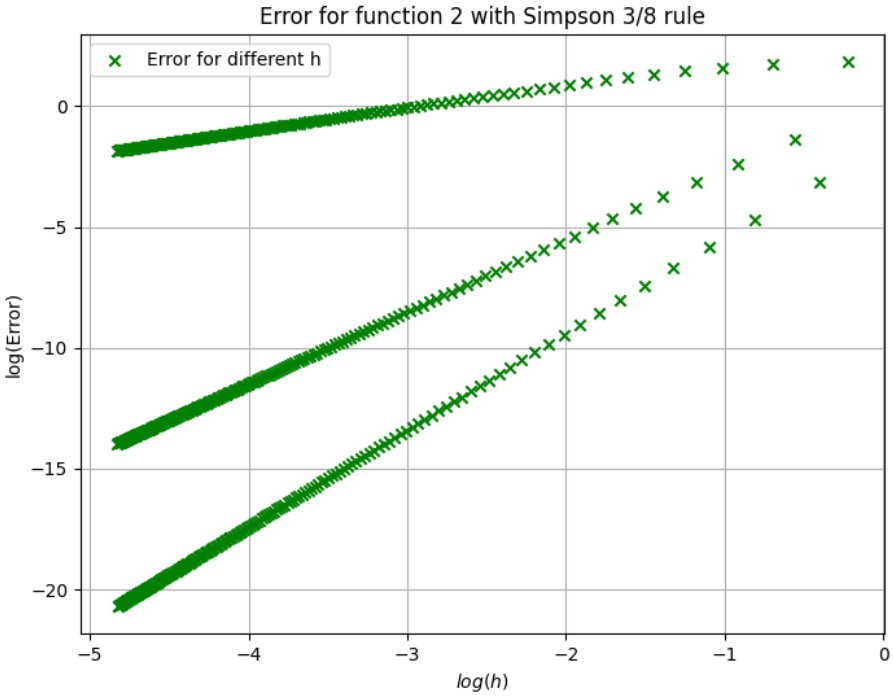

Figure 7: logarithmic plot for f_2 with Simpson rule

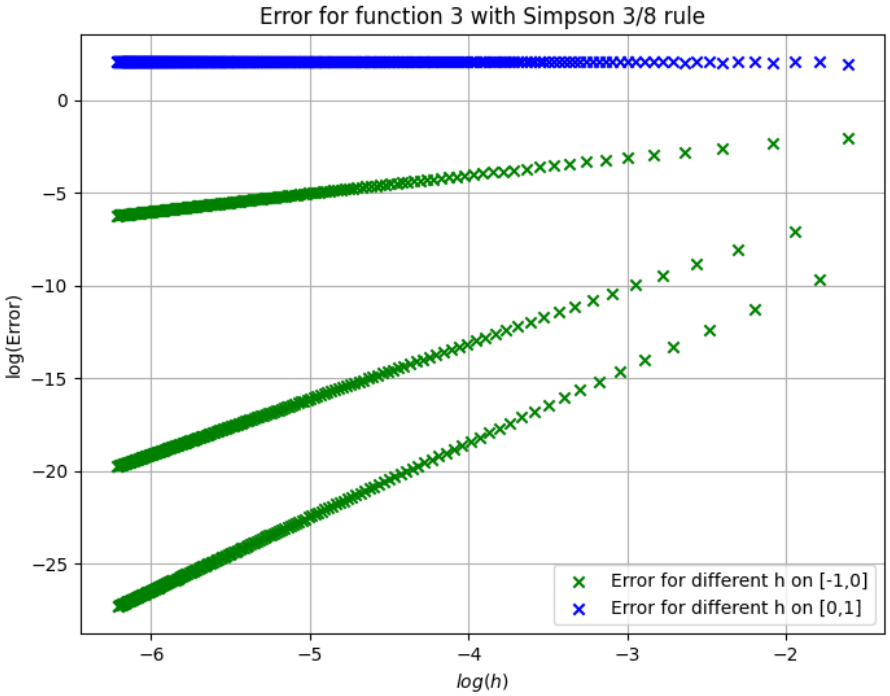

Figure 8: logarithmic plot for f_3 with Simpson rule

The first thing we see is that for the "green" graph there are 3 different lines, this is because we are not only using Simpson rule.The graph with the highest slope is the graph for the Simpson rule becouse Simpson has an error h^4 due to the logarithmic scale the slope is 4, we also see that the graph in the middle has a slope of 2 which represents the trapezoidal addition. The last green graph comes also from the Trapezoidal rule but if we only know 1 more point.
We also see here that the blue graph error is small because we only see one graph.