PHY407 Formal Lab Report 2

Natalia Tabja

October 2024

1 Question 1: Calculating Potential Energy of QM Harmonic Oscillator

1.1 (a) Function to calculate $H_n(x)$

We define a recursive function to compute the Hermite polynomials $H_n(x)$.

```
import numpy as np
from matplotlib import pyplot as plt
import math
from pylab import *

def H(n, x):
    if n == 0:
        return 1
    elif n == 1:
        return 2 * x
    else:
        return 2 * x * H(n - 1, x) - 2 * (n - 1) * H(n - 2, x)
```

1.2 (b) Plot of the Harmonic Oscillator Wavefunctions for $n = 0, 1, 2, 3$

The following code generates and plots the wavefunctions $\psi_n(x)$ for $n = 0, 1, 2, 3$ over the range $-4 \leq x \leq 4$.

```
def psi(n, x):
    return 1/(np.sqrt(2**n * math.factorial(n) * np.pi)) * np.exp(-x**2/2) * H(n,x)

n_vals = [0, 1, 2, 3]
x_axis = np.arange(-4, 4, 0.1)

plt.figure()
plt.title("Harmonic Oscillator Wavefunctions")
plt.xlabel("x")
plt.ylabel("  (x)")
for n in n_vals:
    psi_n = [psi(n, x) for x in x_axis]
    plt.plot(x_axis, psi_n, label=f"  (x) for n = {n}")
plt.legend()
plt.show()
```

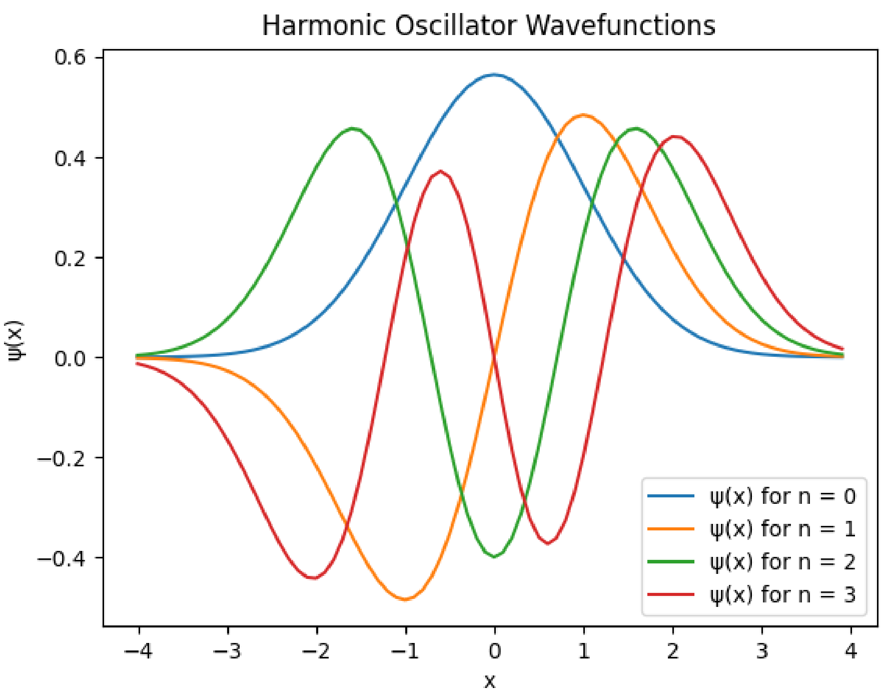

Figure 1: Harmonic Oscillator Wavefunctions

1.3 (c) Calculation of Potential Energy using Gaussian Quadrature

The function below uses Gaussian quadrature to compute the potential energy for $n = 0$ through $n = 10$.

```
def gaussxw(N):
    a = linspace(3,4*N-1,N)/(4*N+2)
    x = cos(pi*a+1/(8*N*N*tan(a)))

    epsilon = 1e-15
    delta = 1.0
    while delta>epsilon:
        p0 = ones(N, float)
        p1 = copy(x)
        for k in range(1, N):
            p0, p1 = p1, ((2*k+1)*x*p1 - k*p0) / (k+1)
        dp = (N+1)*(p0 - x*p1) / (1 - x*x)
        dx = p1 / dp
        x -= dx
        delta = max(abs(dx))

    w = 2*(N+1)*(N+1)/(N*N*(1-x*x)*dp*dp)
    return x, w

def gaussxwab(N, a, b):
    x, w = gaussxw(N)
    return 0.5 * (b - a) * x + 0.5 * (b + a), 0.5 * (b - a) * w

def quantum_uncertainty(n):
    N = 100
    z, w = gaussxwab(N, 0.0, 1.0)
    uncertainty = 0
    for i in range(N):
        uncertainty += w[i] * g(n, z[i])
    return uncertainty
```

```
def potential_energy(n):
    return quantum_uncertainty(n) / 2

# Output the potential energy for n = 0 through 10
for n in range(11):
    energy = potential_energy(n)
    print(f"Potential Energy for n = {n}: {energy:.5f}")
```

1.4 Results

The computed potential energies for $n = 0$ through $n = 10$ are as follows:

```
Potential Energy for n = 0: 0.07052
Potential Energy for n = 1: 0.21157
Potential Energy for n = 2: 0.35262
Potential Energy for n = 3: 0.49367
Potential Energy for n = 4: 0.63471
Potential Energy for n = 5: 0.77576
Potential Energy for n = 6: 0.91681
Potential Energy for n = 7: 1.05786
Potential Energy for n = 8: 1.19890
Potential Energy for n = 9: 1.33995
Potential Energy for n = 10: 1.48100
```

2 Question 2: Relativistic Particle on a Spring

2.1 (a) Calculating the Period of a Particle for $N = 8$ and $N = 16$

The following Python code numerically calculates the period of the particle using Gaussian quadrature for $N = 8$ and $N = 16$.

```
import numpy as np
from pylab import *

# Define the Gaussian quadrature helper functions

def gaussxw(N):
    a = linspace(3, 4 * N - 1, N) / (4 * N + 2)
    x = cos(pi * a + 1 / (8 * N * N * tan(a)))

    # Newton's method to find the roots
    epsilon = 1e-15
    delta = 1.0
    while delta > epsilon:
        p0 = ones(N, float)
        p1 = copy(x)
        for k in range(1, N):
            p0, p1 = p1, ((2 * k + 1) * x * p1 - k * p0) / (k + 1)
        dp = (N + 1) * (p0 - x * p1) / (1 - x * x)
        dx = p1 / dp
        x -= dx
        delta = max(abs(dx))

    # Calculate the weights
    w = 2 * (N + 1) * (N + 1) / (N * N * (1 - x * x) * dp * dp)
    return x, w

def gaussxwab(N, a, b):
    x, w = gaussxw(N)
    return 0.5 * (b - a) * x + 0.5 * (b + a), 0.5 * (b - a) * w
```

```
# Constants
m = 1  # kg
k = 12  # N/m
c = 299792458  # Speed of light in m/s

# Part a
x0 = 0.01  # m

# Define the function g(x)
def g(x, x0):
    term1 = k * (x0**2 - x**2)
    term2 = 2 * m * c**2 + term1 / 2
    numerator = term1 * term2
    denominator = 2 * (m * c**2 + term1 / 2)**2
    return c * np.sqrt(numerator / denominator)

def period(x0, N):
    x_vals, w = gaussxwab(N, 0.0, x0)

    # Compute the integral using Gaussian quadrature
    integral = 0
    for i in range(N):
        integral += w[i] * 1 / g(x_vals[i], x0)

    return 4 * integral

# Period for N = 8 and N = 16
T_8 = period(x0, 8)
T_16 = period(x0, 16)
print(f"Period for N = 8: {T_8} seconds")
print(f"Period for N = 16: {T_16} seconds")

# Estimate the fractional error
fractional_error = abs(T_16 - T_8) / T_16
print(f"Fractional error between N = 8 and N = 16: {fractional_error}")
```

Output:

```
Period for N = 8: 1.7301762343365568 seconds
Period for N = 16: 1.7707154902422433 seconds
Fractional error between N = 8 and N = 16: 0.022894279814619195
```

2.2 (b) Plotting the Integrand and Weighted Values

The following code plots the integrand values $4/g(x)$ and weighted values $4w/g(x)$ for $N = 8$ and $N = 16$.

```
def integrand_values(x0, N):
    x_vals, w = gaussxwab(N, 0.0, x0)
    vals = []
    # Compute the integral using Gaussian quadrature
    for i in range(N):
        vals.append(4 / g(x_vals[i], x0))
    return vals

def weighted_values(x0, N):
    x_vals, w = gaussxwab(N, 0.0, x0)
    vals = []
    # Compute the integral using Gaussian quadrature
    for i in range(N):
        vals.append(4 * w[i] / g(x_vals[i], x0))
    return vals

n_vals = [8, 16]

# First plot: Integrand values 4/g(x)
plt.figure()
```

```
for N in n_vals:
    x_vals, w = gaussxwab(N, 0.0, x0)
    integrand_vals = integrand_values(x0, N)
    plt.plot(x_vals, integrand_vals, label=f'4/g(x), N={N}')
plt.xlabel('x')
plt.ylabel('4/g(x)')
plt.title('Integrand values for N=8 and N=16')
plt.legend()
plt.show()

# Second plot: Weighted values 4w/g(x)
plt.figure()
for N in n_vals:
    x_vals, w = gaussxwab(N, 0.0, x0)
    weighted_vals = weighted_values(x0, N)
    plt.plot(x_vals, weighted_vals, label=f'4w/g(x), N={N}')
plt.xlabel('x')
plt.ylabel('4w/g(x)')
plt.title('Weighted values for N=8 and N=16')
plt.legend()
plt.show()
```

Plots:

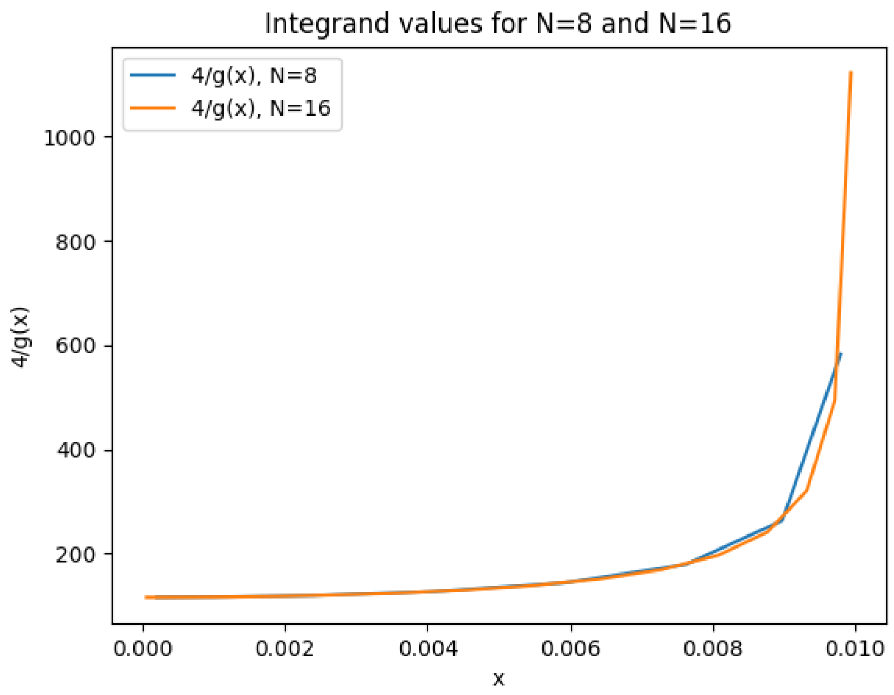

Figure 2: Integrand values for N=8 and N=16

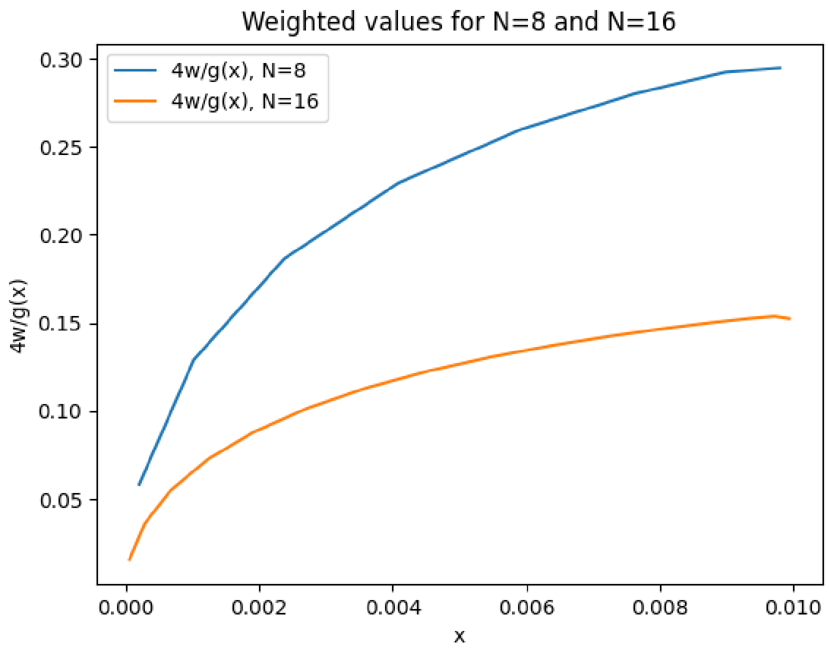

Figure 3: Weighted Values for N=8 and N=16

- **Integrand Values:** As $x \to x_0$, the integrand values exhibit near-singularity behavior. The values for $N = 8$ and $N = 16$ are quite similar for $x \leq 0.008$, but for $x > 0.008$, the singularity is approached more rapidly by the values corresponding to $N = 16$. This could affect the accuracy of the integral calculation since undefined values cannot be integrated. Larger values of N sample more points near the singularity, increasing the likelihood of encountering an undefined value.
- **Weighted Values:** As $x \to x_0$, the weighted values start to plateau, though at different values for each N (around 0.30 for $N = 8$ and 0.15 for $N = 16$). Both graphs follow the same shape, but the weighted values for $N = 16$ seem to be scaled by approximately a factor of 2. This may suggest that as N increases, the method assigns less weight to the upper limit region to balance out the instability or rapid changes in the integrand.

2.3 (c) Plot of T vs x_0

Using Gaussian quadrature with $N = 16$, the following plot shows T as a function of x_0 for $1\,\text{m} \leq x_0 \leq 10x_c$.

```
x_c = c * np.sqrt(m / k)
x0_vals = np.linspace(1, 10 * x_c, 500)
periods = [period(x0, 16) for x0 in x0_vals]

# Defining the limits
classical_limit = 2 * np.pi * np.sqrt(m / k)
relativistic_limit = [(4 * x0 / c) for x0 in x0_vals]

plt.figure()
plt.plot(x0_vals, periods, label='Gaussian Quadrature Period')
plt.plot(x0_vals, relativistic_limit, label='Highly Relativistic Limit')
plt.axhline(classical_limit, color='orange', label='Classical Limit')
plt.xlabel('x0 (m)')
plt.ylabel('Period (s)')
plt.title('Plot of x_0 vs Period')
plt.legend()
plt.show()
```

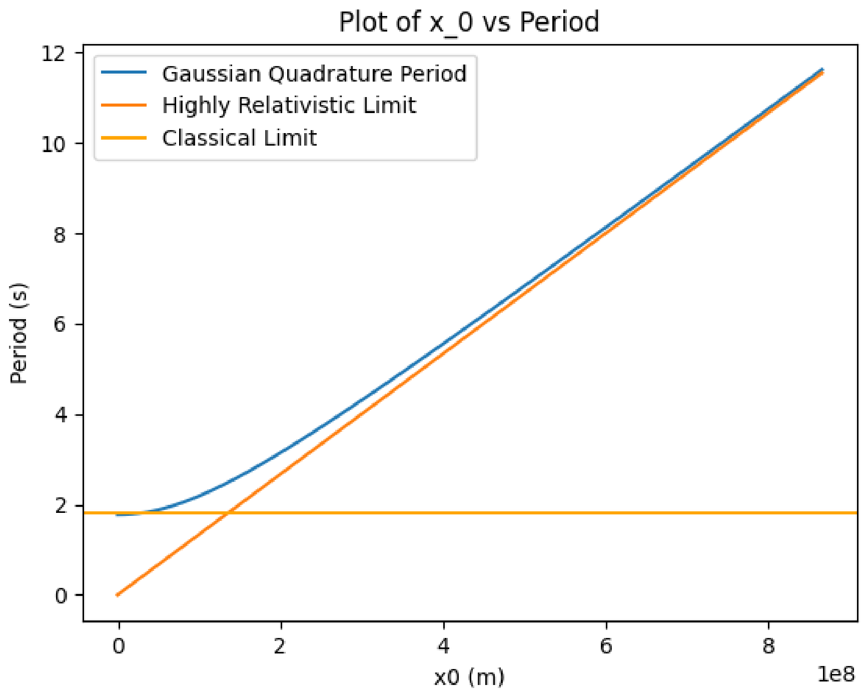

Figure 4: Period vs x0

The plot indeed seems reasonable. For values of $x_0 \leq 3.00 \times 10^7$ meters (approx.), the period closely follows the classical limit. However, as x_0 increases past this point, the term $k \cdot x_0^2$ approaches $m \cdot c^2$. The value 3.00×10^7 meters is reasonable since it is close to the speed of light (multiplied by the spring constant k), making it the threshold where relativistic behavior dominates. This behavior supports the physical expectation that for very large x_0, the system transitions from classical to relativistic dynamics. The behavior of the plot becomes even closer to the relativistic limit when x_0 passes 3.00×10^8 meters (i.e., the speed of light), which makes sense given the nature of the system.

3 Question 3: Central Differences for Numerical Differentiation

3.1 Code

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

def f(x):
    return np.exp(-(x**2))

# Part a)

def central_diff(f, x0, h):
    return (f(x0 + h/2) - f(x0 - h/2))/h

h_start = 10e-16
h_stop = 10e0
h_vals = []
h = h_start

while h <= h_stop:
    h_vals.append(h)
    h *= 10

slopes_central = [central_diff(f, 1/2, h) for h in h_vals]
for i in range(len(slopes_central)):
    print(f"Slope for h={h_vals[i]:.1e}: {slopes_central[i]:.6f}")

# Part b)

def f_prime(x):
    return -2 * x * f(x)

true_slope = f_prime(1/2)
relative_errors_cntr = [np.abs((slope - true_slope)/true_slope) for slope in slopes_central]
for i in range(len(slopes_central)):
    print(f"Relative Error for h={h_vals[i]:.1e}: {relative_errors_cntr[i]:.12f}")

min_error = min(relative_errors_cntr)
h_min = h_vals[relative_errors_cntr.index(min_error)]
print(f"Value of h that yields the smallest error: h={h_min:.1e}\n")

# Part c)

def forward_diff(f, x0, h):
    return (f(x0 + h) - f(x0))/h

slopes_forward = [forward_diff(f, 1/2, h) for h in h_vals]
for i in range(len(slopes_forward)):
    print(f"Slope for h={h_vals[i]:.1e}: {slopes_forward[i]:.6f}")

relative_errors_fwrd = [np.abs((slope - true_slope)/true_slope) for slope in slopes_forward]
for i in range(len(slopes_central)):
    print(f"Relative Error for h={h_vals[i]:.1e}: {relative_errors_fwrd[i]:.12f}")

min_error_f = min(relative_errors_fwrd)
h_min_f = h_vals[relative_errors_fwrd.index(min_error_f)]
print(f"Value of h that yields the smallest error: h={h_min_f:.1e}\n")

# Part d)

plt.figure()
plt.title("Relative Errors for Both Methods")
plt.xlabel("h")
plt.ylabel("|Relative Error|")
plt.yscale("log")
plt.xscale("log")
plt.plot(h_vals, relative_errors_cntr, marker='o', markersize=3, label="Relative Errors for
    Central Difference Method")
plt.plot(h_vals, relative_errors_fwrd, marker='o', markersize=3, linestyle='--', label="
    Relative Errors for Forward Difference Method")
plt.legend()

# Part f)

def g(x):
    return np.exp(2 * x)

# Central difference approximation for the nth derivative
def central_diff_nth_derivative(f, x0, h, n):
    if n == 1:
        # First derivative
        return (f(x0 + h) - f(x0 - h)) / (2 * h)
    elif n == 2:
        # Second derivative
        return (f(x0 + h) - 2 * f(x0) + f(x0 - h)) / h**2
    elif n == 3:
        # Third derivative
        return (f(x0 + 2*h) - 2 * f(x0 + h) + 2 * f(x0 - h) - f(x0 - 2*h)) / (2 * h**3)
    elif n == 4:
        # Fourth derivative
        return (f(x0 + 2*h) - 4 * f(x0 + h) + 6 * f(x0) - 4 * f(x0 - h) + f(x0 - 2*h)) / h
            **4
    elif n == 5:
        # Fifth derivative
        return (f(x0 + 3*h) - 5 * f(x0 + 2*h) + 10 * f(x0 + h) - 10 * f(x0 - h) + 5 * f(x0 -
             2*h) - f(x0 - 3*h)) / (2 * h**5)

for n in range(1, 6):
    derivative = central_diff_nth_derivative(g, 0, 10e-6, n)
    print(f"The {n}th derivative of g(x) at x = 0 is approximately: {derivative:.6f}")
```

3.2 Written Answers

3.3 Parts a and b: Central Difference Approximation

The central difference approximation was used to calculate the derivative of $f(x) = \exp(-x^2)$ numerically. Below is the output of the slopes and relative errors for different values of h, along with the value of h that yielded the smallest relative error:

```
Slope for h=1.0e-15: -0.777156
Slope for h=1.0e-14: -0.777156
Slope for h=1.0e-13: -0.779377
Slope for h=1.0e-12: -0.778821
Slope for h=1.0e-11: -0.778799
Slope for h=1.0e-10: -0.778801
Slope for h=1.0e-09: -0.778801
Slope for h=1.0e-08: -0.778801
Slope for h=1.0e-07: -0.778801
Slope for h=1.0e-06: -0.778801
Slope for h=1.0e-05: -0.778801
Slope for h=1.0e-04: -0.778801
Slope for h=1.0e-03: -0.778801
Slope for h=1.0e-02: -0.778785
Slope for h=1.0e-01: -0.777180
Slope for h=1.0e+00: -0.632121

Relative Error for h=1.0e-15: 0.002111792733
Relative Error for h=1.0e-14: 0.002111792733
Relative Error for h=1.0e-13: 0.000739316431
Relative Error for h=1.0e-12: 0.000026539140
Relative Error for h=1.0e-11: 0.000001971951
Relative Error for h=1.0e-10: 0.000000879158
Relative Error for h=1.0e-09: 0.000000023825
Relative Error for h=1.0e-08: 0.000000004686
Relative Error for h=1.0e-07: 0.000000001016
Relative Error for h=1.0e-06: 0.000000000018
Relative Error for h=1.0e-05: 0.000000000025
Relative Error for h=1.0e-04: 0.000000002085
Relative Error for h=1.0e-03: 0.000000208333
Relative Error for h=1.0e-02: 0.000020833120
Relative Error for h=1.0e-01: 0.002081199345
Relative Error for h=1.0e+00: 0.188341136053

Value of h that yields the smallest error: h=1.0e-06
```

The analytical derivative of $f(x) = \exp(-x^2)$ is given by $f'(x) = -2x\exp(-x^2)$. The relative error for each numerical derivative value was calculated using the true analytical derivative as a reference. The value of h that yielded the smallest relative error was found to be $h = 1.0e-06$, which matches expectations from the error analysis in Section 5.10.

The optimal h for the central difference method is derived from:

$$h = \left(24C\frac{|f(x)|}{|f^{(3)}(x)|}\right)^{1/3}$$

For $f(x) = \exp(-x^2)$, we substitute into the formula:

$$h = \left(24C\frac{\exp(-x^2)}{4(2xe^{-x^2} - 2x^3e^{-x^2})}\right)^{1/3}$$

$$h = \left(24C\frac{\exp(-x^2)}{8xe^{-x^2}(1-x^2)}\right)^{1/3}$$

$$h = \left(24C\frac{1}{8x(1-x^2)}\right)^{1/3}$$

Substituting $x = 0.5$ and $C = 10^{-16}$:

$$h = \left(24 \cdot 10^{-16}\frac{1}{8(0.5)(1-0.5^2)}\right)^{1/3} = 9.28318 \times 10^{-6}.$$

Thus, the expected value for h is approximately 9.28×10^{-6}, which matches the order of magnitude of the observed result.

3.4 Parts a and b: Forward Difference Approximation

We repeated the calculation using the forward difference method. Below is the output for slopes and relative errors for different values of h:

```
Slope for h=1.0e-15: -0.777156
Slope for h=1.0e-14: -0.777156
Slope for h=1.0e-13: -0.779377
Slope for h=1.0e-12: -0.778821
Slope for h=1.0e-11: -0.778799
Slope for h=1.0e-10: -0.778801
Slope for h=1.0e-09: -0.778801
Slope for h=1.0e-08: -0.778801
Slope for h=1.0e-07: -0.778801
Slope for h=1.0e-06: -0.778801
Slope for h=1.0e-05: -0.778805
Slope for h=1.0e-04: -0.778840
Slope for h=1.0e-03: -0.779190
Slope for h=1.0e-02: -0.782630
Slope for h=1.0e-01: -0.811245
Slope for h=1.0e+00: -0.673402

Relative Error for h=1.0e-15: 0.002111792733
Relative Error for h=1.0e-14: 0.002111792733
Relative Error for h=1.0e-13: 0.000739316431
Relative Error for h=1.0e-12: 0.000026539140
Relative Error for h=1.0e-11: 0.000001971951
Relative Error for h=1.0e-10: 0.000000879158
Relative Error for h=1.0e-09: 0.000000023825
Relative Error for h=1.0e-08: 0.000000009569
Relative Error for h=1.0e-07: 0.000000049485
Relative Error for h=1.0e-06: 0.000000499960
Relative Error for h=1.0e-05: 0.000004999909
Relative Error for h=1.0e-04: 0.000049991667
Relative Error for h=1.0e-03: 0.000499166625
Relative Error for h=1.0e-02: 0.004916628412
Relative Error for h=1.0e-01: 0.041658647035
Relative Error for h=1.0e+00: 0.135335283237

Value of h that yields the smallest error: h=1.0e-08
```

The formula for optimal h in the forward difference method is:

$$h = \sqrt{\frac{4C|f(x)|}{|f^{(2)}(x)|}}.$$

For $f(x) = \exp(-x^2)$, we derived:

$$h = \sqrt{\frac{4C}{4x^2}} = \sqrt{\frac{C}{x^2}}.$$

Substituting $C = 10^{-16}$ and $x = 0.5$:

$$h = \sqrt{\frac{10^{-16}}{0.5^2}} = \sqrt{\frac{10^{-16}}{0.25}} = \sqrt{4 \times 10^{-16}} = 2 \times 10^{-8}.$$

Thus, the expected value for h is $h = 2 \times 10^{-8}$, which is consistent with the order of magnitude of observed result for the forward difference method.

Part (d): Plot of Relative Errors

The central difference scheme doesn't always clearly beat the forward difference scheme in terms of accuracy. The relative error for both methods up to about $h = 10^{-9}$ are nearly the same. However, after that point, the central difference method is consistently more accurate than the forward difference method.

However, for small values of h, the errors for both methods converge and become similar due to round-off errors.

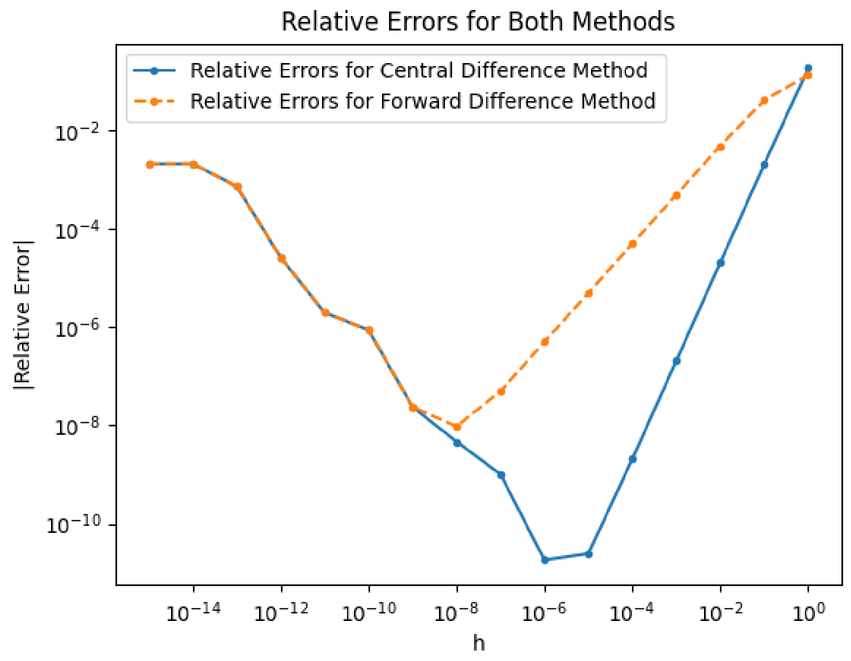

Figure 5: Absolute Value of Relative Errors for Central and Forward Difference Methods

Part (e): Error Behavior at Different h Values

At high values of h, the approximation error dominates, leading to significant discrepancies between the numerical and analytical derivatives. This is because larger h values make the approximation less accurate. On the other hand, at low values of h, round-off errors become significant due to the limited precision of floating-point arithmetic in computers, which introduces inaccuracies in the computation.

Part (f): Higher-Order Derivatives of $g(x)$

The first five derivatives of $g(x) = \exp(2x)$ at $x = 0$ were computed numerically using the central difference method with $h = 10^{-6}$. The results are as follows:

- First derivative: 2.000000
- Second derivative: 3.999999
- Third derivative: 8.104628
- Fourth derivative: 66613.381478
- Fifth derivative: 599999999951750037504.000000

The first three derivatives were computed with a high degree of accuracy, as they are close to the expected analytical values:

- First derivative: 2.000000
- Second derivative: 4.000000
- Third derivative: 8.000000

However, the fourth and fifth derivatives are clearly unreasonable. Analytically, the nth derivative of $g(x)$ is $2^n \exp(2x)$, and at $x = 0$, this simplifies to 2^n. Thus, the expected values for the fourth and fifth derivatives should have been:

- Fourth derivative: 16.000000
- Fifth derivative: 32.000000

The large errors in the higher-order derivatives are likely due to the accumulation of round-off errors and numerical instability. These inaccuracies become more pronounced for higher derivatives, as central difference methods are sensitive to small changes in h and are particularly prone to amplifying errors when calculating higher-order derivatives.