PHY407 Formal Report 1

Natalia Tabja: Student Number: 1007818747

Question 1: Exploring numerical issues with standard deviation calculations

a) Pseudocode

Algorithm 1 Standard Deviation Calculation Pseudocode

LOAD dataset
COMPUTE true_std = standard deviation of dataset using built-in function
COMPUTE n = number of data points in dataset
COMPUTE mean = mean of dataset
FUNCTION std_dev_one(data, mean, n):
    COMPUTE sum of squared differences between each data point and the mean
    DIVIDE the sum by (n - 1)
    RETURN the square root of the result
END FUNCTION
FUNCTION std_dev_two(data, mean, n):
    COMPUTE sum of squared data points
    SUBTRACT n * mean^2 from the sum
    DIVIDE the result by (n - 1)
    RETURN the square root of the result
END FUNCTION
CALCULATE std1 = std_dev_one(data, mean, n)
CALCULATE std2 = std_dev_two(data, mean, n)
COMPUTE rel_err1 = (std1 - true_std) / true_std
COMPUTE rel_err2 = (std2 - true_std) / true_std
PRINT true_std, rel_err1, rel_err2

b) Implementation of Pseudocode in a)

Algorithm 2 Python Code for Standard Deviation Calculation

```
import numpy as np

# Load data from the file
data = np.loadtxt("cdata.txt")
n = len(data)
mu = np.mean(data)

# Standard deviation calculation using numpy
true_std = np.std(data, ddof = 1)
```

```
# Standard deviation calculation formula 1
def std_dev_one(data, mu, n):
   return np.sqrt(np.sum((data - mu) ** 2) / (n - 1))

# Standard deviation calculation formula 2
def std_dev_two(data, mu, n):
   return np.sqrt((np.sum(data**2) - n * mu**2) / (n - 1))

std_one = std_dev_one(data, mu, n)
rel_err1 = (std_one - true_std) / true_std

std_two = std_dev_two(data, mu, n)
rel_err2 = (std_two - true_std) / true_std

print(f"Numpy Calculation: {true_std}")
print(f"Standard Deviation One: {std_one}, Relative Error: {rel_err1}")
print(f"Standard Deviation Two: {std_two}, Relative Error: {rel_err2}")
```

Output:

Numpy Calculation: 0.07901054781905067

Standard Deviation One: 0.07901054781905067, **Relative Error:** 0.0

Standard Dev Two: 0.07901054763832621, **Relative Error:** -2.2873460336752e-09

The relative error with greater magnitude is that corresponding to the second formula for standard deviation.

c) Evaluating Standard Dev Calculations

```
import numpy as np

# Generate sequences that follow a normal distribution
normal_sequence_1 = np.random.normal(0.0, 1.0, 2000)  # Data with mean 0 and sigma 1
normal_sequence_2 = np.random.normal(1.e7, 1.0, 2000)  # Data with mean 10^7 and sigma 1

# Calculate means and standard deviations for both datasets
mean_1 = np.mean(normal_sequence_1)
mean_2 = np.mean(normal_sequence_2)
true_std_1 = np.std(normal_sequence_1, ddof=1)
true_std_2 = np.std(normal_sequence_2, ddof=1)

# Standard deviation calculation using sum of squared differences
def std_dev_one(data, mu, n):
    return np.sqrt(np.sum((data - mu) ** 2) / (n - 1))

def std_dev_two(data, mu, n):
    return np.sqrt((np.sum(data**2) - n * mu**2) / (n - 1))

# Test both formulas on normal_sequence_1 (mean 0, sigma 1)
std_formula_one_seq1 = std_dev_one(normal_sequence_1, mean_1, len(normal_sequence_1))
relative_error_formula_one_seq1 = (std_formula_one_seq1 - true_std_1) / true_std_1
```

```
std_formula_two_seq1 = std_dev_two(normal_sequence_1, mean_1, len(normal_sequence_1))
relative_error_formula_two_seq1 = (std_formula_two_seq1 - true_std_1) / true_std_1

# Test both formulas on normal_sequence_2 (mean 10^7, sigma 1)
std_formula_one_seq2 = std_dev_one(normal_sequence_2, mean_2, len(normal_sequence_2))
relative_error_formula_one_seq2 = (std_formula_one_seq2 - true_std_2) / true_std_2

std_formula_two_seq2 = std_dev_two(normal_sequence_2, mean_2, len(normal_sequence_2))
relative_error_formula_two_seq2 = (std_formula_two_seq2 - true_std_2) / true_std_2

# Output results for normal_sequence_1
print(f"Standard Deviation for Sequence 1 using Formula 1: {std_formula_one_seq1}, Relative Error:
     {relative_error_formula_one_seq1}")
print(f"Standard Deviation for Sequence 1 using Formula 2: {std_formula_two_seq1}, Relative Error:
     {relative_error_formula_two_seq1}")

# Output results for normal_sequence_2
print(f"Standard Deviation for Sequence 2 using Formula 1: {std_formula_one_seq2}, Relative Error:
     {relative_error_formula_one_seq2}")
print(f"Standard Deviation for Sequence 2 using Formula 2: {std_formula_two_seq2}, Relative Error:
     {relative_error_formula_two_seq2}")
```

Output:

Standard Dev for Sequence 1 using Formula 1: 1.0046143032986068,

Relative Error: 0.0

Standard Dev for Sequence 1 using Formula 2: 1.0046143032986068,

Relative Error: 0.0

Standard Dev for Sequence 2 using Formula 1: 1.0133541732930265,

Relative Error: 0.0

Standard Dev for Sequence 2 using Formula 2: 1.020058949340706,

Relative Error: 0.006616419238588076

Comparing the results of applying formula 1 versus formula 2 to both of the normally-distributed sequences, we can analyze their accuracy/performance.

For the first sequence, with a mean of 0 and sigma of 1, both formulas for standard deviation yielded the same estimate, namely `1.0046143032986068`. This was the same result as numpy's built-in standard deviation function.

For the second sequence with a mean of 1×10^7 and sigma of 1, formula 1 still produced a standard deviation equivalent to that of numpy.std, but formula 2's result had a relative error of about 0.66%.

Overall, the second formula tends to produce greater error, in particular when dealing with a larger mean. This is due to the way this formula performs its calculations, i.e by subtracting

$n \cdot \mu^2$ from the sum of squared data points. Since each data point is close to the mean, squaring each data point is like squaring the mean, and doing so n times will of course approximate a result close to $n \cdot \mu^2$. Therefore, these two terms will nearly cancel each other out, leading to catastrophic cancellation.

This phenomenon is more pronounced when dealing with large numbers due to the limitations of floating-point precision in computers. Computers store numbers as two values: a mantissa (which holds the number of significant digits in the number), and an exponent (which scales the mantissa by a power of 2). The number of digits that can be stored in the mantissa is finite, so while small numbers may be stored accurately, the same cannot be said for very large numbers. Therefore two very large numbers with slight differences may be represented in such a way that to the computer they are equal, leading to inaccuracies in these kinds of calculations.

d) Workaround for One-Pass Method Issues

One potential workaround for the catastrophic cancellation is to cast the data to `np.float64` before performing the calculations to ensure that the values are stored and manipulated with higher precision.

Pseudocode:

Algorithm 3 One-Pass Method with float64 Precision

INPUT: data (array of numbers), mean (μ), number of data points (n)
CONVERT data to `float64` precision for higher accuracy
CALCULATE sum of squared data points: sum_squares = $\sum \text{data}^2$
COMPUTE variance = (sum_squares - $n \times \mu^2$) / (n - 1)
RETURN square root of variance as the standard deviation

Code:

```
import numpy as np

# One-pass method with float64 precision
def std_dev_one_pass_high_precision(data, mu, n):
    data = data.astype(np.float64)  # Use float64
    sum_squares = np.sum(data**2)
    variance = (sum_squares - n * mu**2) / (n - 1)
    return np.sqrt(variance)
```

```
# Generate sequence with large mean (1.e7) and sigma 1
normal_sequence_2 = np.random.normal(1.e7, 1.0, 2000).astype(np.float64)
n = len(normal_sequence_2)
mu_2 = np.mean(normal_sequence_2)
true_std_2 = np.std(normal_sequence_2, ddof=1)

high_precision_result = std_dev_one_pass_high_precision(normal_sequence_2, mu_2, n)
rel_error = (high_precision_result - true_std_2) / true_std_2

print(f"True Standard Deviation (numpy): {true_std_2}")
print(f"High Precision One-Pass Method (Sequence 2): {high_precision_result}")
print(f"Relative Error (High Precision One-Pass Method): {rel_error}")
```

Output:

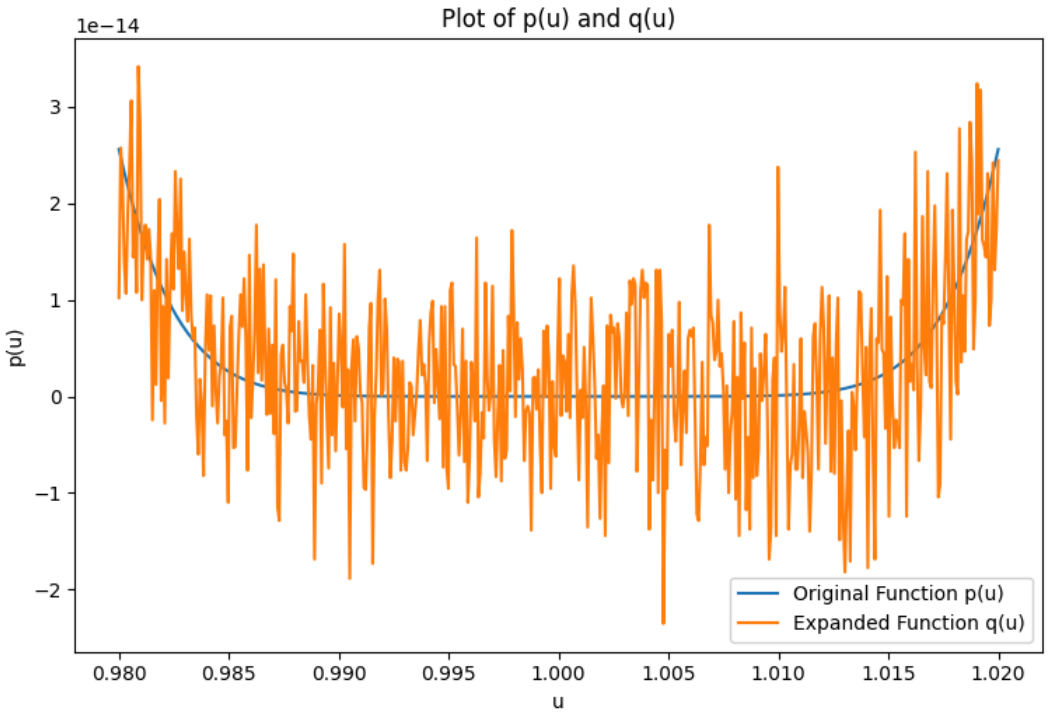
```
True Standard Deviation (numpy): 1.009505895577033
High Precision One-Pass Method (Sequence 2): 1.0121819283745703
Relative Error (High Precision One-Pass Method): 0.002650834244021561
```

By increasing the precision to float64, the relative error in the one-pass method was significantly reduced. While not completely eliminating the error, it was reduced from -1.79% to 0.27%. This makes the one-pass method much more accurate for datasets with large means while preserving its computational efficiency.

Question 2: Exploring Roundoff Error

a) Plotting p(u) and q(u)

The plot of q(u) is clearly the function that exhibits more noise in this plot. The original function p(u) is a lot smoother because it only involves one subtraction (namely $1 - u$) and its exponentiation to the power of eight, leaving fewer opportunities for roundoff errors. On the other hand, the expansion q(u) involves multiple terms that can have similar values around $u = 1$ (where both functions approach zero), leading to catastrophic cancellation as described in the previous section.

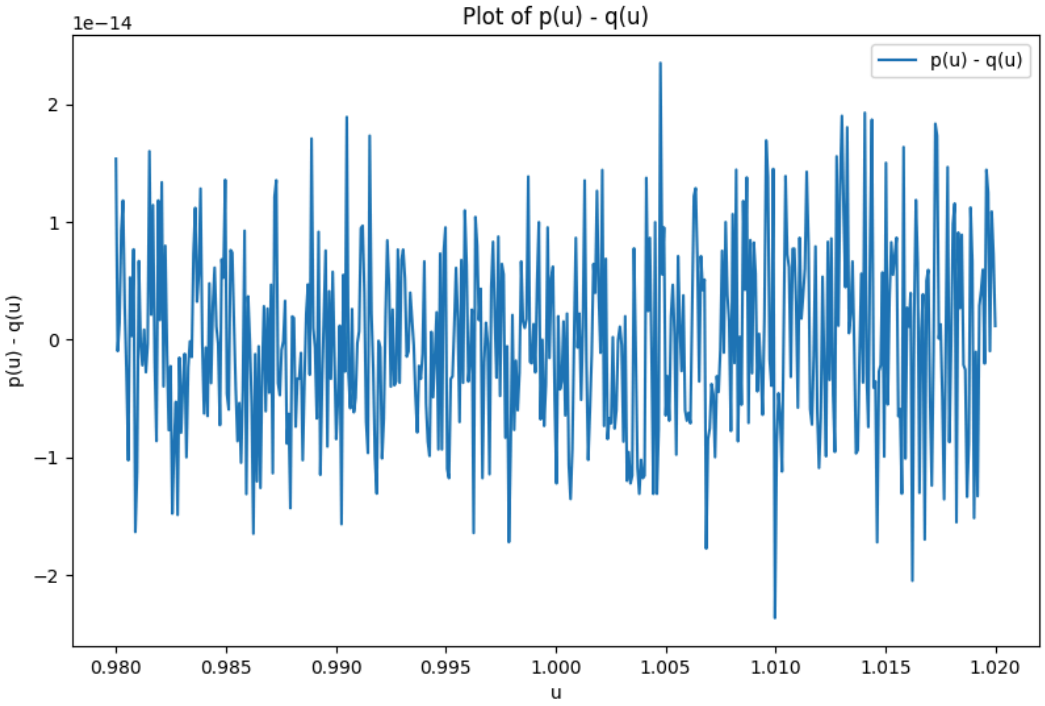


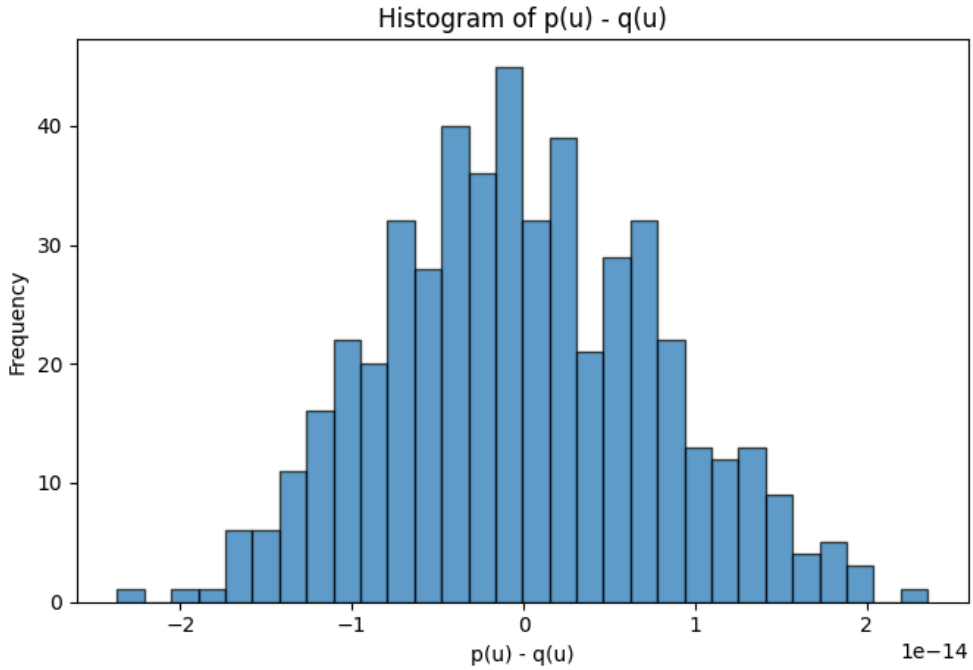
b) Plotting p(u) - q(u)

The estimated error from equation [3] was about 2.24×10^{-15}, while the actual standard deviation of $p(u) - q(u)$ is about 8.00×10^{-15}. The terms N and $\overline{x^2}$ in equation [3] were calculated as follows:

- N = number of data points from 0.98-1.02 for which $p(u) - q(u)$was calculated (500 in this case).
- $\overline{x^2}$ = the mean of the squares of all the plotted data points

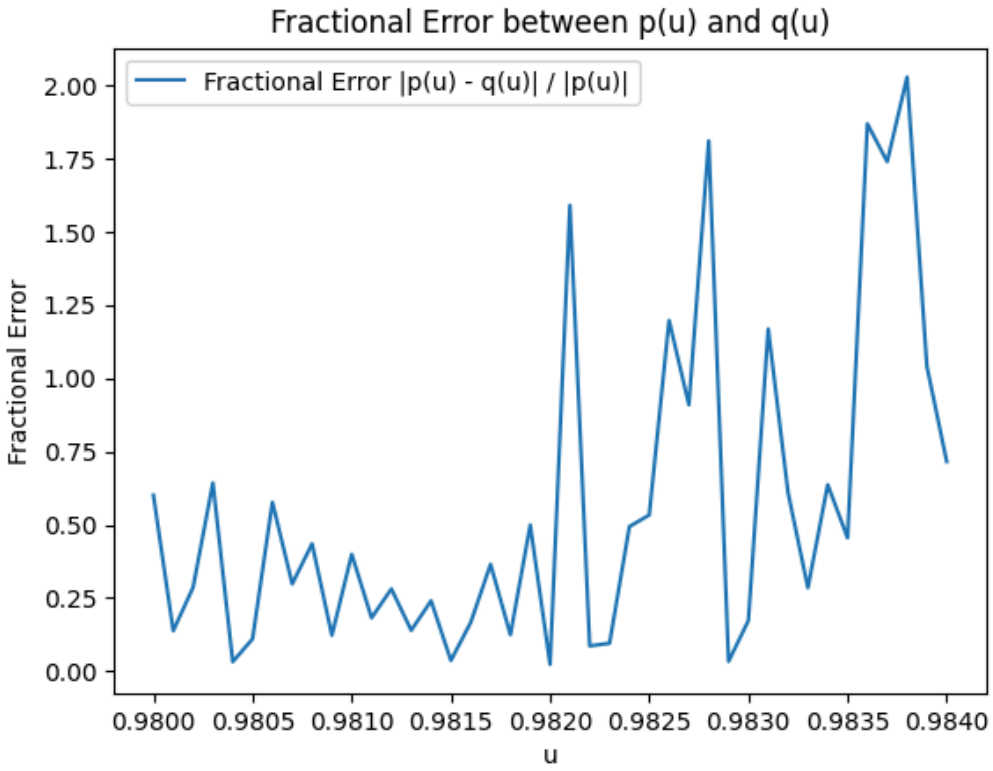
I do think there is a connection between the histogram and the statistical quantity in equation [3]. The histogram shows the distribution of the differences $p(u) - q(u)$, with most values centered around zero and spread out symmetrically. The spread of this distribution, measured by the standard deviation, is of the same order of magnitude as the estimated roundoff error from equation [3], indicating that the distribution of the differences reflects the expected behavior of roundoff errors.


Plot of p(u) - q(u)
1e−14
p(u) - q(u)
2
1
0
−1
−2
p(u) - q(u)
0.980
0.985
0.990
0.995
1.000
1.005
1.010
1.015
1.020
u


Histogram of p(u) - q(u)
Frequency
40
30
20
10
0
−2
−1
0
1
2
p(u) - q(u)
1e−14

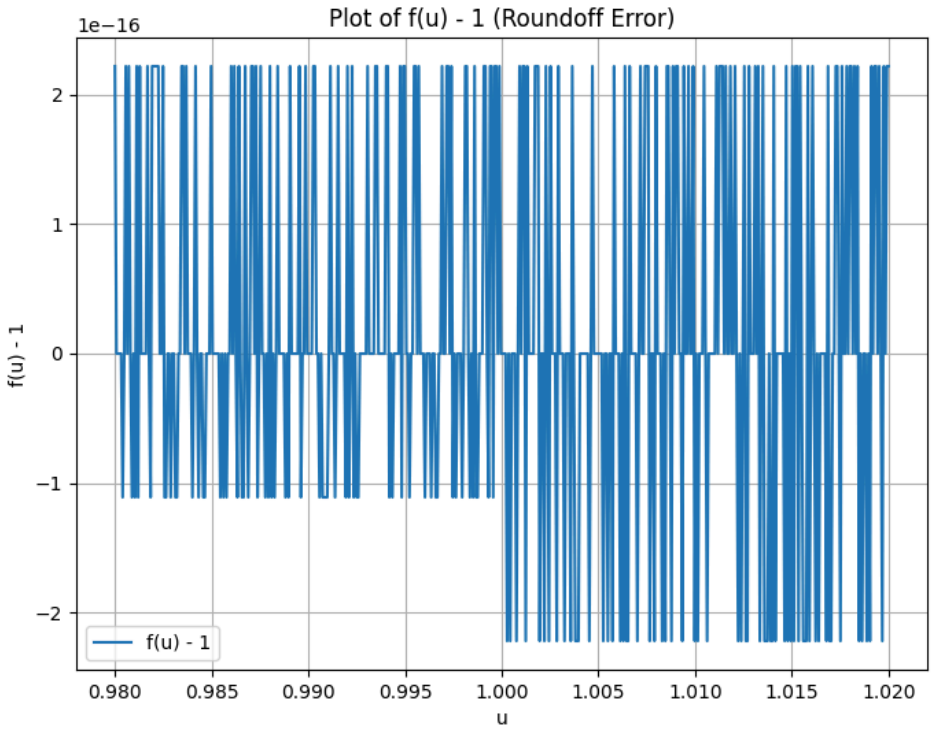
c) Comparing performance of calculations on different sequences

The fractional error between $p(u)$ and $q(u)$ begins to grow significantly as u approaches 1. From the graph, the error exceeds 100% (i.e, fractional error > 1) starting around u=0.982, and continues to fluctuate but generally increases as u gets closer to 1. This behavior indicates the increasing impact of roundoff errors and catastrophic cancellation near this critical point.



d) Plot of Roundoff Error

The observed standard deviation of $f(u) - 1$ is 1.40338×10^{-16}, which is close to the theoretical error estimate of 1.00000×10^{-16} from Equation (4.5). We set $x = 1$ in the error estimate because the function $f(u) = \frac{u^8}{u^4 \times u^4}$ is mathematically equal to 1, and we are calculating the deviation from this value caused by roundoff error. The small difference between the observed and theoretical errors indicates that the roundoff error is consistent with machine precision limits, and the fluctuations around zero in the plot confirm that the deviations are primarily due to numerical roundoff.



Question 3: Trapezoidal and Simpson's Rules for Integration

a) Evaluating the integral

If we rewrite the integral as $4\int_0^1 \frac{1}{1+x^2}dx$, we can solve this analytically:

$$I = 4\int_0^1 \frac{1}{1+x^2}dx = 4 \times [arctan(1) - arctan(0)] = 4 \times [\frac{\pi}{4} - 0] = \pi$$

b) Comparing Trapezoidal vs Simpson's Rule using N=4

Using 4 slices, the Trapezoidal Rule yielded an answer of about 3.13118 (with a corresponding error of about 0.01042), whereas Simpson's Rule gave about 3.14157 (with a corresponding error of about 2.40261 e-05)

```
import numpy as np

# Function to integrate
def f(x):
    return 4 / (1 + x**2)

# Trapezoidal rule
def trapezoidal_rule(a, b, N):
    h = (b - a) / N
    x = np.linspace(a, b, N + 1)
    y = f(x)
```

```
    result = h * (0.5 * y[0] + np.sum(y[1:-1]) + 0.5 * y[-1])
    return result

# Simpson's rule
def simpsons_rule(a, b, N):
    if N % 2 != 0:
        return None  # N must be even for Simpson's rule
    h = (b - a) / N
    x = np.linspace(a, b, N + 1)
    y = f(x)
    result = (h / 3) * (y[0] + 4 * np.sum(y[1:-1:2]) + 2 * np.sum(y[2:-2:2]) + y[-1])
    return result

# Exact value of the integral
exact_value = np.pi

# Parameters for N = 4 slices
a = 0
b = 1
N = 4

# Apply both methods
I_trap_4 = trapezoidal_rule(a, b, N)
I_simp_4 = simpsons_rule(a, b, N)

# Print results and compare
print(f"Exact value of the integral: {exact_value}")
print(f"Trapezoidal Rule (N=4): {I_trap_4}")
print(f"Simpson's Rule (N=4): {I_simp_4}")
print(f"Trapezoidal Error: {abs(I_trap_4 - exact_value)}")
print(f"Simpson's Error: {abs(I_simp_4 - exact_value)}")
```

Output:

Exact value of the integral: 3.141592653589793

Trapezoidal Rule (N=4): 3.1311764705882354

Simpson's Rule (N=4): 3.14156862745098

Trapezoidal Error: 0.010416183001557666

Simpson's Error: 2.4026138813137976e-05

c) Comparing Trapezoidal vs Simpson's Rule for Higher N

For the trapezoidal rule, 16384 slices were required to approximate the integral with an error of $O(10^{-9})$, compared to just 32 being required to achieve the same results using Simpson's method. Moreover, Simpson's method was more time-efficient; it took 0.000078 seconds to reach this accuracy compared to the 0.000306 seconds using the Trapezoidal rule. Overall, Simpson's Rule had much better performance.

```python
import time

# Function to estimate the required slices for error O(10^-9)
def find_slices_for_error(method, a, b, exact_value, error_threshold=1e-9):
    N = 2  # Start with N = 2 slices
    while True:
        result = method(a, b, N)
        error = abs(result - exact_value)
        if error < error_threshold:
            break
        N *= 2  # Double the number of slices
    return N, result, error

# Trapezoidal method timing
start_time = time.time()
N_trap, result_trap, error_trap = find_slices_for_error(trapezoidal_rule, a, b, exact_value)
trap_time = time.time() - start_time

# Simpson's method timing
start_time = time.time()
N_simp, result_simp, error_simp = find_slices_for_error(simpsons_rule, a, b, exact_value)
simp_time = time.time() - start_time

# Print results
print(f"Exact value of the integral: {exact_value}")
print(f"\nTrapezoidal Rule:")
print(f"Slices required (N): {N_trap}")
print(f"Result: {result_trap}")
print(f"Error: {error_trap}")
print(f"Time taken: {trap_time} seconds")

print(f"\nSimpson's Rule:")
print(f"Slices required (N): {N_simp}")
print(f"Result: {result_simp}")
print(f"Error: {error_simp}")
print(f"Time taken: {simp_time} seconds\n")
```

	Trapezoidal Rule	**Simpson's Rule**
Slices required (N):	16384	32
Result	3.1415926529689115	3.1415926535528365
Error	6.208815683805824e-10	3.695665995451236e-11
Time taken (s)	0.000306	0.000078

d) Adapting the “Practical Estimation of Errors” from the Textbook

```
def practical_error_trapezoidal(a, b, N1, N2):
    I1 = trapezoidal_rule(a, b, N1)
    I2 = trapezoidal_rule(a, b, N2)
    error_est = (1 / 3) * abs(I2 - I1)
    return error_est

# N_1 = 16 and N_2 = 32
N1 = 16
N2 = 32

error_estimation_trap = practical_error_trapezoidal(a, b, N1, N2)
I2 = trapezoidal_rule(a, b, N2)
print(f"Estimated error for trapezoidal rule with N2=32: {error_estimation_trap}")
print(f"Trapezoidal rule result with N2=32: {I2}")
```

e) Practical Estimation Method and Simpson’s Rule

The practical estimation method for the trapezoidal rule is based on the fact that the error is proportional to h^2, where h is the step size. For Simpson's rule, the error is proportional to h^4, so a simple subtraction like in the trapezoidal method won't work. To adapt the practical error estimation method for Simpson's rule, one would need to account for the h^4 dependence in the error, which would likely involve a more complex formula similar to Romberg's method, refining the estimate by using higher-order terms.