Chapter 2

Numerical Integration Methods

2.1 Historical Background

The concept of numerical integration dates back to early quadrature methods developed in the 17th century. These methods, collectively known as the Newton–Cotes formulas, approximate integrals by interpolating the integrand with polynomials. Among the most well-known are the **Trapezoidal Rule**, the **Midpoint Rule**, and **Simpson's Rule**. These serve as fundamental techniques in scientific computing, engineering, and applied mathematics.

2.2 Mathematical Foundations

The goal of numerical integration is to approximate a definite integral

$$I = \int_a^b f(x)\,dx \tag{2.1}$$

by evaluating $f(x)$ at discrete points in $[a, b]$. The interval is divided into n subintervals of equal width

$$h = \frac{b-a}{n}, \qquad x_i = a + ih. \tag{2.2}$$

2.3 Midpoint Rule

2.3.1 Geometric Interpretation

The Midpoint Rule approximates the area under $f(x)$ using rectangles whose heights are determined by the function's value at the midpoint of each subinterval.

2.3.2 Derivation

Over a single subinterval $[x_{i-1}, x_i]$, the integral is approximated by

$$\int_{x_{i-1}}^{x_i} f(x)\,dx \approx hf\left(\frac{x_{i-1}+x_i}{2}\right). \tag{2.3}$$

Summing over all n intervals yields

$$I_M = h\sum_{i=1}^{n} f\left(\frac{x_{i-1}+x_i}{2}\right). \tag{2.4}$$

2.3.3 Error Derivation (via Taylor Expansion)

Expanding $f(x)$ about the midpoint x_m gives

$$f(x) = f(x_m) + f'(x_m)(x-x_m) + \frac{f''(\xi)}{2}(x-x_m)^2. \tag{2.5}$$

Integrating and simplifying shows the local truncation error:

$$E_M = -\frac{(b-a)}{24}h^2 f''(\xi), \tag{2.6}$$

where $\xi \in [a,b]$. Hence, the Midpoint Rule has **second-order accuracy** ($O(h^2)$).

2.4 Trapezoidal Rule

2.4.1 Geometric Interpretation

This rule approximates the curve using trapezoids connecting $f(x_i)$ and $f(x_{i+1})$. It effectively replaces $f(x)$ with a straight line between sample points.

2.4.2 Derivation

For one subinterval,

$$\int_{x_i}^{x_{i+1}} f(x)\,dx \approx \frac{h}{2}[f(x_i) + f(x_{i+1})]. \tag{2.7}$$

Summing over n subintervals gives

$$I_T = \frac{h}{2}\left[f(x_0) + 2\sum_{i=1}^{n-1} f(x_i) + f(x_n)\right]. \tag{2.8}$$

2.4.3 Error Derivation

Using the Taylor expansion around x_i,

$$f(x_{i+1}) = f(x_i) + f'(x_i)h + \frac{f''(\xi)}{2}h^2. \tag{2.9}$$

Integrating and comparing with the true integral yields

$$E_T = -\frac{(b-a)}{12}h^2 f''(\xi). \tag{2.10}$$

Thus, like the Midpoint Rule, the Trapezoidal Rule has **second-order accuracy**.

2.5 Simpson's Rule

2.5.1 Geometric Interpretation

Simpson's Rule approximates $f(x)$ using a quadratic polynomial through every three consecutive points. It captures curvature and provides much higher accuracy for smooth functions.

2.5.2 Derivation

Over two subintervals $[x_0, x_2]$, we approximate $f(x)$ by a parabola:

$$f(x) \approx ax^2 + bx + c, \tag{2.11}$$

which passes through x_0, x_1, and x_2. Integrating yields

$$\int_{x_0}^{x_2} f(x)\,dx \approx \frac{h}{3}\left[f(x_0) + 4f(x_1) + f(x_2)\right]. \tag{2.12}$$

For n even,

$$I_S = \frac{h}{3}\left[f(x_0) + 4\sum_{i=1,3,\dots}^{n-1} f(x_i) + 2\sum_{i=2,4,\dots}^{n-2} f(x_i) + f(x_n)\right]. \tag{2.13}$$

2.5.3 Error Derivation

From the fourth-order term in the Taylor expansion, we obtain

$$E_S = -\frac{(b-a)}{180}h^4 f^{(4)}(\xi). \tag{2.14}$$

Hence Simpson's Rule is **fourth-order accurate**, converging much faster than the previous two methods.

2.6 Numerical Examples

2.6.1 Example 1: $f(x) = x^2$ on $[0, 2]$

Exact: $I = 8/3$.

$$\begin{aligned} I_M &= 2.657, & E_M &= 0.0097, \\ I_T &= 2.667, & E_T &= 0.0003, \\ I_S &= 2.667, & E_S &= 0.0000. \end{aligned}$$

2.6.2 Example 2: $f(x) = \sin(x)$ on $[0, \pi]$

Exact: $I = 2$.

$$\begin{aligned} I_M &= 1.999, & E_M &= 0.001, \\ I_T &= 1.983, & E_T &= 0.017, \\ I_S &= 2.000, & E_S &= 0.000. \end{aligned}$$

2.6.3 Example 3: $f(x) = e^{-x^2}$ on $[0, 1]$

Exact: $I = 0.746824$.

$$\begin{aligned} I_M &= 0.7471, & E_M &= 0.0003, \\ I_T &= 0.7429, & E_T &= 0.0039, \\ I_S &= 0.7468, & E_S &= 0.0000. \end{aligned}$$

2.7 Comparison and Discussion

Function	**Exact**	**Midpoint Error**	**Trapezoidal Error**	**Simpson Error**
x^2	2.6667	0.0097	0.0003	0.0000
$\sin(x)$	2.0000	0.0010	0.0170	0.0000
e^{-x^2}	0.7468	0.0003	0.0039	0.0000

Table 2.1: Comparison of numerical errors across methods.

Method	**Order of Accuracy**	**Smoothness Required**	**Computational Cost**
Midpoint Rule	$O(h^2)$	C^2	Low
Trapezoidal Rule	$O(h^2)$	C^2	Low
Simpson's Rule	$O(h^4)$	C^4	Moderate

Table 2.2: Summary of method accuracy and requirements.

2.7.1 Summary of Method Properties

2.8 Conclusion

In this chapter, we explored the three cornerstone methods of numerical integration. The Midpoint and Trapezoidal Rules, both second-order accurate, are efficient for rough approximations. Simpson's Rule, with its fourth-order convergence, is highly accurate for smooth functions. Together, these methods form the foundation for advanced quadrature schemes in computational mathematics.

2.9 Further Reading

- Burden, R. L., & Faires, J. D. (2011). *Numerical Analysis.* Brooks/Cole.
- Chapra, S. C., & Canale, R. P. (2015). *Numerical Methods for Engineers.* McGraw-Hill.
- Atkinson, K. E. (1989). *An Introduction to Numerical Analysis.* Wiley.

2.10 Python Implementation (Loop-Based)

The Python code below is provided for illustrative purposes only.

```
""" Dr. Samir Kenouche - Numerical Integration Methods """

import math

# Midpoint rule
def midpoint_rule(fun, a, b, n):
    h = (b - a) / n
    total_area = 0
    for i in range(n):
        x_mid = a + (i + 0.5) * h
        total_area = total_area + fun(x_mid)
    return h * total_area

# Trapezoidal rule
def trapezoidal_rule(fun, a, b, n):
    h = (b - a) / n
    total_area = fun(a) + fun(b)
    for i in range(1, n):
       total_area = total_area + 2 * fun(a + i * h)
    return (h / 2) * total_area

# Simpson rule
def simpson_rule(fun, a, b, n):
    if n % 2 != 0:
        raise ValueError("n must be even for Simpson's rule")
    h = (b - a) / n
    total_area = fun(a) + fun(b)
    for i in range(1, n):
        x = a + i * h
        total_area = total_area + 4 * fun(x) if i % 2 != 0 else 2 * fun(x)
    return (h / 3) * total_area

# Example functions
def fun1(x): return x**2
def fun2(x): return math .sin(x)
def fun3(x): return math .exp(-x **2)

# Result for each method
int_Midpoint = print(midpoint_rule (fun1, 0, 2, 8)) # Midpoint_rule method
int_Trapez  = print(midpoint_rule (fun1, 0, 2, 8)) # Trapezoidal_rule method
int_Simpson = print(midpoint_rule (fun1, 0, 2, 8)) # Simpson_rule method
```