Analysis of Numerical Integrations Methods

Niranjan Krishna Kumar
Shobhit Singh

Key-words:

1. Introduction

In this report, we will be analysing the three numerical integration methods, namely Midpoint Rule, Trapezoid Rule and the Simpsons Rule in several aspects such as time complexity and accuracy . On the basis of the result, we hope to conclude the best numerical integration method.

2. Numerical Integration Methods

2.1. Midpoint Rule

Midpoint rule uses the Riemann sum of equal sub intervals and their respective midpoints i.e. in mathematical terms,
Assume $f(x)$ is continuous on the interval $[a, b]$
Let n be a positive integer, then $\Delta x = \frac{b-a}{n}$
If $[a, b]$ is divided into n sub intervals of length Δx.
Let m_i be the midpoint of the i^{th} sub interval of $[a, b]$
Then the limit of its Riemann Sum i.e. $M_n = \sum_{i=1}^{n} f(x_i) \cdot \Delta x$ is an approximation for the integral of the function.
$\implies \lim_{n\to\infty} M_n = \int_a^b f(x)dx$

```
 def midpoint_rule_sum(f,a,b,n):
l = numpy.linspace(a,b,n)
sum = 0
for i in range(n-1):
   sum = sum + ((l[i+1]-l[i])* f ((l[i+1]+l[i]) /2))
return sum
```

2.2. Trapezoidal Rule

Trapezoidal Rule for approximation uses trapezoids instead of rectangles(like in midpoint rule) to approximate the integral.
Assume $f(x)$ is continuous on the interval $[a, b]$
Let n be a positive integer, then $\Delta x = \frac{b-a}{n}$
If $[a, b]$ is divided into n sub intervals of length Δx.
Let the endpoints of each sub interval be P $= \{x_0, x_1, x_2 \ldots, x_{n-1}, x_n\}$
Sum of the areas of the Trapezoids,
$A_n = \frac{\Delta x}{2} \cdot \{f(x_0) + 2 \cdot f(x_1) + 2 \cdot f(x_2) \cdots + 2 \cdot f(x_{n-1} + f(x_n))\}$
Hence the integral approximate is,
$\lim_{n\to\infty} A_n = \int_a^b f(x)dx$

```
 def trapezoidal_sum(f,a,b,n):
l = numpy.linspace(a,b,n)
sum = 0
for i in range(n-1):
   sum = sum + ( 0.5 * (f(l[i]) + f(l[i+1])) * (l[i+1] - l[i] ) )
```

```
    return sum
```

2.3. Simpsons Rule

Trapezoid rule uses piece-wise linear function, in Simpsons rule we use piece-wise quadratic equations instead,
Assume $f(x)$ is continuous on the interval $[a, b]$
Let n be a positive even integer, then $\Delta x = \frac{b-a}{n}$
If $[a, b]$ is divided into n sub intervals of length Δx.
Let the endpoints of each sub interval be P $= \{x_0, x_1, x_2 \ldots, x_{n-1}, x_n\}$
Sum of Areas is,
$S_n = \frac{\Delta x}{3}\{f(x_1) + 2 \cdot f(x_2) + 4 \cdot f(x_2) + \cdots + 2 \cdot f(x_{n-2} + 4 \cdot f(x_{n-1} + f(x_n)))\}$
Hence,
$\lim_{n\to\infty} S_n = \int_a^b f(x)dx$

```
def simpsons(f,a,b,n):
    h = (b - a) / n
    x = a
    res = f(x)
    res += f(b)
    for i in range(1, n):
        x += h
        res += 2 * f(x) if i % 2 == 0 else 4 * f(x)
    return (h / 3) * res
```

3. Time Complexity Analysis

Time complexity describes the rate time taken by algorithm grows when the input is very large. We write time complexity in terms of O (big O). A time complexity of $O(n^2)$ would mean that the time taken by algorithm grows exponentially as the input value increases and hence it would be considered a pretty bad algorithm.

3.1. Time Complexity of Midpoint rule

```
l = numpy.linspace(a,b,n) \\O(n)
sum = 0                   \\O(1)
for i in range(n-1):      \\O(n)
    sum = sum + ((l[i+1]-l[i])* f ((l[i+1]+l[i]) /2))
```

Final time complexity = O(n)+O(1)+O(n) = O($2n + 1$) = O(n) i.e. Linear growth.

3.2. Time Complexity of Trapezoid rule

```
l = numpy.linspace(a,b,n) \\O(n)
sum = 0                   \\O(1)
for i in range(n-1):      \\O(n)
   sum = sum + ( 0.5 * (f(l[i]) + f(l[i+1])) * (l[i+1] - l[i]))
```

Final time complexity = O(n)+O(1)+O(n) = O($2n + 1$) = O(n) i.e. Linear growth.

3.3. Time Complexity of Simpsons rule

```
h = (b - a) / n    \\O(1)
x = a              \\O(1)
res = f(x)         \\O(1)
res += f(b)        \\O(1)
for i in range(1,n): \\O(n)
    x += h
    res += 2 * f(x) if i % 2 == 0 else 4 * f(x)
```

Final time complexity = O(1)+O(1)+O(1)+O(1)+O(n) = O($n+4$) = O(n) i.e. Linear growth.

4. Runtime analysis

In the second section, we compared the three algorithms on the basis of actual runtime.

4.1. Methodology

To calculate runtime, we chose the time module of Python 3.9. Any other method like using Linux's inbuilt time command could have also worked.

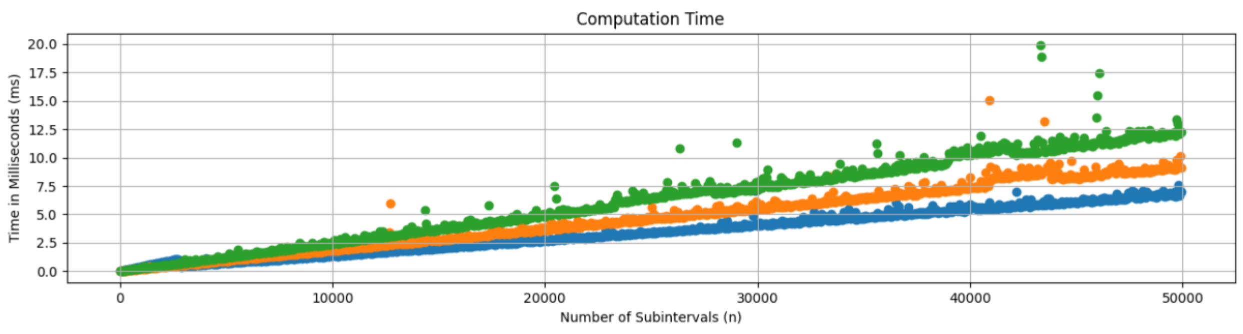
Example of implementation :

```
def midpoint_rule_sum(f,a,b,n):
    start = time.time()
    l = numpy.linspace(a,b,n)
    sum = 0
    for i in range(n-1):
        sum = sum + ((l[i+1]-l[i])* f ((l[i+1]+l[i]) /2))
    return time.time()-start
```

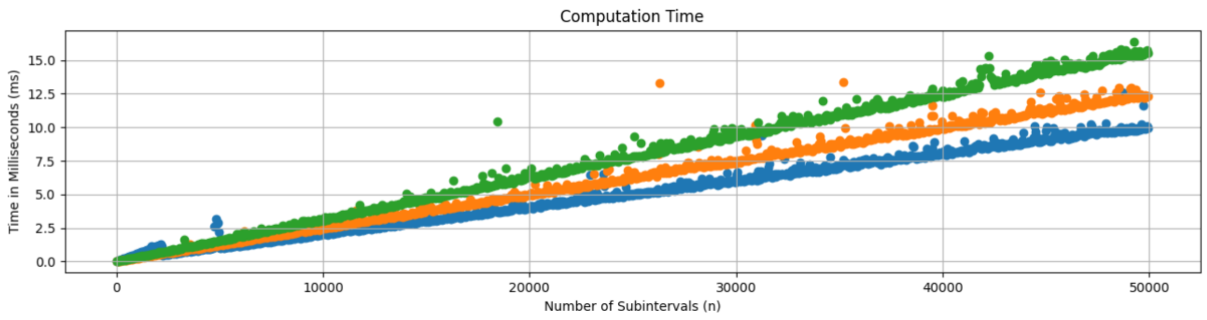
This code was run on four randomly chosen functions and randomly chosen limits.

4.2. Results

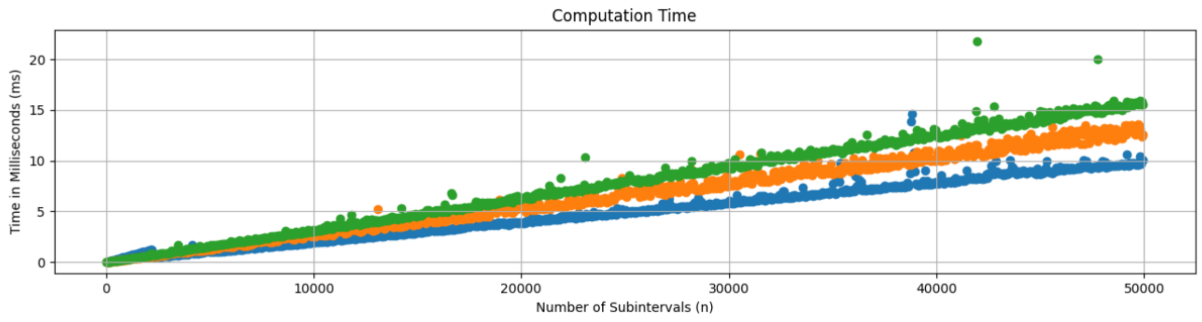
$$\int_1^{10} x^2 \, dx$$



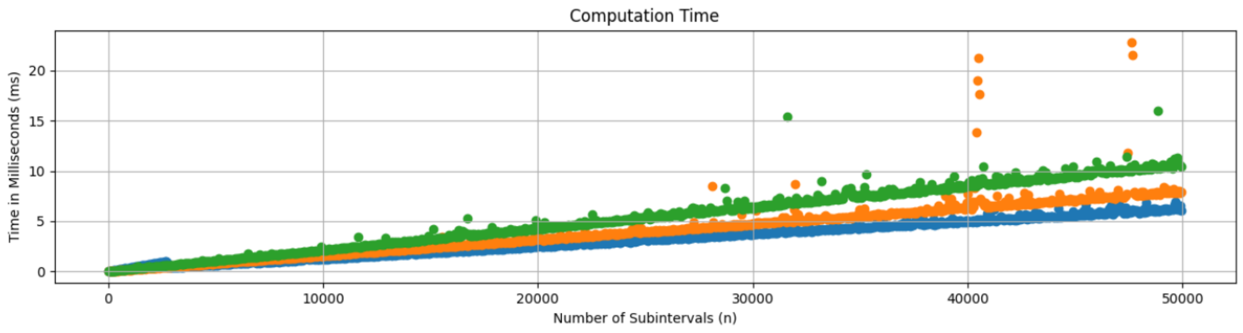
$$\int_1^{10} sin(x)\ dx$$



$$\int_1^{10} e^x\ dx$$



$$\int_1^{10} 2\ dx$$



5. Error analysis

In the final part we compared the algorithms on the basis of how precise they were under various conditions.

5.1. Methodology

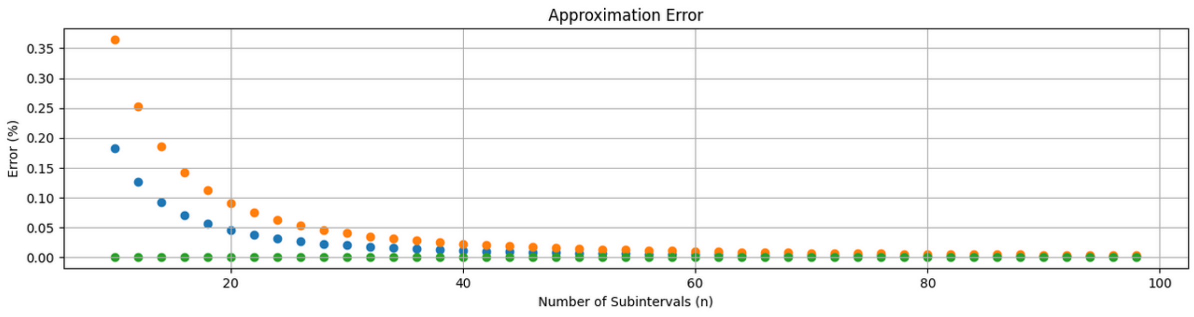
To calculate the error, we chose definite integrals which could be calculated algebraically. After that we run the algorithm and calculate the difference between the expected and measured value.

Implementation:

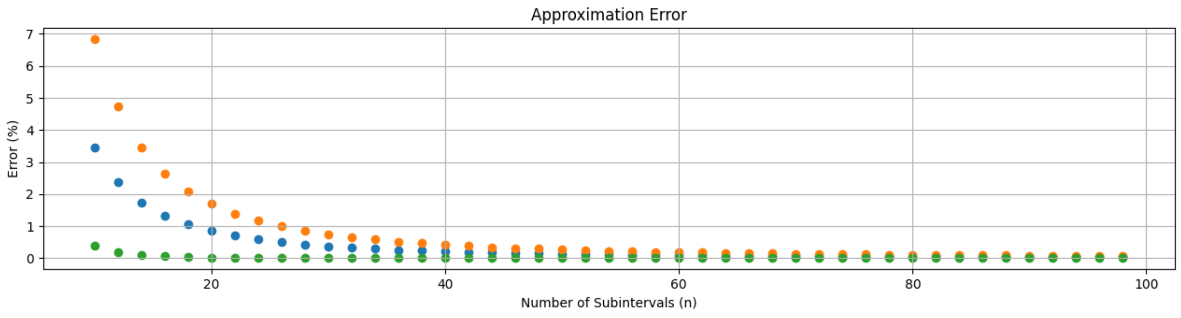
```
def f(x):
    return (x*x)

def expected(a,b):
    return (b**3)/3 - (a**3)/3

def midpoint_rule_sum(f,a,b,n):
    l = numpy.linspace(a,b,n)
    sum = 0
    for i in range(n-1):
        sum = sum + ((l[i+1]-l[i])* f ((l[i+1]+l[i]) /2))
    error = expected(a,b)-sum
    return sum, error
```

5.2. Results

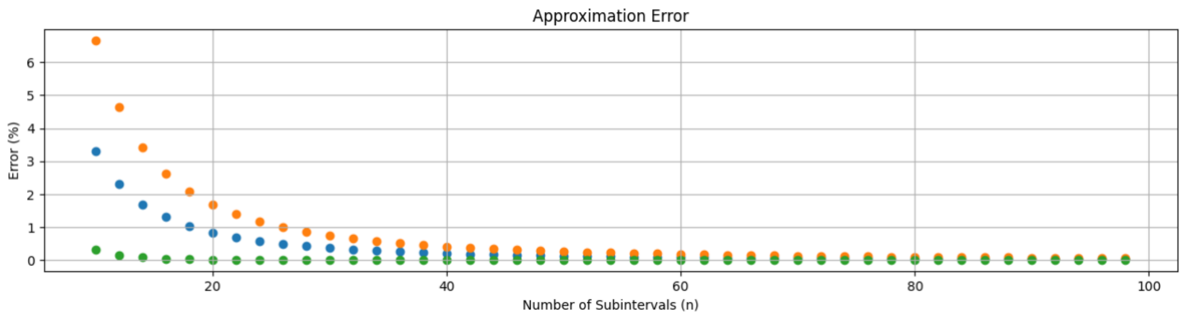
$$\int_1^{10} x^2 \; dx$$



$$\int_1^{10} sin(x) \; dx$$



$$\int_1^{10} e^x \, dx$$



5.3. Algebraic Error Bounds

Apart from comparing the precision of algorithm computationally, we can also work on deriving hard error bounds on the three algorithms.

Assume $f(x)$ is continuous on the interval $[a, b]$
Let n be a positive integer, then $\Delta x = \frac{b-a}{n}$
If $[a, b]$ is divided into n sub intervals of length.

$$\text{Midpoint rule error bound } |E_M| \leq \frac{K(b-a)^3}{24n^2} \qquad |f''(x)| \leq K \tag{1}$$

$$\text{Trapezoidal rule error bound } |E_T| \leq \frac{K(b-a)^3}{12n^2} \qquad |f''(x)| \leq K \tag{2}$$

$$\text{Simpson's rule error bound } |E_S| \leq \frac{K(b-a)^5}{180n^4} \qquad \left|f^{(4)}(x)\right| \leq K \tag{3}$$

6. Derivation for Error Bound of Midpoint Rule

Assume$f(x)$is continuous on the interval$[a, b]$

Let n be a positive integer, then$\Delta x = \frac{b-a}{n}$

If$[a, b]$is divided into n sub intervals of length.

$$\int_a^b f(x)dx = M_n + E_m$$

$$E_m = \int_a^b f(x)dx - M_n$$

Let us consider a single interval with bound$x_i and x_{i+1}$

Let the length of the interval i.e.$x_{i+1} - x_i = h$

$$\text{Midpoint} = x_i + \frac{h}{2}$$

$$\text{Height of the interval} = f(x_i + \frac{h}{2})$$

$$M_i = h \cdot f(x_i + \frac{h}{2})$$

$$\int_{x_i}^{x_{i+1}} f(x)dx = M_i + E_i$$

Let$t = x - x_i$, then$at x = x_i, t = 0$and$at x = x_{i+1}, t = h$

$$\implies \int_0^h f(t + x_i)dt$$

$Let g(t) = f(t + x_i)$(For ease of proving)

$$\int_0^h g(t)dt$$

We need the h/2 term so,

$$\int_0^h g(t)dt = \int_{\frac{h}{2}}^h g(t)dt + \int_0^{\frac{h}{2}}$$

$$g(t)dt$$

Now we do integration by parts on two times so that we get double derivative of the function as it is more accurate than the single derivative.

$$\text{Lets first integrate} \int_0^{\frac{h}{2}} g(t)dt by parts$$

$$\int_0^{\frac{h}{2}} g(t)dt = g(t)(t + C_1)]_0^{\frac{h}{2}} + \int_0^{\frac{h}{2}} (t + C_1)g'(t)dt$$

We do integration by parts again,

$$\int_0^{\frac{h}{2}} g(t)dt = g(t)(t + C_1)]_0^{\frac{h}{2}} - (\frac{(t + C_1)^2}{2} + C_2)g'(t)]_0^{\frac{h}{2}} + \int_0^{\frac{h}{2}} (\frac{(t + C_1)}{2} + C_2)g"(t)$$

Similarly,

$$\int_{\frac{h}{2}}^h g(t)dt = g(t)(t + C_3)]_{\frac{h}{2}}^h - (\frac{(t + C_3)^2}{2} + C_4)g'(t)]_{\frac{h}{2}}^h + \int_{\frac{h}{2}}^h (\frac{(t + C_3)}{2} + C_4)g"(t)$$

$C_1, C_2, C_3 and C_4$are integral constants and cancel out so we can take whatever values we need

We need the g(t) terms to add up to the area

$$= g(t)(t + C_3)]_{\frac{h}{2}}^h + g(t)(t + C_1)]_{\frac{h}{2}}^h$$

$$= g(h/2)(h/2 + C_1) - g(0)(C_1) + g(h)(h + C_1) - g(h/2)(h/2 + C_3)$$

$$Lets take C_1 = 0 and C_2 = h/2$$

$$= g(h/2)(h/2) - g(h/2)(-h/2)$$

$$= h \cdot g(\frac{h}{2}) = M_i$$

Now we need the g'(t) terms to become 0

$$= -(\frac{t^2}{2} + C_4)g'(t)]^{h}_{\frac{h}{2}} - (\frac{(t-h)^2}{2} + C_4)g'(t)]^{h}_{\frac{h}{2}}$$

Lets simplify and take $C_2 = C_4 = 0$

Now what is remaining is

$$\int_0^h g(t)dt = h \cdot g(\frac{h}{2}) + \int_0^{\frac{h}{2}} (\frac{(t)}{2})g"(t)dt + \int_{\frac{h}{2}}^h (\frac{(t-h)}{2})g"(t)dt$$

$$\implies E_i = \int_0^{\frac{h}{2}} (\frac{(t)}{2})g"(t)dt + \int_{\frac{h}{2}}^h (\frac{(t-h)}{2})g"(t)dt g(t) = f(t+x_i)$$

$$\implies E_i = \int_0^{\frac{h}{2}} (\frac{(t)}{2})f"(t+x_i)dt + \int_{\frac{h}{2}}^h (\frac{(t-h)}{2})f"(t+x_i)dt$$

$$E_m = E_1 + E_2 + \dots + E_{n-1}$$

$$E_m = \int_0^{\frac{h}{2}} (\frac{(t)}{2})f"(t+x_0)dt + \int_{\frac{h}{2}}^h (\frac{(t-h)}{2})f"(t+x_0)dt + \dots + \int_0^{\frac{h}{2}} (\frac{(t)}{2})f"(t+x_{n-1})dt + \int_{\frac{h}{2}}^h (\frac{(t-h)}{2})f"(t+x_{n-1})dt$$

$$E_m = \int_0^{\frac{h}{2}} (\frac{(t)}{2})f"(t+x_0) + ... + f"(t+x_{n-1})dt + int^h_{\frac{h}{2}}(\frac{(t-h)}{2})f"(t+x_0) + ... + f"(t+x_{n-1})dt$$

$$\text{Let} k \geq |max(f"(x))|$$

$$\implies |E_m| \leq \int_0^{\frac{h}{2}} (\frac{(t)}{2})nkdt + int^h_{\frac{h}{2}}(\frac{(t-h)}{2})nkdt |E_m| \leq \frac{nkh^3}{24}$$

$$\text{h for the entire interval} = \frac{b-a}{3}$$

$$\implies |E_m| \leq \frac{nk(b-a)^3}{24n^3}$$

$$\implies |E_m| \leq \frac{k(b-a)^3}{24n^2}$$

7. Conclusion

→ All the algorithms have the same time complexity O(n) i.e. Linear which is fairly good.
→ In terms of run-time, Simpsons is the fastest algorithm followed by Trapezoid and Midpoint rule.
→ The run-time plots for all three algorithms are linear, which confirms our guess for time complexity.
→ In the error section, we see that the Simpsons is the most precise followed by Midpoint Trapezoid however the error rate goes down exponentially as we increase the number of intervals.
→ The observation that Trapezoid rule is less precise than midpoint rule can be explained through algebraically through error bounds.

8. References

1. Davies, A. (2021, December 4). How is Numerical Integration Done Using Python? [web log]. Retrieved December 14, 2021, from https://python.plainenglish.io/how-is-numerical-integration-done-using-python-4585344e5800.
2. Deriving the Trapezoidal Rule Error. Some Additional Material On Integration (UCSanDiego). (n.d.). Retrieved December 14, 2021, from http://math.ucsd.edu/ ebender/20B/77_Trap.pdf
3. Bounding the error in the midpoint rule for numerical integration (UPenn). (n.d.). Retrieved December 14, 2021, from https://www2.math.upenn.edu/ rimmer/math104/midpoint.pdf
4. Hive.blog. 2021. Approximate Integration: Midpoint Rule Error Bound: Proof — Hive. [online] Available at: <https://hive.blog/mathematics/@mes/approximate-integration-midpoint-rule-error-bound-proof> [Accessed 19 December 2021].
5. Tutorialspoint.com. 2021. Algorithms and Complexities. [online] Available at: <https://www.tutorialspoint.com/Algor and-Complexities> [Accessed 19 December 2021].