Numerical Integration & Differentiation Lab

E/20/168 – Jayasinghe BVRR

15 June 2024

Important: All the python code has been attached as a separate pdf. Please refer that.

Numerical Quadrature

Q1: Explain that the absolute error for the composite trapezoidal rule decays at the rate of $1/n^2$, and the absolute error for the composite Simpson's rule decays at the rate of $1/n^4$, where n is the number of subintervals.

Composite Trapezoidal Rule

The composite trapezoidal rule approximates the integral of a function $f(x)$ over an interval $[a, b]$ by dividing into the n number of intervals.

$$I_t = \frac{h}{2}\left[f(a) + 2\sum_{i=1}^{n-1} f(x_i) + f(b)\right] \quad ; \ where\ h = \frac{b-a}{n}$$

Error E_t of the composite rule is

$$E_t = -\frac{b-a}{12} \times h^2 \times f``(\xi) = -\frac{(b-a)^3}{12n^2} \times f``(\xi)$$

Thus, the absolute error of the composite trapezoidal rule decays at $\frac{1}{n^2}$ rate.

Composite Simpson's Rule

This uses quadratic polynomials to approximate the functions over the sub intervals.

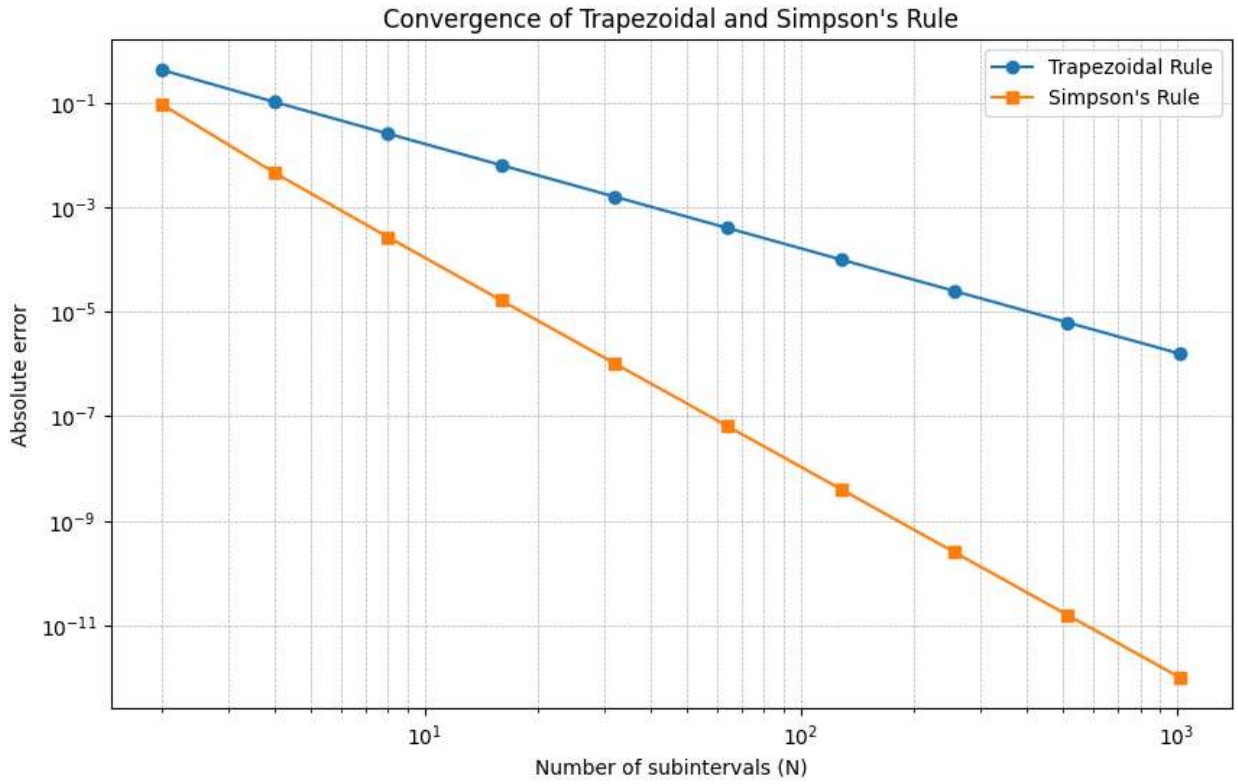
$$I_s = \frac{h}{3}\left[f(a) + 4\sum_{i=1,odd}^{n-1} f(x_i) + 2\sum_{i=2,even}^{n-2} f(x_i) + f(b)\right]$$

Error E_t of the composite Simpson's rule is

$$E_t = -\frac{b-a}{180} \times h^4 \times f^4(\xi) = -\frac{(b-a)^5}{180n^4} \times f^4(\xi)$$

This shows that as the number of subintervals n increases, the absolute error decreases proportionally to $\frac{1}{n^4}$.

Following graph shows a comparison between the convergences of trapizoidal and Simpson's rules.



Q2: Determine n that ensures the composite Simpson's rule approximates $\int_1^2 xlogx\ dx$ with an absolute error of at most 10^{-6}. Numerically verify if the value for n is reasonable.

Considering the error estimation of the composite Simpson's rule,

$E_s = -\frac{(b-a)}{180} h^4 f^4(\xi)$ and $|E_s| < 10^{-6}$,

By differentiating $f(x)$,

$$\max f^4(\xi) = \frac{2}{x^3} = \frac{2}{1} = 2$$

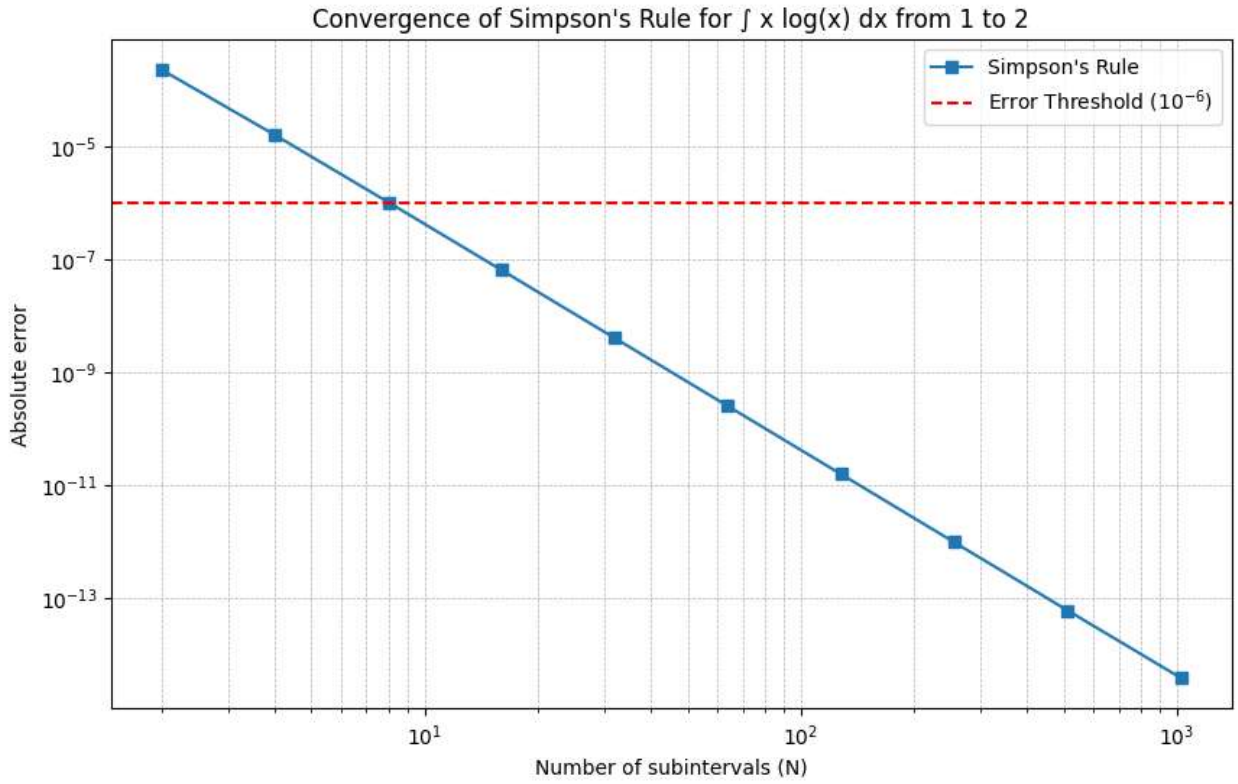
Therefore, $E_s\ max$ is,

$$|E_s| = \frac{h^4}{90}$$

$$h < (9 \times 10^{-5})^{\frac{1}{4}}$$

$$n > \frac{1}{(9 \times 10^{-5})^{\frac{1}{4}}}$$

So, minimum n should be 12.




```
1. Numerical integral: 0.6362945608313058
2. Exact integral: 0.6362943611198906
3. Absolute error: 1.9971141518304592e-07
```

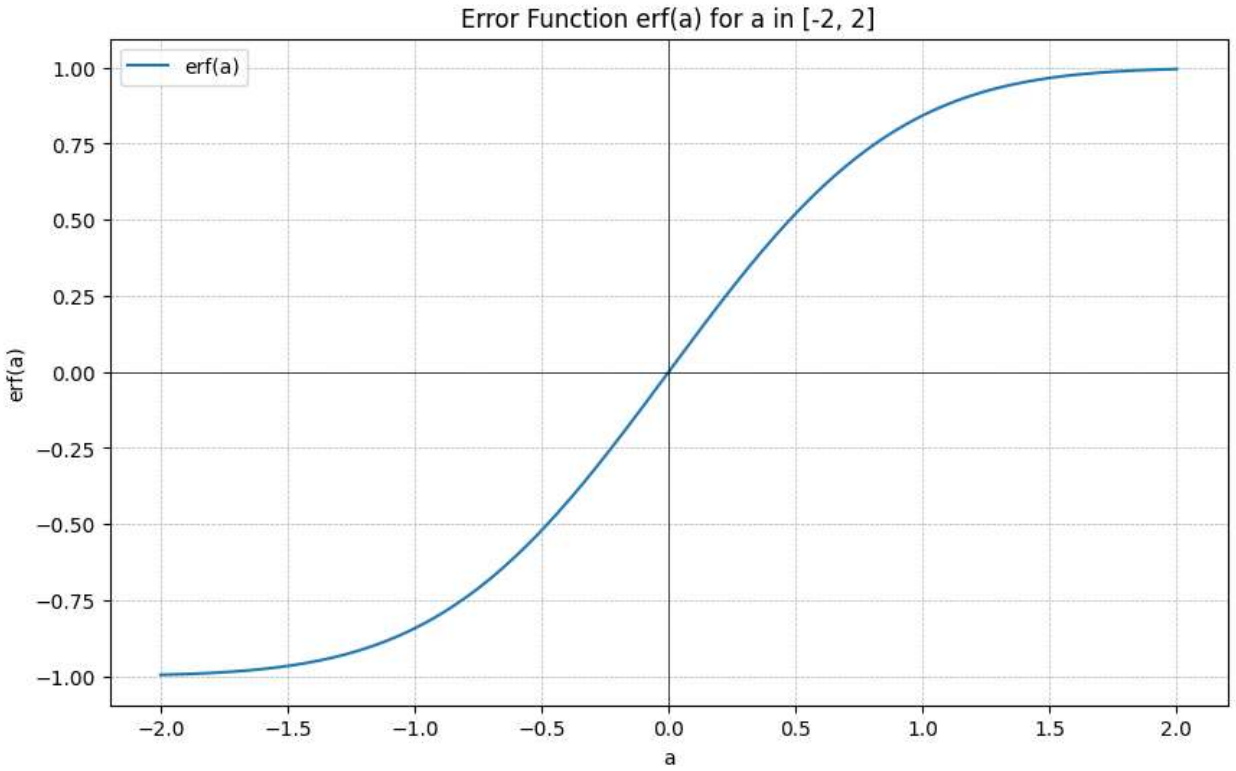
Q3: Consider the error function that occur in many areas of engineering and statistics: $erf(a) = \frac{2}{\sqrt{\pi}} \int_0^a e^{-x^2}\, dx$

a) Value of the integral for a = 1.5,

```python
# Define the integrand function
def integrand(x):
    return np.exp(-x**2)

# Define the upper limit of the integral
a = 1.5

# Compute the integral using scipy.integrate.quad
result, error = quad(integrand, 0, a)

# Compute the error function value
erf_value = (2 / np.sqrt(np.pi)) * result
```

```
1. The value of the integral from 0 to 1.5 is approximately: 0.8561883936249011
2. The value of the error function erf(1.5) is approximately: 0.9661051464753108
```

b)



c) *Evaluate the integral using the Gauss integration scheme with 2, 3, 4 and 5 integration points. Calculate the relative error in each case.*

```
Number of points: 2
Gauss-Legendre Integral: 0.8633384521923847
Relative Error: 0.008351034212472744

Number of points: 3
Gauss-Legendre Integral: 0.8556538986506874
Relative Error: 0.0006242726229337497

Number of points: 4
Gauss-Legendre Integral: 0.8562100942814074
Relative Error: 2.534565601202751e-05

Number of points: 5
Gauss-Legendre Integral: 0.8561877895596879
Relative Error: 7.055283833338084e-07
```

a) *Calculate the number of trapeziums required by the trapezoidal rule to produce an error similar to that produced by the 3-point Gauss integration scheme, then evaluate the integral using the trapezoidal rule with the obtained number of trapeziums.*

```
Maximum |f''(x)| on [0, 1.5]: 0.8925206405935732
Number of trapezoids required: 22
Trapezoidal Integral: 0.8560659282046451
Relative Error: 0.000143035599603806
```

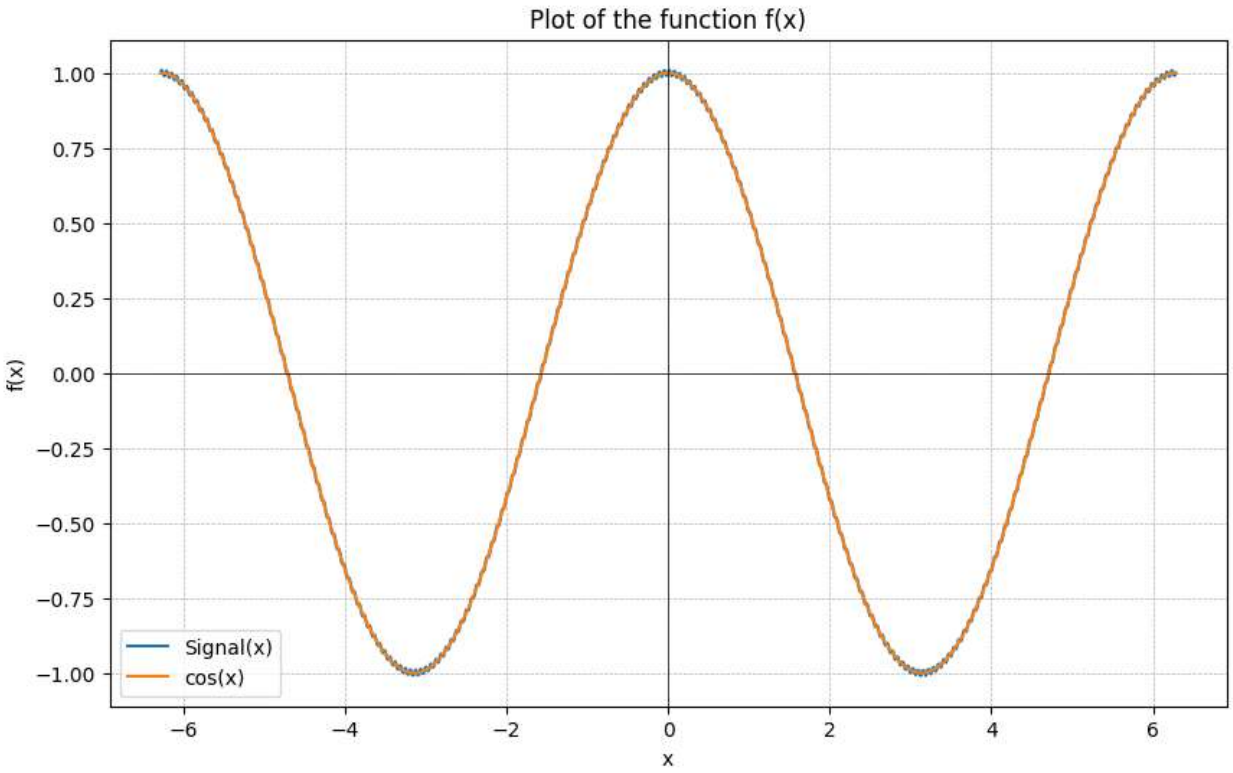
Numerical Differentiation

Q1: Use the forward difference formula with h = 10 -8 to estimate f'(2).

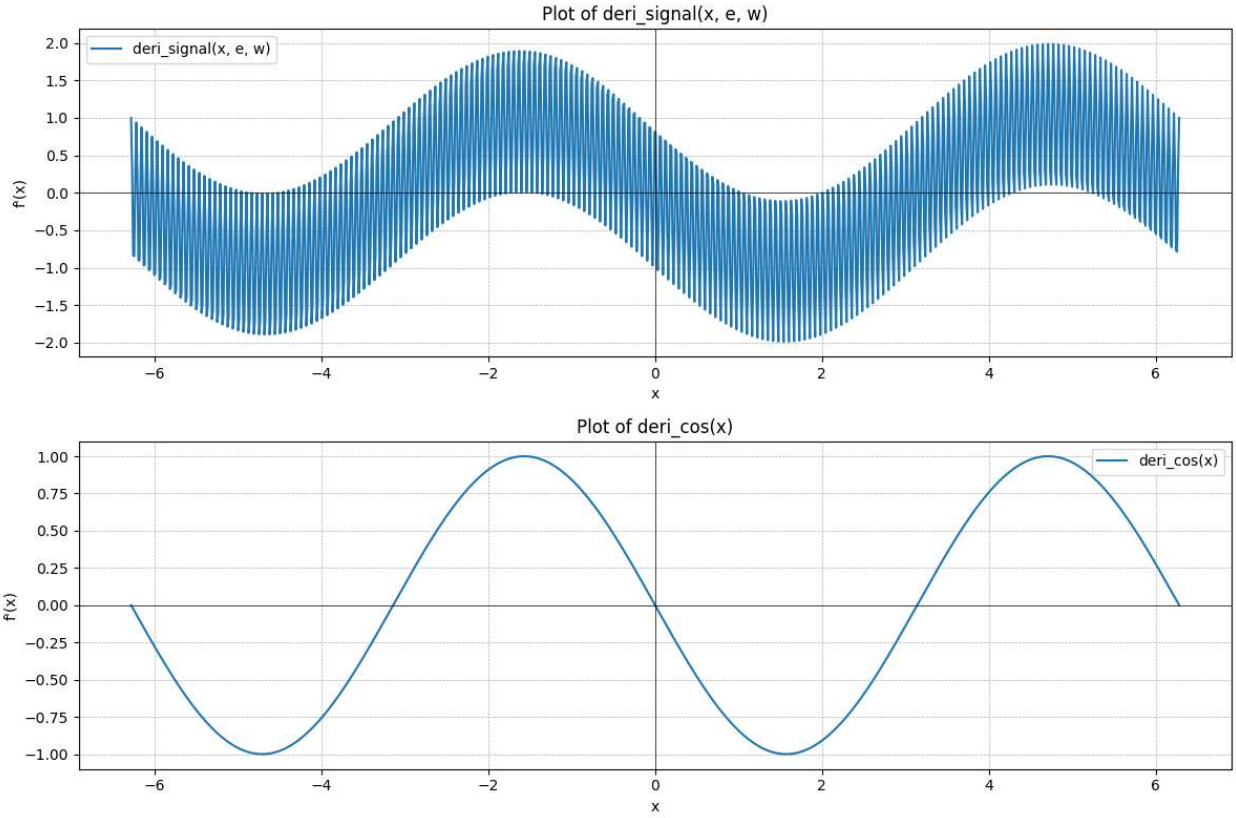
```
def forward_diff(f,x,h):
    return (f(x+h)-f(x))/h

x = 2
h = 1e-8
print(f"Estimated deri. f`(2): {forward_diff(f,x,h)}")
```

```
Estimated deri. f`(2): 3.079641430758784
```

Q2: plot in the same graph the function $f = cos(x)$ and the function with noise $f\epsilon, \omega(x) = cos(x) + \epsilon sin(\omega x)$.



plot the derivatives f' and $f'\epsilon, \omega$ and comment on your observations.



Based on the obtained results, the derivative of the signal exhibits rapid oscillations, indicating a high frequency of changes in its slope. In contrast, the derivative of cos(x) displays smooth oscillations, which correspond to the regular and predictable nature of its sine wave pattern. This difference highlights the variability and potential complexity in the signal's behavior compared to the steady periodic nature of the cosine function's derivative.

Q3: Suppose a formula for the first derivative using finite difference formula is given as: $f'(xi) \approx \frac{-f(xi+3)+9f(xi+1)-8f(xi)}{6h}$ Where the points $xi, xi+1, xi+2, xi+3$ are all equally spaced with step size h.

According to the Tayler's series,

$$f(x_i) = f_i$$

$$f(x_i + h) = f_i + \frac{hf_i'}{1!} + \frac{h^2 f_i''}{2!} + \frac{h^3 f_i'''}{3!} + O(h^4)$$

$$f(x_i + 2h) = f_i + \frac{2hf_i'}{1!} + \frac{4h^2 f_i''}{2!} + \frac{8h^3 f_i'''}{3!} + O(h^4)$$

$$f(x_i + 3h) = f_i + \frac{3hf_i'}{1!} + \frac{9h^2 f_i''}{2!} + \frac{27h^3 f_i'''}{3!} + O(h^4)$$

Using above expansions,

$$-f(x_{i+3}) + 9f(x_{i+1}) - 8f(x_i) = 6hf_i' - 3h^3 f_i''' + O(h^4)$$

$$\frac{-f(x_{i+3}) + 9f(x_{i+1}) - 8f(x_i)}{6h} = f_i' - \frac{h^2 f_i'''}{2} + O(h^3)$$

The major term of the truncation error is $\frac{h^2 f_i'''}{2}$. Therefore, this approximation is second-order accurate.

Q4: The following table gives the values of $f(x) = sin\, x$ .Estimate $f'(0.1)$; $f'(0.3)$ using an appropriate three-point formula.

X	f(x)
0.1	0.09983
0.2	0.19867
0.3	0.29552
0.4	0.38942

Just to find the derivative at $x = 0.1$, forward difference formular is used while at $x = 0.3$ backward difference formular is used.

```
# Forward difference formula for f'(0.1)
def forward_difference(f_values, h,i):
    f_prime = (-f_values[i+3] + 9*f_values[i+1] - 8*f_values[i]) / (6 * h)
    return f_prime

# Backward difference formula for f'(0.3)
def backward_difference( f_values, h,i):
    f_prime = (3*f_values[i] - 4*f_values[i-1] + 3*f_values[i-2]) / (2 * h)
    return f_prime
```

Answers:

```
f'(0.1) ≈ 0.99995
f'(0.3) ≈ 1.95685
```