cs374-lab1

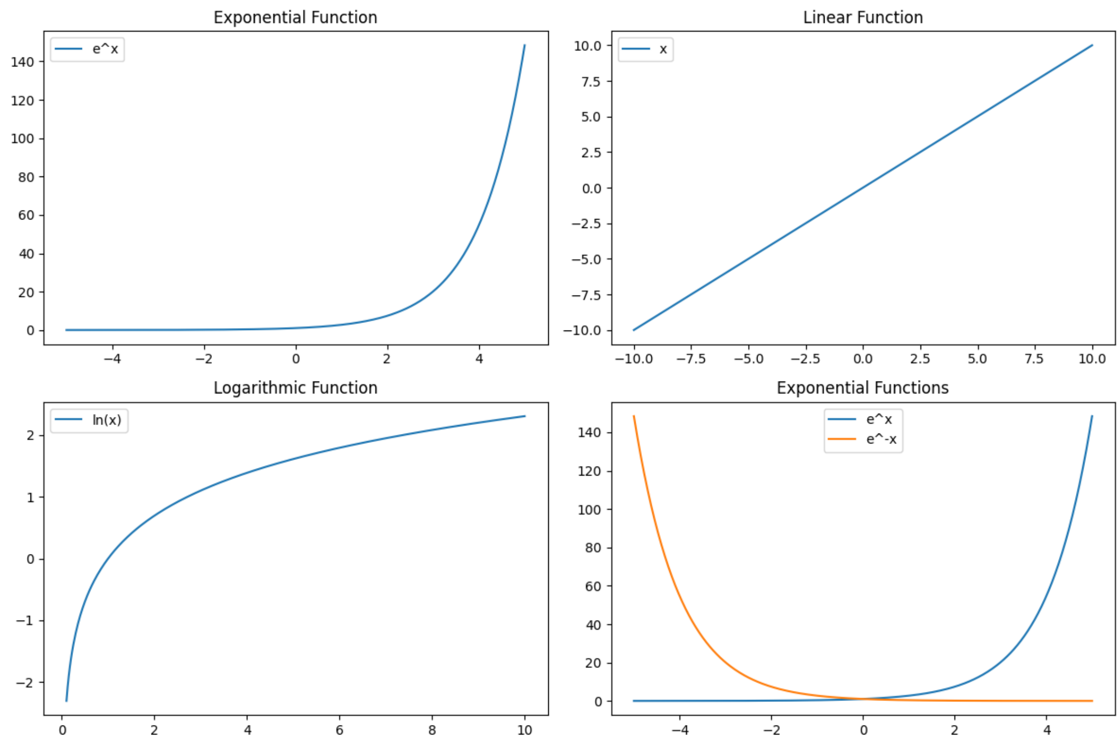
August 8, 2024

1 Basic Plotting Techniques

2 Question 1

With the help of a single code, plot the following functions: A. y=e^x B. y=x C. y=lnx Use suitable ranges of x for each of the functions and judge their properties on various scales of x. Extending this exercise, plot e±x on the same graph and compare them.

[ ]:
```
import numpy as np
import matplotlib.pyplot as plt

x_exp = np.linspace(-5, 5, 400)
x_linear = np.linspace(-10, 10, 400)
x_log = np.linspace(0.1, 10, 400)

y_exp = np.exp(x_exp)
y_linear = x_linear
y_log = np.log(x_log)


fig, axs = plt.subplots(2, 2, figsize=(12, 8))

axs[0, 0].plot(x_exp, y_exp,label='e^x'); axs[0, 0].legend(); axs[0, 0].
 ↪set_title('Exponential Function')
axs[0, 1].plot(x_linear, y_linear,label='x'); axs[0, 1].legend(); axs[0, 1].
 ↪set_title('Linear Function')
axs[1, 0].plot(x_log, y_log,label='ln(x)'); axs[1, 0].legend(); axs[1, 0].
 ↪set_title('Logarithmic Function')
axs[1, 1].plot(x_exp, np.exp(x_exp), label='e^x'); axs[1, 1].plot(x_exp, np.
 ↪exp(-x_exp), label='e^-x'); axs[1, 1].legend(); axs[1, 1].
 ↪set_title('Exponential Functions')

fig.tight_layout()

plt.show()
```



3 Question 2

For a fixed parameter k, plot the function y = sin(kx) for a few suitably chosen values of k. What is the role of k in determining the profile of the function? Thereafter, for k **= 1, plot sinx and (sin(x))^2 on the same graph within − < x < . Compare both.**

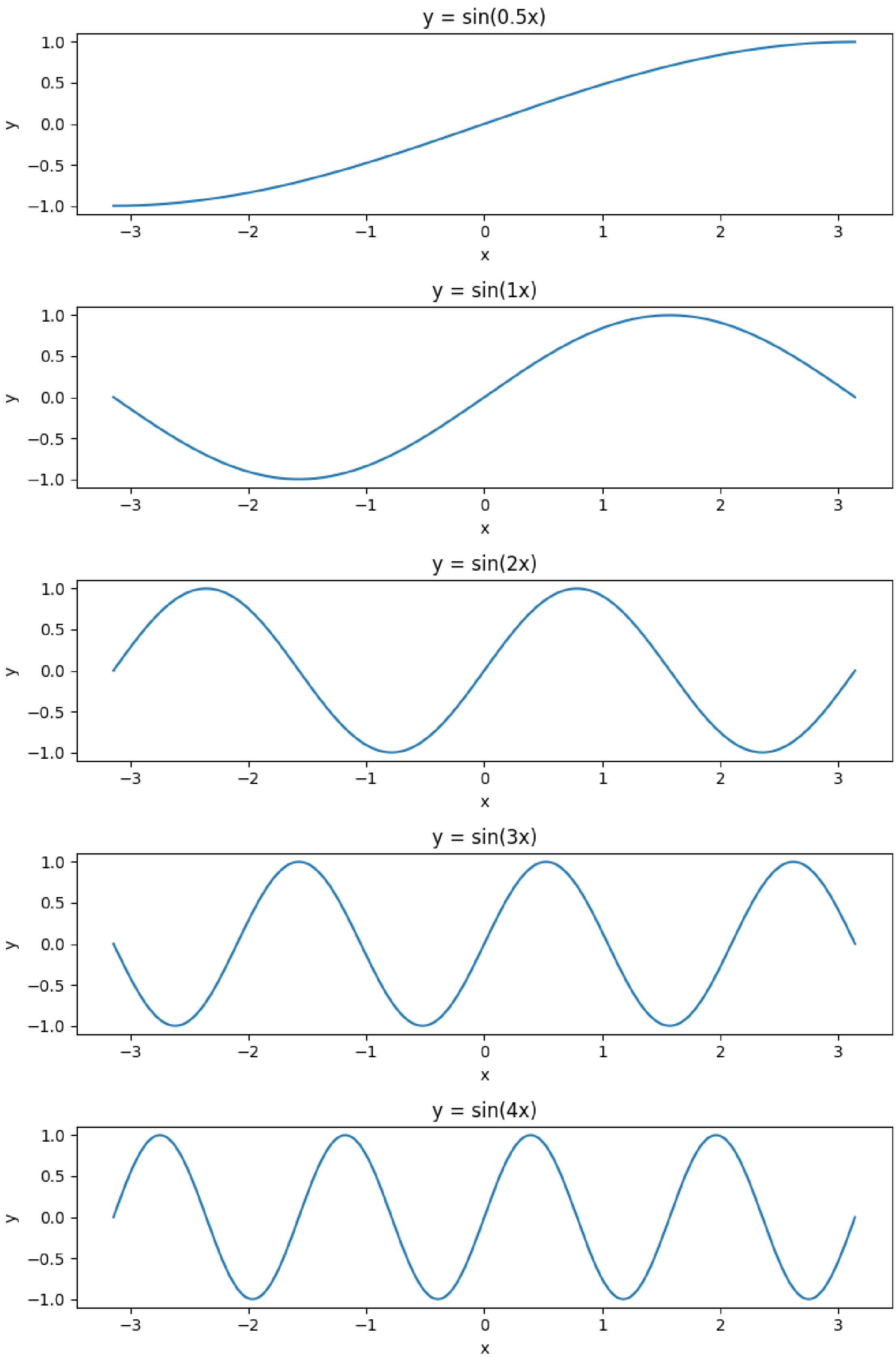
Answer

Role of k is that it determines the frequency of the sine wave. Higher k value results in higher frequency means it will oscillate more rapidly

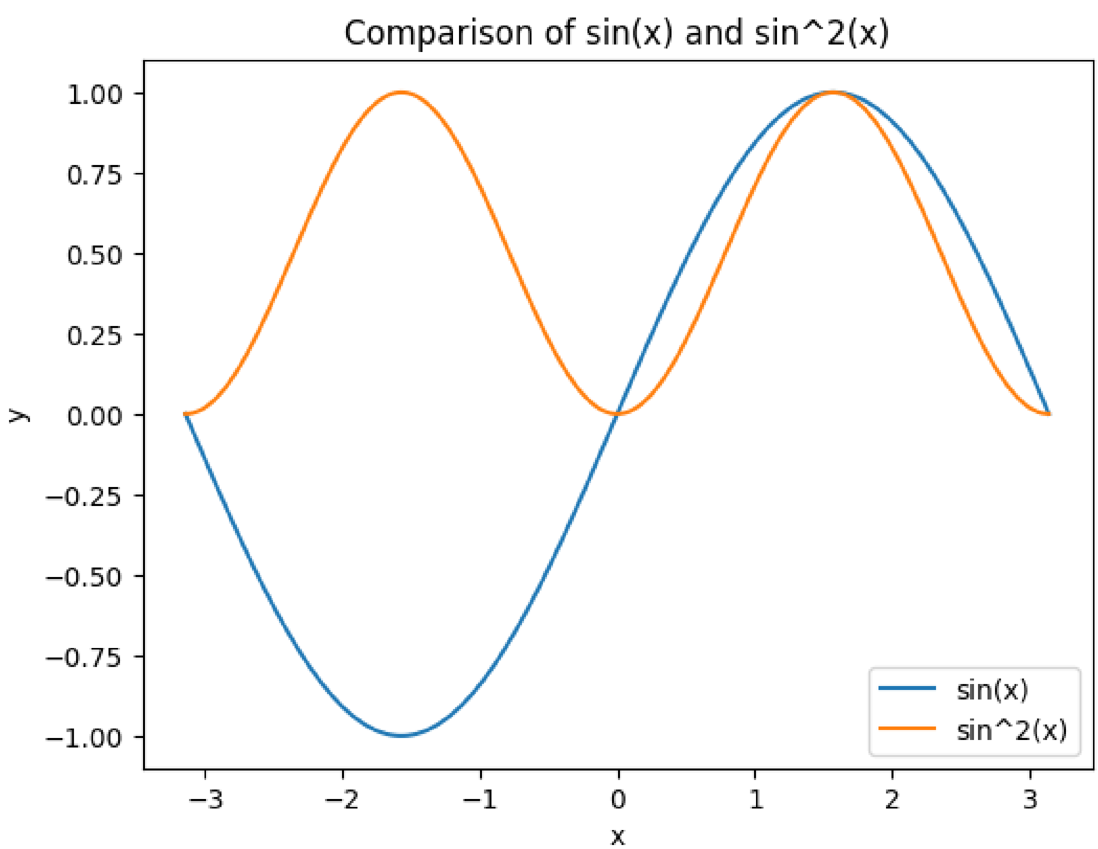
[ ]:

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


x = np.linspace(-np.pi, np.pi, 400)

k_values = [0.5, 1, 2, 3, 4]

fig, axs = plt.subplots(len(k_values), 1, figsize=(8, 12))

for i, k in enumerate(k_values):
    axs[i].plot(x, np.sin(k*x))
    axs[i].set_title(f"y = sin({k}x)")
```

```python
    axs[i].set_xlabel("x")
    axs[i].set_ylabel("y")


fig.tight_layout()

plt.show()
```


y = sin(0.5x)
1.0
0.5
0.0
−0.5
−1.0
y
−3
−2
−1
0
1
2
3
x
y = sin(1x)
1.0
0.5
0.0
−0.5
−1.0
y
−3
−2
−1
0
1
2
3
x
y = sin(2x)
1.0
0.5
0.0
−0.5
−1.0
y
−3
−2
−1
0
1
2
3
x
y = sin(3x)
1.0
0.5
0.0
−0.5
−1.0
y
−3
−2
−1
0
1
2
3
x
y = sin(4x)
1.0
0.5
0.0
−0.5
−1.0
y
−3
−2
−1
0
1
2
3
x

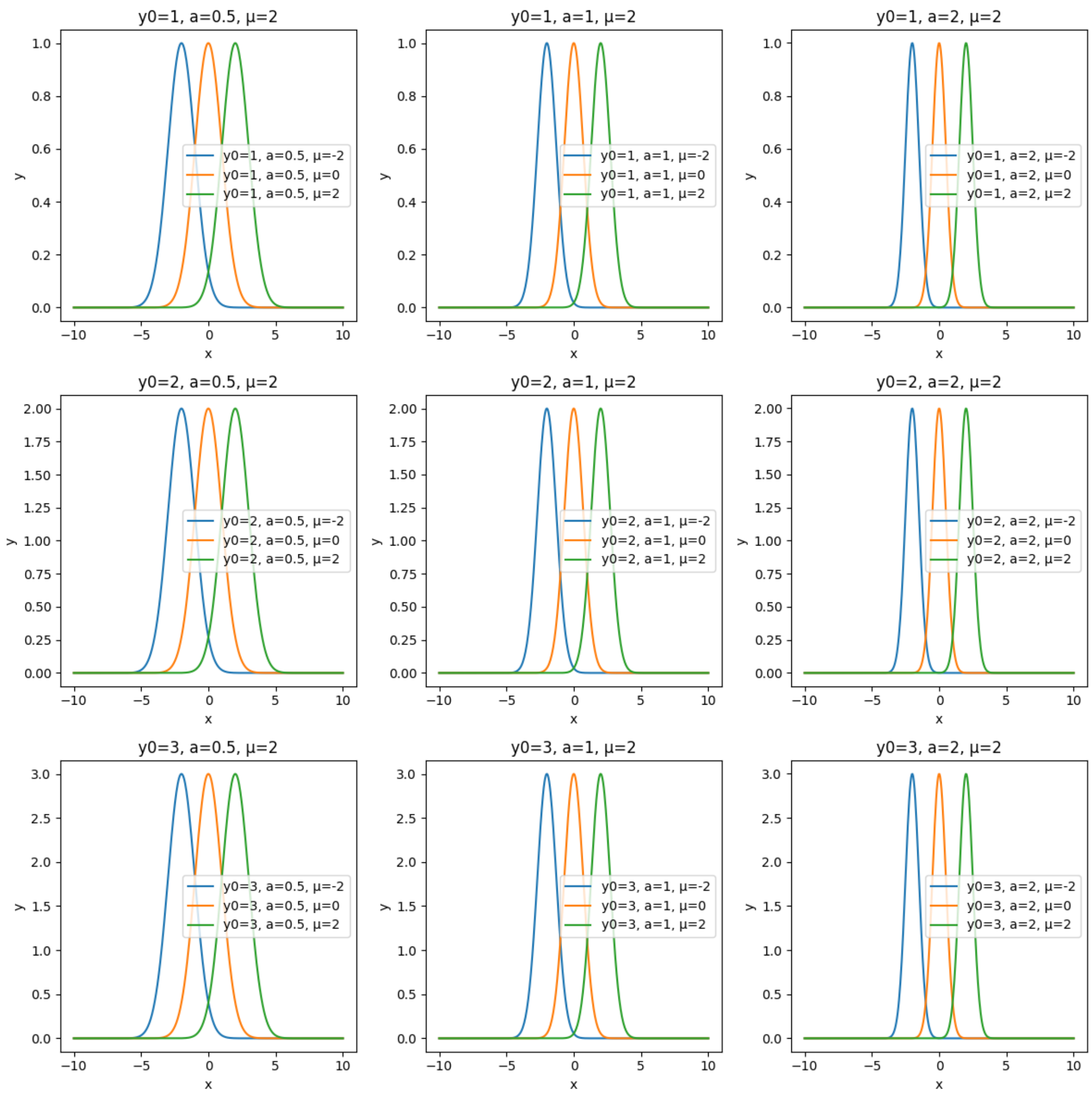
[ ]:
```
import numpy as np
import matplotlib.pyplot as plt


x = np.linspace(-np.pi, np.pi, 400)


plt.plot(x, np.sin(x), label="sin(x)")
plt.plot(x, np.sin(x)**2, label="sin^2(x)")
plt.xlabel("x")
plt.ylabel("y")
plt.title("Comparison of sin(x) and sin^2(x)")
plt.legend()

plt.show()
```



4 Question 3

Plot the Gaussian function y = y0$^{e-a(x-)}$2 for a few suitably chosen values of the fixed parameters y0, a and . Examine the shifting profile of the function, with changes in the parameters (= vt simulates a single wave pulse, like a tsunami, travelling with a velocity v). Then for y0 = a = 1 and = 0, consider a first-order expansion of the Gaussian function to obtain the Lorentz function. Plot both of them together and compare their behaviour For every value of x take the difference between the two functions and plot it against x over 0 < x < 10.

[ ]:
```
import numpy as np
import matplotlib.pyplot as plt


x = np.linspace(-10, 10, 400)


y0_values = [1, 2, 3]
a_values = [0.5, 1, 2]
mu_values = [-2, 0, 2]


fig, axs = plt.subplots(len(y0_values), len(a_values), figsize=(12, 12))


for i, y0 in enumerate(y0_values):
    for j, a in enumerate(a_values):
        for k, mu in enumerate(mu_values):
            axs[i, j].plot(x, y0*np.exp(-a*(x-mu)**2), label=f"y0={y0}, a={a},␣
↪ ={mu}")
            axs[i, j].set_title(f"y0={y0}, a={a},  ={mu}")
            axs[i, j].set_xlabel("x")
            axs[i, j].set_ylabel("y")
            axs[i, j].legend()


fig.tight_layout()

plt.show()
```

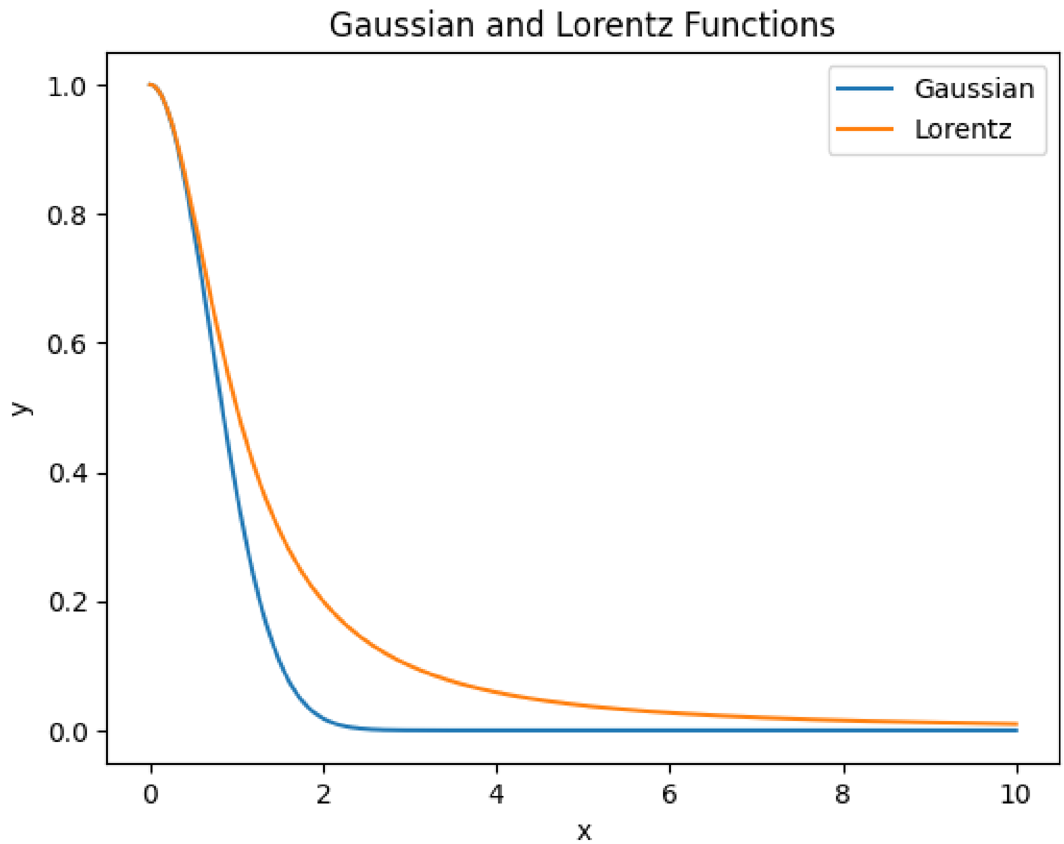


The nature of the graphs above can be justified by stating that 'a' value is basically the standard deviation term which defines the width of the bell yo defines the height and mu basically controls the shifting of the curve

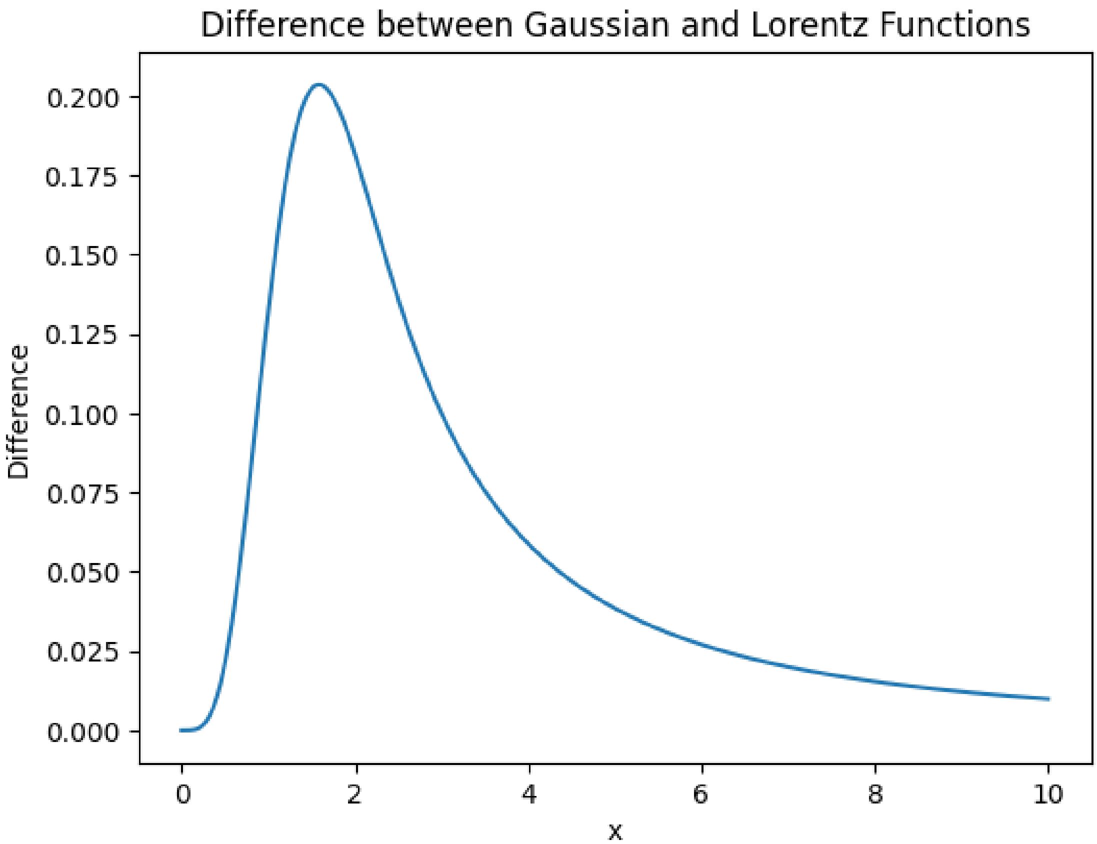
[ ]:

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

def gaussian(x, y0, a, mu):
    return y0 * np.exp(-a * (x - mu)**2)

def lorentz(x, y0, a, mu):
    return y0 / (1 + a * (x - mu)**2)
```

```python
y0 = 1
a = 1
mu = 0

x = np.linspace(0, 10, 400)

gaussian_values = gaussian(x, y0, a, mu)
lorentz_values = lorentz(x, y0, a, mu)

plt.plot(x, gaussian_values, label='Gaussian')
plt.plot(x, lorentz_values, label='Lorentz')
plt.xlabel('x')
plt.ylabel('y')
plt.title('Gaussian and Lorentz Functions')
plt.legend()
plt.show()

# Plot the difference between the two functions
difference = np.abs(gaussian_values - lorentz_values)

plt.plot(x, difference)
plt.xlabel('x')
plt.ylabel('Difference')
plt.title('Difference between Gaussian and Lorentz Functions')
plt.show()
```



Gaussian and Lorentz Functions
Gaussian
Lorentz
1.0
0.8
0.6
0.4
0.2
0.0
y
0
2
4
6
8
10
x

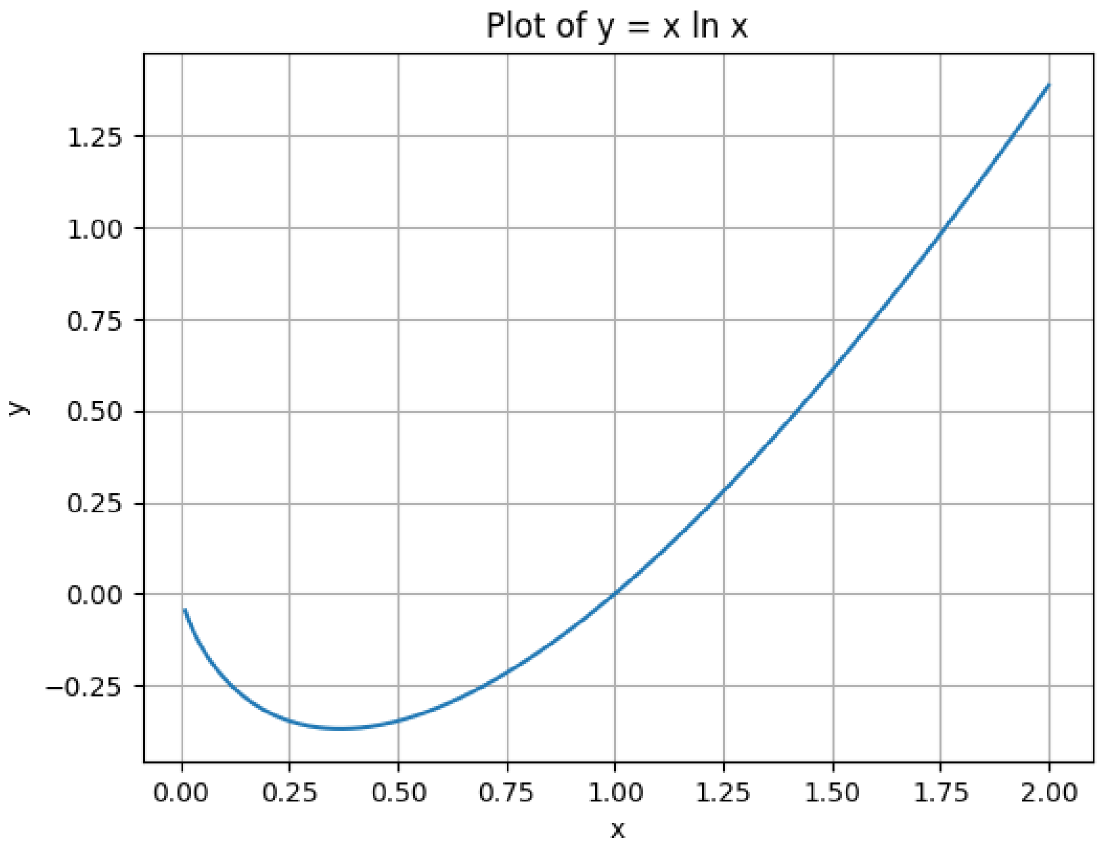


5 Question 4

Plot y = x ln x and carefully examine it for 0 < x < 2. Provide an analytical justification for what you observe. Also note the growth of the function for very large x.

[ ]:
```
import numpy as np
import matplotlib.pyplot as plt

# Define the x range
x = np.linspace(0.01, 2, 400)

y = x * np.log(x)

plt.plot(x, y)
plt.xlabel("x")
plt.ylabel("y")
plt.title("Plot of y = x ln x")
plt.grid(True)
```

```
plt.show()
```



We can justify this behaviour by seeing the derivate of the function i.e 1+ln(x).

At x=0 f'(x) is negative infinity and remains negative till x=1/e gradually becoming less negative.

Then for x=1/e it is zero and for x>1/e derivate is positive and graph increases rapidly.

As x approaches infinity, ln x grows logarithmically, but x grows exponentially. Therefore, the product x ln x grows exponentially, making the function grow very rapidly for very large x.

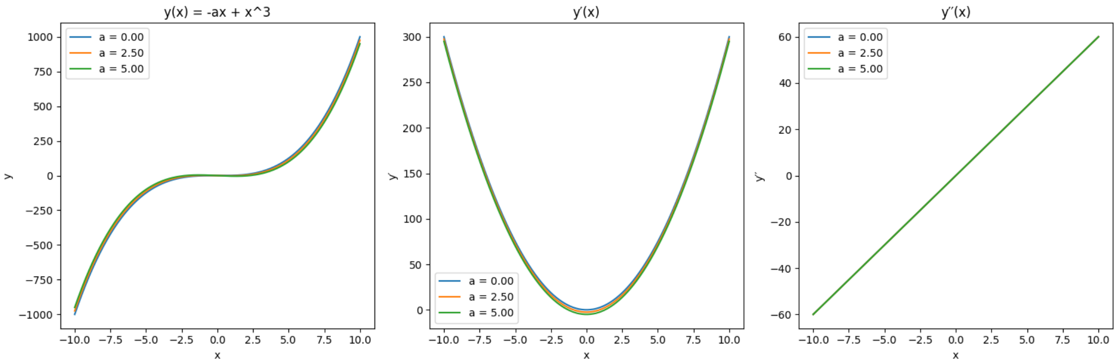
6 Question 5

Plot y(x), y (x) and y (x) for the following polynomial functions:A. y=−ax+x3 B.y=−ax2 +x4 Change a continuously over a suitable range of values (a 0) to observe the shift in the function profiles and their two derivatives. Carefully check all conditions for a = 0.

7 1. y=-ax+x^3

[7]:
```
import numpy as np
import matplotlib.pyplot as plt

def y(x, a):
    return -a*x + x**3

def y_prime(x, a):
    return -a + 3*x**2

def y_double_prime(x, a):
    return 6*x

a_vals = np.linspace(0, 5, 3)

fig, axs = plt.subplots(1, 3, figsize=(15, 5))


for a_val in a_vals:

    x_vals = np.linspace(-10, 10, 400)

    y_vals = y(x_vals, a_val)
    y_prime_vals = y_prime(x_vals, a_val)
    y_double_prime_vals = y_double_prime(x_vals, a_val)


    axs[0].plot(x_vals, y_vals, label=f"a = {a_val:.2f}")
    axs[1].plot(x_vals, y_prime_vals, label=f"a = {a_val:.2f}")
    axs[2].plot(x_vals, y_double_prime_vals, label=f"a = {a_val:.2f}")

axs[0].set_title("y(x) = -ax + x^3")
axs[0].set_xlabel("x")
axs[0].set_ylabel("y")
axs[1].set_title("y (x)")
axs[1].set_xlabel("x")
axs[1].set_ylabel("y ")
axs[2].set_title("y  (x)")
axs[2].set_xlabel("x")
axs[2].set_ylabel("y  ")

axs[0].legend()
axs[1].legend()
axs[2].legend()
```

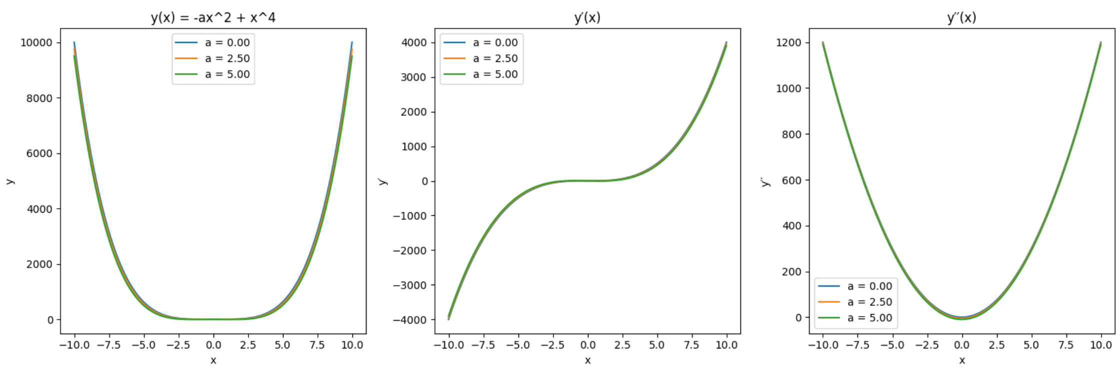
```
plt.tight_layout()
plt.show()
```



8 2. y=−ax2 +x4

[8]:
```
import numpy as np
import matplotlib.pyplot as plt

def y(x, a):
    return -a*(x**2) + x**4

def y_prime(x, a):
    return -2*a*x + 4*x**3

def y_double_prime(x, a):
    return -2*a + 12*x**2

a_vals = np.linspace(0, 5, 3)
fig, axs = plt.subplots(1, 3, figsize=(15, 5))


for a_val in a_vals:

    x_vals = np.linspace(-10, 10, 400)

    y_vals = y(x_vals, a_val)
    y_prime_vals = y_prime(x_vals, a_val)
    y_double_prime_vals = y_double_prime(x_vals, a_val)


    axs[0].plot(x_vals, y_vals, label=f"a = {a_val:.2f}")
    axs[1].plot(x_vals, y_prime_vals, label=f"a = {a_val:.2f}")
    axs[2].plot(x_vals, y_double_prime_vals, label=f"a = {a_val:.2f}")
```

```
axs[0].set_title("y(x) = -ax^2 + x^4")
axs[0].set_xlabel("x")
axs[0].set_ylabel("y")
axs[1].set_title("y (x)")
axs[1].set_xlabel("x")
axs[1].set_ylabel("y ")
axs[2].set_title("y  (x)")
axs[2].set_xlabel("x")
axs[2].set_ylabel("y  ")

axs[0].legend()
axs[1].legend()
axs[2].legend()


plt.tight_layout()
plt.show()
```



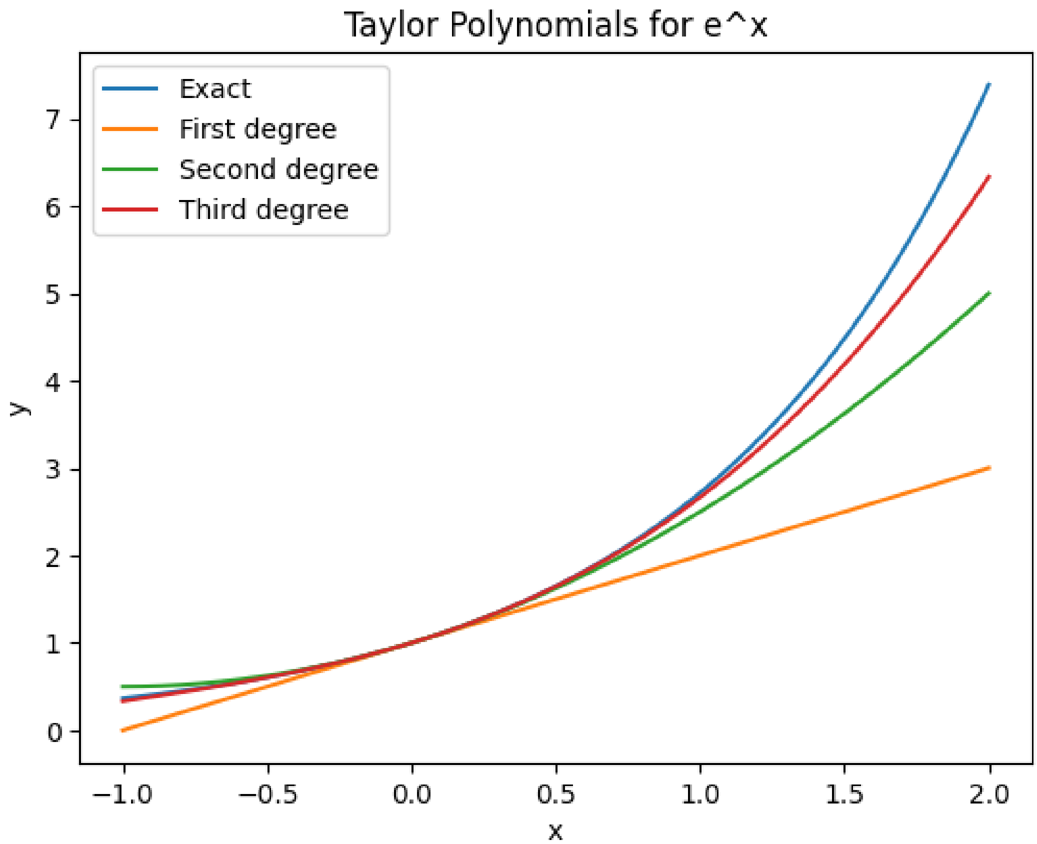
9 Taylor's Polynomial Exercise

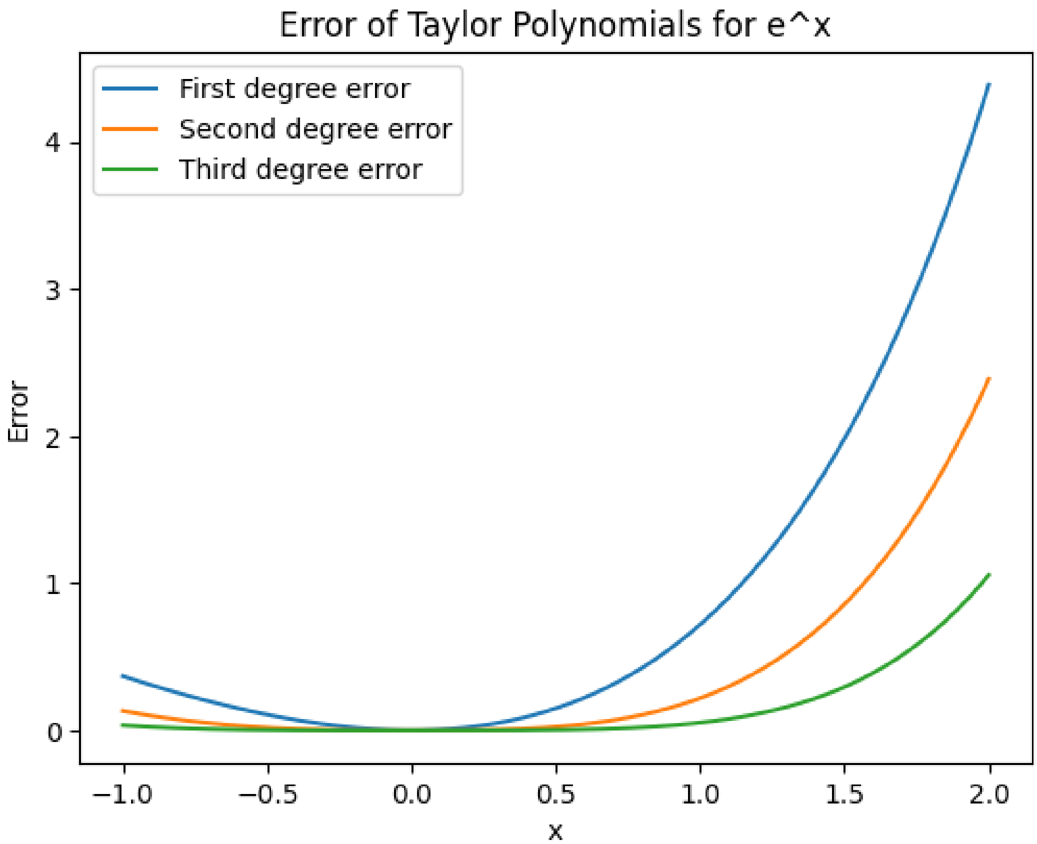
10 Question 1

11 e^x

[9]:
```
import numpy as np
import matplotlib.pyplot as plt
import math

def exp_taylor_series(x, a, n):
    result = 0
    for i in range(n+1):
        term = (x-a)**i / math.factorial(i)
```

```
        result += term
    return result

a = 0
x = np.linspace(-1, 2, 400)

exact_values = np.exp(x)
first_degree = exp_taylor_series(x, a, 1)
second_degree = exp_taylor_series(x, a, 2)
third_degree = exp_taylor_series(x, a, 3)

plt.plot(x, exact_values, label='Exact')
plt.plot(x, first_degree, label='First degree')
plt.plot(x, second_degree, label='Second degree')
plt.plot(x, third_degree, label='Third degree')
plt.xlabel('x')
plt.ylabel('y')
plt.title('Taylor Polynomials for e^x')
plt.legend()
plt.show()


x = np.linspace(-1, 2, 400)
exact_values = np.exp(x)
first_degree_errors = np.abs(exact_values - exp_taylor_series(x, a, 1))
second_degree_errors = np.abs(exact_values - exp_taylor_series(x, a, 2))
third_degree_errors = np.abs(exact_values - exp_taylor_series(x, a, 3))

plt.plot(x, first_degree_errors, label='First degree error')
plt.plot(x, second_degree_errors, label='Second degree error')
plt.plot(x, third_degree_errors, label='Third degree error')
plt.xlabel('x')
plt.ylabel('Error')
plt.title('Error of Taylor Polynomials for e^x')
plt.legend()
plt.show()
```

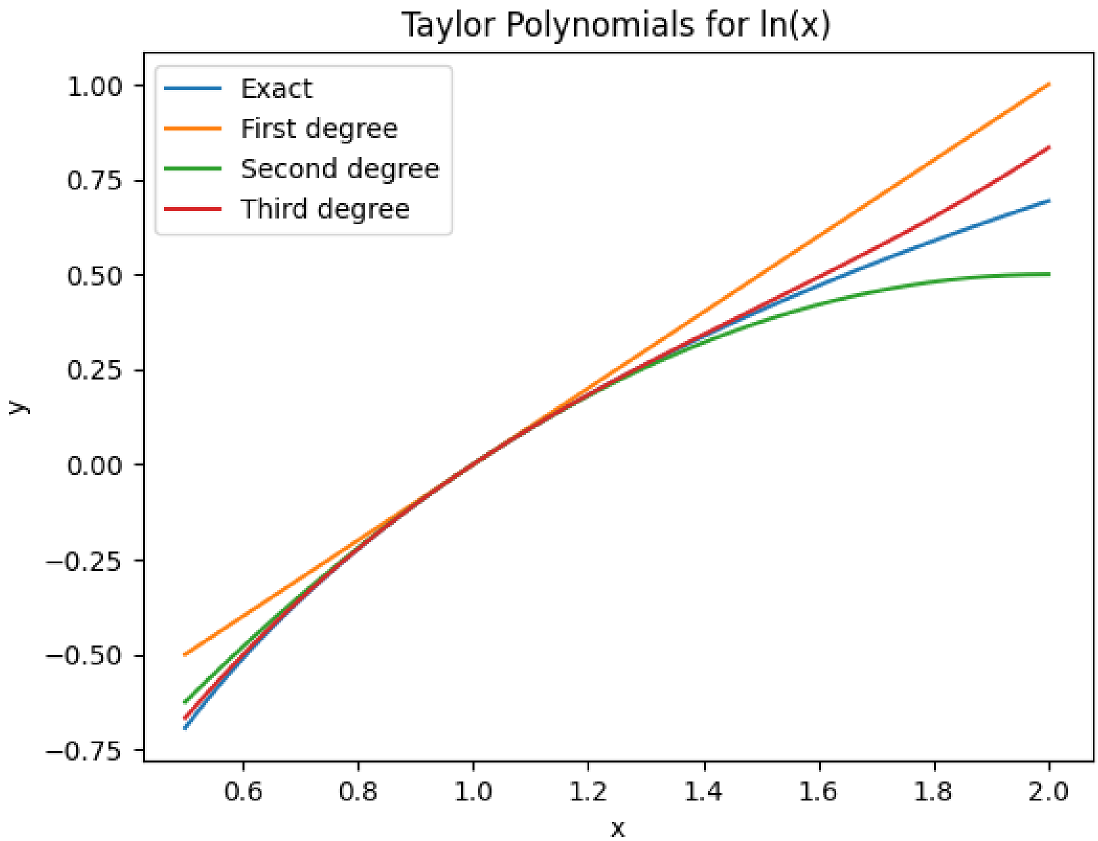

Taylor Polynomials for e^x
Exact
First degree
Second degree
Third degree
y
x
0
1
2
3
4
5
6
7
−1.0
−0.5
0.0
0.5
1.0
1.5
2.0

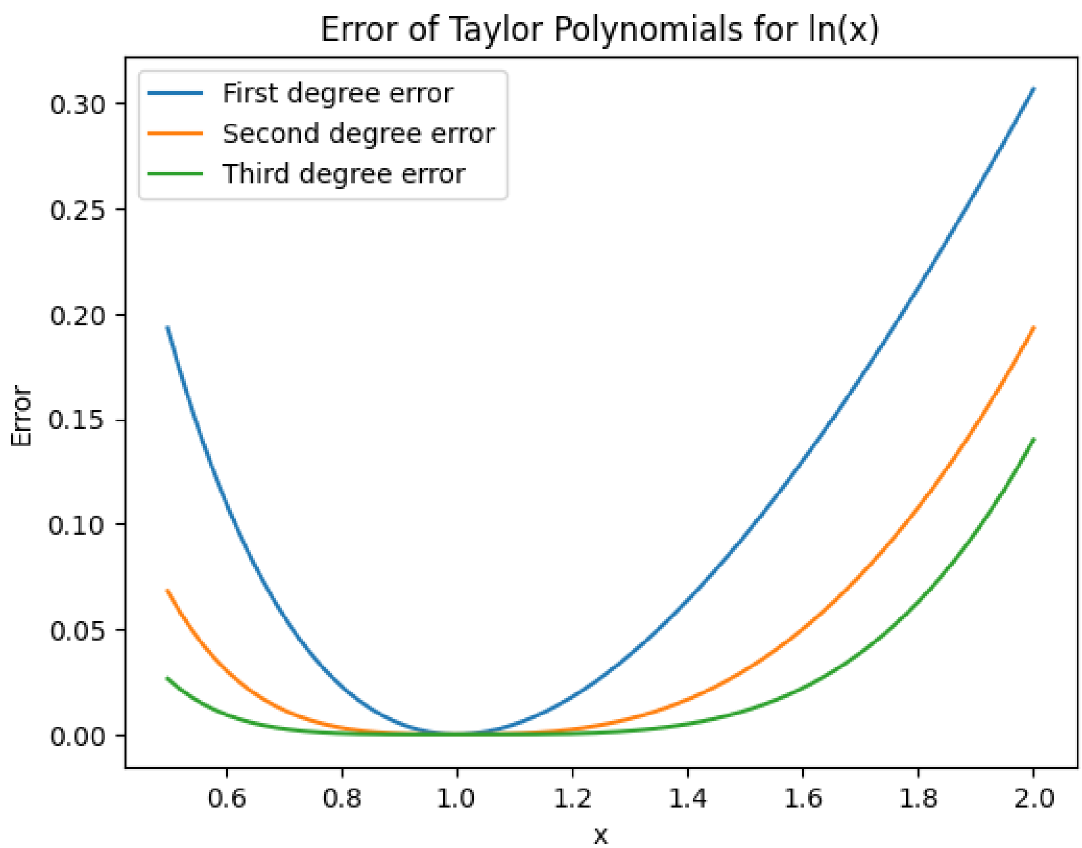


12 ln(x)

[10]:
```
import numpy as np
import matplotlib.pyplot as plt

def ln_taylor_series(x, a, n):
    result = 0
    for i in range(1, n+1):
        sign = (-1)**(i-1)
        term = ((x-a)**i) / (i * (a**i))
        result += sign * term
    return result + np.log(a)

a = 1
x = np.linspace(0.5, 2, 400)

exact_values = np.log(x)
first_degree = ln_taylor_series(x, a, 1)
second_degree = ln_taylor_series(x, a, 2)
```

```python
third_degree = ln_taylor_series(x, a, 3)

plt.plot(x, exact_values, label='Exact')
plt.plot(x, first_degree, label='First degree')
plt.plot(x, second_degree, label='Second degree')
plt.plot(x, third_degree, label='Third degree')
plt.xlabel('x')
plt.ylabel('y')
plt.title('Taylor Polynomials for ln(x)')
plt.legend()
plt.show()

x = np.linspace(0.5, 2, 400)
exact_values = np.log(x)
first_degree_errors = np.abs(exact_values - ln_taylor_series(x, a, 1))
second_degree_errors = np.abs(exact_values - ln_taylor_series(x, a, 2))
third_degree_errors = np.abs(exact_values - ln_taylor_series(x, a, 3))

plt.plot(x, first_degree_errors, label='First degree error')
plt.plot(x, second_degree_errors, label='Second degree error')
plt.plot(x, third_degree_errors, label='Third degree error')
plt.xlabel('x')
plt.ylabel('Error')
plt.title('Error of Taylor Polynomials for ln(x)')
plt.legend()
plt.show()
```

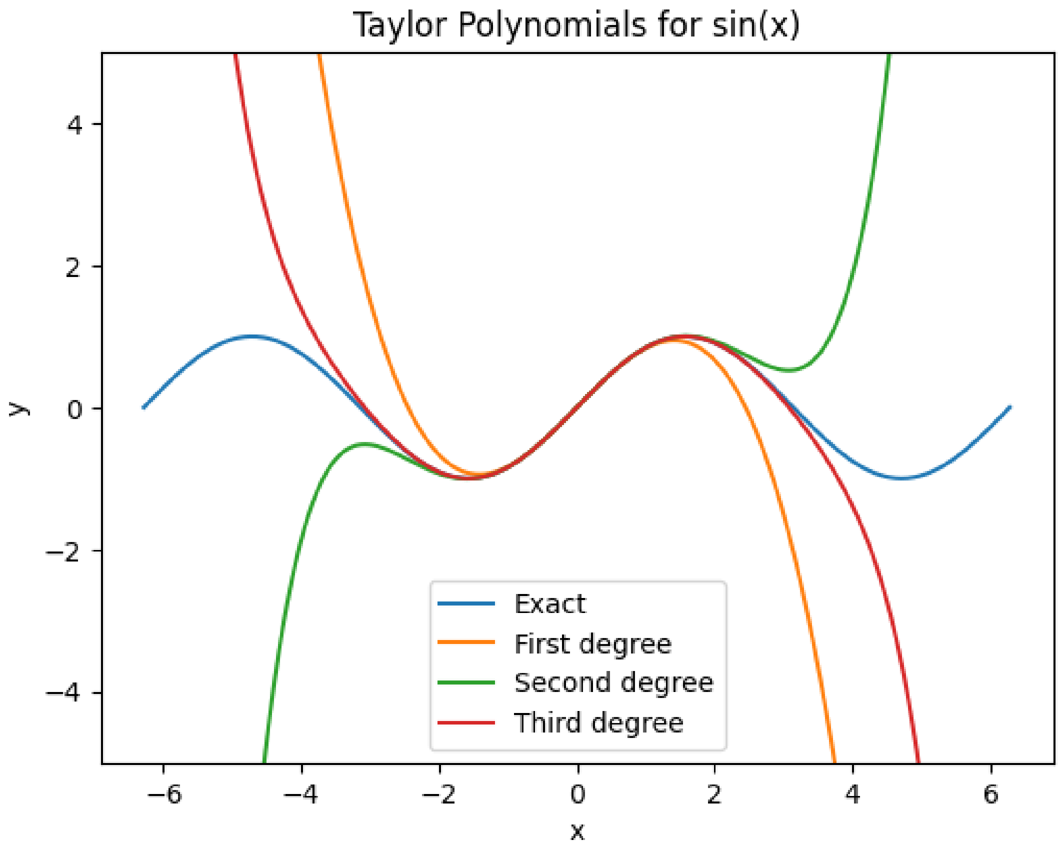

Taylor Polynomials for ln(x)
Exact
First degree
Second degree
Third degree
1.00
0.75
0.50
0.25
0.00
−0.25
−0.50
−0.75
y
0.6
0.8
1.0
1.2
1.4
1.6
1.8
2.0
x

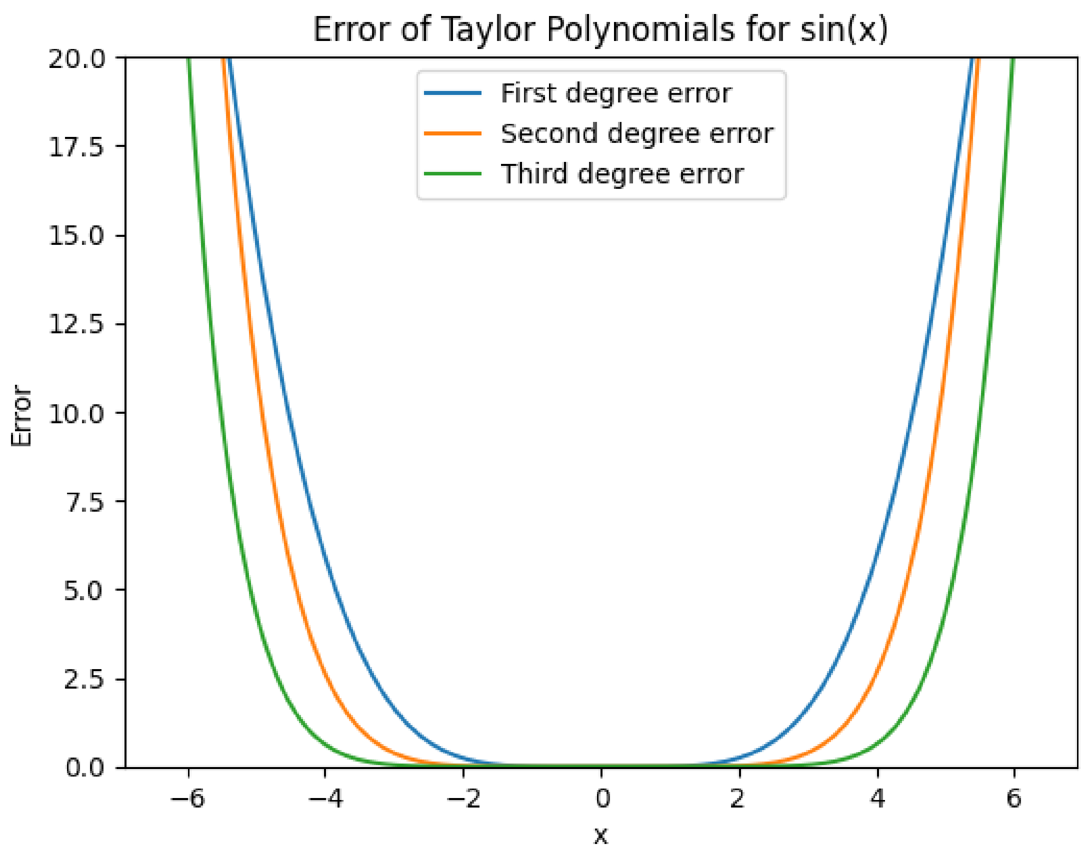


13 sin(x)

[13]:
```
import numpy as np
import matplotlib.pyplot as plt

def sin_taylor_series(x, a, n):
    result = 0
    for i in range(n+1):
        term = ((-1)**i) * (x-a)**(2*i+1) / math.factorial(2*i+1)
        result += term
    return result

a = 0
x = np.linspace(-2*np.pi, 2*np.pi, 400)

exact_values = np.sin(x)
first_degree = sin_taylor_series(x, a, 1)
second_degree = sin_taylor_series(x, a, 2)
third_degree = sin_taylor_series(x, a, 3)
```

```python
plt.plot(x, exact_values, label='Exact')
plt.plot(x, first_degree, label='First degree')
plt.plot(x, second_degree, label='Second degree')
plt.plot(x, third_degree, label='Third degree')
plt.xlabel('x')
plt.ylabel('y')
plt.title('Taylor Polynomials for sin(x)')
plt.ylim(-5,5)
plt.legend()
plt.show()


x = np.linspace(-2*np.pi, 2*np.pi, 400)
exact_values = np.sin(x)
first_degree_errors = np.abs(exact_values - sin_taylor_series(x, a, 1))
second_degree_errors = np.abs(exact_values - sin_taylor_series(x, a, 2))
third_degree_errors = np.abs(exact_values - sin_taylor_series(x, a, 3))

plt.plot(x, first_degree_errors, label='First degree error')
plt.plot(x, second_degree_errors, label='Second degree error')
plt.plot(x, third_degree_errors, label='Third degree error')
plt.xlabel('x')
plt.ylabel('Error')
plt.title('Error of Taylor Polynomials for sin(x)')
plt.legend()
plt.ylim(0,20)
plt.show()
```

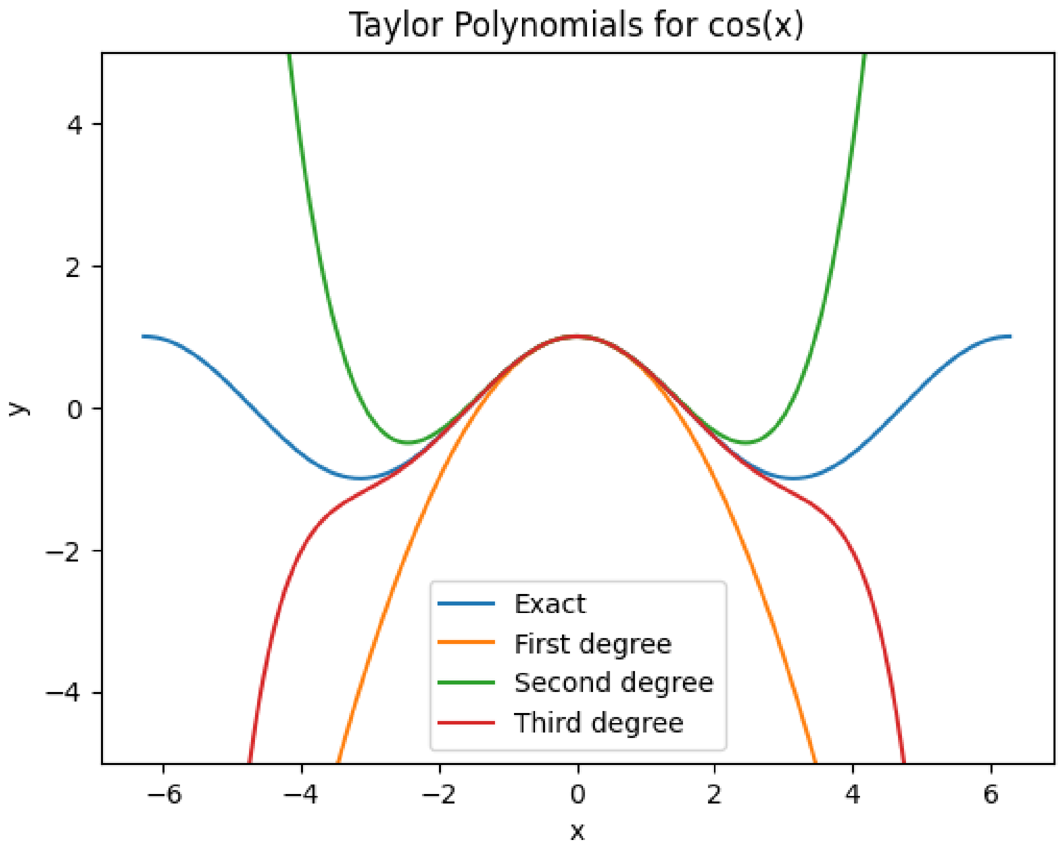

Taylor Polynomials for sin(x)
4
2
0
−2
−4
y
−6
−4
−2
0
2
4
6
x
Exact
First degree
Second degree
Third degree

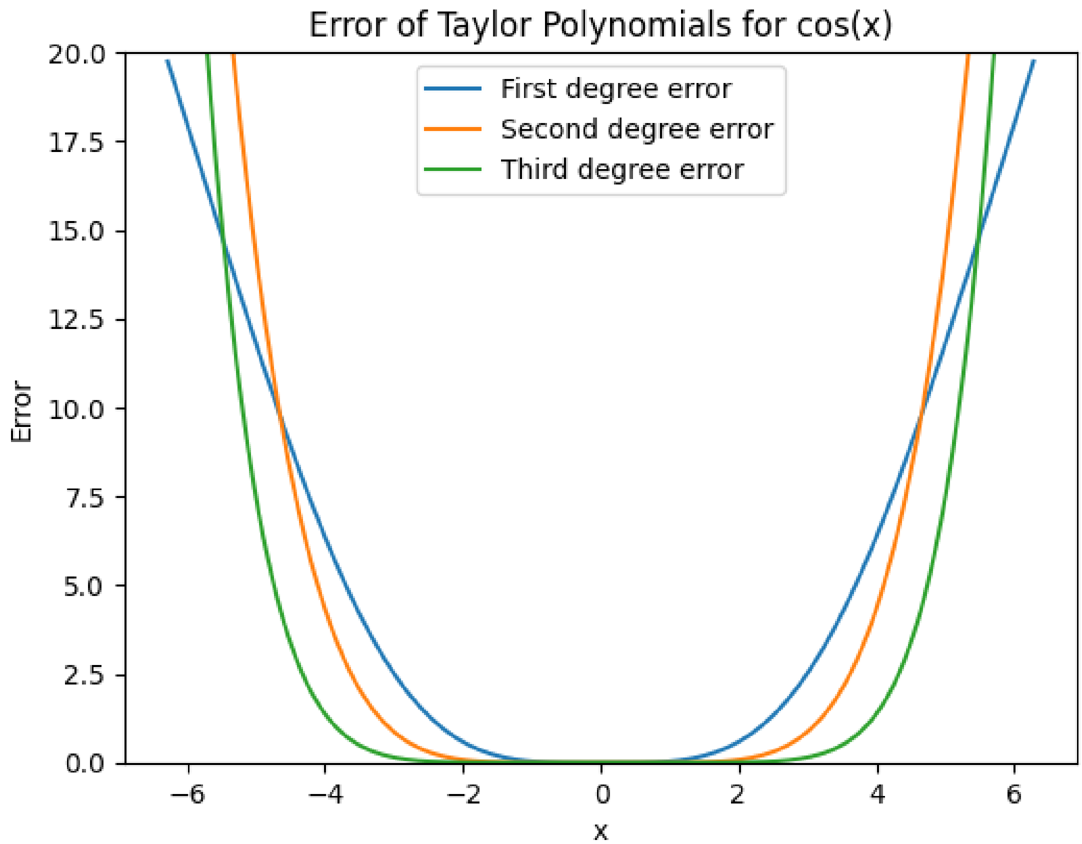


14 cos(x)

[14]:
```
import numpy as np
import matplotlib.pyplot as plt

def cos_taylor_series(x, a, n):
    result = 0
    for i in range(n+1):
        term = ((-1)**i) * (x-a)**(2*i) / math.factorial(2*i)
        result += term
    return result

a = 0
x = np.linspace(-2*np.pi, 2*np.pi, 400)

exact_values = np.cos(x)
first_degree = cos_taylor_series(x, a, 1)
second_degree = cos_taylor_series(x, a, 2)
third_degree = cos_taylor_series(x, a, 3)
```

```
plt.plot(x, exact_values, label='Exact')
plt.plot(x, first_degree, label='First degree')
plt.plot(x, second_degree, label='Second degree')
plt.plot(x, third_degree, label='Third degree')
plt.xlabel('x')
plt.ylabel('y')
plt.title('Taylor Polynomials for cos(x)')
plt.legend()
plt.ylim(-5,5)
plt.show()


x = np.linspace(-2*np.pi, 2*np.pi, 400)
exact_values = np.cos(x)
first_degree_errors = np.abs(exact_values - cos_taylor_series(x, a, 1))
second_degree_errors = np.abs(exact_values - cos_taylor_series(x, a, 2))
third_degree_errors = np.abs(exact_values - cos_taylor_series(x, a, 3))

plt.plot(x, first_degree_errors, label='First degree error')
plt.plot(x, second_degree_errors, label='Second degree error')
plt.plot(x, third_degree_errors, label='Third degree error')
plt.xlabel('x')
plt.ylabel('Error')
plt.title('Error of Taylor Polynomials for cos(x)')
plt.legend()
plt.ylim(0,20)
plt.show()
```


Taylor Polynomials for cos(x)
4
2
0
−2
−4
y
−6
−4
−2
0
2
4
6
x
Exact
First degree
Second degree
Third degree



15 Question 2

For the function y=ln(x), construct the approximate polynomial function with error <0.01 at x=2 and a=1

[61]:
```
import numpy as np
import matplotlib.pyplot as plt

x=2
a=1
true_val=np.log(2)
max_error=0.01
i=1
ans=0
cnt=1
while(1):
  sign = (-1)**(i-1)
  term = ((x-a)**i) / (i * (a**i))
  ans += (sign * term)
```

```
    ans+=np.log(a)
    error=abs(true_val-ans)
    print(cnt ,"->", error)
    cnt=cnt+1
    if(error<max_error):
        break
    i=i+1

print("Degree at which error is less than 0.01")
print(i)
```

```
1 -> 0.3068528194400547
2 -> 0.1931471805599453
3 -> 0.14018615277338797
4 -> 0.10981384722661203
5 -> 0.09018615277338793
6 -> 0.0764805138932787
7 -> 0.0663766289638642
8 -> 0.058623371036135796
9 -> 0.052487740074975364
10 -> 0.047512259925024614
11 -> 0.043396830984066326
12 -> 0.039936502349267045
13 -> 0.03698657457380994
14 -> 0.03444199685476146
15 -> 0.03222466981190519
16 -> 0.030275330188094807
17 -> 0.028548199223669912
18 -> 0.027007356331885668
19 -> 0.025624222615482695
20 -> 0.02437577738451735
21 -> 0.023243270234530322
22 -> 0.022211275220015092
23 -> 0.021266985649550096
24 -> 0.020399681017116533
25 -> 0.019600318982883502
26 -> 0.018861219478654934
27 -> 0.018175817558382046
28 -> 0.017538468155903653
29 -> 0.01694429046478596
30 -> 0.016389042868547365
31 -> 0.015869021647581638
32 -> 0.015380978352418362
33 -> 0.014922051950611914
34 -> 0.014489712755270445
35 -> 0.014081715816158136
36 -> 0.013696061961619654
```

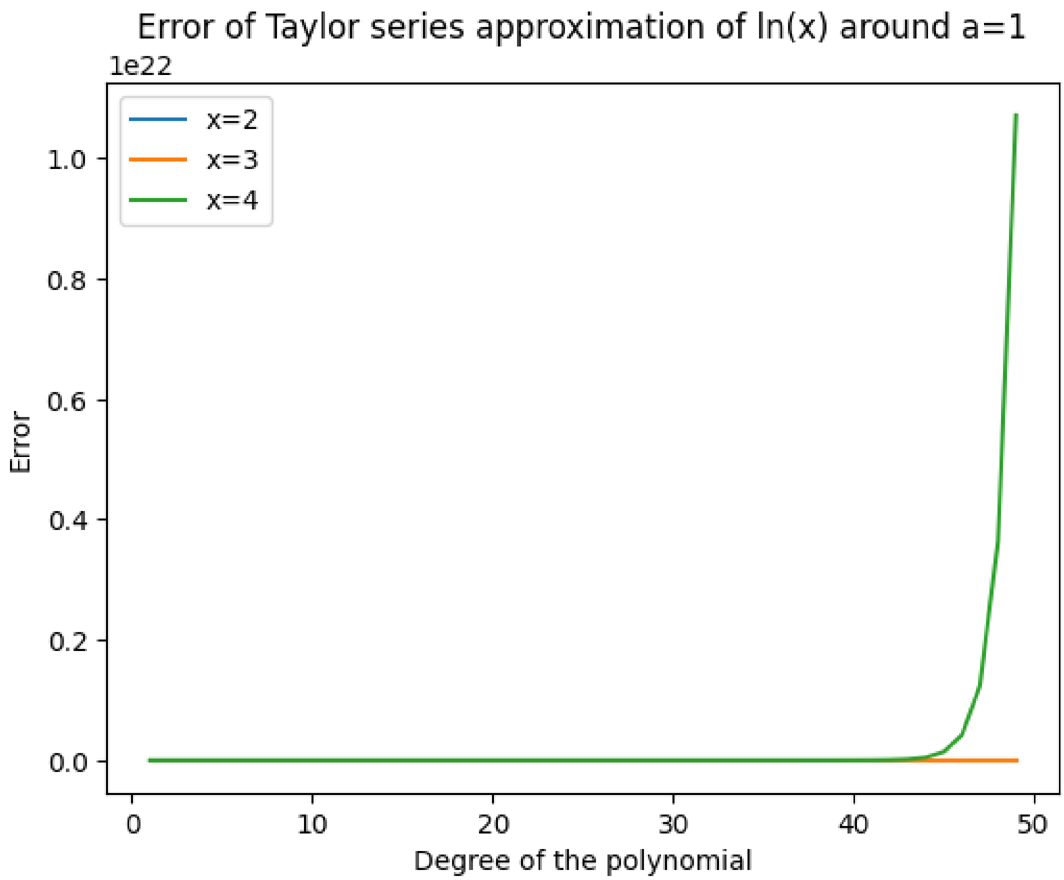
```
37 -> 0.013330965065407319
38 -> 0.012984824408276863
39 -> 0.012656201232748798
40 -> 0.012343798767251224
41 -> 0.012046445135187822
42 -> 0.011763078674336014
43 -> 0.0114927352791524
44 -> 0.011234537448120308
45 -> 0.010987684774101947
46 -> 0.010751445660680647
47 -> 0.010525150084000234
48 -> 0.010308183249333136
49 -> 0.010099980015973009
50 -> 0.009900019984027009
Degree at which error is less than 0.01
50
```

16 Question 3

Plot the degree of the polynomial vs the error at x=2,3,4 in a single figure.

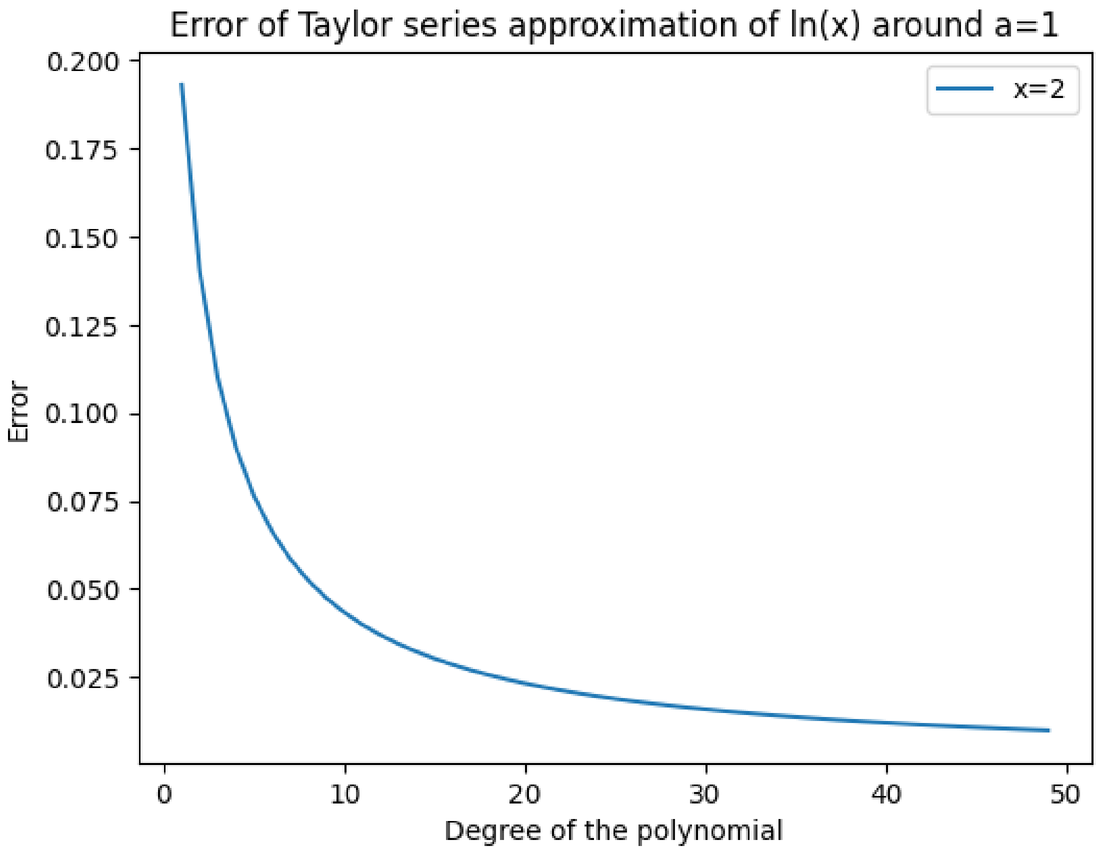
[60]:
```
import numpy as np
import matplotlib.pyplot as plt


def ln(x):
    return np.log(x)

def ln_taylor(x, n):
    taylor_sum = 0
    for i in range(n+1):
        taylor_sum += ((-1)**i) * (x-1)**(i+1) / (i+1)
    return taylor_sum


xs = [2,3,4]

exact_values = [ln(x) for x in xs]


degrees = np.arange(1, 50)
errors = np.zeros((len(xs), len(degrees)))
for i, x in enumerate(xs):
    for j, n in enumerate(degrees):
        approx_value = ln_taylor(x, n)
        errors[i, j] = np.abs(approx_value - exact_values[i])
```

```
for i, x in enumerate(xs):
    plt.plot(degrees, errors[i], label=f'x={x}')

plt.xlabel('Degree of the polynomial')
plt.ylabel('Error')
plt.title('Error of Taylor series approximation of ln(x) around a=1')
plt.legend()
plt.show()
```



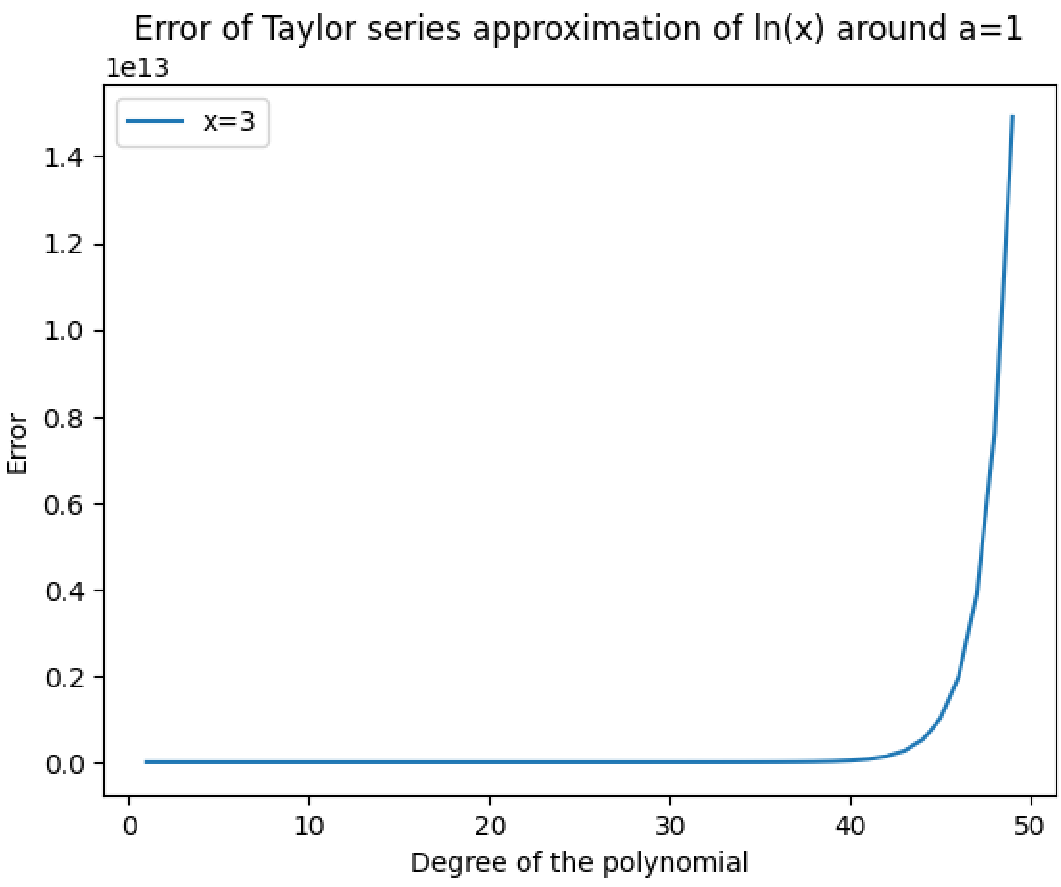
[59]:
```
import numpy as np
import matplotlib.pyplot as plt


def ln(x):
    return np.log(x)

def ln_taylor(x, n):
    taylor_sum = 0
```

```
    for i in range(n+1):
        taylor_sum += ((-1)**i) * (x-1)**(i+1) / (i+1)
    return taylor_sum


xs = [2]

exact_values = [ln(x) for x in xs]


degrees = np.arange(1, 50)
errors = np.zeros((len(xs), len(degrees)))
for i, x in enumerate(xs):
    for j, n in enumerate(degrees):
        approx_value = ln_taylor(x, n)
        errors[i, j] = np.abs(approx_value - exact_values[i])


for i, x in enumerate(xs):
    plt.plot(degrees, errors[i], label=f'x={x}')

plt.xlabel('Degree of the polynomial')
plt.ylabel('Error')
plt.title('Error of Taylor series approximation of ln(x) around a=1')
plt.legend()
plt.show()
```



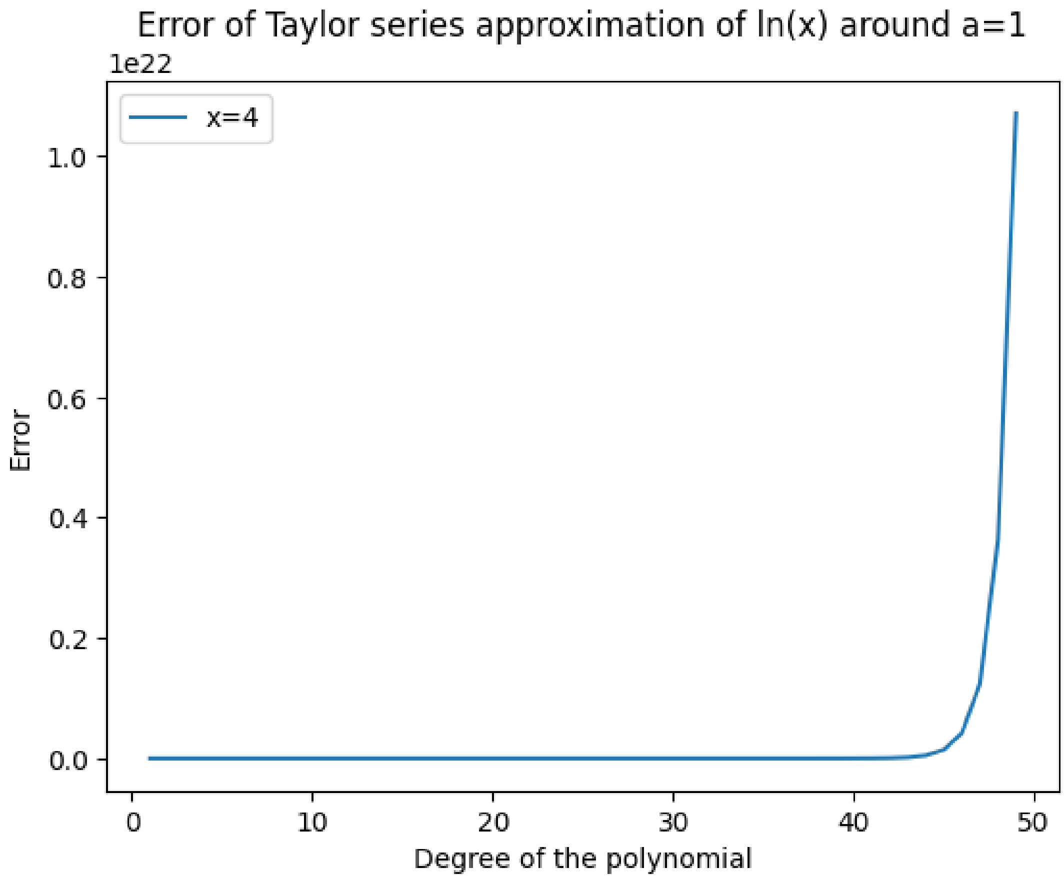
[58]:
```
# @title
import numpy as np
import matplotlib.pyplot as plt


def ln(x):
    return np.log(x)

def ln_taylor(x, n):
    taylor_sum = 0
    for i in range(n+1):
        taylor_sum += ((-1)**i) * (x-1)**(i+1) / (i+1)
    return taylor_sum


xs = [3]

exact_values = [ln(x) for x in xs]
```

```
degrees = np.arange(1, 50)
errors = np.zeros((len(xs), len(degrees)))
for i, x in enumerate(xs):
    for j, n in enumerate(degrees):
        approx_value = ln_taylor(x, n)
        errors[i, j] = np.abs(approx_value - exact_values[i])


for i, x in enumerate(xs):
    plt.plot(degrees, errors[i], label=f'x={x}')

plt.xlabel('Degree of the polynomial')
plt.ylabel('Error')
plt.title('Error of Taylor series approximation of ln(x) around a=1')
plt.legend()
plt.show()
```



[57]:
```
import numpy as np
import matplotlib.pyplot as plt
```

```
def ln(x):
    return np.log(x)

def ln_taylor(x, n):
    taylor_sum = 0
    for i in range(n+1):
        taylor_sum += ((-1)**i) * (x-1)**(i+1) / (i+1)
    return taylor_sum


xs = [4]

exact_values = [ln(x) for x in xs]


degrees = np.arange(1, 50)
errors = np.zeros((len(xs), len(degrees)))
for i, x in enumerate(xs):
    for j, n in enumerate(degrees):
        approx_value = ln_taylor(x, n)
        errors[i, j] = np.abs(approx_value - exact_values[i])


for i, x in enumerate(xs):
    plt.plot(degrees, errors[i], label=f'x={x}')

plt.xlabel('Degree of the polynomial')
plt.ylabel('Error')
plt.title('Error of Taylor series approximation of ln(x) around a=1')
plt.legend()
plt.show()
```



For ln(x) till x=2 it stays convergent but after 2 in starts diverging that is why the error plot increases rapidly for x=3 and x=4 that is why we have plotted individual graphs