# DS288 (AUG) 3:0 Numerical Methods


Naman Pesricha
namanp@iisc.ac.in
SR - 24115


**Homework-3**

---

**Q1 Exercise Set 3.2, Problem #12 in Text (Page-124). Be sure to read the note about inverse interpolation directly above the problem. Solve this problem using an iterated interpolation approach (i.e., Neville's algorithm). Report the relative error of your result and what value you used for the exact solution. [1.5 points]**

---

| $i$ | $y = x - e^x$ | $x = Q_{i0}$ | $Q_{i1}$ | $Q_{i2}$ | $Q_{i3}$ |
|---|---|---|---|---|---|
| 0 | -0.440818 | 0.3 | - | - | - |
| 1 | -0.270320 | 0.4 | 5.585473e-01 | - | - |
| 2 | -0.106531 | 0.5 | 5.650416e-01 | 5.671112e-01 | - |
| 3 | 0.051188 | 0.6 | 5.675448e-01 | 5.671463e-01 | **5.671426e-01** |

Table 1: Inverse interpolation table using NEVILLES for finding root of $x = f^{-1}(y)$. $Q_{ij}$ is the polynomial approximation that agrees with $[x_{i-j}, x_{i-j+1}, ..., x_i]$.

| $i$ | Relative Error $Q_{i,0}$ | Relative Error $Q_{i,1}$ | Relative Error $Q_{i,2}$ | Relative Error $Q_{i,3}$ |
|---|---|---|---|---|
| 1 | 4.710331e-01 | - | - | - |
| 2 | 2.947109e-01 | 1.515662e-02 | - | - |
| 3 | 1.183886e-01 | 3.705734e-03 | 5.655048e-05 | - |
| 4 | 5.793370e-02 | 7.079698e-04 | 5.254266e-06 | **1.175861e-06** |

Table 2: Relative errors for the polynomials $Q_{ij}$

Value used for exact solution (computed from BISECTIONMETHOD)

$$\boxed{x^* = 0.5671432904}$$

We get the least relative error in $Q_{43}$

$$\boxed{ComputedValue = 5.671426 \times 10^{-01}}$$

$$\boxed{RelativeError(Q_{43}) = 1.175861 \times 10^{-06}}$$

---

**Q2 In some applications one is faced with the problem of interpolating points which lie on a curved path in the plane, for example in computer printing of enlarged letters. Often the complex shapes (i.e., alphabet characters) cannot be represented as a function of x because they are not single-valued. One approach is to use *Parametric Interpolation.* Assume that the points along a curve are numbered $P_1, P_2, ..., P_n$ as the curved path is traversed and let $d_i$ be the (straight-line) distance between $P_i$ and $P_{i+1}$. Then define $t_i = \Sigma_{j=1}^{i=1} dj$ , for for i =1, 2, ..., n (i.e.,$t_1 = 0, t_2 = d_1, t_3 = d_1 + d_2$, etc). If $P_i = (x_i, y_i)$, one can consider two sets of data $(t_i, x_i)$**

**and $(t_i, y_i)$ for i = 1, 2, ...n which can be interpolated independently to generate the functions f(t) and g(t), respectively. Then $P(f(t), g(t))$ for $0 \leq t \leq t_n$ is a point in the plane and as t is increased from 0 to $t_n$, P(t) interpolates the desired shaped (hopefully!). Interpolation of the data given below via this method should produce a certain letter. Adapt your algorithm from problem (1) to perform the interpolations on f(t) and g(t) where t is increased from 0.0 to 12.0 in steps of $dt$ (see data below). Report the value of $dt$ you use to achieve 'reasonable' results. Turn in a plot of your interpolated shape (not the numeric values of P(t)) as well as plots of f(t) and g(t) individually. [3 points]**

---

For a very low value of dt($= 0.1250$), we can visualize the value of $P(t)$, $f(t)$ and $g(t)$

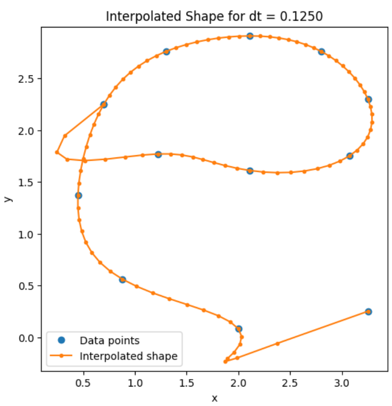


Figure 1: $P(f(t), g(t))$ at dt $=$ 0.125

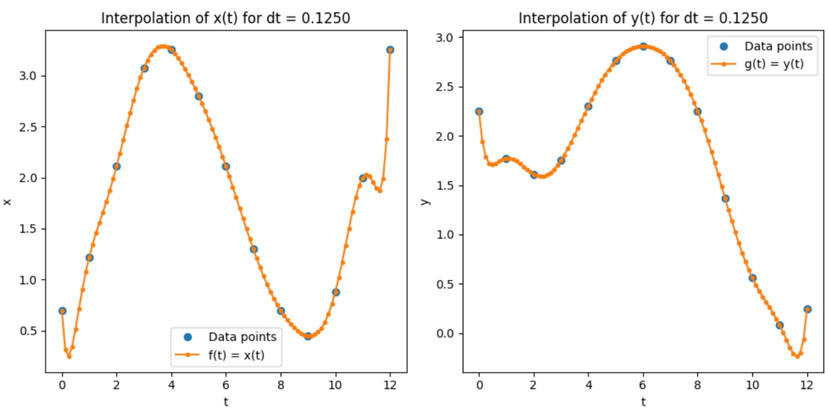


Figure 2: $x = f(t)$ and $y = g(t)$ for dt $= 0.1250$

This curve is smooth but we can see the **spurious oscillations** at the end points. The underlying polynomial is LAGRANGE which is a **Global Interpolating Polynomial** and hence the oscillations are observed. We can use a larger value of

$$\boxed{dt = 0.8}$$

to get a more reasonable curve which looks more like the letter **'e'** (but is not as smooth.) We can see that there's a clear tradeoff between number of smoothness ($\downarrow dt$) and avoiding oscillations at end points ($\uparrow dt$).

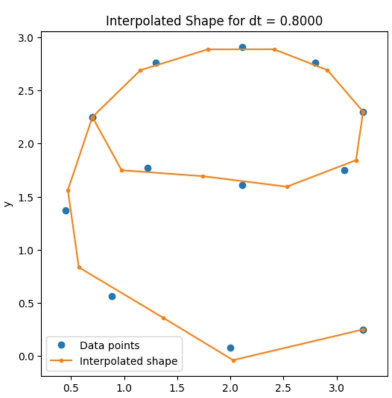


Figure 3: $P(f(t), g(t))$ at dt $= 0.8$

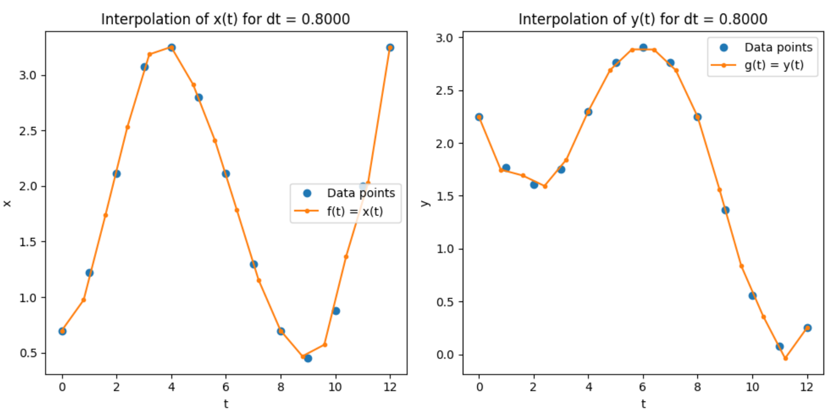


Figure 4: $x = f(t)$ and $y = g(t)$ for dt $= 0.1250$

**Q3 Repeat problem (2) with a natural cubic spline. Also report all four coefficients for each of the cubics which comprise the interpolants for both f(t) and g(t). How does your letter compare with that produced in problem (2)? Explain any differences. [3.5 points]**

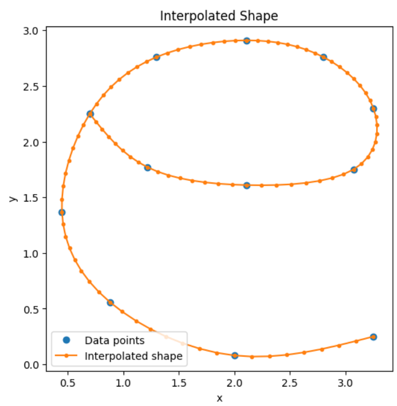


Figure 5: $P(t)$ for cubic spline.

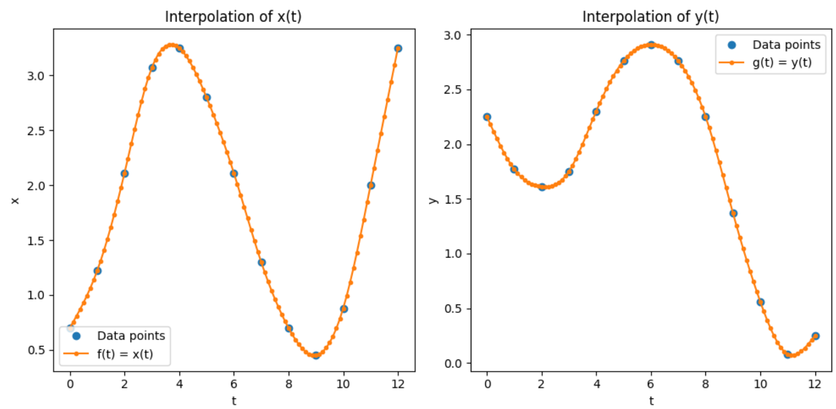


Figure 6: $x = f(t)$ and $y = g(t)$ for dt=0.1250 for cubic spline

| $S_i$ | **a** | **b** | **c** | **d** |
|---|---|---|---|---|
| 0 | 8.2085e-02 | -1.1102e-16 | 4.3791e-01 | 7.0000e-01 |
| 1 | -4.0427e-02 | 2.4626e-01 | 6.8417e-01 | 1.2200e+00 |
| 2 | -2.2038e-01 | 1.2497e-01 | 1.0554e+00 | 2.1100e+00 |
| 3 | 7.1935e-02 | -5.3616e-01 | 6.4422e-01 | 3.0700e+00 |
| 4 | 8.2637e-02 | -3.2035e-01 | -2.1229e-01 | 3.2500e+00 |
| 5 | -1.2481e-02 | -7.2441e-02 | -6.0508e-01 | 2.8000e+00 |
| 6 | 8.7289e-02 | -1.0989e-01 | -7.8740e-01 | 2.1100e+00 |
| 7 | -6.6750e-03 | 1.5198e-01 | -7.4531e-01 | 1.3000e+00 |
| 8 | 7.9411e-02 | 1.3196e-01 | -4.6137e-01 | 7.0000e-01 |
| 9 | 1.9031e-02 | 3.7019e-01 | 4.0779e-02 | 4.5000e-01 |
| 10 | -1.4553e-01 | 4.2728e-01 | 8.3825e-01 | 8.8000e-01 |
| 11 | 3.1069e-03 | -9.3207e-03 | 1.2562e+00 | 2.0000e+00 |

Table 3: Coefficients for $x = f(t)$ for the equation $S_i = a_i(t-t_i)^3 + b_i(t-t_i)^2 + c_i(t-t_i) + d_i$

| $S_j$ | **a** | **b** | **c** | **d** |
|---|---|---|---|---|
| 0 | 7.2213e-02 | 1.1102e-16 | -5.5221e-01 | 2.2500e+00 |
| 1 | -4.1067e-02 | 2.1664e-01 | -3.3557e-01 | 1.7700e+00 |
| 2 | 7.2053e-02 | 9.3440e-02 | -2.5493e-02 | 1.6100e+00 |
| 3 | -1.3715e-01 | 3.0960e-01 | 3.7755e-01 | 1.7500e+00 |
| 4 | -2.3466e-02 | -1.0184e-01 | 5.8531e-01 | 2.3000e+00 |
| 5 | 1.1010e-02 | -1.7224e-01 | 3.1123e-01 | 2.7600e+00 |
| 6 | -1.0574e-02 | -1.3921e-01 | -2.1795e-04 | 2.9100e+00 |
| 7 | -2.8714e-02 | -1.7093e-01 | -3.1036e-01 | 2.7600e+00 |
| 8 | 1.1543e-01 | -2.5707e-01 | -7.3836e-01 | 2.2500e+00 |
| 9 | 6.9979e-03 | 8.9216e-02 | -9.0621e-01 | 1.3700e+00 |
| 10 | 1.1658e-01 | 1.1021e-01 | -7.0679e-01 | 5.6000e-01 |
| 11 | -1.5332e-01 | 4.5995e-01 | -1.3663e-01 | 8.0000e-02 |

Table 4: Coefficients for $y = g(t)$ for the equation $S_j = a_j(t-t_i)^3 + b_j(t-t_j)^2 + c_j(t-t_j) + d_j$

The letter **'e'** for the $\boxed{dt = 0.1250}$ using cubic spline is **smoother** and has **no spurious oscillations at the endpoints** compared to the approach used in Q2. This difference is arising because:

1. **Smoothness** : Cubic spline is a **piecewise interpolation polynomial** that matches the **value**, **derivative** and **second derivative** at each point ensuring smoothness.
2. **No spurious oscillations** : The fitting between points $t_i$ and $t_{i+1}$ **depends only on** $t_i$ and $t_{i+1}$ and it's derivatives and **not faraway points** that shouldn't have influence in the vicinity of $t_i$ and $t_{i+1}$.

**Q4 Consider the oscillograph record of the free-damped vibrations of a structure. From vibration theory, it is known that for viscous damping (damping proportional to velocity) the envelop of such a vibration (i.e., the curve through the peaks of the oscillations) is an exponential function of the form**

$$y = be^{-2\pi ax}$$

**where x is the cycle number, y is the corresponding amplitude and a is a damping factor. Using the three data points shown in the figure, determine a and b that result from a best fit based on the least-squares criterion. Use a linear least squares approach by suitable change of variable. You may solve this problem "by hand" if you wish.**
**An alternate approach to this problem would be to construct a nonlinear least-squares fit using the data directly as given. Would this approach lead to exactly the same a and b values you determine above (assuming perfect math, i.e., no rounding errors in either case)?. [2 points]**

---

| Name | Prediction | Error |
|---|---|---|
| Transformed approach | $pred_{transformed}$ | $E_{transformed}(y_i, \hat{y}_i) = \sum[\log(y_i) - \log(\hat{y_i})]^2$ |
| Non-Linear least squares | $pred_{non-linear}$ | $E_{non-linear}(y_i, \hat{y}_i) = \sum[(y_i) - (\hat{y_i})]^2$ |

Table 5: Terminologies used in answer 4.

By using the **Transformed approach**, we can transform the problem to a linear problem by taking *log* on both sides.

$$y = be^{-2\pi ax} \equiv \log y = \log b - 2\pi ax$$

Solving this linear equation using linear least squares we get.

$\boxed{a_{transformed} = 0.00609}$ $\boxed{b_{transformed} = 16.86397}$

Using the **Non-linear approach**, we get the following values of coefficients:

$\boxed{a_{non-linear} = 0.00618}$ $\boxed{b_{non-linear} = 16.96953}$

We will get **different** a and b values using the **non-linear approach** (assuming perfect maths) as both approaches are minimizing different error functions $E_{transformed}$ and $E_{non-linear}$. From the table below, we can see that the $pred_{non-linear}$ minimizes $E_{non-linear}$ and $pred_{transformed}$ minimizes $E_{transformed}$. Moreover, the **transformed** approach is not the least squares approximation of the original problem (as the $E_{transformed}$ is not least squares error).

$pred_{transformed}$ = [ 16.8639, 9.14577, 4.95999 ] , $pred_{non-linear}$ = [ 16.9695, 9.11345, 4.89436 ]

| Approach | $pred_{tarnsformed}$ | $pred_{non-linear}$ |
|---|---|---|
| $E_{transformed}(y_i, \hat{y}_i)$ | **0.000387** | 0.000616 |
| $E_{non-linear}(y_i, \hat{y}_i)$ | 0.041354 | **0.024959** |

Table 6: $pred_{non-linear}$ minimizes $E_{non-linear}$ and $pred_{transformed}$ minimizes $E_{transformed}$.

## CODE (Python)

```python
# %% [markdown]
# # Q1

# %%
import numpy as np
np.set_printoptions(precision=6, suppress=False, formatter={'float_kind': '{: .6e}'.format})

# Define the data points
x = np.array([0.3, 0.4, 0.5, 0.6])
e_power_x = np.array([ 0.740818,  0.670320, 0.606531,  0.548812])
y = x - e_power_x

# Define the function for which we want to find the root
def f(x):
    return x - np.exp(-x)

# Bisection method
def bisection_method(f, a, b, tol=1e-15, max_iter=1000):
    if f(a) * f(b) >= 0:
        raise ValueError("The function must have different signs at the endpoints a and b.")

    for _ in range(max_iter):
        c = (a + b) / 2
        if f(c) == 0 or (b - a) / 2 < tol:
            return c
        if f(c) * f(a) < 0:
            b = c
        else:
            a = c
    return c


# Use bisection method to find the root
exact_solution = bisection_method(f, 0, 1)
print(f"Root found by bisection method (Exact approximation):{exact_solution}")

# Neville's Algorithm for inverse interpolation
def nevilles(x,y,p):
    n = len(x)
    Q = np.zeros((n,n))
    Q[:,0] = y

    for i in range(1,n):
        for j in range(i,n):
            Q[j,i] = ( (p - x[j-i])*Q[j,i-1] - (p - x[j])*Q[j-1,i-1])/(x[j] - x[j-i])

    return Q


# Apply Neville's Algorithm to approximate f^(-1)(0)
p = 0
Q = nevilles(y,x,p)  # We want to find f^(-1)(0)
n = len(Q)
approximation = Q[n-1,n-1]

print(Q)

# Calculate the relative error
relative_error = abs((approximation - exact_solution) / exact_solution)

```

```python
# Report the results
print(f"Approximation of f^(-1)(0): {approximation}")
print(f"Relative error: {relative_error}")
print(y)

# Calculate the error matrix
error_matrix = np.abs(Q - exact_solution)/exact_solution

# Print the error matrix
print("Error Matrix:")
print(error_matrix)


# %% [markdown]
# # Q2

# %%
import numpy as np
import matplotlib.pyplot as plt

# Given data points
t = np.array([0.0, 1.0, 2.0, 3.0, 4.0, 5.0, 6.0, 7.0, 8.0, 9.0, 10.0, 11.0, 12.0])
x = np.array([0.70, 1.22, 2.11, 3.07, 3.25, 2.80, 2.11, 1.30, 0.70, 0.45, 0.88, 2.00, 3.25])
y = np.array([2.25, 1.77, 1.61, 1.75, 2.30, 2.76, 2.91, 2.76, 2.25, 1.37, 0.56, 0.08, 0.25])

# Neville's Algorithm for interpolation
def nevilles(x,y,p):
    n = len(x)
    Q = np.zeros((n,n))
    Q[:,0] = y

    for i in range(1,n):
        for j in range(i,n):
            Q[j,i] = ( (p - x[j-i])*Q[j,i-1] - (p - x[j])*Q[j-1,i-1])/(x[j] - x[j-i])

    return Q

num_points = 97

# Define the range of t for interpolation
t_interp = np.linspace(0, 12, num_points)

# Interpolated values using Neville's method
x_interp = np.array([nevilles(t, x, t_val)[-1, -1] for t_val in t_interp])
y_interp = np.array([nevilles(t, y, t_val)[-1, -1] for t_val in t_interp])
print(x_interp)
print(t_interp)

# Plot the interpolated shape
plt.figure(figsize=(6, 6))
plt.xlim(-1, 4)
plt.ylim(-1, 4)
plt.plot(x, y, 'o', label='Data points')
plt.plot(x_interp, y_interp, label=f'Interpolated shape', marker='o', markersize=3)
plt.xlabel('x')
plt.ylabel('y')
plt.legend()
plt.title(f'Interpolated Shape for dt = {12/(num_points-1):.4f}')
plt.axis('equal')
plt.show()

# Plot the interpolated shape
plt.figure(figsize=(10, 5))

# Plot f(t) and g(t)
```

```
plt.subplot(1, 2, 1)
plt.plot(t, x, 'o', label='Data points')
plt.plot(t_interp, x_interp, label='f(t) = x(t)', marker='o', markersize=3)
plt.xlabel('t')
plt.ylabel('x')
plt.legend()
plt.title(f'Interpolation of x(t) for dt = {12/(num_points-1):.4f}')

plt.subplot(1, 2, 2)
plt.plot(t, y, 'o', label='Data points')
plt.plot(t_interp, y_interp, label='g(t) = y(t)', marker='o', markersize=3)
plt.xlabel('t')
plt.ylabel('y')
plt.legend()
plt.title(f'Interpolation of y(t) for dt = {12/(num_points-1):.4f}')

plt.tight_layout()
plt.show()


# %% [markdown]
# # Q3

# %%
t = np.array([0.0, 1.0, 2.0, 3.0, 4.0, 5.0, 6.0, 7.0, 8.0, 9.0, 10.0, 11.0, 12.0])
x = np.array([0.70, 1.22, 2.11, 3.07, 3.25, 2.80, 2.11, 1.30, 0.70, 0.45, 0.88, 2.00, 3.25])
y = np.array([2.25, 1.77, 1.61, 1.75, 2.30, 2.76, 2.91, 2.76, 2.25, 1.37, 0.56, 0.08, 0.25])

import numpy as np
import matplotlib.pyplot as plt
from scipy.interpolate import CubicSpline

cs_x = CubicSpline(t, x, bc_type='natural')
cs_y = CubicSpline(t, y, bc_type='natural')

t_interp = np.linspace(0, 12, 97)

x_interp = cs_x(t_interp)
y_interp = cs_y(t_interp)

coeffs_x = cs_x.c
coeffs_y = cs_y.c

print(coeffs_x)
print(coeffs_y)
# Plot the interpolated shape
plt.figure(figsize=(6, 6))
plt.xlim(-1, 4)
plt.ylim(-1, 4)
plt.plot(x, y, 'o', label='Data points')
plt.plot(x_interp, y_interp, label='Interpolated shape', marker='o', markersize=3)
plt.xlabel('x')
plt.ylabel('y')
plt.legend()
plt.title('Interpolated Shape')
plt.axis('equal')
plt.show()

# Plot the interpolated shape
plt.figure(figsize=(10, 5))

# Plot f(t) and g(t)
plt.subplot(1, 2, 1)
plt.plot(t, x, 'o', label='Data points')
plt.plot(t_interp, x_interp, label='f(t) = x(t)', marker='o', markersize=3)
```

```
plt.xlabel('t')
plt.ylabel('x')
plt.legend()
plt.title('Interpolation of x(t)')

plt.subplot(1, 2, 2)
plt.plot(t, y, 'o', label='Data points')
plt.plot(t_interp, y_interp, label='g(t) = y(t)', marker='o', markersize=3)
plt.xlabel('t')
plt.ylabel('y')
plt.legend()
plt.title('Interpolation of y(t)')

plt.tight_layout()
plt.show()



# %% [markdown]
# # Q4

# %%
import numpy as np

# Data points
data = np.array([[0, 17], [16, 9], [32, 5]])

# Transform y to ln(y)
Y = np.log(data[:, 1])
X = data[:, 0]

# Model: ln(y) = -2*pi*a*x + ln(b) => y' = a_1*x + a_0
sum_x = np.sum(X)
sum_x2 = np.sum(X**2)
sum_y = np.sum(Y)
sum_xy = np.sum(X*Y)
m = len(X)

a_0 = (sum_x2*sum_y - sum_x*sum_xy)/(m*sum_x2 - sum_x**2)
a_1 = (m*sum_xy - sum_x*sum_y)/(m*sum_x2 - sum_x**2)

# Retreiving a and b from a_0 and a_1

b = np.exp(a_0)
a = -a_1/(2*np.pi)
Y_hat = b*np.exp(-2*np.pi*a*X)
Y = data[:, 1]
print(Y)
print(Y_hat)
print(f"Damping factor a: {a:.5f}")
print(f"Coefficient b: {b:.5f}")
print(f"Least Squares Error in Y: {np.sum((Y-Y_hat)**2):.5f}")
print(np.abs(Y-Y_hat))




# %%
from scipy.optimize import curve_fit

data = np.array([[0, 17], [16, 9], [32, 5]])
X = data[:, 0]
Y = data[:, 1]

# Define the exponential function
```

```python
def exponential_func(x, b, a):
    return b * np.exp(-2 * np.pi * a * x)

# Use curve_fit to find the best-fit parameters
params, _ = curve_fit(exponential_func, X, Y, p0=[17, 0.01])

# Extract the parameters
b_nonlinear, a_nonlinear = params

Y_hat = b_nonlinear*np.exp(-2*np.pi*a_nonlinear*X)

print(Y)
print(Y_hat)
print(f"Damping factor a (nonlinear fit): {a_nonlinear:.5f}")
print(f"Coefficient b (nonlinear fit): {b_nonlinear:.5f}")
print(f"Least Squares Error in Y: {np.sum((Y-Y_hat)**2):.5f}")
print(np.abs(Y-Y_hat))


# %%
data_from_transformation = np.array([ 1.686397e+01,  9.145774e+00,  4.959993e+00])
data_from_non_linear_fit = np.array([ 1.696953e+01, 9.113459e+00, 4.894368e+00])
actual_output = np.array([ 17, 9, 5])

def error_function_transformed(actual, predicted):
    return np.sum((np.log(actual) - np.log(predicted))**2)

def error_function_non_linear(actual, predicted):
    return np.sum((actual - predicted)**2)

print(f"Error for transformed prediction using transformed LSE method: {error_function_transformed(actual_output, data_from_transformation)}")
print(f"Error for non-linear prediction using transformed LSE method: {error_function_transformed(actual_output, data_from_non_linear_fit)}")
print(f"Error for transformed prediction using non-linear LSE method: {error_function_non_linear(actual_output, tra)}")
print(f"Error for non-linear prediction using non-linear LSE method: {error_function_non_linear(actual_output, data_from_non_linear_fit)}")

# %%
```