Q 1. Data for inverse interpolation to find an approximated root of the function $f(x) = x - e^{-x}$ is as follows:

x	0.3	0.4	0.5	0.6
e^{-x}	0.740818	0.670320	0.606531	0.548812
$f(x) = x - e^{-x}$	0.440818	0.270320	0.106531	-0.051188

In order to get the root of the $f(x)$ *using inverse interpolation*, we need to interpolate the value of $x = f^{-1}(x - e^{-x})$. If we calculate $f^{-1}(0)$, we get the approximated value of the root x.

As bisection method guarantee convergence to the root, hence we use bisection method to calculate the exact value of the root with tolerance 1×10^{-10}, and initial guess values as 0.5 and 0.6.

Value of root for $f(x) = x - e^{-x}$ calculated using bisection method (exact solution) and inverse Neville's method given in the table:

Neville's Method	0.5671423
Bisection Method	0.5671432903

Relative error is inverse Neville's method is given as:

$$\text{Relative Error} = \frac{|\text{Exact Value} - \text{Approximate Value}|}{|\text{Exact Value}|} = \frac{|0.5671432903 - 0.567143|}{|0.5671432903|} = \mathbf{5.1186 \times 10^{-7}}$$

Q 2. Parametric curves are interpolated for the data given in the table 1 using the same method used in problem (1), Neville's method. As there are 13 number of data points, $f(t)$ and $g(t)$ are 12th degree polynomial.

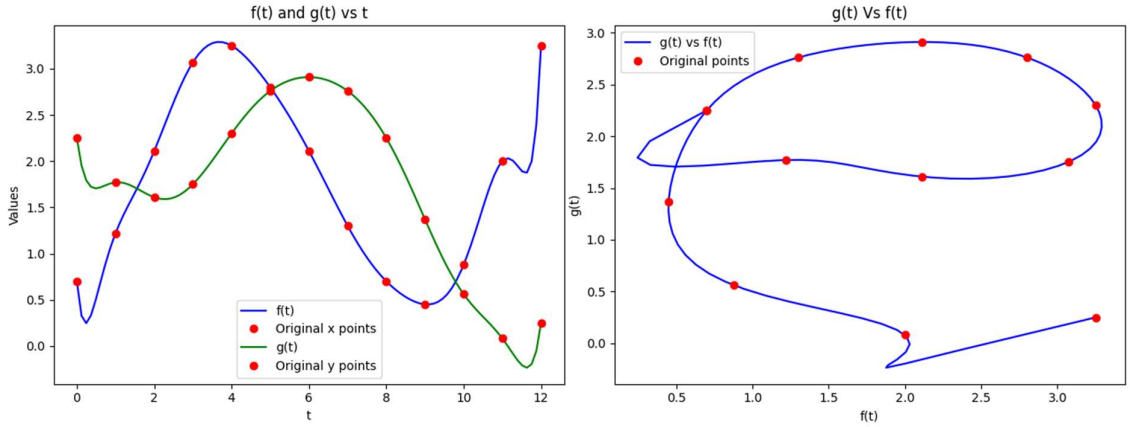

Fig 1 (a) Evaluation of functions $f(t)$ (blue) and $g(t)$ (green) at intervals of $\mathbf{\Delta t = 0.12}$ .The functions are plotted against the variable t, with red dots indicating the discrete data points sampled from the original functions. Fig (b) Parametric plot of $g(t)$ versus $f(t)$, showing the trajectory of the functions across the sampled points. The red dots represent the original values at each time step.

Conclusion: When the interpolated curve is plotted using exactly 13 points, the result resembles a simple line connecting the points. However, as we increase the number of points for interpolating the curves f(t) and g(t), oscillations (similar oscillation seen in bird example discussed in class) begin to appear near the endpoints, as shown in Figure 1 (a). Consequently, when plotting g(t) versus f(t) we do not obtain the precise shape of the letter 'e', can be seen in fig 1(b). These spurious oscillations occur because Neville's method uses global polynomial interpolation, which introduces instabilities near the boundaries.

Table 1.

t_i	x_i	y_i
0	0.70	2.25
1	1.22	1.77
2	2.11	1.61
3	3.07	1.75
4	3.25	2.30
5	2.80	2.76
6	2.11	2.91
7	1.30	2.76
8	0.70	2.25
9	0.45	1.37
10	0.88	0.56
11	2.00	0.08
12	3.25	0.25

Q 3. Data given in table 2 and table 3 is interpolated using Natural Cubic spline interpolation independently for $f(t)$ with t and $g(t)$ with t respectively.

$$S_i(t) = a_i + b_i(t - t_i) + c_i(t - t_i)^2 + d_i(t - t_i)^3$$

$S_i(t)$ is the cubic polynomial for the interval $[t_i, t_{i+1}]$.

a_i, b_i, c_i, and d_i are the coefficients for this interval.

Cubic Coefficients for $f(t)$ (x-values) given in table 2, and for $g(t)$ (y-values) given in table 3.

Table 2

i	a_i	b_i	c_i	d_i
0	0.7	0.437914	0	0.082085
1	1.22	0.684170	0.246256	-0.04042
2	2.11	1.055402	0.124974	-0.22037
3	3.07	0.644220	-0.536156	0.071935
4	3.25	-0.212285	-0.320350	0.082636
5	2.8	-0.605077	-0.072441	-0.012481
6	2.11	-0.787403	-0.109885	0.087289
7	1.3	-0.745307	0.151982	-0.006675
8	0.7	-0.461368	0.131956	0.079411
9	0.45	0.040779	0.370190	0.019030
10	0.88	0.838251	0.427282	-0.145534
11	2	1.256213	-0.009320	0.003106

Table 3

i	a_i	b_i	c_i	d_i
0	2.25	-0.552213	0	0.072213
1	1.77	-0.335573	0.21664	-0.041067
2	1.61	-0.025493	0.09344	0.072053
3	1.75	0.377547	0.3096	-0.137147
4	2.3	0.585306	-0.101841	-0.023466
5	2.76	0.311228	-0.172238	0.01101
6	2.91	-0.000218	-0.139208	-0.010574
7	2.76	-0.310356	-0.17093	-0.028714
8	2.25	-0.738358	-0.257071	0.115429
9	1.37	-0.906213	0.089216	0.006998
10	0.56	-0.706789	0.110209	0.11658
11	0.08	-0.136632	0.459948	-0.153316

Conclusion: The interpolation of the data from Table 1 using natural cubic spline interpolation results in a smooth curve, as depicted in Figure 2(a). Each interval between data points is modelled by a cubic polynomial, ensuring that both the curve and its first and second derivatives are continuous across intervals. This prevents oscillations at the endpoints, which commonly occur in other global interpolation methods. Compared to the earlier method used in problem (2), the natural cubic spline provides a more accurate interpolation with smoother transitions between points. Additionally, all four coefficients for each cubic for both $f(t)$ and $g(t)$ were calculated and reported in table 2 and table 3 respectively. **The letter 'e' shape seen in** Figure 2(b)**, representing $g(t)$ vs $f(t)$, shows a much closer fit to the original data points, especially compared to the earlier approach. The smoothness is due to the local nature of cubic splines, which reduces spurious oscillations while maintaining continuity.**

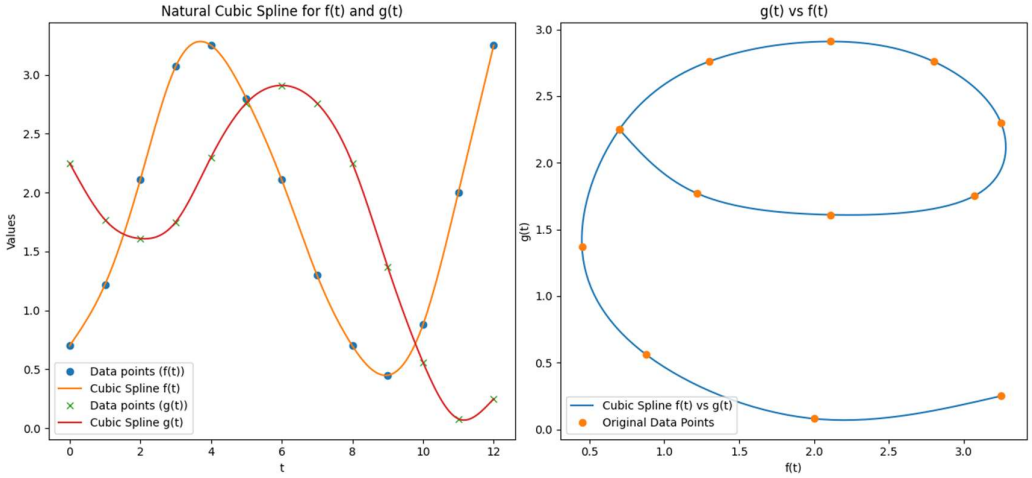

Figure 1 (a): This plot shows the interpolated curves for both f(t) and g(t) using natural cubic splines. **Figure 2 (b):** This figure represents the relationship between $g(t)$ and $f(t)$, The smooth curve produced closely matches the data, forming a loop that resembles the letter 'e.'

Q4:

(i) Linear least squares Approach:

$y = be^{-2\pi a n}$

$\ln(y) = \ln(b) - 2\pi a n$

$Y = B + An$

i	ni	yi
1	0	17
2	16	9
3	32	5

$$B = \frac{\sum_{i=1}^{3} n_i^2 \sum_{i=1}^{3} Y_i - \sum_{i=1}^{m} n_i Y_i \sum_{i=1}^{m} n_i}{m\left[\sum_{i=1}^{m} n_i^2\right] - \left(\sum_{i=1}^{m} n_i\right)^2} = \frac{1280 \times 6.639076 - 86.65761 \times 48}{3(1280) - (48)^2} = 2.825179$$

$B = 2.825179$, $B = \ln(b) \Rightarrow$ $\boxed{b = 16.86397}$

$$A = \frac{m\sum_{i=1}^{m} n_i Y_i - \sum_{i=1}^{m} n_i \sum_{i=1}^{m} Y_i}{m\left[\sum_{i=1}^{m} n_i^2\right] - \left(\sum_{i=1}^{m} n_i\right)^2} = \frac{3 \times 86.65761 - 48 \times 6.639076}{3 \times 1280^2 - 48^2}$$

$A = -0.08824 \Rightarrow$ $\boxed{a = 0.00608}$

$$\boxed{y = 16.86397\, e^{-0.08824 \times n}}$$

the error at i^{th} data point given as:

$$\epsilon_{linear,i} = Y_i - (B + An)$$

where $Y_i = \ln(y_i)$ ← data point

$B = \ln(b)$

$A = -2\pi a$

Made with Goodnotes

(ii) Non-linear least squares Approach: the model is given as:

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

the error at the i-th data point for non-linear model:

$$\epsilon_{nonlinear,i} = y_i - be^{-2\pi an}$$

the error in the linear model is determined by the difference between the logarithms of the actual and predicted values. on the other hand, in the non-linear model, the error is calculated directly as the difference between the actual and predicted values, without using any logarithmic transformation. when comparing the error from both approaches, it's evident that they are not identical, however, they can be close when the errors are small. in such cases, the relationship can be approximated as:

$$\ln(\epsilon_{nonlinear,i}) \simeq \epsilon_{linear,i} \quad (\text{for small errors})$$

this means that for minor errors, the linear and nonlinear errors will be approximately related via the logarithmic transformation. However, as the error becomes larger, this weakens, leading to more noticable differences between the linear and nonlinear error calculations.

Made with Goodnotes

Appendix:

1.

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

x = np.array([0.3, 0.4, 0.5, 0.6])
y = np.array([0.740818, 0.670320, 0.606531, 0.548812])

fun = x - y

def neville(x, y, xbar):
    n = len(x)
    Q = np.zeros((n, n))
    Q[:, 0] = y
    for i in range(1, n):
        for j in range(1, i + 1):
            Q[i, j] = ((xbar - x[i - j]) * Q[i, j - 1] - (xbar - x[i]) *
Q[i - 1, j - 1]) / (x[i] - x[i - j])
    return Q[n - 1, n - 1]

def Bisection_method(func, a, b, tol):
    if func(a) * func(b) > 0:
        print("No root found.")
        return None
    while (b - a) / 2 > tol:
        midpoint = a + (b - a) / 2
        if func(midpoint) == 0:
            return midpoint
        elif func(a) * func(midpoint) < 0:
            b = midpoint
        else:
            a = midpoint
    return midpoint

func_x = lambda x: x - np.exp(-x)
x_root = Bisection_method(func_x, 0.5, 0.6, 1e-10)
print(x_root)

fun_value = 0
# Example value for interpolation
print(neville(fun, x, fun_value))
```

2.

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

t_i = 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_i = 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_i = 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])

def neville(x, y, xbar):
    n = len(x)
    Q = np.zeros((n, n))
    Q[:, 0] = y
    for i in range(n):
        for j in range(1, i + 1):
            if x[i] == x[i - j]:
                raise ValueError(f"Division by zero detected at i={i}, j={j}")
            Q[i, j] = ((xbar - x[i - j]) * Q[i, j - 1] - (xbar - x[i]) * Q[i - 1, j - 1]) / (x[i] - x[i - j])
    return Q[n - 1, n - 1]

# Create a vector of t values ranging from 0 to 12, 0.12 apart
T = np.linspace(0, 12, 100)
X = np.zeros(no_of_points)
Y = np.zeros(no_of_points)

# Calculate interpolated values for X and Y
for j, t in enumerate(T):
    X[j] = neville(t_i, x_i, t)
    Y[j] = neville(t_i, y_i, t)

# Create subplots
fig, ax = plt.subplots(1, 2, figsize=(15, 5))
# Plot x(t) vs t on the first subplot
ax[0].plot(T, X, 'b-', label='f(t)')
ax[0].plot(t_i, x_i, 'ro', label='Original x points')
ax[0].set_title("f(t) vs t")
ax[0].set_xlabel("t")
ax[0].set_ylabel("f(t)")
ax[0].legend()
# Plot y(t) vs t on the first subplot
ax[0].plot(T, Y, 'g-', label='g(t)')
ax[0].plot(t_i, y_i, 'ro', label='Original y points')
ax[0].set_title("f(t) and g(t) vs t")
ax[0].set_xlabel("t")
ax[0].set_ylabel("Values")
ax[0].legend()
# Plot y(t) vs x(t) on the second subplot
ax[1].plot(X, Y, 'b-', label='g(t) vs f(t)')
ax[1].plot(x_i, y_i, 'ro', label='Original points')
ax[1].set_title("g(t) Vs f(t)")
ax[1].set_xlabel("f(t)")
ax[1].set_ylabel("g(t)")
ax[1].legend()
plt.show()
```

3.

```
import numpy as np
from scipy.interpolate import CubicSpline
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])

# Natural Cubic Splines for f(t) and g(t)
cs_f = CubicSpline(t, x, bc_type='natural')
cs_g = CubicSpline(t, y, bc_type='natural')

# Get the spline coefficients for f(t) and g(t)
coeffs_f = cs_f.c  # Coefficients for f(t)
coeffs_g = cs_g.c  # Coefficients for g(t)


# Print the coefficients of f(t)
print("Cubic Spline Coefficients for f(t):")
for i in range(len(t) - 1):
    print(f"Interval {i}: a = {coeffs_f[3, i]:.6f}, b = {coeffs_f[2, i]:.6f}, c =
{coeffs_f[1, i]:.6f}, d = {coeffs_f[0, i]:.6f}")

# Print the coefficients of g(t)
print("\nCubic Spline Coefficients for g(t):")
for i in range(len(t) - 1):
    print(f"Interval {i}: a = {coeffs_g[3, i]:.6f}, b = {coeffs_g[2, i]:.6f}, c =
{coeffs_g[1, i]:.6f}, d = {coeffs_g[0, i]:.6f}")

# Plot the splines along with the original data points
t_new = np.linspace(min(t), max(t), 200)

fx, ax = plt.subplots(1, 2, figsize=(15, 6))
# Plot f(t) and g(t) splines on one plot
ax[0].plot(t, x, 'o', label='Data points (f(t))')
ax[0].plot(t_new, cs_f(t_new), label='Cubic Spline f(t)')
ax[0].plot(t, y, 'x', label='Data points (g(t))')
ax[0].plot(t_new, cs_g(t_new), label='Cubic Spline g(t)')
ax[0].set_title('Natural Cubic Spline for f(t) and g(t)')
ax[0].set_xlabel('t')
ax[0].set_ylabel('Values')
ax[0].legend()

# Plot f(t) vs g(t)
ax[1].plot(cs_f(t_new), cs_g(t_new), label='Cubic Spline f(t) vs g(t)')
ax[1].plot(x, y, 'o', label='Original Data Points')
ax[1].set_title('g(t) vs f(t)')
ax[1].set_xlabel('f(t)')
ax[1].set_ylabel('g(t)')
ax[1].legend()

plt.show()
```