Name:	Aneesh Panchal	**Homework No:**	Homework 3
SR No:	06-18-01-10-12-24-1-25223	**Course Code:**	DS288
Email ID:	aneeshp@iisc.ac.in	**Course Name:**	Numerical Methods
Date:	October 8, 2024	**Term:**	AUG 2024

Solution 1

Here we are given function as $f(x) = e^{-x}$ and we have to find the root for the equation,

$$g(x) = x - f(x) = x - e^{-x}$$

For this purpose the method or, algorithm used is described as follows,

1. Find values of $g(x_i) = x_i - y_i$ for every given values of i and construct Neville's Table using values $(x_i, g(x_i))$.
2. Find value in 1st column such that $g(x_i) \times g(x_{i+1}) < 0$, then proceed for the next order polynomial in that region.
3. Find root for the next order polynomial which lies in that region then, update the region accordingly. [Similar to Bisection Method using Neville's Interpolating Polynomials]
4. Recursively propagate to highest degree polynomial which leads to best root possible using given number of points.

Results:

Neville's polynomials with **Bisection method** is implemented for finding the roots of the equation $g(x) = x - e^{-x}$.

Newton Method is used for finding the True value (exact solution) of the equation $g(x)$ with relative tolerance $1e-6$ because it have 2nd order convergence and $g'(x)$ is well defined in whole domain.

Relative Error: **3.36467638805107e-6**
Predicted Value of x: **0.567145198663419**
True Value of x: **0.567143290409781**

Solution 2

Figure 1 shows the plot for function $f(t)$ and Figure 2 shows the plot for function $g(t)$.
Figure 3 shows the interpolated shape $P(t) \equiv P(f(t), g(t))$ using $dt = 0.1$, Figure 4 shows the same using $dt = 0.5$ and Figure 5 shows the same using $dt = 1$. It is clear from the Figures 3, 4 and 5 that for $dt = 1$ generates the alphabet (**e**) which is somewhat recognizable, this is due to the reason that Neville's method generates the global polynomial which is not a good fit for such problems.

Results:

For reasonable results (plotting of alphabet (**e**)) using Neville's interpolating polynomial method,
Value of **dt** used: **1**
(this is because Neville's method generates the **global interpolating polynomial**)

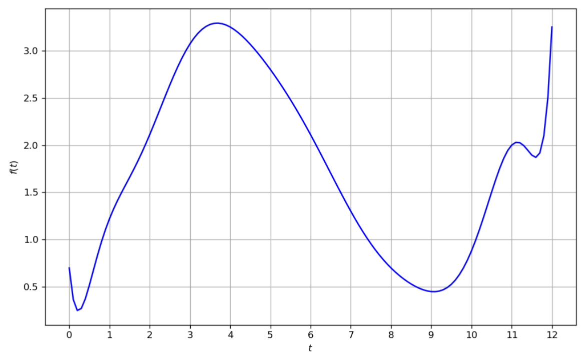


Figure 1: Plot for Function $f(t)$

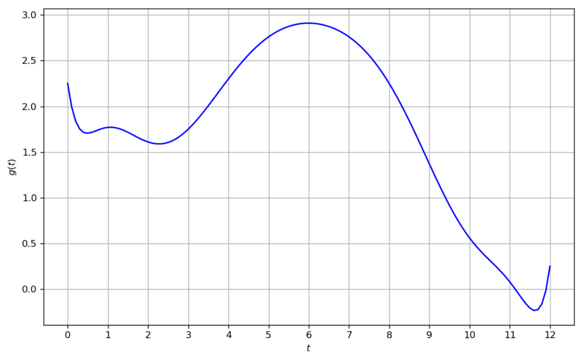


Figure 2: Plot for Function $g(t)$

Name:	Aneesh Panchal	**Homework No:**	Homework 3
SR No:	06-18-01-10-12-24-1-25223	**Course Code:**	DS288
Email ID:	aneeshp@iisc.ac.in	**Course Name:**	Numerical Methods
Date:	October 8, 2024	**Term:**	AUG 2024

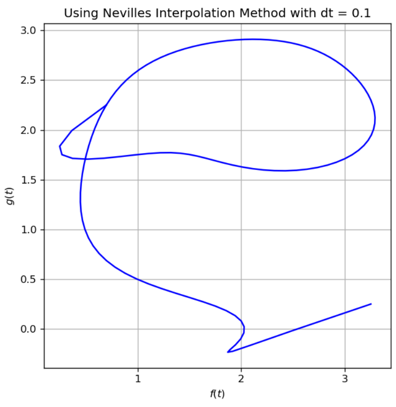


Figure 3: Neville's Method ($dt = 0.1$)

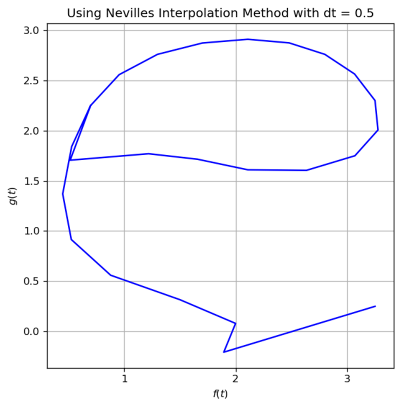


Figure 4: Neville's Method ($dt = 0.5$)

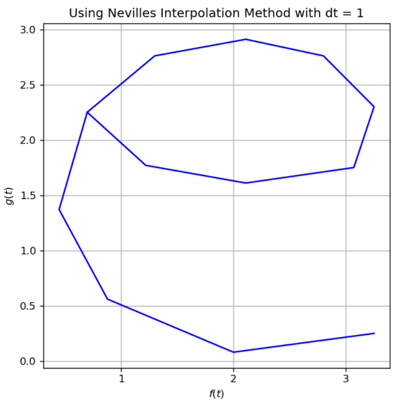


Figure 5: Neville's Method ($dt = 1$)

Solution 3

Let Cubic splines interpolating polynomials for the functions $f(t)$ and $g(t)$ be of the form,

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

$$g_i(t) = p_i + q_i(t - t_i) + r_i(t - t_i)^2 + s_i(t - t_i)^3$$

Analysis for $f(t)$

Here we are using Natural Cubic splines with total number of t values as 13. Hence, we require total of 12 cubic equation for $f(t)$ and hence, 48 constraints are required.
$f_i(t_i) = x_i$ gives **12** constraints.
$f_i(t_{i+1}) = x_{i+1}$ gives **12** constraints.
$f'_i(t_{i+1}) = f'_{i+1}(t_{i+1})$ gives **11** constraints.
$f''_i(t_{i+1}) = f''_{i+1}(t_{i+1})$ gives **11** constraints.
Finally, Natural Splines constraints gives the remaining **2** constraints, which are, $f''_0(t_0) = 0$ and $f''_{11}(t_{12}) = 0$

Analysis for $g(t)$

Here we are using Natural Cubic splines with total number of t values as 13. Hence, we require total of 12 cubic equation for $g(t)$ as well and hence, 48 constraints are required.
$g_i(t_i) = y_i$ gives **12** constraints.
$g_i(t_{i+1}) = y_{i+1}$ gives **12** constraints.
$g'_i(t_{i+1}) = g'_{i+1}(t_{i+1})$ gives **11** constraints.
$g''_i(t_{i+1}) = g''_{i+1}(t_{i+1})$ gives **11** constraints.
Finally, Natural Splines constraints gives the remaining **2** constraints, which are, $g''_0(t_0) = 0$ and $g''_{11}(t_{12}) = 0$

Results:

The coefficients of f(t) and g(t) for different intervals are provided in Table 1 and 2. The alphabet (letter) produced using Natural Cubic Spline is much better than what we got using Neville's interpolation method. Some of the comparisons and differences observed are as follows,

1. **Neville's method provides a global interpolation** method and **Natural Cubic Spline method provides a local interpolation** method. This is the reason why Natural Cubic Spline performs better than Neville's method for **dt = 0.1**.
2. Due to being global interpolation method, Neville's method exhibits oscillations around edge corners which can be seen in Figure 3 but forms smooth curve when natural cubic spline interpolation is used (due to its local behaviour) which can be seen in Figure 9.

Table 1: Coefficient values for Cubic Splines for function $f(t)$

Interval	a_i	b_i	c_i	d_i
0.0 - 1.0	0.70000000	0.437914579	0.00000000	0.0820854211
1.0 - 2.0	1.22000000	0.684170842	0.246256263	-0.0404271053
2.0 - 3.0	2.11000000	1.05540205	0.124974947	-0.220377000
3.0 - 4.0	3.07000000	0.644220947	-0.536156052	0.0719351051
4.0 - 5.0	3.25000000	-0.212285842	-0.320350737	0.0826365797
5.0 - 6.0	2.80000000	-0.605077578	-0.0724409981	-0.0124814240
6.0 - 7.0	2.11000000	-0.787403846	-0.109885270	0.0872891163
7.0 - 8.0	1.30000000	-0.745307038	0.151982079	-0.00667504128
8.0 - 9.0	0.70000000	-0.461368004	0.131956955	0.0794110488
9.0 - 10.0	0.45000000	0.0407790526	0.370190101	0.0190308461
10.0 - 11.0	0.88000000	0.838251794	0.427282640	-0.145534433
11.0 - 12.0	2.00000000	1.25621377	-0.00932065991	0.00310688664

Table 2: Coefficient values for Cubic Splines for function $g(t)$

Interval	p_i	q_i	r_i	s_i
0.0 - 1.0	2.25000000	-0.552213337	-1.38777878e-17	0.0722133367
1.0 - 2.0	1.77000000	-0.335573327	0.216640010	-0.0410666837
2.0 - 3.0	1.61000000	-0.0254933572	0.0934399590	0.0720533982
3.0 - 4.0	1.75000000	0.377546755	0.309600154	-0.137146909
4.0 - 5.0	2.30000000	0.585306335	-0.101840574	-0.0234657614
5.0 - 6.0	2.76000000	0.311227903	-0.172237858	0.0110099546
6.0 - 7.0	2.91000000	-0.000217948718	-0.139207994	-0.0105740572
7.0 - 8.0	2.76000000	-0.310356108	-0.170930166	-0.0287137258
8.0 - 9.0	2.25000000	-0.738357617	-0.257071343	0.115428961
9.0 - 10.0	1.37000000	-0.906213422	0.0892155383	0.00699788381
10.0 - 11.0	0.56000000	-0.706788694	0.110209190	0.116579504
11.0 - 12.0	0.08000000	-0.136631802	0.459947703	-0.153315901

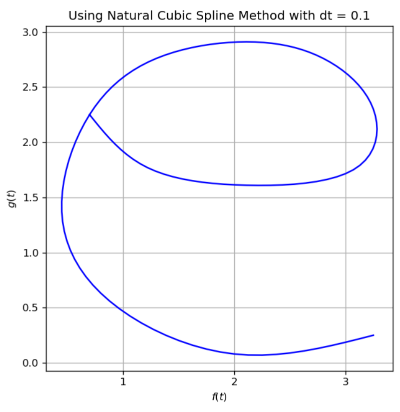


Figure 6: Cubic Spline ($dt = 0.1$)

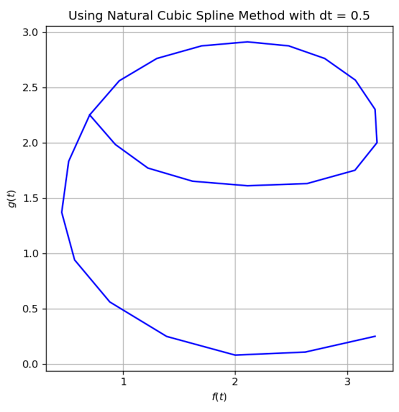


Figure 7: Cubic Spline ($dt = 0.5$)

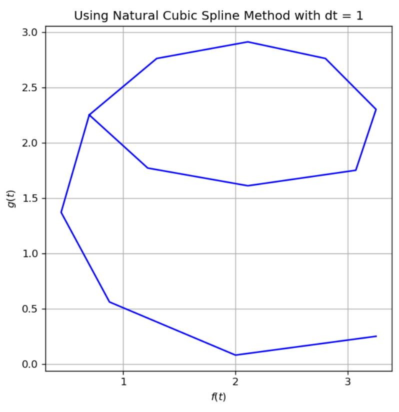


Figure 8: Cubic Spline ($dt = 1$)

Name: Aneesh Panchal
SR No: 06-18-01-10-12-24-1-25223
Email ID: aneeshp@iisc.ac.in
Date: October 8, 2024

Homework No: Homework 3
Course Code: DS288
Course Name: Numerical Methods
Term: AUG 2024

Solution 4

Given exponential function is of the form,

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

Linear Least Square Fit

For change of variables, taking natural log both side we get,

$$\log y = \log b - 2\pi ax$$

Now, the linear least square error function is of the form,

$$E(a,b) = \sum_{i=1}^{N} (\log(y_i) - \log b - 2\pi a x_i)^2 \tag{1}$$

Taking partial derivative with respect to variables and calculating them to 0, we get the system as follows,

$$\begin{bmatrix} 2\pi \sum_{i=1}^{N} x_i^2 & -\sum_{i=1}^{N} x_i \\ 2\pi \sum_{i=1}^{N} x_i & -N \end{bmatrix} \begin{bmatrix} a \\ \log(b) \end{bmatrix} = \begin{bmatrix} -\sum_{i=1}^{N} x_i \log(y_i) \\ -\sum_{i=1}^{N} \log(y_i) \end{bmatrix}$$

Now using Linear Least Squares Fit on the given data points, and we get the values of the unknown coefficients a and b as follows,

$$a = \mathbf{0.006086559661783669} \quad \text{and} \quad b = \mathbf{16.86397450267915}$$

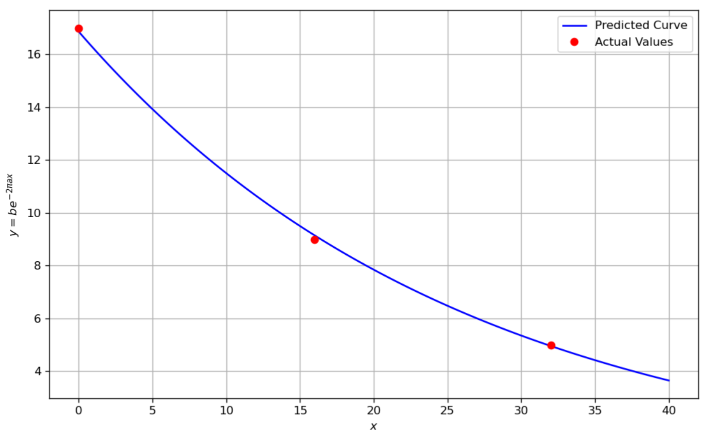


Figure 9: Parameter Estimation using Linear Least Square Method

Non-Linear Least Squares Fit

Here, the non-linear least square error function is of the form,

$$E_N(a,b) = \sum_{i=1}^{N} (y_i - be^{-2\pi a x_i})^2 \tag{2}$$

Results:

No, the non-linear least square fit approach will not lead to the same values of the parameters because, when the number of cycles (x) becomes very large then the error value from equation (1) becomes very large as y will become very small which is not in the case of equation (2). Hence, non-linear least square fit would be better approach than linear approach with change of variables. Hence, if we dont know the whole data then it is better to estimate the parameter using non-linear approach if non-linear functions are implemented.

1 Appendix: Assignment 3 Programming

[1]:
```
import numpy as np
import matplotlib.pyplot as plt
import matplotlib.ticker as mticker
import sympy as sym
import math

x = sym.symbols('x')
t = sym.symbols('t')
```

1.1 Question 1

[2]:
```
# Root of val = x - e^(-x) and here, e^(-x) = f_exp
xval = [0.3, 0.4, 0.5, 0.6]
f_exp = [0.740818, 0.670320, 0.606531, 0.548812]

val = [xval[i] - f_exp[i] for i in range(len(xval))]
NevilleTable = sym.Matrix(len(xval), len(xval), lambda i, j: 0)

for i in range(len(val)):
    NevilleTable[i,0] = val[i]

for i in range(len(xval)):
    for j in range(i):
        k = j+1
        NevilleTable[i,k] = ((x - xval[i-k])*NevilleTable[i,k-1] - (x -␣
↪xval[i])*NevilleTable[i-1,k-1])/(xval[i] - xval[i-k])

print(sym.simplify(NevilleTable))
```

```
Matrix([[-0.440818000000000, 0, 0, 0], [-0.270320000000000, 1.70498*x -
0.952312, 0, 0], [-0.106531000000000, 1.63789*x - 0.925476,
-0.335449999999998*x**2 + 1.939795*x - 0.992566, 0], [0.0511880000000000,
1.57719*x - 0.895126, -0.303499999999995*x**2 + 1.91104*x - 0.986175999999999,
0.106500000000005*x**3 - 0.463250000000006*x**2 + 1.98985*x -
0.998956000000001]])
```

[3]:
```
xbound = [0,0]
k = 0
while True:
    if (NevilleTable[k,0].subs(x,xval[k]))*(NevilleTable[k+1,0].subs(x,xval[k+1])) < 0:
        xbound[0] = xval[k]
        xbound[1] = xval[k+1]
        k = k+1
        break
    k = k+1

for j in range(1,len(xval)):
    xvalues = sym.nroots(NevilleTable[k,j])
    xvalues = [root for root in xvalues if root.is_real]
    xvalues = [float(root) for root in xvalues]
    for i in range(len(xvalues)):
        if xvalues[i]<=xbound[1] and xvalues[i]>=xbound[0]:
            xvalue = xvalues[i]
    print(f"For i = {j}  --> ", xvalue)

    if j < len(xval) - 1:
        if (NevilleTable[k-1,j+1].subs(x,xbound[0]))*(NevilleTable[k,j+1].
↪subs(x,xvalue)) < 0:
            xbound[1] = xvalue
        elif (NevilleTable[k,j+1].subs(x,xvalue))*(NevilleTable[k,j+1].
↪subs(x,xbound[1])) < 0:
            xbound[0] = xvalue
```

```
For i = 1  -->  0.567544810707651
For i = 2  -->  0.5671201348931916
For i = 3  -->  0.567145198663419
```

[4]:
```
def f(x):
    return x - sym.exp(-x)
def fp(x):
    return sym.diff(f(x),x)

p = [0]
tol = 1e-6
iter = 0
while True:
    p_new = p[iter] - (f(x).subs(x,p[iter])/fp(x).subs(x,p[iter]))
    p_new = sym.N(p_new)
    p.append(p_new)
    if abs((p[iter+1]-p[iter])/p[iter+1])<=tol:
        iter = iter+1
        break
    iter = iter+1

x_true = p[iter]
f_true = f(x).subs(x,p[iter])

print("True Value of x: ", x_true)
print("True Value of f(x): ", f_true)
```

```
True Value of x:  0.567143290409781
True Value of f(x):  -4.44089209850063e-15
```

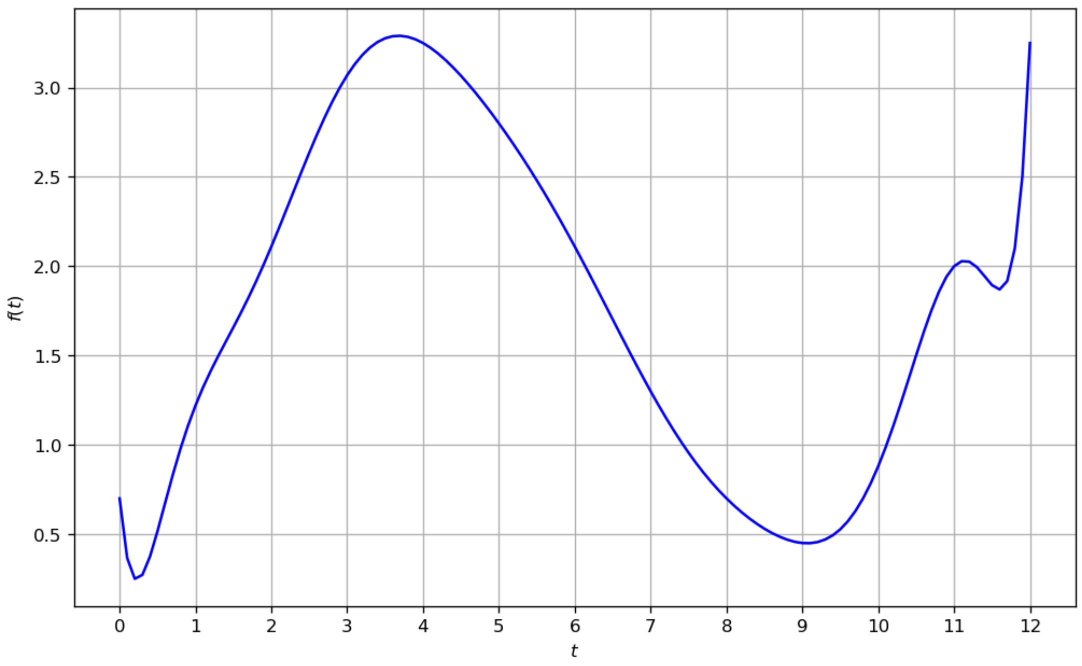
[5]:
```
x_pred = xvalue
f_pred = NevilleTable[len(xval)-1,len(xval)-1].subs(x,xvalue)

print("Predicted Value of x: ", x_pred)
print("Predicted Value of f(x): ", f_pred)

rel_err_x = abs(x_pred - x_true)/abs(x_true)
print("\nRelative Error: ", rel_err_x)
```

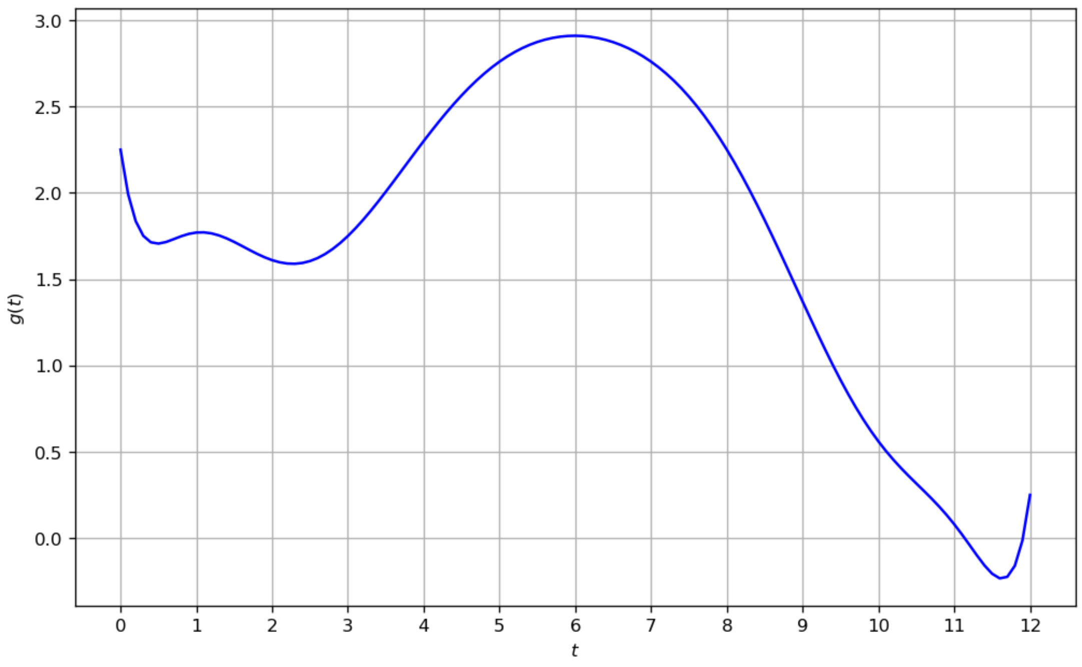
```
Predicted Value of x:  0.567145198663419
Predicted Value of f(x):  -2.07510197176505e-15

Relative Error:  3.36467638805107e-6
```

1.2 Question 2

[6]:
```
tval = [0,1,2,3,4,5,6,7,8,9,10,11,12]
xval = [0.7,1.22,2.11,3.07,3.25,2.8,2.11,1.3,0.7,0.45,0.88,2,3.25]
yval = [2.25,1.77,1.61,1.75,2.3,2.76,2.91,2.76,2.25,1.37,0.56,0.08,0.25]

NevilleTablex = sym.Matrix(len(xval), len(xval), lambda i, j: 0)
NevilleTabley = sym.Matrix(len(yval), len(yval), lambda i, j: 0)

for i in range(len(tval)):
    NevilleTablex[i,0] = xval[i]
    NevilleTabley[i,0] = yval[i]

for i in range(len(tval)):
    for j in range(i):
        k = j+1
        NevilleTablex[i,k] = ((t - tval[i-k])*NevilleTablex[i,k-1] - (t -␣
↪tval[i])*NevilleTablex[i-1,k-1])/(tval[i] - tval[i-k])
        NevilleTabley[i,k] = ((t - tval[i-k])*NevilleTabley[i,k-1] - (t -␣
↪tval[i])*NevilleTabley[i-1,k-1])/(tval[i] - tval[i-k])

f = NevilleTablex[len(tval)-1,len(tval)-1]
g = NevilleTabley[len(tval)-1,len(tval)-1]
```

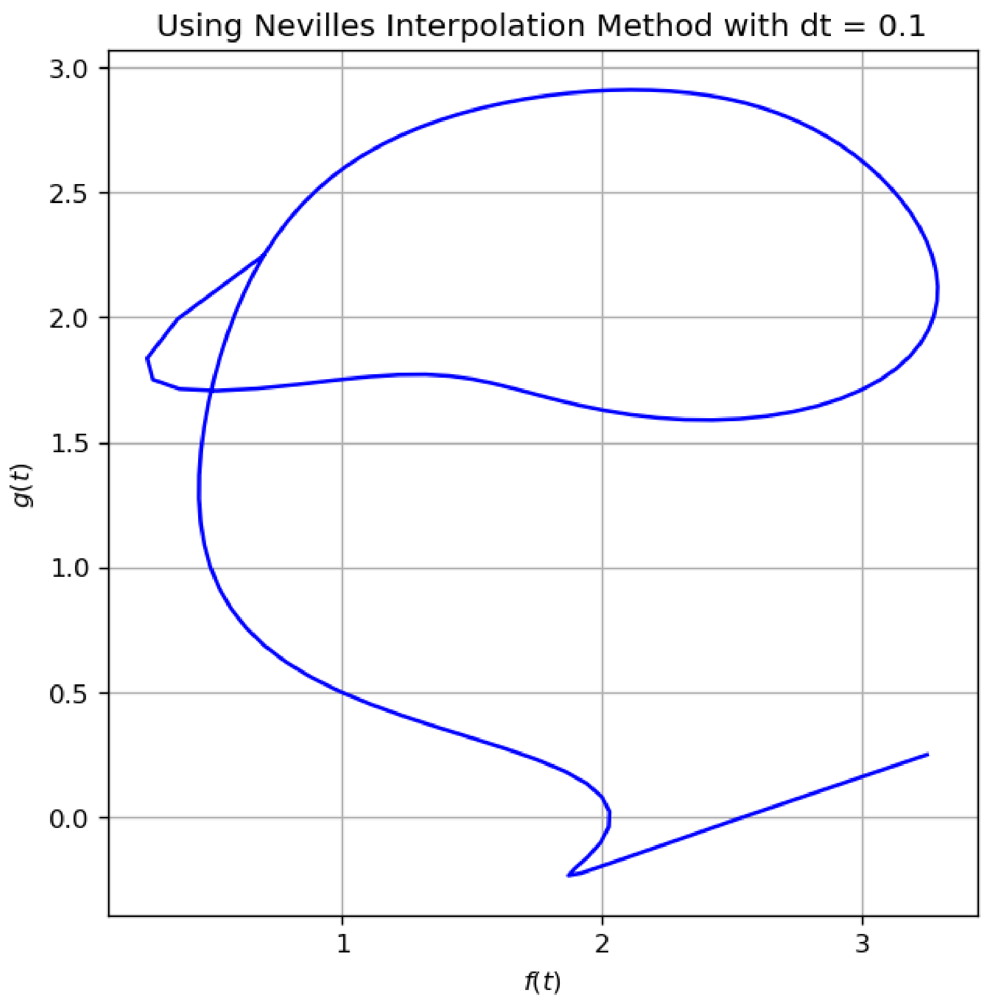
[7]:
```
values = np.arange(0, 12 + 0.1, 0.1)
fvalues = [f.subs(t,i) for i in values]
gvalues = [g.subs(t,i) for i in values]

plt.figure(figsize=(10, 6), dpi=120)
plt.plot(values, fvalues,'b')
plt.gca().xaxis.set_major_locator(mticker.MultipleLocator(1))
plt.grid(True, which="both", ls="-")
plt.xlabel(r"$t$")
plt.ylabel(r"$f(t)$")
plt.savefig('ft.png')
plt.show()

plt.figure(figsize=(10, 6), dpi=120)
plt.plot(values, gvalues,'b')
plt.gca().xaxis.set_major_locator(mticker.MultipleLocator(1))
plt.grid(True, which="both", ls="-")
plt.xlabel(r"$t$")
plt.ylabel(r"$g(t)$")
plt.savefig('gt.png')
plt.show()
```

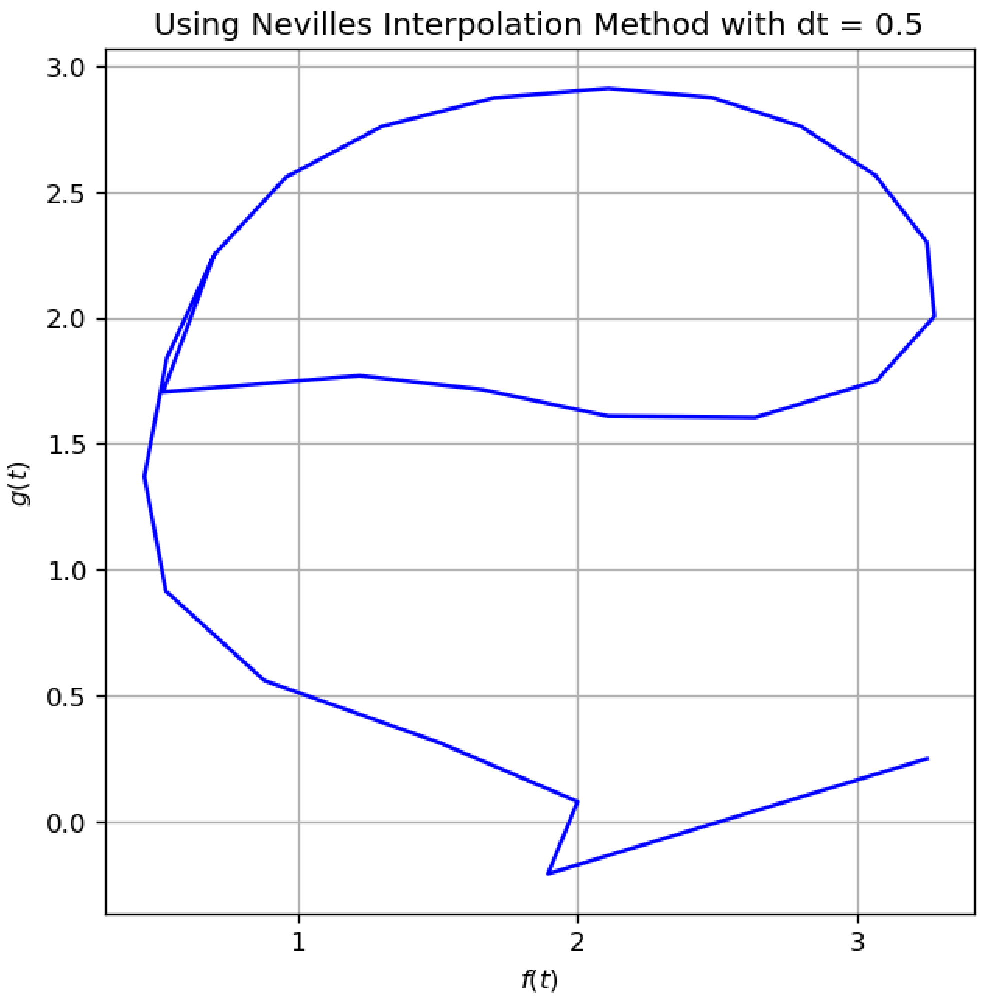


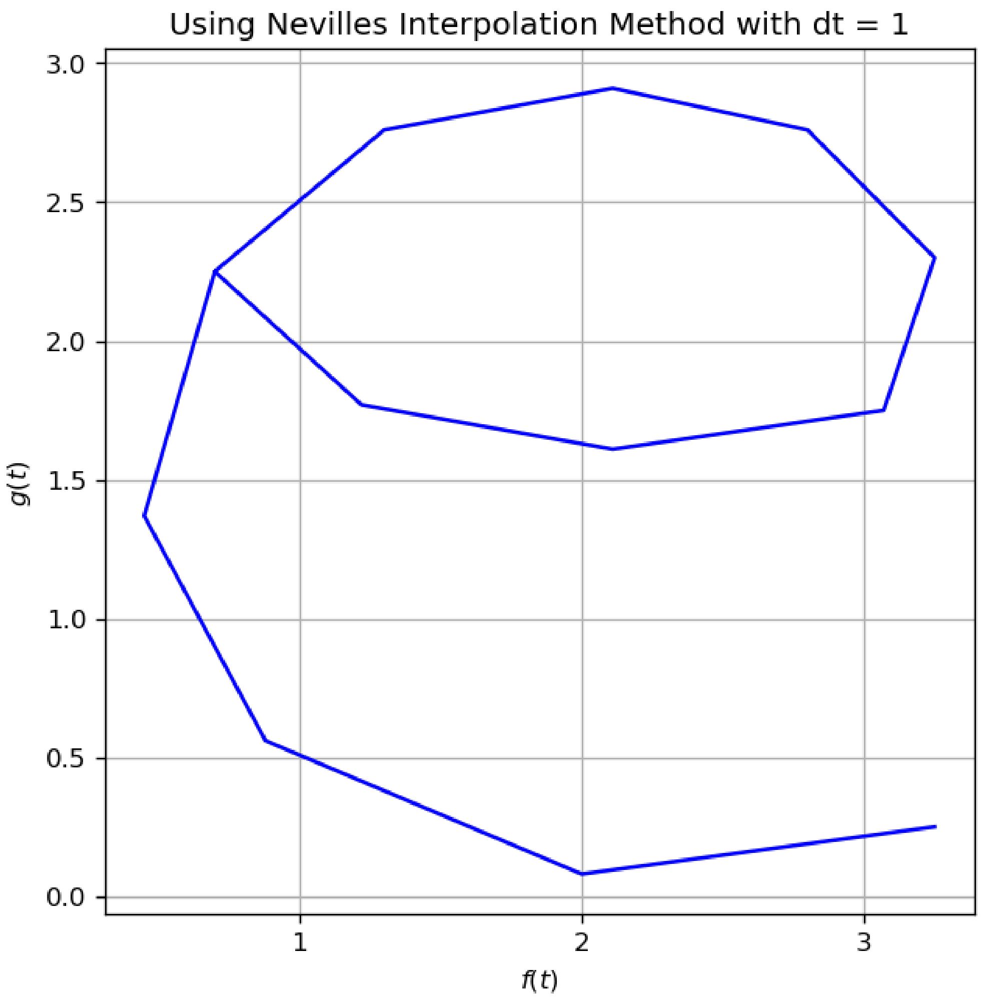


[8]:
```
dtval = [0.1, 0.5, 1]
for dt in dtval:
    a = 0
```

```
b = 12
fval = []
gval = []
while a<=b:
    fval.append(f.subs(t,a))
    gval.append(g.subs(t,a))
    a = a + dt
plt.figure(figsize=(6, 6), dpi=120)
plt.plot(fval,gval,'b')
plt.gca().xaxis.set_major_locator(mticker.MultipleLocator(1))
plt.grid(True, which="both", ls="-")
plt.xlabel(r"$f(t)$")
plt.ylabel(r"$g(t)$")
plt.title(f"Using Nevilles Interpolation Method with dt = {dt}")
plt.savefig(f'fgNeville{dt}.png')
plt.show()
```




Using Nevilles Interpolation Method with dt = 0.5
3.0
2.5
2.0
1.5
1.0
0.5
0.0
$g(t)$
1
2
3
$f(t)$



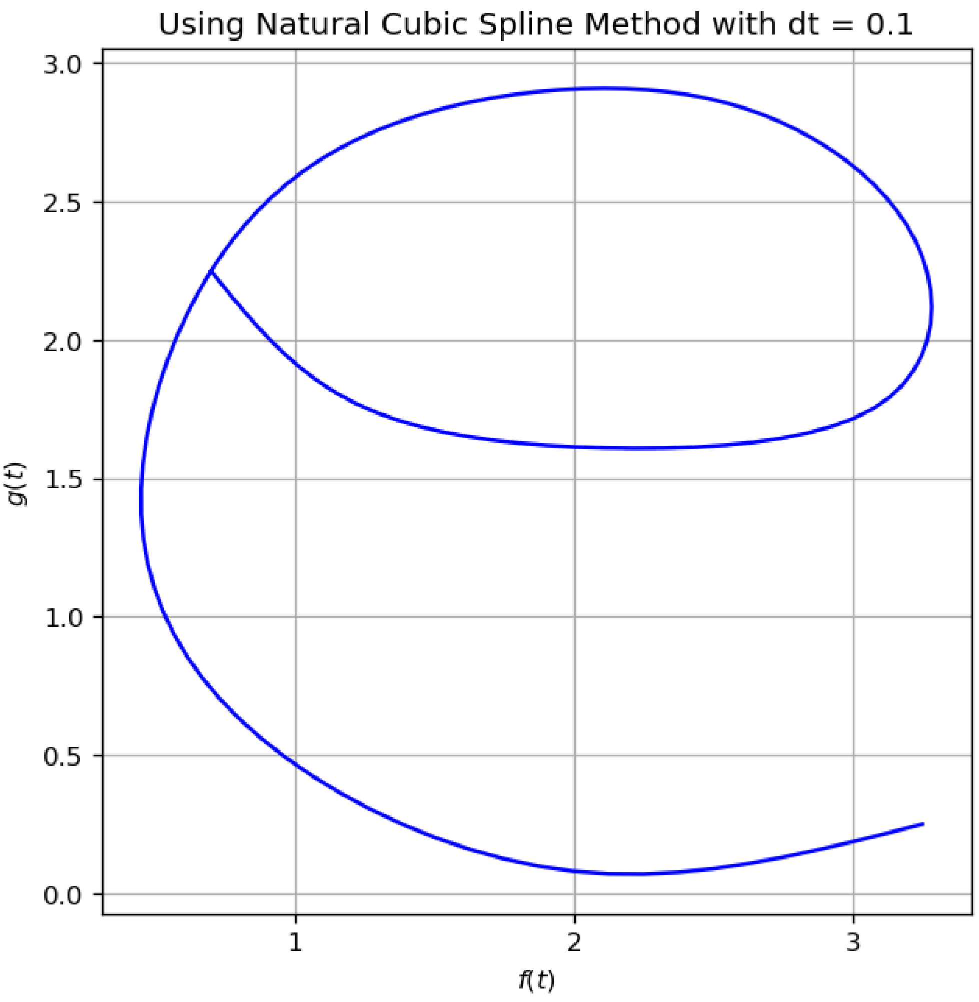
1.3 Question 3

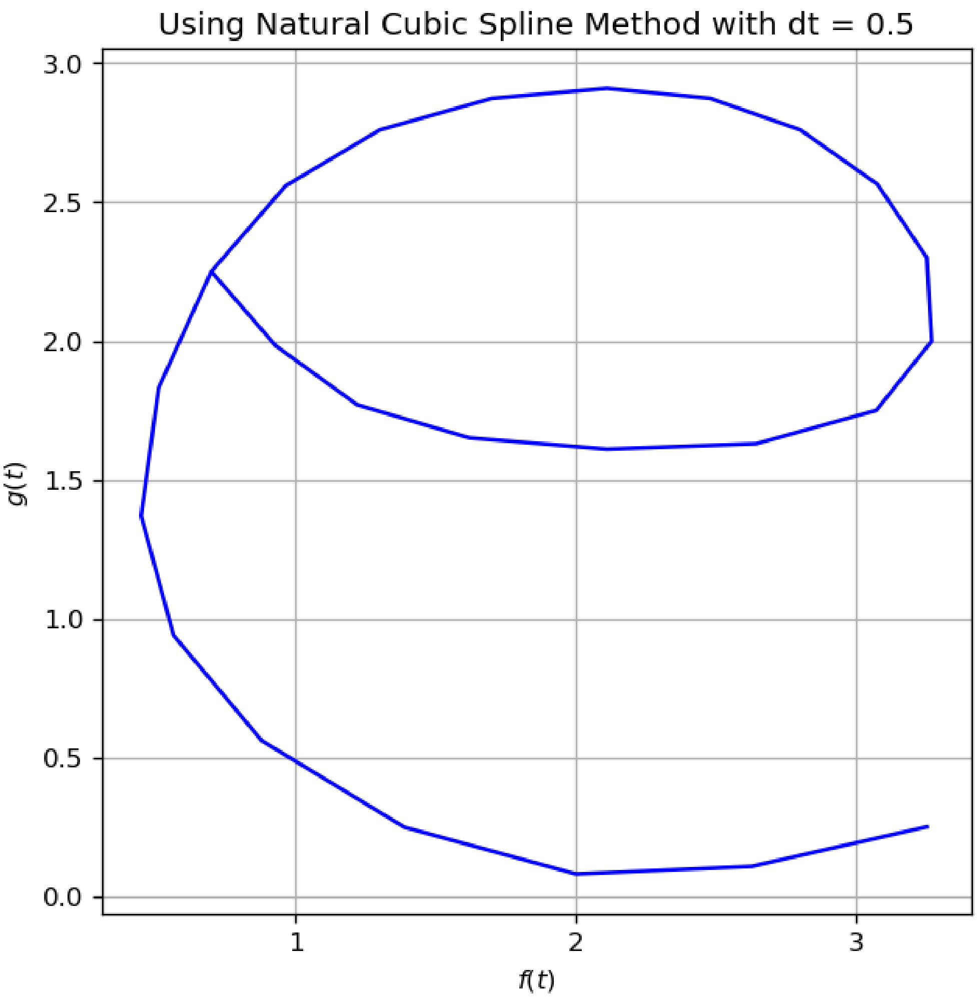
[9]:
```
tval = [0,1,2,3,4,5,6,7,8,9,10,11,12]
xval = [0.7,1.22,2.11,3.07,3.25,2.8,2.11,1.3,0.7,0.45,0.88,2,3.25]
yval = [2.25,1.77,1.61,1.75,2.3,2.76,2.91,2.76,2.25,1.37,0.56,0.08,0.25]

Matx = np.zeros((4*(len(tval)-1),4*(len(tval)-1)))
bx = []
Maty = np.zeros((4*(len(tval)-1),4*(len(tval)-1)))
by = []

row = 0
col = 0

for i in range(len(tval)-1):
    Matx[row,4*col] = 1
    bx.append(xval[i])
    Maty[row,4*col] = 1
    by.append(yval[i])

    row = row + 1
    col = col + 1

col = 0
for i in range(len(tval)-1):
    Matx[row,4*col] = 1
    Matx[row,4*col+1] = tval[i+1] - tval[i]
    Matx[row,4*col+2] = (tval[i+1] - tval[i])**2
    Matx[row,4*col+3] = (tval[i+1] - tval[i])**3
    bx.append(xval[i+1])

    Maty[row,4*col] = 1
    Maty[row,4*col+1] = tval[i+1] - tval[i]
    Maty[row,4*col+2] = (tval[i+1] - tval[i])**2
    Maty[row,4*col+3] = (tval[i+1] - tval[i])**3
    by.append(yval[i+1])

    row = row + 1
    col = col + 1

col = 0
for i in range(len(tval)-2):
    Matx[row,4*col+1] = 1
    Matx[row,4*col+2] = 2*(tval[i+1] - tval[i])
    Matx[row,4*col+3] = 3*(tval[i+1] - tval[i])**2
    Matx[row,4*(col+1)+1] = -1
    bx.append(0)

    Maty[row,4*col+1] = 1
    Maty[row,4*col+2] = 2*(tval[i+1] - tval[i])
    Maty[row,4*col+3] = 3*(tval[i+1] - tval[i])**2
    Maty[row,4*(col+1)+1] = -1
    by.append(0)

    row = row + 1
    col = col + 1

col = 0
for i in range(len(tval)-2):
    Matx[row,4*col+2] = 2
    Matx[row,4*col+3] = 6*(tval[i+1] - tval[i])
    Matx[row,4*(col+1)+2] = -2
    bx.append(0)

    Maty[row,4*col+2] = 2
    Maty[row,4*col+3] = 6*(tval[i+1] - tval[i])
    Maty[row,4*(col+1)+2] = -2
    by.append(0)
    row = row + 1
    col = col + 1

# Natural Spline
Matx[row,2] = 2
bx.append(0)
Matx[row+1,-2] = 2
Matx[row+1,-1] = 6
bx.append(0)

Maty[row,2] = 2
by.append(0)
Maty[row+1,-2] = 2
Maty[row+1,-1] = 6
by.append(0)

# Find solution to linear system of equation
coeffx = np.linalg.solve(Matx, bx)
coeffy = np.linalg.solve(Maty, by)
print(coeffx)
print(coeffy)

fvalues = sym.Matrix(1, len(tval)-1, lambda i, j: 0)
gvalues = sym.Matrix(1, len(tval)-1, lambda i, j: 0)
vals = 0
for i in range(len(fvalues)):
    fvalues[i] = coeffx[vals] + coeffx[vals+1]*(t - tval[i]) + coeffx[vals+2]*(t -␣
↪tval[i])**2 + coeffx[vals+3]*(t - tval[i])**3
    gvalues[i] = coeffy[vals] + coeffy[vals+1]*(t - tval[i]) + coeffy[vals+2]*(t -␣
↪tval[i])**2 + coeffy[vals+3]*(t - tval[i])**3
    vals = vals + 4
```

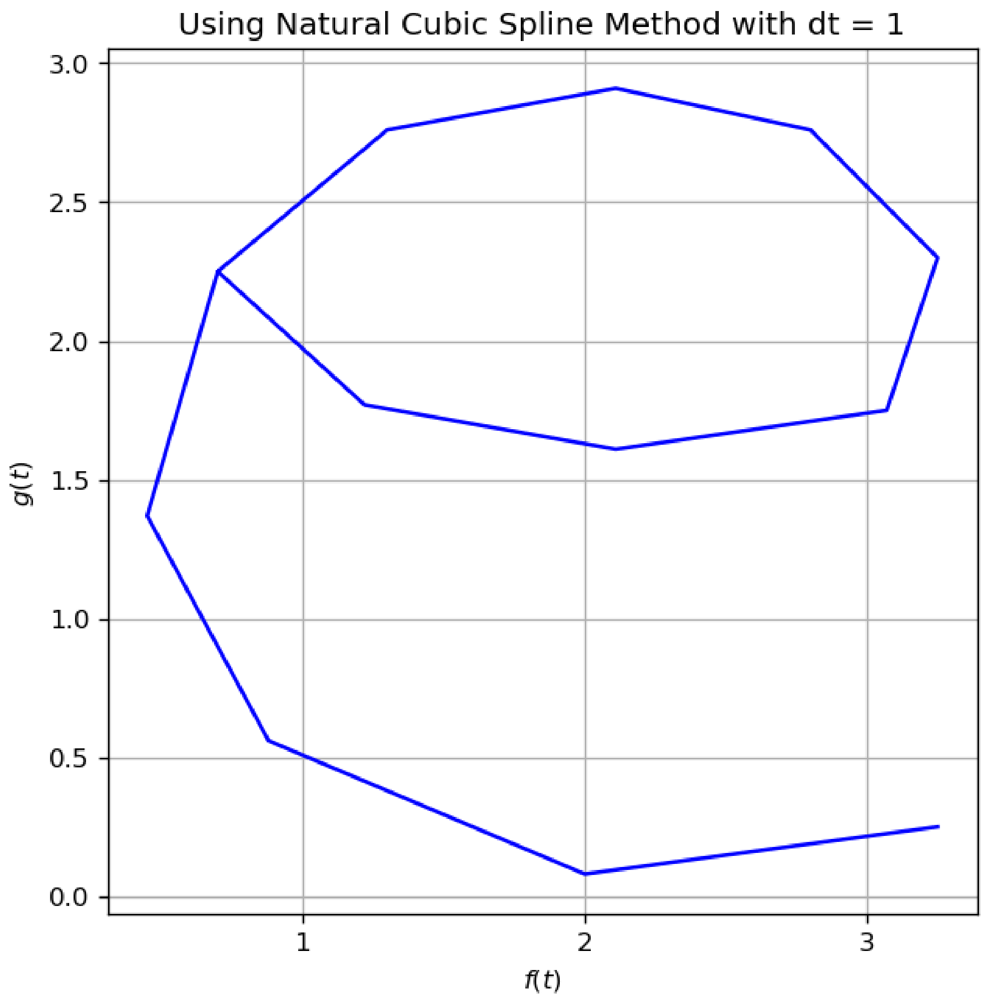
```
[ 7.00000000e-01  4.37914579e-01  0.00000000e+00  8.20854211e-02
  1.22000000e+00  6.84170842e-01  2.46256263e-01 -4.04271053e-02
  2.11000000e+00  1.05540205e+00  1.24974947e-01 -2.20377000e-01
  3.07000000e+00  6.44220947e-01 -5.36156052e-01  7.19351051e-02
  3.25000000e+00 -2.12285842e-01 -3.20350737e-01  8.26365797e-02
  2.80000000e+00 -6.05077578e-01 -7.24409981e-02 -1.24814240e-02
  2.11000000e+00 -7.87403846e-01 -1.09885270e-01  8.72891163e-02
  1.30000000e+00 -7.45307038e-01  1.51982079e-01 -6.67504128e-03
  7.00000000e-01 -4.61368004e-01  1.31956955e-01  7.94110488e-02
  4.50000000e-01  4.07790526e-02  3.70190101e-01  1.90308461e-02
  8.80000000e-01  8.38251794e-01  4.27282640e-01 -1.45534433e-01
  2.00000000e+00  1.25621377e+00 -9.32065991e-03  3.10688664e-03]
[ 2.25000000e+00 -5.52213337e-01 -1.38777878e-17  7.22133367e-02
  1.77000000e+00 -3.35573327e-01  2.16640010e-01 -4.10666837e-02
  1.61000000e+00 -2.54933572e-02  9.34399590e-02  7.20533982e-02
  1.75000000e+00  3.77546755e-01  3.09600154e-01 -1.37146909e-01
  2.30000000e+00  5.85306335e-01 -1.01840574e-01 -2.34657614e-02
  2.76000000e+00  3.11227903e-01 -1.72237858e-01  1.10099546e-02
  2.91000000e+00 -2.17948718e-04 -1.39207994e-01 -1.05740572e-02
  2.76000000e+00 -3.10356108e-01 -1.70930166e-01 -2.87137258e-02
  2.25000000e+00 -7.38357617e-01 -2.57071343e-01  1.15428961e-01
  1.37000000e+00 -9.06213422e-01  8.92155383e-02  6.99788381e-03
```

```
 5.60000000e-01 -7.06788694e-01  1.10209190e-01  1.16579504e-01
 8.00000000e-02 -1.36631802e-01  4.59947703e-01 -1.53315901e-01]
```

[10]:
```python
dtval = [0.1, 0.5, 1]

def find_index(tvalue, t_values):
    k = 0
    for i in range(len(t_values)-1):
        if tvalue>t_values[i+1]:
            k = k+1
        else:
            break
    return k

for dt in dtval:
    a = 0
    b = 12
    t_values = np.arange(a,b + dt,dt)
    func_count = 0
    fval = []
    gval = []
    for i in range(len(t_values)):
        tvalue = t_values[i]
        func_count = find_index(tvalue, tval)
        if tvalue > b:
            break
        fval.append(fvalues[func_count].subs(t,tvalue))
        gval.append(gvalues[func_count].subs(t,tvalue))
    plt.figure(figsize=(6, 6), dpi=120)
    plt.plot(fval,gval,'b')
    plt.gca().xaxis.set_major_locator(mticker.MultipleLocator(1))
    plt.grid(True, which="both", ls="-")
    plt.xlabel(r"$f(t)$")
    plt.ylabel(r"$g(t)$")
    plt.title(f"Using Natural Cubic Spline Method with dt = {dt}")
    plt.savefig(f'fgCubic{dt}.png')
    plt.show()
```


Using Natural Cubic Spline Method with dt = 0.1
3.0
2.5
2.0
1.5
1.0
0.5
0.0
$g(t)$
1
2
3
$f(t)$


Using Natural Cubic Spline Method with dt = 0.5
3.0
2.5
2.0
1.5
1.0
0.5
0.0
$g(t)$
1
2
3
$f(t)$

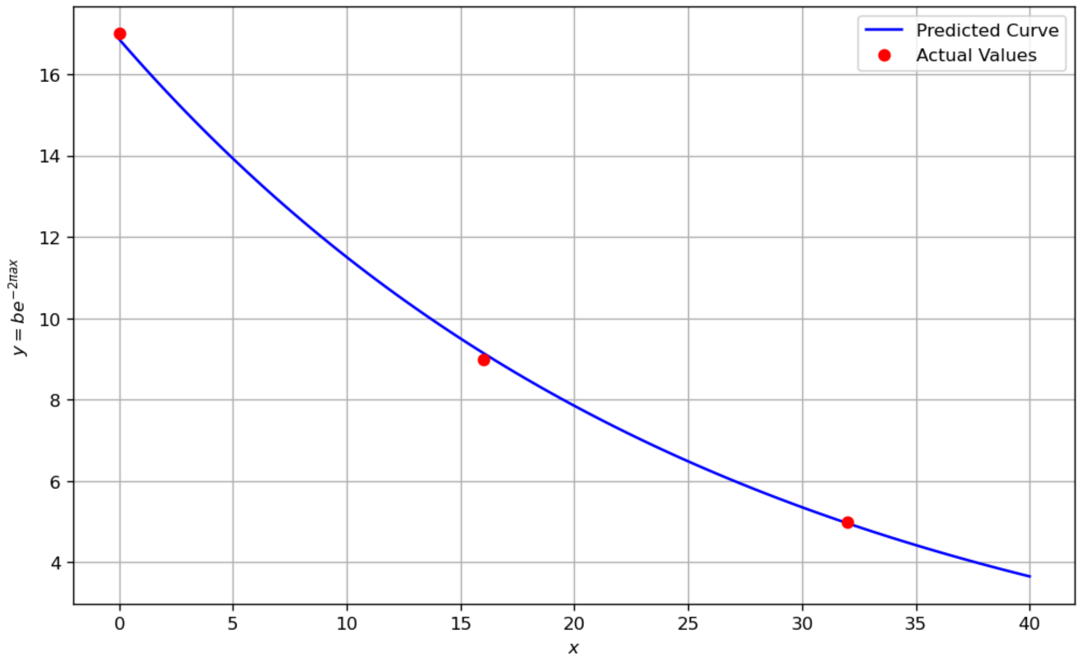


1.4 Question 4

[11]:
```
xval = [0,16,32]
yval = [17,9,5]

Mat = np.zeros((2,2))
Mat[0,0] = np.sum([2*np.pi*xi**2 for xi in xval])
Mat[0,1] = -np.sum([xi for xi in xval])
Mat[1,0] = np.sum([2*np.pi*xi for xi in xval])
Mat[1,1] = -len(xval)
bmat = []
bmat.append(-np.sum([xval[i]*np.log(yval[i]) for i in range(len(xval))]))
bmat.append(-np.sum([np.log(yi) for yi in yval]))

coeff = np.linalg.solve(Mat, bmat)
aval = coeff[0]
bval = np.exp(coeff[1])
```

```
print("Value of a: ", aval)
print("Value of b: ", bval)

func = bval*sym.exp(-2*sym.pi*aval*x)

x_vals = np.linspace(0, 40, 100)
y_vals = [func.subs(x, val).evalf() for val in x_vals]

plt.figure(figsize=(10, 6), dpi=120)
plt.plot(x_vals, y_vals, 'b', label = f"Predicted Curve")
plt.plot(xval, yval, 'ro', label = f"Actual Values")
plt.legend()
plt.grid(True, which="both", ls="-")
plt.xlabel(r"$x$")
plt.ylabel(r"$y = be^{-2\pi ax}$")
plt.savefig('LinearLeastSquare.png')
plt.show()
```

```
Value of a:  0.006086559661783669
Value of b:  16.86397450267915
```


Predicted Curve
Actual Values
$y = be^{-2\pi ax}$
x