Lecture 4

Numerical Interpolation / TDMA

Jung-Il Choi

School of Mathematics and Computing (Computational Science and Engineering)

YONSEI UNIVERSITY

0. Interpolation

- **Problem statement**
  - For given data

$$(x_i, y_i)\ with\ i = 1, \dots, n$$

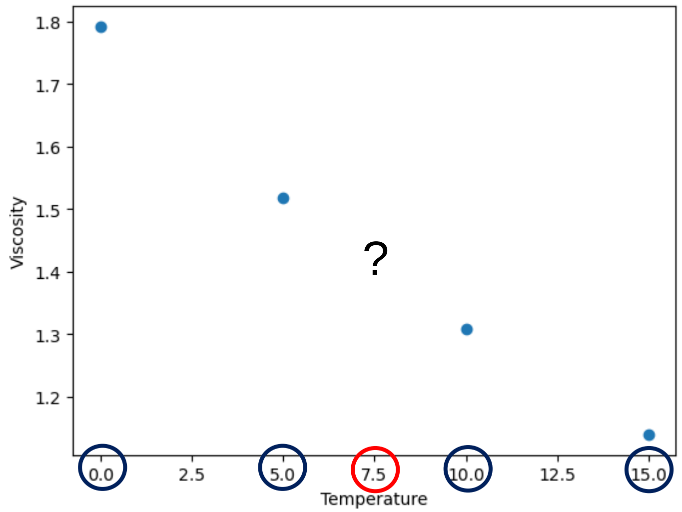
  determine function $f: \mathbb{R} \to \mathbb{R}\ such\ that$

$$f(x_i) = y_i\ with\ i = 1, \dots, n$$

  - Given a new x^*, we can interpolate its function value $\hat{y}(x^*)$. $\hat{y}(x)$ is **interpolating function.**

  - Example

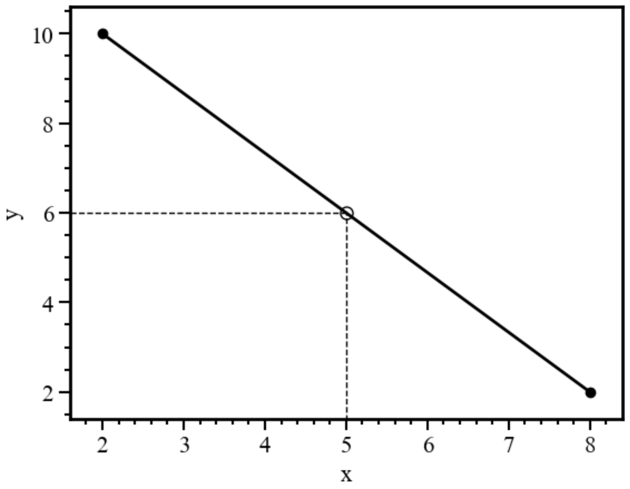
Temperature	0°	5°	10°	15°
Viscosity	1.792	1.519	1.308	1.140



1. Polynomial Interpolation

- **Linear interpolation**
  - The estimated point is assumed to lie on the line joining the nearest points to the left and right.
  - Linear interpolation at x is

$$p(x) = \left(\frac{x - x_1}{x_0 - x_1}\right) y_0 + \left(\frac{x - x_0}{x_1 - x_0}\right) y_1$$
$$= y_0 + \left(\frac{y_1 - y_0}{x_1 - x_0}\right)(x - x_0)$$



x	2	8
y	10	2

$$\rightarrow \quad \hat{y}(5) = 10 + \left(\frac{2 - 10}{8 - 2}\right)(5 - 2) = 6$$

1. Polynomial Interpolation

- **Lagrange interpolation**
  - Lagrange polynomial interpolation finds a single polynomial, $P(x)$.
  - As an interpolation function, it should have the property $P(x_i) = f(x_i)$ for every point in the dataset.
  - It is useful to write them as a linear combination of Lagrange basis polynomials, $l_i(x)$

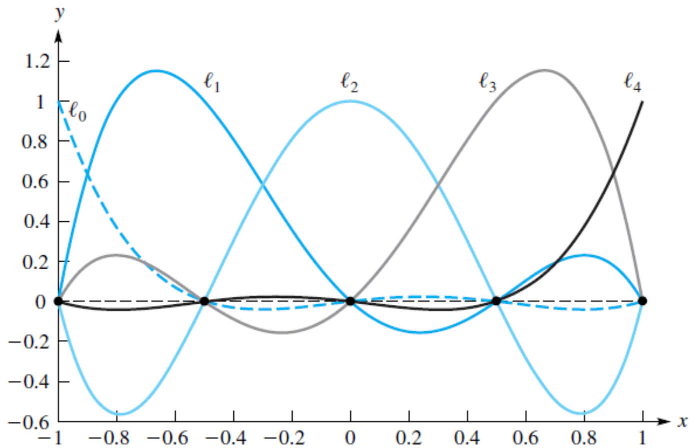
$$\boxed{\ell_i(x) = \prod_{\substack{j \neq i \\ j=0}}^{n} \left(\frac{x - x_j}{x_i - x_j} \right) \qquad (0 \leqq i \leqq n)}$$

$$\ell_i(x) = \left(\frac{x - x_0}{x_i - x_0} \right) \left(\frac{x - x_1}{x_i - x_1} \right) \cdots \left(\frac{x - x_{i-1}}{x_i - x_{i-1}} \right) \left(\frac{x - x_{i+1}}{x_i - x_{i+1}} \right) \cdots \left(\frac{x - x_n}{x_i - x_n} \right)$$

  - And

$$\boxed{p_n(x) = \sum_{i=0}^{n} \ell_i(x) f(x_i)}$$

$$p_n(x_j) = \sum_{i=0}^{n} \ell_i(x_j) f(x_i) = \ell_j(x_j) f(x_j) = f(x_j)$$



1. Polynomial Interpolation

- **Lagrange interpolation**

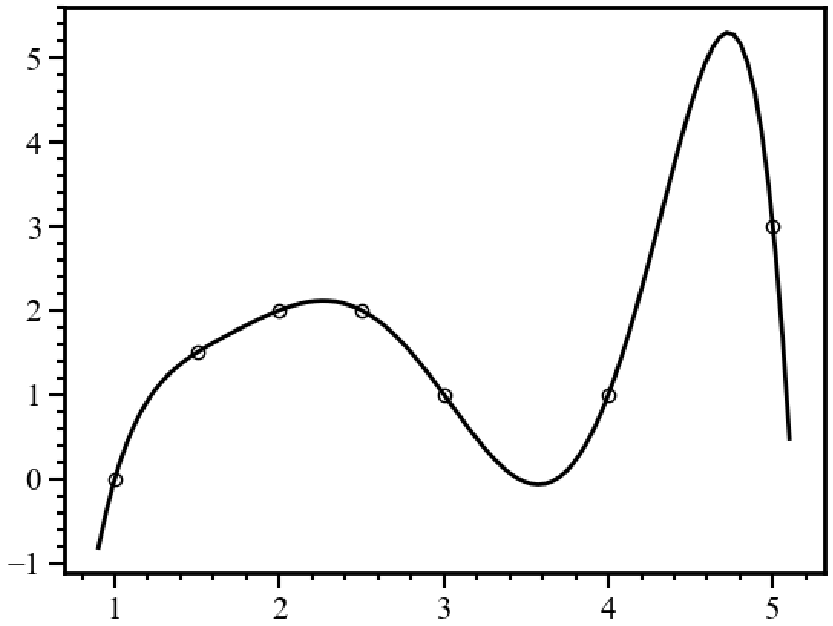
```
In [1]: import numpy as np
        import matplotlib.pyplot as plt
```

```
In [2]: def Lagrange(x,y,xval):
            yval = 0

            deg = len(x) - 1

            for i in range(deg+1):
                LagBase = 1.
                for k in range(deg+1):
                    if(k != i):
                        LagBase *= (xval-x[k])/(x[i]-x[k])
                yval += y[i]*LagBase
            return yval
```

```
In [3]: x = np.array([1, 1.5, 2, 2.5, 3, 4, 5])
        y = np.array([0, 1.5, 2,   2, 1, 1, 3])
```

```
In [4]: xa = np.linspace(0.9,5.1,100)
        ya = Lagrange(x,y,xa)
```

Interpolation result(line-by-line code)

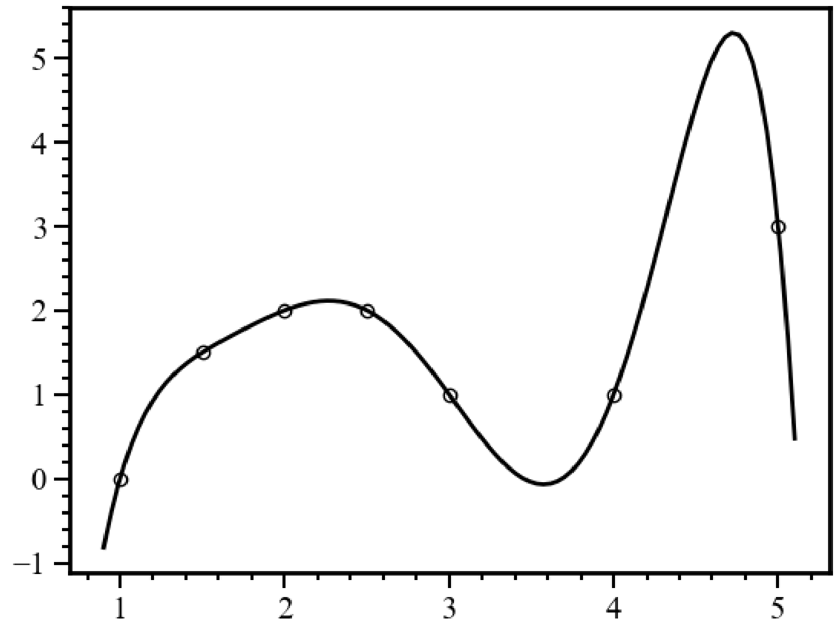


1. Polynomial Interpolation

- **Lagrange interpolation**

```
In [7]: from scipy import interpolate

In [8]: LagPoly = interpolate.lagrange(x,y)
        print(LagPoly)

            6         5         4         3         2
-0.2429 x + 3.933 x - 25.12 x + 81.25 x - 141.8 x + 128.8 x - 46.86

In [9]: xa = np.linspace(0.9,5.1,100)
        ya = LagPoly(xa)
```

Interpolation result(Scipy)



1. Polynomial Interpolation

- **Bivariate functions**

The methods we have discussed for interpolating functions of one variable by polynomials extend to *some* cases of functions of two or more variables. An important case occurs when a function $(x, y) \mapsto f(x, y)$ is to be approximated on a rectangle. This leads to what is known as **tensor-product interpolation**. Suppose the rectangle is the Cartesian product of two intervals: $[a, b] \times [\alpha, \beta]$. That is, the variables x and y run over the intervals $[a, b]$, and $[\alpha, \beta]$, respectively. Select n nodes x_i in $[a, b]$, and define the *Lagrangian polynomials*

$$\ell_i(x) = \prod_{\substack{j \neq i \\ j=1}}^{n} \frac{x - x_j}{x_i - x_j} \qquad (1 \leqq i \leqq n)$$

Similarly, we select m nodes y_i in $[\alpha, \beta]$ and define

$$\overline{\ell}_i(y) = \prod_{\substack{j \neq i \\ j=1}}^{m} \frac{y - y_j}{y_i - y_j} \qquad (1 \leqq i \leqq m)$$

Then the function

$$P(x, y) = \sum_{i=1}^{n} \sum_{j=1}^{m} f(x_i, y_j) \ell_i(x) \overline{\ell}_j(y)$$

2. TDMA

Tri-dianonal matrix algorithm

- **Linear algebra**

An $n \times n$ system of linear equations can be written in matrix form

$$\boldsymbol{A}\boldsymbol{x} = \boldsymbol{b}$$

where the coefficient matrix $\boldsymbol{A}$ has the form

$$\boldsymbol{A} = \begin{bmatrix} a_{11} & a_{12} & a_{13} & \cdots & a_{1n} \\ a_{21} & a_{22} & a_{23} & \cdots & a_{2n} \\ a_{31} & a_{32} & a_{33} & \cdots & a_{3n} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ a_{n1} & a_{n2} & a_{n3} & \cdots & a_{nn} \end{bmatrix}$$

2. TDMA

Diagonal 만 저장해서 저장공간 낭비를 줄임

연관이 있는 앞 뒤에 대해서만 고려하겠다!

$$\begin{pmatrix} d_0 & a_0 & 0 & \dots & \dots & \dots & 0 \\ b_1 & d_1 & a_1 & 0 & \dots & \dots & 0 \\ 0 & b_2 & d_2 & a_2 & 0 & \dots & 0 \\ 0 & 0 & * & * & * & 0 & 0 \\ 0 & 0 & 0 & * & * & * & 0 \\ 0 & \dots & \dots & 0 & b_{N-1} & d_{N-1} & a_{N-1} \\ 0 & \dots & \dots & \dots & 0 & b_N & d_N \end{pmatrix} \begin{pmatrix} u_0 \\ u_1 \\ u_2 \\ \vdots \\ \vdots \\ u_{N-1} \\ u_N \end{pmatrix} = \begin{pmatrix} c_0 \\ c_1 \\ c_2 \\ \vdots \\ \vdots \\ c_{N-1} \\ c_N \end{pmatrix}$$

Gauss elimination (forward sweep)

$$\begin{aligned} d'_i &= d_i - \frac{b_i}{d'_{i-1}} a_{i-1} \\ b'_i &= b_i - \frac{b_i}{d'_{i-1}} d'_{i-1} = 0 \\ c'_i &= c_i - \frac{b_i}{d'_{i-1}} c'_{i-1} \\ i &= 1, 2, \dots, N \end{aligned}$$

$$\begin{pmatrix} d_0 & a_0 & 0 & \dots & \dots & \dots & 0 \\ 0 & d'_1 & a_1 & 0 & \dots & \dots & 0 \\ 0 & 0 & d'_2 & a_2 & 0 & \dots & 0 \\ 0 & 0 & 0 & * & * & 0 & 0 \\ 0 & 0 & 0 & 0 & * & * & 0 \\ 0 & \dots & \dots & 0 & 0 & d'_{N-1} & a_{N-1} \\ 0 & \dots & \dots & \dots & 0 & 0 & d'_N \end{pmatrix} \begin{pmatrix} u_0 \\ u_1 \\ u_2 \\ \vdots \\ \vdots \\ u_{N-1} \\ u_N \end{pmatrix} = \begin{pmatrix} c_0 \\ c'_1 \\ c'_2 \\ \vdots \\ \vdots \\ c'_{N-1} \\ c'_N \end{pmatrix}$$

2. TDMA

Back substitution (forward sweep)

Find unknowns starting with u_N

$$u_N = c'_N / d'_N.$$

then going backward $i = N-1, N-2, \ldots, 0$

$$u_i = (c'_i - a_i u_{i+1}) / d'_i$$

Forward + backward sweeps require $\sim N$ arithmetic operations

Standard Gauss elimination, which does not take into account special structure of matrix, requires $\sim N^3$ arithmetic operations

2. TDMA

- **Example**

$$\begin{bmatrix} 2 & 3 & 0 & 0 \\ 6 & 3 & 9 & 0 \\ 0 & 2 & 5 & 2 \\ 0 & 0 & 4 & 3 \end{bmatrix} \begin{bmatrix} x_0 \\ x_1 \\ x_2 \\ x_3 \end{bmatrix} = \begin{bmatrix} 21 \\ 69 \\ 34 \\ 22 \end{bmatrix}$$

$$\downarrow$$

$$\begin{bmatrix} 2 & 3 & 0 & 0 \\ 6 & 3 & 9 & 0 \\ 0 & 2 & 5 & 2 \\ 0 & 0 & 4 & 3 \end{bmatrix} \begin{bmatrix} 3 \\ 5 \\ 4 \\ 2 \end{bmatrix} = \begin{bmatrix} 21 \\ 69 \\ 34 \\ 22 \end{bmatrix}$$

In [11]:
```python
import numpy as np

def TDMA(a,b,c,d):
    n = len(c)
    dp = np.zeros(n)
    cp = np.zeros(n)
    x = np.zeros(n)

    # forward sweep
    dp[0] = d[0]
    cp[0] = c[0]

    for i in range(1,n):
        dp[i] = d[i] - b[i]*a[i-1]/dp[i-1]
        cp[i] = c[i] - b[i]*cp[i-1]/dp[i-1]

    # backward substitution
    x[n-1] = cp[n-1]/dp[n-1]

    for i in range(n-2,-1,-1):
        x[i] = (cp[i] - a[i]*x[i+1])/dp[i]

    return x
```

In [24]:
```python
upp = np.array([3,9,2,0])
dig = np.array([2,3,5,3])
low = np.array([0,6,2,4])

rhs = np.array([21,69,34,22])

X = TDMA(upp,low,rhs,dig)

print(X)
```

```
[3. 5. 4. 2.]
```

3. Piecewise Interpolation

- **Cubic Spline**

Let $g_i(x)$ be the cubic in the interval $x_i \le x \le x_{i+1}$ and let $g(x)$ denote the collection of all the cubics for the entire range of x. Since g is piecewise cubic its second derivative, g'', is piecewise linear. For the interval $x_i \le x \le x_{i+1}$, we can write the equation for the corresponding straight line as

$$\boxed{g_i''(x) = g''(x_i)\frac{x - x_{i+1}}{x_i - x_{i+1}} + g''(x_{i+1})\frac{x - x_i}{x_{i+1} - x_i}.} \tag{1.3}$$

Integrating (1.3) twice we obtain

$$g_i(x) = \frac{g''(x_i)}{x_i - x_{i+1}}\frac{(x - x_{i+1})^3}{6} + \frac{g''(x_{i+1})}{x_{i+1} - x_i}\frac{(x - x_i)^3}{6} + C_1 x + C_2.$$

The undetermined constants C_1 and C_2

$$g_i(x_i) = f(x_i) \equiv y_i \qquad g_i(x_{i+1}) = f(x_{i+1}) \equiv y_{i+1}$$

3. Piecewise Interpolation

- **Cubic Spline**

$$g_i(x) = \frac{g''(x_i)}{6}\left[\frac{(x_{i+1}-x)^3}{\Delta_i} - \Delta_i(x_{i+1}-x)\right] + \frac{g''(x_{i+1})}{6}\left[\frac{(x-x_i)^3}{\Delta_i} - \Delta_i(x-x_i)\right] + f(x_i)\frac{x_{i+1}-x}{\Delta_i} + f(x_{i+1})\frac{x-x_i}{\Delta_i}, \qquad \Delta_i = x_{i+1} - x_i$$

$g''(x_i)$ and $g''(x_{i+1})$ unknowns.

the continuity of the first derivatives: $g'_i(x_i) = g'_{i-1}(x_i)$

$$\frac{\Delta_{i-1}}{6}g''(x_{i-1}) + \frac{\Delta_{i-1}+\Delta_i}{3}g''(x_i) + \frac{\Delta_i}{6}g''(x_{i+1}) = \frac{f(x_{i+1})-f(x_i)}{\Delta_i} - \frac{f(x_i)-f(x_{i-1})}{\Delta_{i-1}} \qquad i = 1, 2, 3, \ldots, N-1.$$

$N-1$ equations for the $N+1$ unknowns

3. Piecewise Interpolation

- **Cubic Spline**

$N-1$ equations for the $N+1$ unknowns $\rightarrow$ *required 2 more constraints*

$$\begin{bmatrix} \frac{\Delta_0+\Delta_1}{3} & \frac{\Delta_1}{6} & 0 & \cdots & 0 & 0 & 0 \\ \frac{\Delta_1}{6} & \frac{\Delta_1+\Delta_2}{3} & \frac{\Delta_2}{6} & \cdots & 0 & 0 & 0 \\ \vdots & & & \ddots & & & \vdots \\ 0 & 0 & 0 & \cdots & \frac{\Delta_{n-3}}{6} & \frac{\Delta_{n-3}+\Delta_{n-2}}{3} & \frac{\Delta_{n-2}}{6} \\ 0 & 0 & 0 & \cdots & 0 & \frac{\Delta_{n-2}}{6} & \frac{\Delta_{n-2}+\Delta_{n-1}}{3} \end{bmatrix} \begin{bmatrix} g''(x_1) \\ g''(x_2) \\ \vdots \\ g''(x_{n-2}) \\ g''(x_{n-1}) \end{bmatrix} = \begin{bmatrix} \frac{f(x_2)-f(x_1)}{\Delta_1} - \frac{f(x_1)-f(x_0)}{\Delta_0} - \frac{\Delta_0}{6} g''(x_0) \\ \frac{f(x_2)-f(x_1)}{\Delta_1} - \frac{f(x_1)-f(x_0)}{\Delta_0} \\ \vdots \\ \frac{f(x_{n-1})-f(x_{n-2})}{\Delta_{n-2}} - \frac{f(x_{n-2})-f(x_{n-3})}{\Delta_{n-3}} \\ \frac{f(x_n)-f(x_{n-1})}{\Delta_{n-1}} - \frac{f(x_{n-1})-f(x_{n-2})}{\Delta_{n-2}} - \frac{\Delta_{n-1}}{6} g''(x_n) \end{bmatrix}$$

3. Piecewise Interpolation

- **Cubic Spline**
  - Free run-out (natural spline):

$$g''(x_0) = g''(x_N) = 0.$$ 미분이 변화가 없다

  - Parabolic run-out:

$$g''(x_0) = g''(x_1)$$
$$g''(x_{N-1}) = g''(x_N).$$

$$g''(x_0) = \alpha g''(x_1)$$
$$g''(x_{N-1}) = \beta g''(x_N),$$

α and β are constants chosen by the user.

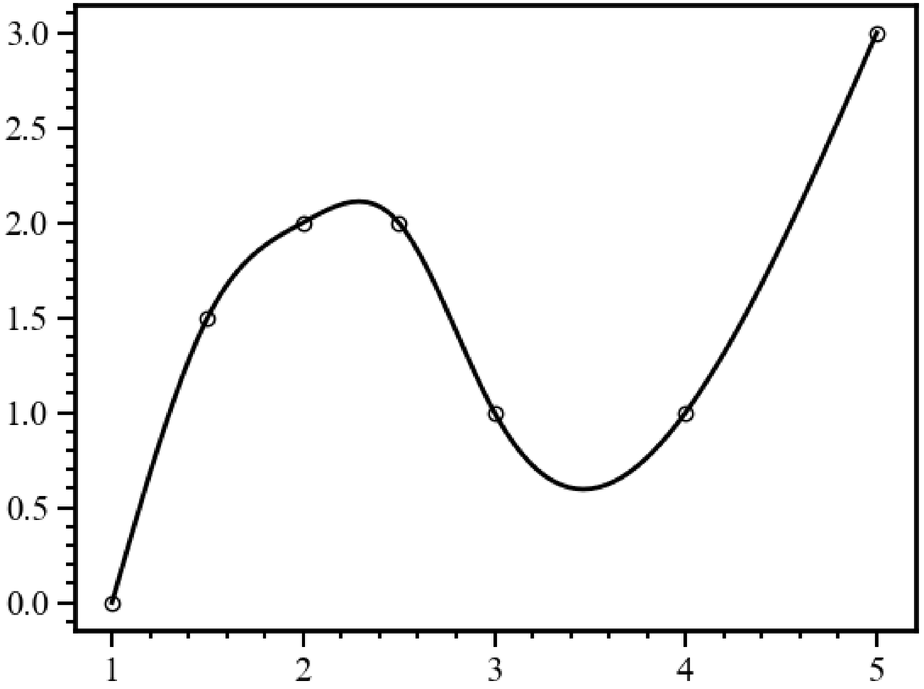
  - Periodic:

$$g''(x_0) = g''(x_{N-1})$$
$$g''(x_1) = g''(x_N).$$

3. Piecewise Interpolation

- **Cubic Spline**

Interpolation result(line-by-line code)



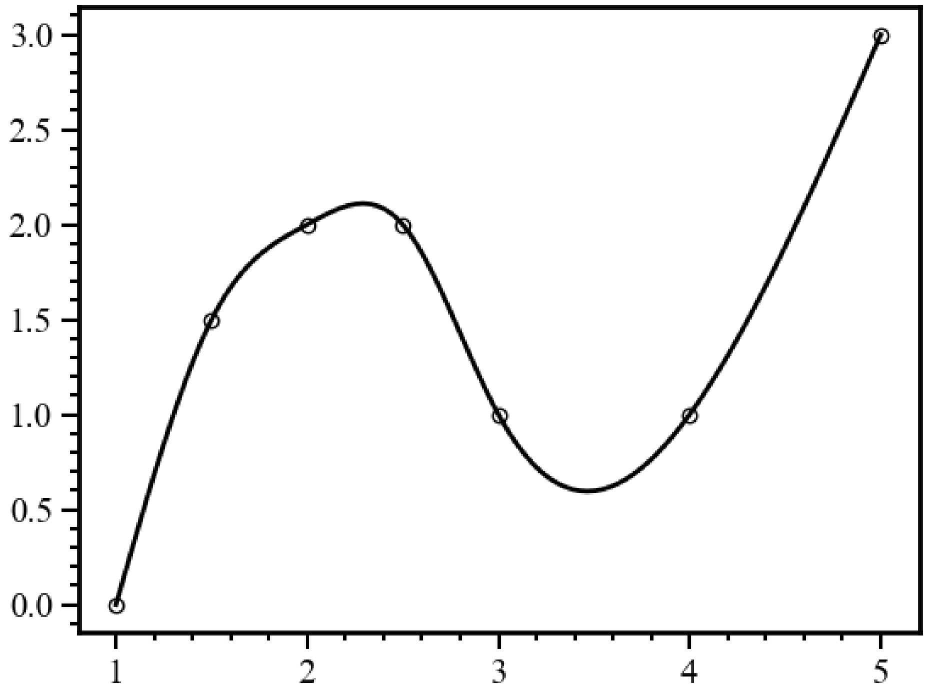

```
In [13]: import numpy as np

def cubic_spline(x,y,xval):
    # number of intervals
    N   = len(x) - 1
    # number of points
    Np  = N + 1
    # number of intervals minus 1
    Nm  = N - 1

    # initialize arrays
    h   = np.zeros(N) # interval widths
    gdp = np.zeros(Np) # second derivatives of spline
    upp = np.zeros(Nm) # upper diagonal of matrix
    low = np.zeros(Nm) # lower diagonal of matrix
    dig = np.zeros(Nm) # diagonal of matrix
    rhs = np.zeros(Nm) # right-hand side of matrix equation

    gdp[0], gdp[N] = 0.0, 0.0 # free-run (boundary conditions)

    # calculate interval widths
    h[:] = x[1:] - x[:-1]

    # set up matrix equation to solve for second derivatives of spline
    upp[  :-1] =  h[ 1:-1]/6
    dig[  :  ] = (h[  :-1]+h[ 1:  ])/3
    low[ 1:  ] =  h[ 1:-1]/6

    rhs[  :  ] = (y[ 2:  ]-y[ 1:-1])/h[ 1:  ] - (y[ 1:-1]-y[  :-2])/h[  :-1]
    rhs[ 0] -= h[ 0]*gdp[ 0]/6
    rhs[-1] -= h[-1]*gdp[-1]/6

    # solve matrix equation to obtain second derivatives of spline
    gdp[1:-1] = TDMA(upp,low,rhs,dig)

    # evaluate spline at specified x values
    Ncs = len(xval)
    yval= np.zeros(Ncs)

    for i in range(N):
        for j in range(Ncs):
            if x[i+1]>=xval[j] and x[i]<xval[j]:
                yval[j] = gdp[i  ]/6 * ( (x[i+1]-xval[j])**3/h[i] - h[i]*(x[i+1]-xval[j]) ) \
                        + gdp[i+1]/6 * ( (xval[j] - x[i])**3/h[i] - h[i]*(xval[j] - x[i]) ) \
                        + y[i]*(x[i+1]-xval[j])/h[i] + y[i+1]*(xval[j]-x[i])/h[i]

    # return spline values at specified x values
    return yval
```

```
In [14]: xa = np.linspace(1,5,100)
ya = cubic_spline(x,y,xa)
```

3. Piecewise Interpolation

- **Cubic Spline**

Interpolation result(line-by-line code)



In [28]:
```
from scipy import interpolate as ip

cs = ip.CubicSpline(x,y,bc_type='natural')

xa = np.linspace(1,5,100)
ya = cs(xa)
```

Q&A Thanks for listening