WEEK 1

2020136006 정성원

모든 코딩은 파이썬을 사용하였으며, numpy 라이브러리와 plt 라이브러리를 사용하였음을 밝힙니다.

1-1. RANDOM 함수를 써서 만든 임의의 숫자 100개를 오름차순으로 정렬하세요.

```
list1 = np.random.randint(1,101, 100)
list1_sorted = np.sort(list1)
print(list1_sorted)
```

Numpy 라이브러리에 있는 randint 변수를 이용하여 1부터 100까지 100의 숫자를 생성하였고, np.sort를 사용하여 오름차순으로 정렬하였다.

1-2. 임의의 두 4 x 4 matrix A, B를 생성하고, 두 matrix를 곱해 matrix C를 생성하세요. 그 이후에 기존의 공식을 사용하여 matrix C의 inverse matrix를 계산하세요. 그리고, C와 inverse C를 곱해 identity matrix가 나옴을 확인하세요.

```
matA = np.random.randint(-5,5, (4,4))
matB = np.random.randint(-5,5, (4,4))
print("Matrix A",matA, "Matrix B",matB, sep='\n')

# 행렬 곱
matC = np.dot(matA, matB)
print('Matrix C',matC,sep='\n')
```

Numpy 라이브러리에 있는 randint 함수를 이용하여 행렬 A,B를 생성하였고 dot 함수를 사용하여 행렬 A와 B의 곱인 행렬 C를 구하였다.

```
# 역행렬
matM = np.zeros((4, 4))
for i in range(4):
    for j in range(4):
        M_int = np.delete(np.delete(matC,i, axis=0), j, axis=1)
        matM[i,j] = M_int[0,0]*(M_int[1,1]*M_int[2,2]
                    - M_int[1,2]*M_int[2,1])
        - M_int[0,1]*(M_int[1,0]*M_int[2,2]
                    - M_int[1,2]*M_int[2,0])
        + M_int[0,2]*(M_int[1,0]*M_int[2,1]
                    - M_int[1,1]*M_int[2,0])

det = matC[0,0] * matM[0,0] - matC[0,1] * matM[0,1] + matC[0,2] * matM[0,2] - matC[0,3] * matM[0,3]
print('determinant:', det)
C = np.zeros((4, 4))
for i in range(4):
    for j in range(4):
        C[i,j] = matM[i,j] * (-1)**(i+j)

matC_inv = 1/det * C.T
print('Inverse Matrix C',matC_inv,sep='\n')
```

4*4행렬의 역행렬을 구하기 위해 여인수를 구하는 과정을 for 문을 이용하여 구하였고 이를 이용하여 determinant를 구하였다. 여인수를 구할 때는 i 행과 j열을 모두 제거하여 [i,j]의 소행렬식을 M행렬에 저장하였다. 이를 이용하여 determinant를 계산하여 하였고 소행렬식에 (-1)^(i+j)를 곱하여 adjugate matrix를 만들었다. 따라서, determinant와 adjugate matrix를 이용하여 역행렬을 만들었다.

```
# 단위행렬
I = np.dot(matC, matC_inv)
print('Identity Matrix',I,sep='\n')
```

```
Identity Matrix
[[ 1.00000000e+00  4.44089210e-16 -2.22044605e-16  1.11022302e-16]
 [ 0.00000000e+00  1.00000000e+00  1.38777878e-17 -1.38777878e-17]
 [ 0.00000000e+00 -5.55111512e-17  1.00000000e+00  5.55111512e-17]
 [ 2.22044605e-16  4.44089210e-16 -2.22044605e-16  1.00000000e+00]]
```

단위행렬인 것을 확인하기 위해 C행렬과 C행렬의 역행렬을 곱하였다. 이 계산 결과가 정확한 단위 행렬로 나오지는 않는다. 이는 컴퓨터 계산의 결과로 매우 작은 값들 (약 10**-16)정도가 남았기 때문인데 실제로 계산해보면 정확한 단위 행렬이 나올 것으로 예상된다.

주어진 함수 $f(x) = x^5 - 9x^4 - x^3 + 17x^2 - 8x - 8$ 에 대하여 다음 문제를 풀어 제출하세요.
(Error bound = 10^{-8})

1. 이분법(bisection method)을 사용하여 주어진 3구간 [-10,-1], [-1,0] 그리고 [0,10] 에서의 $f(x)$의 해를 구하시오.

```
def func(x):
    return x**5-9*x**4-x**3+17*x**2-8*x-8
```

주어진 함수를 def 함수를 이용하여 정의하였다.

```
# [-10, -1]
a, b = -10,-1
count1 = 0

while abs(b-a) >= 10**-8:
    count1 += 1
    m = (a + b) / 2
    if func(m) * func(a) <=0:
        b = m
    else:
        a = m

print('[-10, -1]에서의 해:', m, '\n 반복 횟수:', count1)
```

구간 [-10,-1]에서 While 함수를 이용하여 b와 a의 차이가 10**-8보다 크거나 같을 때까지 시행을 반복하였고 count1을 이용하여 while문이 얼마나 반복되었는지 확인하였다. 이분법 방법인 양 끝의 함수 값과 중간 값의 함수 값의 곱이 음수라면 그 사이에 근이 존재하므로 이를 이용하여 위와 같이 while문을 작성하였다.

```
[-10, -1]에서의 해: -1.3875071099027991
 반복 횟수: 30
```

결과는 위와 같이 위와 같이 나왔다

```
[-10, -1]에서의 해: -1.3875071099027991
 반복 횟수: 30
[-1, 0]에서의 해: -0.5104293450713158
 반복 횟수: 27
[0, 10]에서의 해: 8.910696404054761
 반복 횟수: 30
총 반복 횟수: 87
```

같은 방식으로 구간 [-1,0], [0,10]에서도 진행하였고 총 반복횟수는 구간 [-10,-1], [-1,0], [0,10]에서 계산된 횟수를 다 더한 값으로 구하였다. 결과를 보면 총 반복 횟수는 87회인 것을 볼 수 있다.

2. 뉴턴법(Newton's method)를 사용하여 주어진 3점 $\mathbf{x_0 = -10, -0.1, 10}$ 에서 $f(x)$의 해를 구하시오.

뉴턴법은 특정 점에서 접선을 구하고 접선의 x절편을 구하고 해당 x지점의 함수 값에서 접선을 구하고 이를 반복하여 근을 구하는 방법이다.

```
def func_diff(x):
    return 5*x**4 - 36*x**3 - 3*x**2 + 34*x - 8
```

이를 진행하기 위해서는 원래 함수의 도함수를 구하여야 하는데 주어진 함수의 도함수는 $f' = 5x^4 - 36x^3 - 3x^2 + 34x^1 - 8$이다. 해당 함수를 def 함수를 func_diff로 정의하였다.

```
# -10
x0 = -10
x1 = np.Infinity
count1 = 0

while abs(x0-x1) >= 10**-8:
    if count1 == 0 :
        x0 = -10
        x1 = np.Infinity

    else:
        x0 = x1

    x1 = x0 - func(x0)/func_diff(x0)
    count1 += 1

print('x0 = -10에서의 해:', x1, '\n 반복 횟수:', count1)
```

뉴턴법을 이용하여 x값을 구하는 공식은 $x_{n+1} = x_n - \frac{f(x_n)}{f'(x_n)}$ 이다. while문을 이용하여 $x_{n+1} - x_n$의 값이 10**-8보다 크거나 같을 때 시행하도록 하였다.

```
x0 = -10에서의 해: -1.3875071055826167
 반복 횟수: 13
x0 = -0.1에서의 해: -0.5104293428178256
 반복 횟수: 5
x0 = 10에서의 해: 8.910696402960298
 반복 횟수: 6
총 반복 횟수: 24
```

결과는 위와 같이 나왔는데 이분법과 값이 별로 차이가 나지 않는 것을 확인할 수 있다. 하지만 반복횟수는 약 1/4정도로 줄어든 것을 확인할 수 있다.

3. 뉴턴법을 사용하였을 때, 초기 조건을 $x_0 = 0$ 에 대해 계산할 경우, 계산이 어떠한 양상을 보이는지 서술하시오.

$x_0 = 0$인 경우 수렴하지 않고 1과 0, -1이 반복되는 것으로 확인된다. 어떤 한 값에 수렴하지 못하고 진동하는 것을 확인할 수 있다. 이는 초기 값을 0으로 가깝게 하면 안된다는 것을 확인할 수 있다.

4. 시컨트법(secant method)를 사용하여 주어진 초기조건에서 $f(x)$의 해를 구하시오.

(1) $x_1 = -10, x_2 = -9.9$ / (2) $x_1 = -0.1, x_2 = -0.2$ / (3) $x_1 = 10, x_2 = 9.9$

```
# x1 = -10, x2 = -9.9
x1 = -10
x2 = - 9.9
count1 = 0

while abs(x2-x1) >= 10**-8:

    x3 = x2 - func(x2)*((x2 - x1) / (func(x2) - func(x1)))
    x1 = x2
    x2 = x3
    count1 += 1

print('x1 = -10, x2 = -9.9에서의 해:', x2, '\n 반복 횟수:', count1)
```

시컨트법은 뉴턴법과는 다르게 한 지점의 접선을 구하는 것이 아니라 함수 위의 두 점을 지나는 직선을 구하여 그 직선의 x 절편을 구하고 그 때의 함수 값과 전에 사용한 두 점 중 한 점을 이용하여 다시 반복하여 시행하는 방법이다. 해당방법의 공식은 $x_{n+2} = x_{n+1} - f(x_{n+1})\frac{x_{n+1}-x_n}{f(x_{n+1})-f(x_n)}$ 이고, While 문을 이용하여 두 점의 간격이 10**-8보다 클 때만 시행하도록 하였다. 이를 이용하여 (1),(2),(3)조건을 이용하여 다음과 같이 구했다.

```
x1 = -10, x2 = -9.9에서의 해: -1.3875071055826 35
 반복 횟수: 18
x1 = -0.1, x2 = -0.2에서의 해: -0.5104293428178256
 반복 횟수: 8
x1 = 10, x2 = 9.9에서의 해: 8.910696402960298
 반복 횟수: 6
총 반복 횟수: 32
```

결과는 위 두 방법과 비슷하게 나왔다. 하지만 총 반복횟수는 뉴턴법보다는 많고 이분법보다는 큰 것을 확인할 수 있다.

5. 이분법, 뉴턴법, 시컨트법에 대한 계산을 진행하고, 몇 번의 반복계산 후에 해를 구하였는지 서술하시오.

```
    print('이분법 시행 횟수:', count_bi)
    print('뉴턴법 시행 횟수:', count_new)
    print('시컨트법 시행 횟수:', count_sec)
```

```
이분법 시행 횟수: 87
뉴턴법 시행 횟수: 24
시컨트법 시행 횟수: 32
```

이분법은 87회, 뉴턴법은 24회, 시컨트법은 32회로 나타났다. 뉴턴법과 시컨트법의 계산 회수는 약 8회 정도 차이가 났지만 이분법은 약 63회 정도 차이가 났다.

주어진 함수 $f(x) = \frac{1}{1+16x^2}, [-1, 1]$ 에 대하여 다음 문제를 풀어 제출하시오.

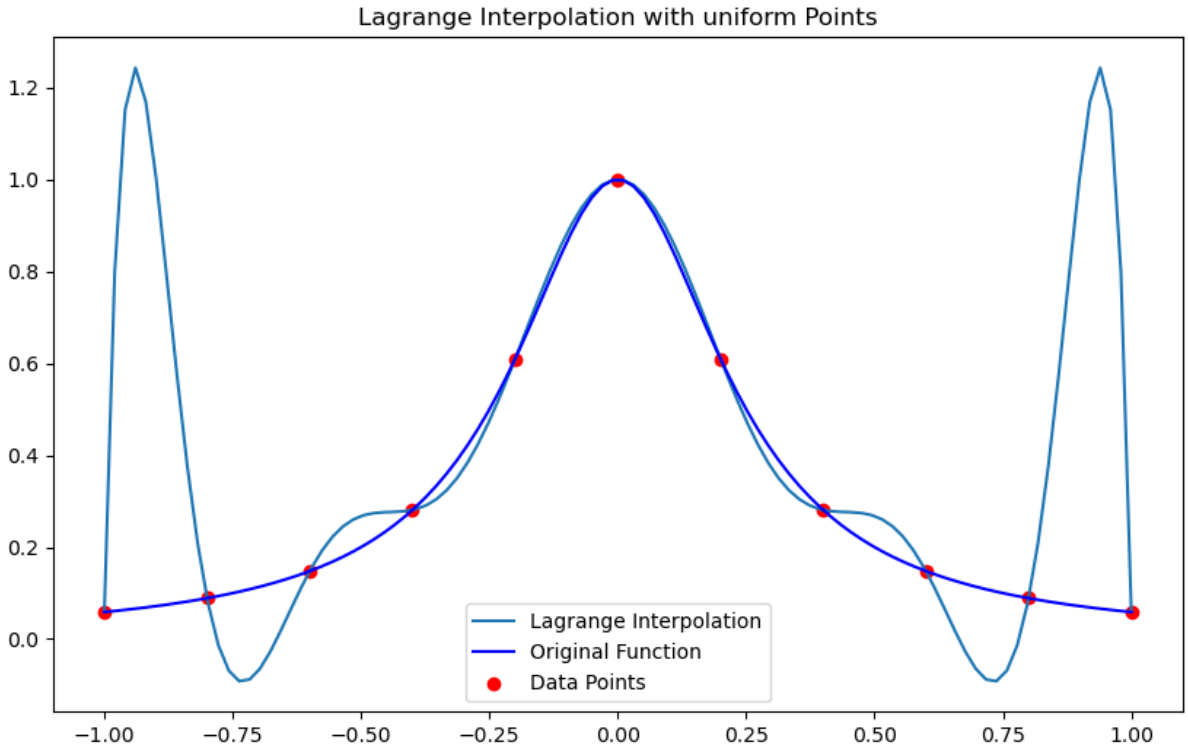
1. h=0.2로 동일하게 간격이 나누어진 uniform nodes를 사용하여, 본 함수의 Lagrange interpolating polynomial p(x) 를 찾고 그리시오.

```
def func(x):
    return 1/(1+16*x**2)
```

함수를 def 함수를 이용하여 func에 저장하였다.

```
a = 100
list = np.linspace(-1, 1, a)
list_x = np.linspace(-1, 1, 11)
list_y = func(list_x)


for k in range(len(list_x)):
    if k == 0:
        list_L = np.zeros(len(list))
    L_int = np.ones(len(list))
    for n in range(len(list_x)):
        if n != k:
            L_int = L_int*(list-list_x[n])/(list_x[k]-list_x[n])
    list_L = list_L + L_int*list_y[k]

plt.figure(figsize=(10, 6))
plt.plot(list, list_L, label='Lagrange Interpolation')
plt.plot(list, func(list), label='Original Function', color='blue')
plt.scatter(list_x, list_y, color='red', label='Data Points')
plt.title('Lagrange Interpolation with uniform Points')
plt.legend()
```

Lagrange interpolating polynomial은 $l_j(x) = \prod_{\substack{0 \le m \le k \\ m=j}} \frac{x-x_m}{x_j-x_m}$를 구하고 $P(x) = \sum_{j=0}^{k} y_j\, l_j(x)$를 구할 수 있다. $\prod$를 수행하기 위해서 for문을 사용하였고 list_L가 p(x)이고 L_int는 l_j이다. N을 이용한 for문이 $\prod$를 수행하는 것이고 k를 이용한 for문은 l_j를 각 uniform node에서의 값을 구하는 것이다.

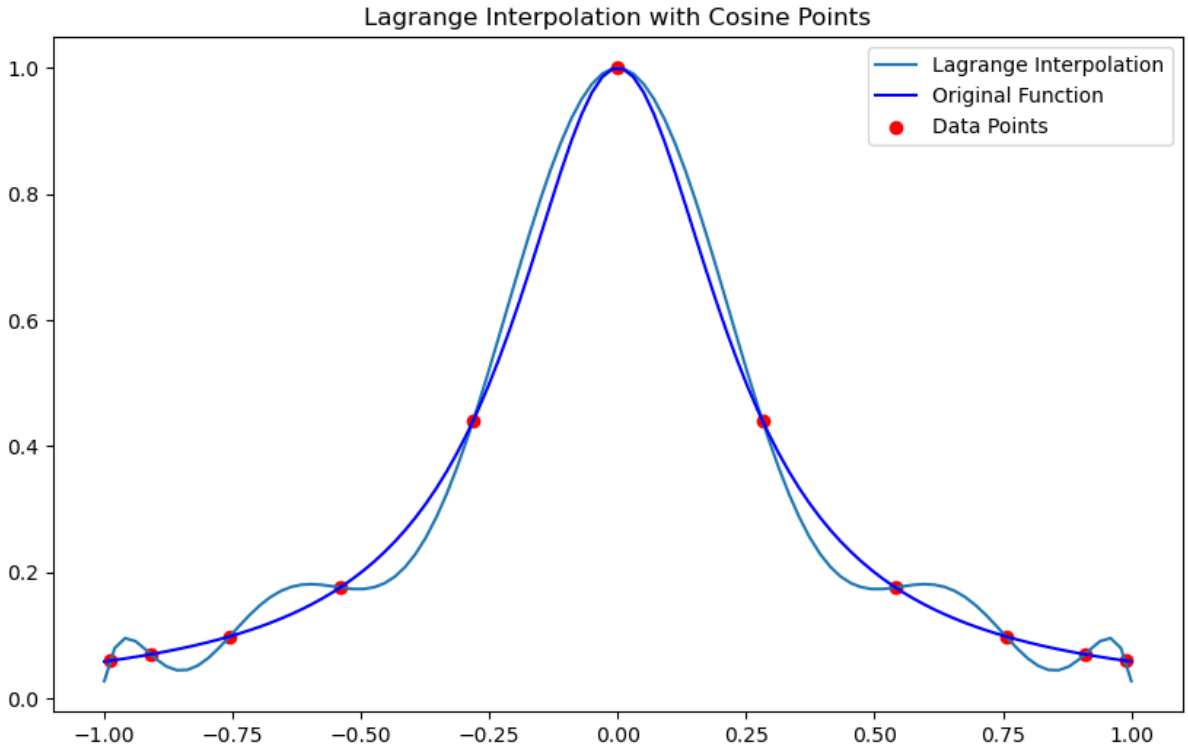


결과를 확인해보면 실제 함수와 internation한 값이 0에 가까울수록 잘 맞지만 0에서 멀어질수록 달라지는 것을 확인할 수 있다.

2. 동일한 과정을 Chebyshev nodes $x_i = cos\left(\frac{2i+1}{2n+2}\pi\right), i = 0, 1, \dots, 10$ 를 사용하여 진행하시오.

Chebyshev nodes를 구하기 위해서 list_x_cost를 만들어서 구하였다.

```
a = 100
list = np.linspace(-1, 1, a)
list_x = np.linspace(0, 10, 11)
list_x_cos = np.cos((2*list_x+1)/(2*10+2)*np.pi)
list_y = func(list_x_cos)


for k in range(len(list_x_cos)):
    if k == 0:
        list_L = np.zeros(len(list))
    L_int = np.ones(len(list))
    for n in range(len(list_x_cos)):
        if n != k:
            L_int = L_int*(list-list_x_cos[n])/(list_x_cos[k]-list_x_cos[n])
    list_L = list_L + L_int*list_y[k]

plt.figure(figsize=(10, 6))
plt.plot(list, list_L, label='Lagrange Interpolation')
plt.plot(list, func(list), label='Original Function', color='blue')
plt.scatter(list_x_cos, list_y, color='red', label='Data Points')
plt.title('Lagrange Interpolation with Cosine Points')
plt.legend()
```

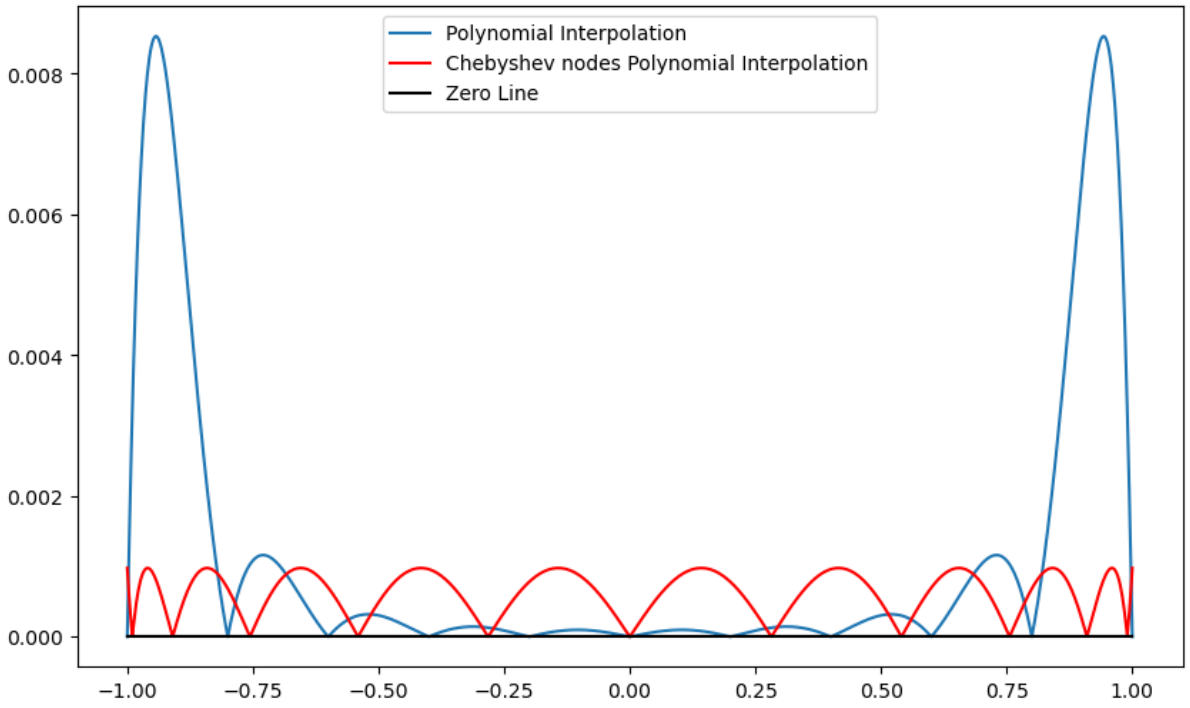


결과를 확인해보면 실제 함수와 internation한 값이 원래 함수와 비슷하지만 진동하는 것처럼 보인다.

3. Chebyshev nodes를 사용하였을 때가, 동일한 간격으로 나누어진 격자 점을 사용한 경우보다 좋은 결과를 보이는지를 서술하시오. [HINT : uniform and Chebyshev nodes를 사용하여 $\prod_{i=0}^{n}|x-x_i|$를 그려보시오.]

힌트에서 주어진 식을 계산하는 것을 for 구문을 이용하여 구하였다.

```
a = 1001
list = np.linspace(-1, 1, a)
list_x = np.linspace(-1, 1, 11)
list_x_cos = np.cos((2*np.linspace(0, 10, 11)+1)/(2*10+2)*np.pi)
pi_uni = np.ones(len(list))
pi_cos = np.ones(len(list))
for i in range(len(list_x)):
    pi_uni = pi_uni * np.abs(list-list_x[i])
    pi_cos = pi_cos * np.abs(list-list_x_cos[i])


plt.figure(figsize=(10, 6))
plt.plot(list, pi_uni, label='Polynomial Interpolation')
plt.plot(list, pi_cos, label='Chebyshev nodes Polynomial Interpolation', color='Red')
plt.plot(list, np.zeros(len(list)), label='Zero Line', color='black')
plt.legend()
```



Uniform node를 사용하면 0에서 가까울 때의 차이가 그렇게 크지 않지만 0에서 멀어질수록 오차의 범위가 더 커지게 된다. 하지만, Chebyshev node를 사용하면 0에 가까울 때 uniform node보다 오차가 작지만 x가 커진다고 해서 오차가 더 커지지 않는다. 이는 곧 두 node의 결과에서 볼 수 있듯 interpolation의 값이 0에서 멀어진다고 해서 원래 함수와의 차이가 커지지 않는다는 것을 확인할 수 있다.

4. Cubic spline interpolation 을 사용하여 1,2에 대해 진행하시오.

**** 양 끝 경계에 대해서는 각 점에서 두 번 미분한 값이 0 이라는 조건을 추가한다.**

Cubic spline interpolation을 하기 위해서는 각 구간의 함수를 따로 정의하여 internation을 하는 것이다.

$$f(x)=\begin{cases} f_1(x) & if \quad x_1 \le x < x_2 \\ f_2(x) & if \quad x_2 \le x < x_3 \\ \vdots & \\ f_{N-1}(x) & if \quad x_{N-1} \le x < x_N \end{cases}$$

경계조건으로 인해 fn(x)은 4차 식으로 사용할 수 있다.

$f_j(x)=a_{1,j}+a_{2,j}(x-x_j)+a_{3,j}(x-x_j)^2+a_{4,j}(x-x_j)^3$ 로 정의할 수 있다.

$h_j=(x_{j+1}-x_j)$ 라하고 $f_j^{(2)}(x_j)=\mathrm{M}_j$ 하면 위 식을 풀면

$$y_j=a_{1,j}$$

$$f_j^{(1)}(x_j)=a_{2,j}$$

$$a_{3,j}=\frac{\mathrm{M}_j}{2}$$

$$a_{4,j}=\frac{(\mathrm{M}_{j+1}-\mathrm{M}_j)}{6h_j}$$

$$h_{j-1}\mathrm{M}_{j-1}+(2h_j+2h_{j-1})\mathrm{M}_j+h_j\mathrm{M}_{j+1}=6\left(\frac{(y_{j+1}-y_j)}{h_j}-\frac{(y_j-y_{j-1})}{h_{j-1}}\right)$$

$$\mathrm{V}_j=6\left(\frac{(y_{j+1}-y_j)}{h_j}-\frac{(y_j-y_{j-1})}{h_{j-1}}\right)$$

를 구할 수 있다. 이를 이용하여 행렬을 구하면 다음과 같다.

$$\begin{bmatrix} 2(h_2+h_1) & h_2 & 0 & 0 & \cdots & 0 & 0 & 0 & 0 & 0 \\ h_2 & 2(h_3+h_2) & h_3 & 0 & \cdots & 0 & 0 & 0 & 0 & 0 \\ 0 & h_3 & 2(h_4+h_3) & h_4 & \cdots & 0 & 0 & 0 & 0 & 0 \\ \cdot & \cdot & \cdot & \cdot & \cdots & & \cdot & \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot & \cdots & & \cdot & \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot & \cdots & & \cdot & \cdot & \cdot & \cdot \\ 0 & 0 & 0 & 0 & 0 & \cdots & h_{N-4} & 2(h_{N-3}+h_{N-4}) & h_{N-4} & 0 \\ 0 & 0 & 0 & 0 & 0 & \cdots & 0 & h_{N-3} & 2(h_{N-2}+h_{N-3}) & h_{N-2} \\ 0 & 0 & 0 & 0 & 0 & \cdots & 0 & 0 & h_{N-2} & 2(h_{N-1}+h_{N-2}) \end{bmatrix} \begin{bmatrix} \mathrm{M}_2 \\ \mathrm{M}_3 \\ \mathrm{M}_4 \\ \cdot \\ \cdot \\ \cdot \\ \mathrm{M}_{N-3} \\ \mathrm{M}_{N-2} \\ \mathrm{M}_{N-1} \end{bmatrix} = \begin{bmatrix} \mathrm{V}_2 \\ \mathrm{V}_3 \\ \mathrm{V}_4 \\ \cdot \\ \cdot \\ \cdot \\ \mathrm{V}_{N-3} \\ \mathrm{V}_{N-2} \\ \mathrm{V}_{N-1} \end{bmatrix}$$

이는 삼각대각행렬로 이는 토마스 알고리즘을 이용하여 풀 수 있다. 이는 아래와 같은 def 함수를 사용하여 구할 수 있다.

```python
# 토마스 알고리즘
def thomas_algorithm(a, b, c, d):
    n = len(d)
    c_prime = np.zeros(n-1)
    d_prime = np.zeros(n)

    c_prime[0] = c[0] / b[0]
    d_prime[0] = d[0] / b[0]

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

    d_prime[n-1] = (d[n-1] - a[n-2] * d_prime[n-2]) / (b[n-1] - a[n-2] * c_prime[n-2])

    x = np.zeros(n)
    x[-1] = d_prime[-1]

    for i in range(n-2, -1, -1):
        x[i] = d_prime[i] - c_prime[i] * x[i+1]

    return x
```

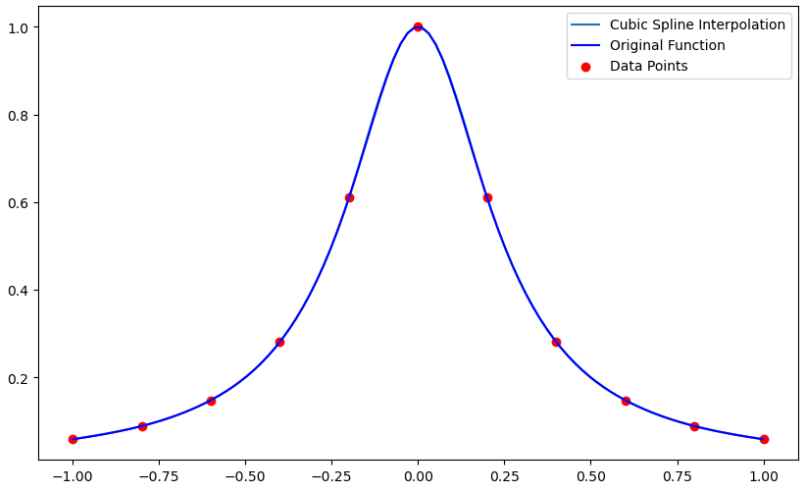
위 행렬을 만들기 위해서 아래와 같은 for문을 사용하여 만들었다.

```python
# Cubic Spline Interpolation(uniform nodes)
a = 100
list = np.linspace(-1, 1, a)
list_x = np.linspace(-1, 1, 11)
list_y = func(list_x)
h_list = list_x[1:] - list_x[:-1]

a = np.zeros(len(h_list)-2)
b = np.zeros(len(h_list)-1)
c = np.zeros(len(h_list)-2)

for i in range(len(h_list)-1):
    if i == 0 :
        b[i] = 2*(h_list[i]+h_list[i+1])
        c[i] = h_list[i+1]
    elif i == len(h_list)-2:
        a[i-1] = h_list[i]
        b[i] = 2*(h_list[i]+h_list[i+1])
    else:
        a[i-1] = h_list[i]
        b[i] = 2*(h_list[i]+h_list[i+1])
        c[i] = h_list[i+1]

V = 6*((list_y[2:]-list_y[1:-1])/h_list[1:]-(list_y[1:-1]-list_y[:-2])/h_list[:-1])

M = thomas_algorithm(a, b, c, V)

M = np.concatenate(([0], M, [0]))
```

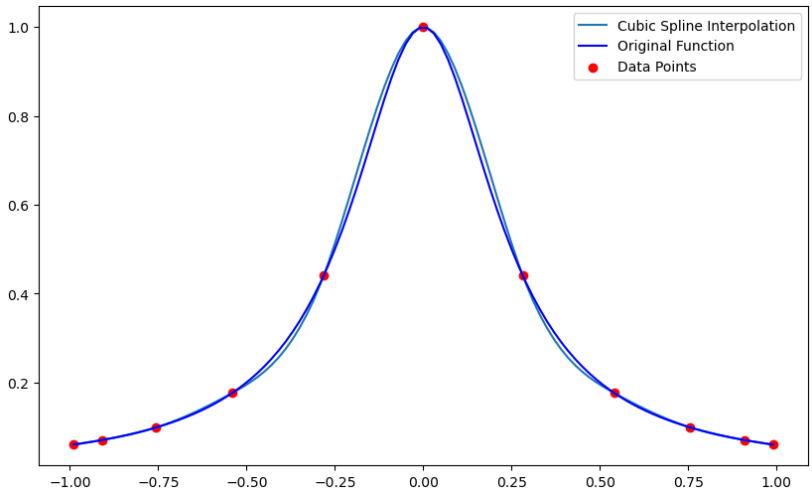
결과적으로 M과 V를 구해 이 값들을 사용하여 $a_{1,j}, a_{2,j} a_{3,j} a_{4,j}$을 구했다. 또한, for문과 if 문을 사용하여 해당 구간에서 어떤 함수를 사용할지에 대해 결정했다.

```
a1 = list_y[:-1]
a2 = (list_y[1:]-list_y[:-1])/h_list - (M[1:]+2*M[:-1])/6*h_list
a3 = M[:-1]/2
a4 = (M[1:]-M[:-1])/(6*h_list)

y = []
for i in range(len(list)):
    for j in range(len(list_x)-1):
        if list[i] >= list_x[j] and list[i] <= list_x[j+1]:
            y.append(a1[j] + a2[j]*(list[i]-list_x[j]) + a3[j]*(list[i]-list_x[j])**2 + a4[j]*(list[i]-list_x[j])**3)


plt.figure(figsize=(10, 6))
plt.plot(list, y, label='Cubic Spline Interpolation')
plt.plot(list, func(list), label='Original Function', color='blue')
plt.scatter(list_x, list_y, color='red', label='Data Points')
plt.legend()
```

결과는 아래와 같다.



Chebyshev node는 똑같은 방식으로 list_x 대신 list_x_cos를 이용하여 계산하였다. 결과는 다음과 같다.



5. Chebyshev node를 사용하여 Lagrangian interpolation과 Cubic spline method를 적용한 결과와 Exact solution 과의 비교를 통한 Error 분석을 하고, 이에 대한 본인의 의견을 서술하시오.

첨부한 코드를 확인하면 원래 함수와 intetpolation으로 구한 함수의 값의 차이의 평균을 구하였다. 결과적으로 Lagrangian interploation에서는 0.0306으로 나왔고 Cubic spline method에서는 0.00876으로 나타났다. 이는 결과적으로 Cubic spline method가 더 정확하게 나타낼 수 있다고 생각한다. 왜냐하면 Lagrangian interpolation으로 해를 구하게 되면 점의 개수가 커지면 커질수록 방정식의 차수가 높아진다. 이는 x가 커질수록 interpolation의 값이 x에 큰 영향을 받는다는 것을 확인할 수 있다. 그에 반해 Cubic spline은 항상 4차식으로 일정하므로 x가 커지면서 interpolation의 값의 영향이 더 많이 받지 않는다고 생각할 수 있다. 물론 주어진 함수 말고 다른 함수에 적용하거나 알고 있는 점의 개수가 변하면 변할 수 있다고 예상되지만 점이 많아지면 많아질수록 Cubic Spline method와 Lgrangian interpolation의 정확도 차이는 커질 것이라고 예상할 수 있다.