Week1 report

김민규

실행 환경: google colab

사용 언어: python

1) CSE SERVER를 사용한 기초계산

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

```
import random

#임의의 숫자 100개 생성
a = [random.randint(0,1000) for i in range(100)]

#오름차순 정렬
numbers = sorted(a)

print(a)
print(numbers)
```

Random 라이브러리, List comprehension을 사용하여 임의의 숫자 100개를 생성하고 sorted 함수를 이용해 오름차순으로 정렬했다. 결과는 다음과 같다.

[847, 604, 516, 564, 690, 200, 982, 134, 312, 766, 175, 980, 484, 274, 984, 764, 601, 713, 666, 87, 488...]

[26, 50, 59, 70, 71, 78, 87, 88, 92, 94, 95, 111, 119, 122, 125, 134, 137, 159, 165, 167, 170, ...]

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.

```
import numpy as np

#임의의 두 4x4 matrix 생성
A = np.random.randint(0,100,(4,4))
B = np.random.randint(0,100,(4,4))

#두 matrix의 곱
C = np.dot(A,B)

#matirx C의 inverse 계산
C_det = np.linalg.det(C) #C의 determinant
C_trans = np.transpose(C) #C의 전치행렬
C_cofactor = np.zeros((4,4))

for row in range(4):
  for col in range(4):
    C_minor = np.delete(np.delete(C_trans, row, axis=0), col, axis=1) #소행렬 계산
    C_cofactor[row,col]=((-1) ** (row + col)) * np.linalg.det(C_minor) #여인자 계산

C_inv = C_cofactor/C_det

#Identity matrix 확인
I = np.dot(C,C_inv)

print(A)
print(B)
print(C)
print(C_inv)
print(I)
```

Determinant와 cofactor를 이용하여 역행렬을 구했다. 결과는 다음과 같다.

```
[[84 93 54 45]
 [72 44 70 97]
 [32 77 72 95]
 [31 39 30 57]]
[[99 32 49 81]
 [ 8 23 99 34]
 [70 60 18 69]
 [35 60 91 85]]
[[14415 10767 18390 17517]
 [15775 13336 17971 20403]
 [12149 12815 19132 18253]
 [ 7476  7109 11107 10752]]
[[ 0.00051766  0.00010125  0.00058597 -0.00203026]
 [ 0.00028341  0.00017686  0.00168546 -0.00365864]
 [ 0.00025072 -0.00027898  0.00021612 -0.00024599]
 [-0.00080632  0.00010085 -0.00174509  0.0041778 ]]
[[ 1.00000000e+00 -1.66971471e-14  8.34629674e-14 -1.49148055e-13]
 [ 1.90935592e-14  1.00000000e+00  9.80376804e-14 -1.52943630e-13]
 [ 1.41853760e-14 -1.55730463e-14  1.00000000e+00 -1.51701568e-13]
 [ 8.71525074e-15 -9.54791801e-15  5.51780843e-14  1.00000000e+00]]
```

Identity matrix가 만들어진 모습을 확인할 수 있다.

2) Root-finding method

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.

이분법이란 사잇값 정리를 이용하여 근사해를 구하는 방법이다. 다음과 같이 코드를 작성하여 이분법을 구현할 수 있다.

```
#Error bound
tol=10**(-8)

#주어진 함수
def f(x):

  return x**5 -9*x**4 - x**3 + 17*x**2 - 8*x - 8

#구간 설정
A = [-10, -1, 0]
B = [-1, 0, 10]

print("이분법 결과")

#bisection 함수 정의
def bisection(a, b):
  if f(a)*f(b)>0:
      print("부적절한 구간 설정")
  else:
    j = 0 #계산 횟수
    while b-a > tol:
      m = a + (b-a)/2
      if f(a)*f(m)<0:
        b = m
      else:
        a = m
      j += 1
    print("근사해:",m," 계산 횟수:",j)

for i in range(len(A)):
  bisection(A[i],B[i])
```

결과는 다음과 같다.

```
이분법 결과
근사해: -1.3875071099027991  계산 횟수: 30
근사해: -0.5104293450713158  계산 횟수: 27
근사해: 8.910696404054761  계산 횟수: 30
```

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

뉴턴법은 접선을 이용하여 근사해를 구하는 방법이다.

```
#Error bound
tol=10**(-8)

#주어진 함수
def f(x):

  return x**5 - 9*x**4 - x**3 + 17*x**2 - 8*x - 8

#도함수
def fp(x):

  return 5*x**4 - 36*x**3 - 3*x**2 + 34*x - 8

#초기값 설정
x_0 = [-10, -0.1, 10]

print("뉴턴법 결과")
#newton 함수 정의
def newton(x):
  i = 0 #계산 횟수
  while True:
    x_n = x - f(x)/fp(x)
    if abs(x_n - x) < tol:
      break
    x = x_n
    i += 1
  print("근사해:",x_n, "계산 횟수", i)

for i in range(len(x_0)):
  newton(x_0[i])
```

결과는 다음과 같다.

```
뉴턴법 결과
근사해: -1.3875071055826167 계산 횟수 12
근사해: -0.5104293428178256 계산 횟수 4
근사해: 8.910696402960298 계산 횟수 5
```

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

초기 조건을 0으로 두었을 경우 2분이 걸려도 근사해가 구해지지 않는 결과가 발생했다. 이는 초기값을 어떻게 설정하는지가 중요한 뉴턴법의 특징 중 하나로 볼 수 있다. 만약 어느 지점에서 미분계수가 0이 되거나, 초기값을 잘못 설정하면 해를 아예 찾을 수 없는 상황이 발생할 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) $x_1 = 10, x_2 = 9.9$

시컨트법이란 두 점을 이은 직선을 이용하여 근사해를 구하는 방법이다.

```python
#Error bound
tol=10**(-8)

#주어진 함수
def f(x):

  return x**5 - 9*x**4 - x**3 + 17*x**2 - 8*x - 8

print("시컨트법 결과")
#secant 함수 정의
def secant(x_1, x_2):
  i = 0 #계산 횟수
  while True:
    x_3 = x_2 - f(x_2)*(x_2 - x_1)/(f(x_2) - f(x_1))
    if abs(x_3 - x_2) < tol:
      break
    x_1 = x_2
    x_2 = x_3
    i += 1
  print("근사해:",x_3, "계산 횟수", i)

secant(-10, -9.9)
secant(-0.1, -0.2)
secant(10, 9.9)
```

```
시컨트법 결과
근사해: -1.387507105582635 계산 횟수 17
근사해: -0.5104293428178256 계산 횟수 5
근사해: 8.910696402960298 계산 횟수 6
```

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

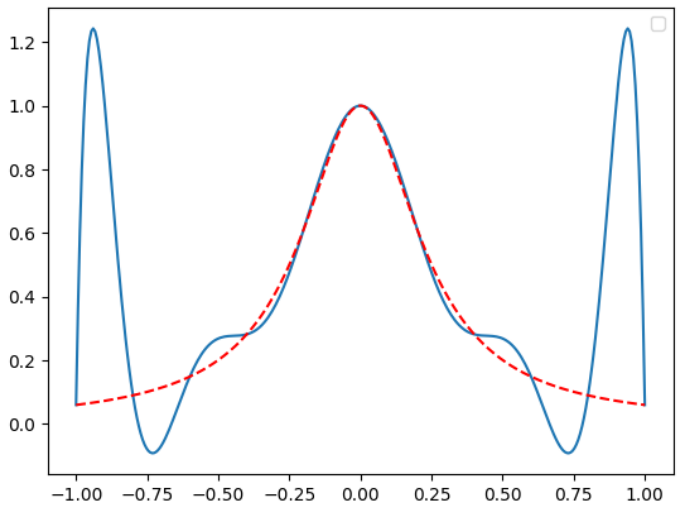
각각의 방법에 대해 계산 횟수를 구한 결과, 전체적으로 뉴턴법->시컨트법->이분법 순으로 근사해를 구하기 위한 계산 횟수가 증가하였다. 근사해를 구하는 안정성 또한 뉴턴법->시컨트법->이분법 순으로 증가한다.

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.

```
import numpy as np
import matplotlib.pyplot as plt
from sympy import symbols, simplify, lambdify

x = symbols('x')

#주어진 함수
def f(x):

  return 1/(1+16*x**2)

#lagrange 함수 정의
def lagrange(x_points,y_points):
  P = 0
  n = len(x_points)
  for i in range(n):
    L = 1
    for j in range(n):
      if j!= i:
        L *= (x - x_points[j])/(x_points[i] - x_points[j])
    P += y_points[i]*L

  return simplify(P)
```

```
#uniform nodes
x_points = np.linspace(-1,1,11)
y_points = [f(x_points[i]) for i in range(11)]

#lagrange polynomial 생성 및 출력
P = lagrange(x_points,y_points)
print("P(x):",P)
P_func = lambdify(x, P, modules = ['numpy'])

#시각화
x_vals = np.linspace(min(x_points),max(x_points), 300)
y_vals = P_func(x_vals)
y_true = f(x_vals)

plt.plot(x_vals,y_vals)
plt.plot(x_vals,y_true,'r--')
plt.legend()
```

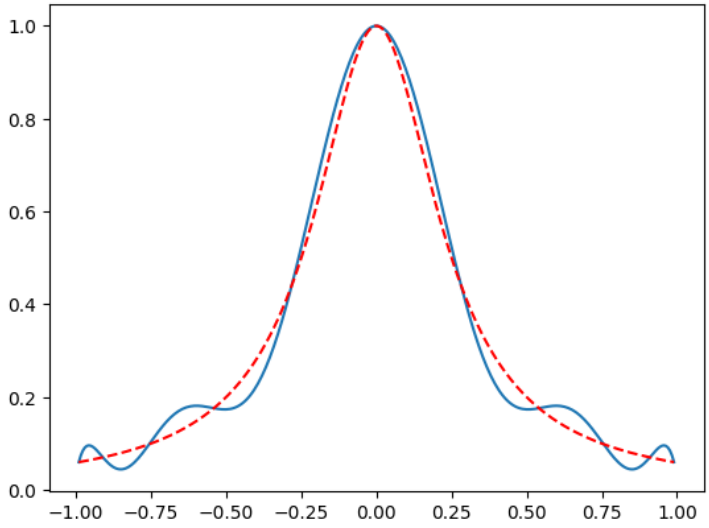


P(x): -33.8021340382322*x**10 - 2.66453525910038e-15*x**9 + 95.0685019825281*x**8 +...

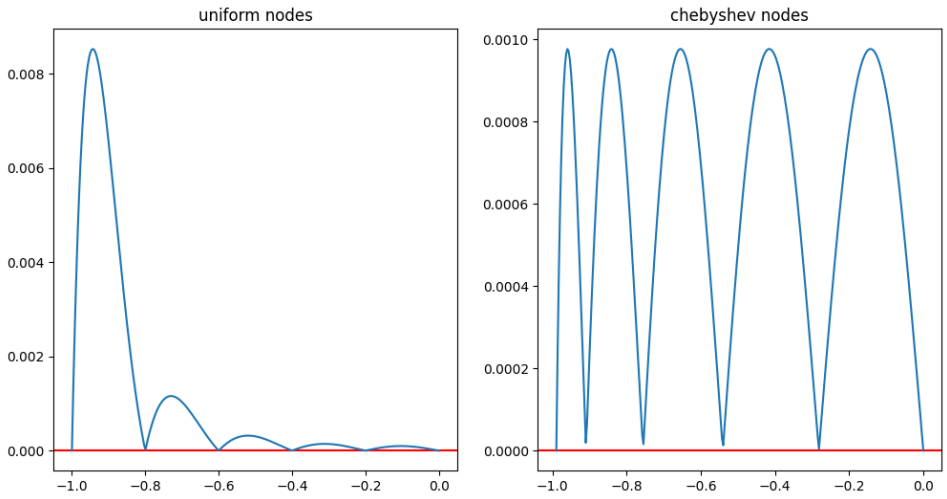
11개의 노드가 존재하므로 10차 다항식의 다항식이 만들어진다.

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 nodes
x_points = [math.cos(math.pi*(2*i + 1)/(2*10+2)) for i in range(11)]
y_points = [f(x_points[i]) for i in range(11)]
```



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.]



uniform nodes를 사용하여 lagrange interpolating을 하면 주어진 구간의 양 끝에서 진동하는 runge 현상이 발생한다. 반면 Chebyshev nodes를 사용하여 lagrange interpolating을 진행하면 구간의 양 끝에서 진동하는 현상이 작게 발생하는 것을 확인할 수 있다. 이는 위의 그래프를 통해 설명할 수 있다. uniform nodes와 달리 chebyshev nodes는 구간

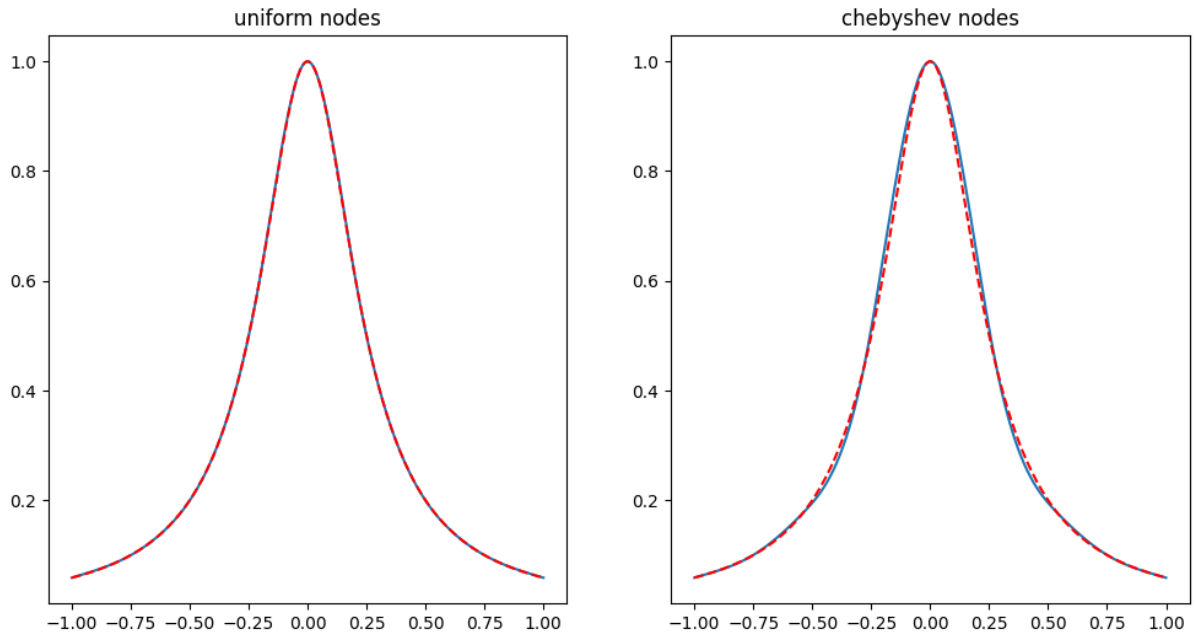
의 양 끝에 nodes가 몰려있다. 이는 interpolating을 할 때 양 끝에서의 결과를 보정해준다. 따라서 Chebyshev nodes를 사용할 때 구간의 양 끝에서 진동이 줄고 더 좋은 결과를 보여준다.

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

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

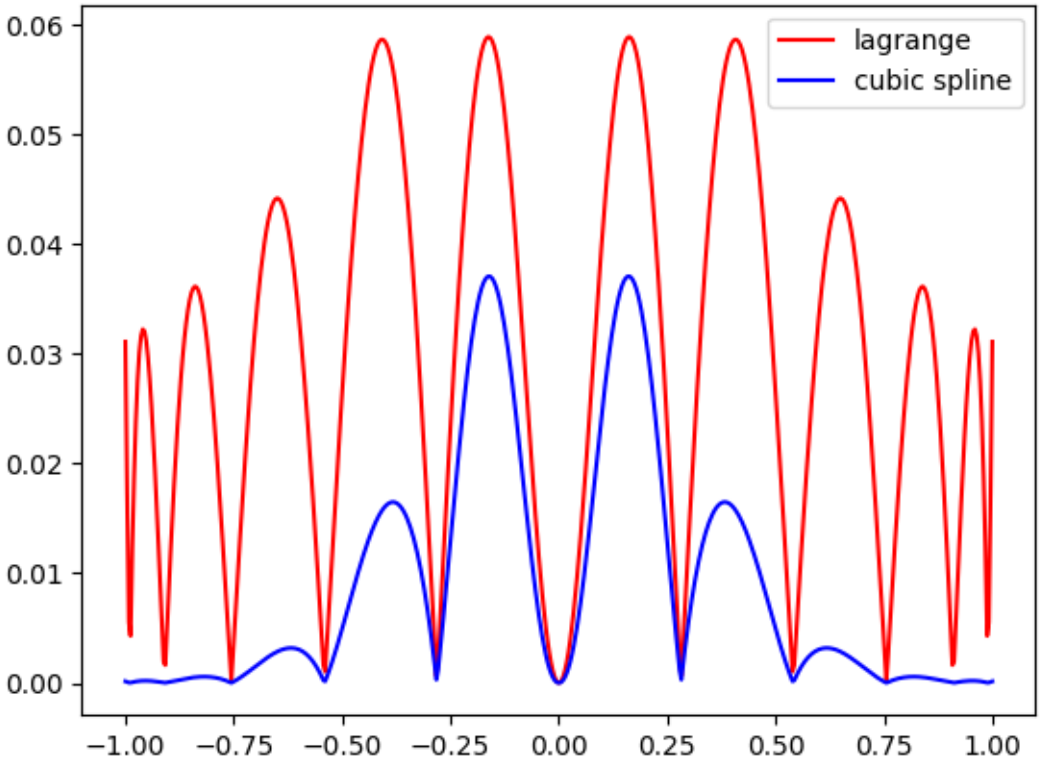
```
import numpy as np
import matplotlib.pyplot as plt
import math
from scipy.interpolate import CubicSpline

#주어진 함수
def f(x):

  return 1/(1+16*x**2)

#uniform and chebyshev node 생성
x_uniform = np.linspace(-1, 1, 11)
y_uniform = [f(x_uniform[i]) for i in range(11)]

x_chebyshev = sorted([math.cos(math.pi*(2*i + 1)/(2*10+2)) for i in range(11)])
y_chebyshev = [f(x_chebyshev[i]) for i in range(11)]

#Cubic spline interpolation
cs_uniform = CubicSpline(x_uniform, y_uniform, bc_type = 'natural')
cs_chebyshev = CubicSpline(x_chebyshev, y_chebyshev, bc_type = 'natural')

#interpolation 결과 시각화
x=np.linspace(-1,1,300)
y_cs_uniform = cs_uniform(x)
y_cs_chebyshev = cs_chebyshev(x)
y_true = f(x)
```

```
plt.figure(figsize=(12,6))

plt.subplot(1, 2, 1)
plt.plot(x,y_cs_uniform)
plt.plot(x, y_true,'r--')
plt.title('uniform nodes')

plt.subplot(1, 2, 2)
plt.plot(x,y_cs_chebyshev)
plt.plot(x, y_true,'r--')
plt.title('chebyshev nodes')
```



Lagrangian polynomial과 달리 양 끝점에서 runge 현상이 발생하지 않는다.

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


lagrange
cubic spline
0.06
0.05
0.04
0.03
0.02
0.01
0.00
−1.00
−0.75
−0.50
−0.25
0.00
0.25
0.50
0.75
1.00