2주차 과제1 #3

2022145079 임혜린

본 과제는 Python, VS Code를 사용하였음을 밝힙니다.

문제를 푸는데 있어 공통적으로 아래의 환경을 사용하였다.

```
###3
import numpy as np
import matplotlib.pyplot as plt
import math
plt.rcParams['font.family'] = 'Malgun Gothic'
plt.rcParams['axes.unicode_minus'] = False
```

2-1

2) Numerical differentiation

주어진 함수 $f(x) = sin((4-x)(4+x)), 0 \le x \le 8$ 에 대하여 다음 문제를 푸시오. Uniform nodes를 사용하여 33개의 격자점을 사용하도록 한다.

1. Second-order one-sided difference shceme을 사용하여 경계에서의 f''를 유도하시오 (서술)

코드

```
###2-1
def f(x):
    return np.sin((4-x)*(4+x))

N=33
h=8/(N-1)
xlist=np.linspace(0,8,N)

def fffr(x):
    global h
    return (2*f(x)-5*f(x+h)+4*f(x+2*h)-f(x+3*h))/h**2

def fffl(x):
    global h
    return (2*f(x)-5*f(x-h)+4*f(x-2*h)-f(x-3*h))/h**2

print(f"""Uniform nodes 33개, Second-order one-sided difference shceme을
사용하여 유도한 결과 f''(0)={fffr(0)}, f''(8)={fffl(8)}입니다.""")
```

경계 0에서는 오른쪽의 세 점(0.25, 0.5, 0.75)을 이용한 3step 2차 후진 차분, 경계 8에서는 왼쪽 세 점 (7.25, 7.5, 7.75)을 이용한 3step 2차 전진 차분을 이용하여 f″의 근사값을 구하였다.

이에는 각 점에서의 테일러 전개를 조합하여 유도한 f″=(2f(x)-5f(x-h)+4f(x-2h)-f(x-3h))/h^2가 사용되었다.

결과

```
Uniform nodes 33개, Second-order one-sided difference shceme을
사용하여 유도한 결과 f''(0)=2.0248034695313226, f''(8)=-19.16647580126187입니다.
```

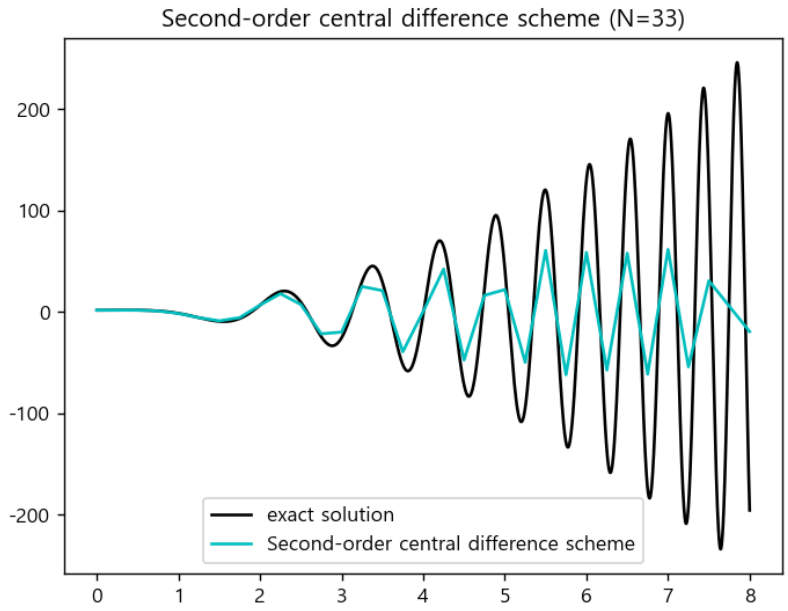
2-2

2. Second-order central difference scheme을 사용하여 exact solution 과 함께 f″를 그리시오. (정확도는 $O(\Delta x^2)$를 유지하도록 한다.)

코드

```
###2-2
N=33
h=8/(N-1)
xlist=np.linspace(0,8,N)
xor=np.linspace(0,8,3200)

def fff(x):
    global h
    return (f(x+h)-2*f(x)+f(x-h))/h**2

def exact_fff(x):
    return -2*np.cos(16-x**2)-4*x**2*(np.sin(16-x**2))

def endpoints(x):
    x[0]=fffr(0)
    x[-1]=fffl(8)

fff_xlist=[fff(x) for x in xlist]
endpoints(fff_xlist)

plt.plot(xor, exact_fff(xor), 'k', label="exact solution")
plt.plot(xlist, fff_xlist, 'c', label='Second-order central difference scheme')
plt.legend(loc='lower center')
plt.title("Second-order central difference scheme (N=33)")

plt.show()
```

결과



2-3

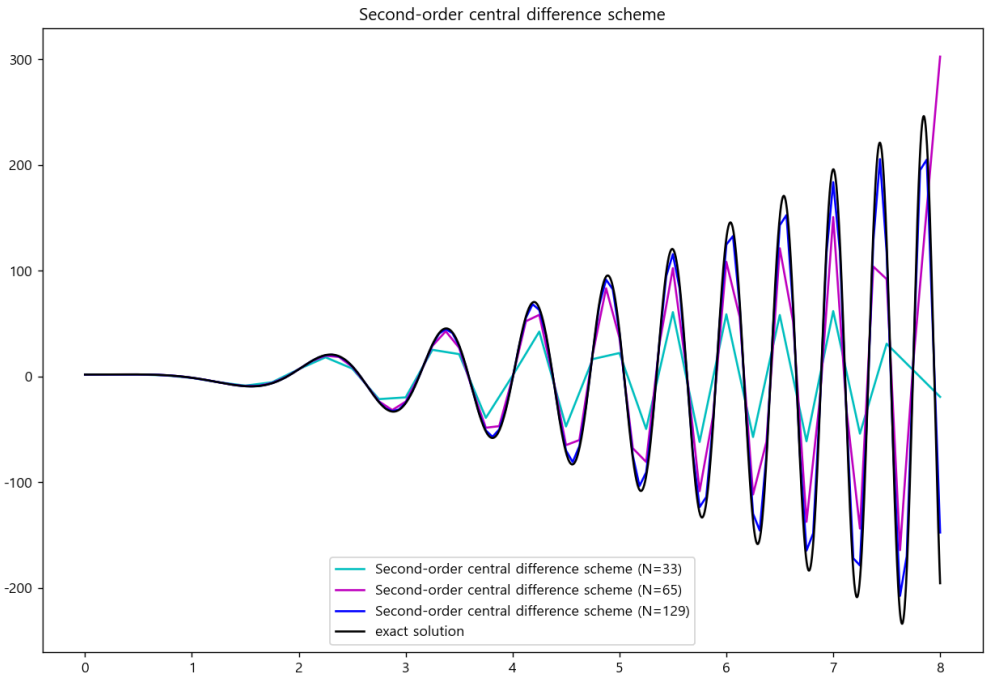
3. Second-order central difference scheme을 사용하여 격자 개수를 바꿔가며 (33,65,129) 정확도를 분석하시오. 정확도 분석은 x축을 $log(\Delta x)$, y축을 $log||e||$ 를 그리도록 한다. 본 방법에서 $||e||$ 는 L_2 $norm\ error$ 를 의미한다.

코드

```
###2-3
N33=33
h=8/(N33-1)
xlist33=np.linspace(0,8,N33)
fff_N33=[fff(x) for x in xlist33]
endpoints(fff_N33)

N65=65
h=8/(N65-1)
xlist65=np.linspace(0,8,N65)
fff_N65=[fff(x) for x in xlist65]
endpoints(fff_N65)
print(f'N=65, x=8에서의 함숫값: {fff_N65[-1]}')
print(f'x=8에서의 exact solution: {exact_fff(8)}')

N129=129
h=8/(N129-1)
xlist129=np.linspace(0,8,N129)
fff_N129=[fff(x) for x in xlist129]
endpoints(fff_N129)
```

정확도 분석에 앞서 N에 따른 세 f″ 그래프를 정의하고 한 번에 그려보았다. 각 경계에서는 중앙 차분을 사용할 수 없기 때문에 2-1처럼 후진 차분과 전진 차분을 사용하였다.

```
plt.figure(figsize=(12,8))
plt.plot(xlist33, fff_N33, 'c', label='Second-order central difference scheme (N=33)')
plt.plot(xlist65, fff_N65, 'm', label='Second-order central difference scheme (N=65)')
plt.plot(xlist129, fff_N129, 'b', label='Second-order central difference scheme (N=129)')
plt.plot(xor, exact_fff(xor), 'k', label="exact solution")
plt.legend(loc='lower center')
plt.title("Second-order central difference scheme")
plt.show()
```

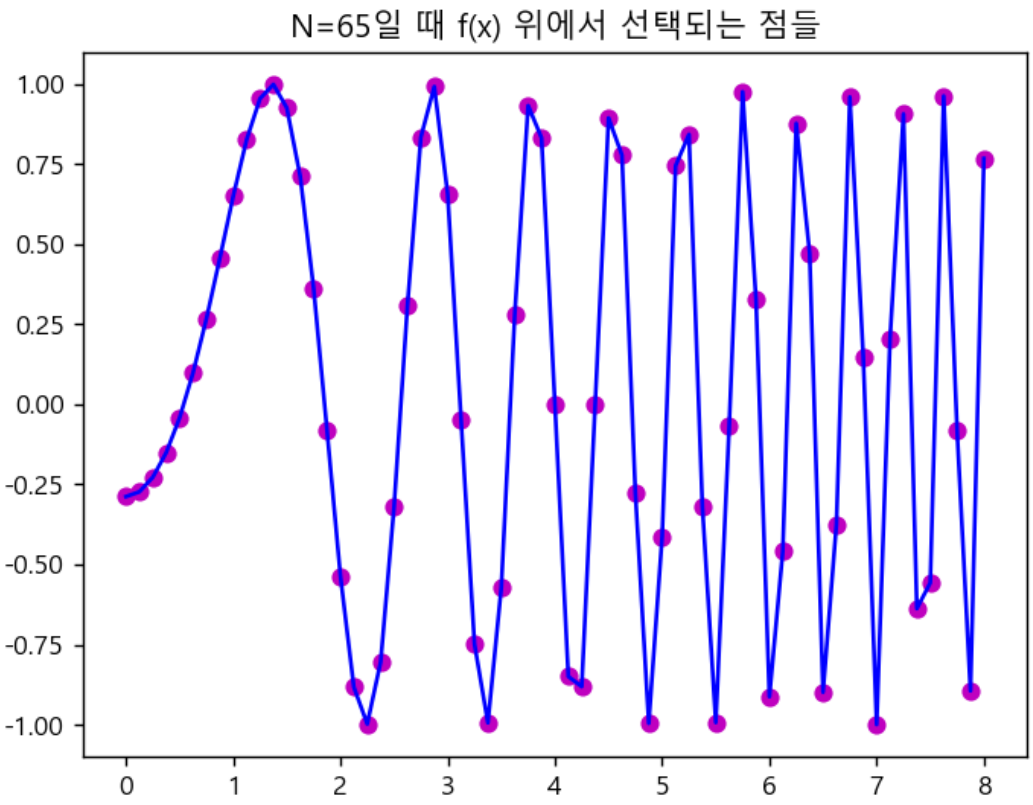



```
N=65, x=8에서의 함숫값: 302.53655332973847
```

그래프를 통해 N=65인 경우 x=8인 지점에서 함숫값이 302로 튀는 것을 확인하였다. Exact solution의 해는 -195로 매우 큰 차이를 보인다. 이 오차가 아래의 오차 검증에 큰 영향을 미쳐 오차 계산 시 이 점을 제외하고 진행하였다.

함숫값이 튀는 원인을 확인하기 위해 f(x)에 N=65인 경우 골라지는 점들을 찍어 확인해 보았다

```
y_f=[np.sin((4-x)*(4+x)) for x in xlist65]
plt.plot(xlist65, y_f, c='b', label='sin((4-x)*(4+x))')
plt.scatter(xlist65, y_f, c='m', label='sin((4-x)*(4+x))')
plt.title("N=65일 때 f(x) 위에서 선택되는 점들")
plt.show()
```

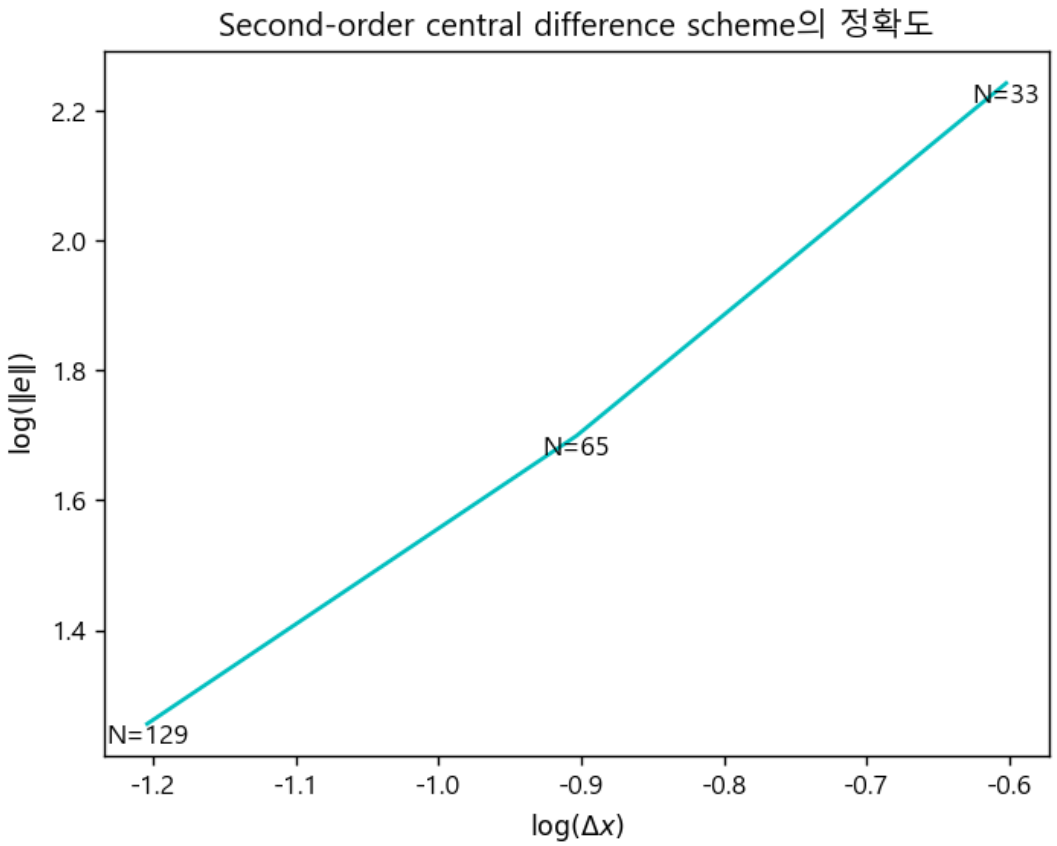


사용한 전진 차분의 식은 (2f(x)-5f(x-h)+4f(x-2h)-f(x-3h))/h^2이다. 가장 가중치가 큰 f(7.75)의 값이 극단적인 것이 주요 원인으로 보이나 직접 값을 확인해 보기 전까지는 이를 예상하기 어려울 것으로 보인다.

```
error33=[(exact_fff(xlist33[i])-fff_N33[i])**2 for i in range(33)]
# x=8일 때 값이 튀는 것을 보정하기 위하여 range(65)가 아닌 range(64)로 진행하였다.
error65=[(exact_fff(xlist65[i])-fff_N65[i])**2 for i in range(64)]
error129=[(exact_fff(xlist129[i])-fff_N129[i])**2 for i in range(129)]
error=[np.log10(math.sqrt(sum(error33)*8/(N33-1))),
       np.log10(math.sqrt(sum(error65)*8/(N65-1))),
       np.log10(math.sqrt(sum(error129)*8/(N129-1)))]
log_h=[np.log10(8/(N33-1)), np.log10(8/(N65-1)), np.log10(8/(N129-1))]



plt.plot(log_h, error, 'c', label='정확도')
plt.text(log_h[0], error[0], 'N=33', fontsize=10, ha='center', va='top')
plt.text(log_h[1], error[1], 'N=65', fontsize=10, ha='center', va='top')
plt.text(log_h[2], error[2], 'N=129', fontsize=10, ha='center', va='top')
plt.title("Second-order central difference scheme의 정확도")
plt.xlabel('$\log(\Delta x)$')
plt.ylabel('$\log(\Vert e \Vert)$')
plt.show()
```

```
from scipy.stats import linregress

print(f'order of accuracy: {linregress(log_h, error).slope}')
# order of accuracy: 1.6388
```

결과



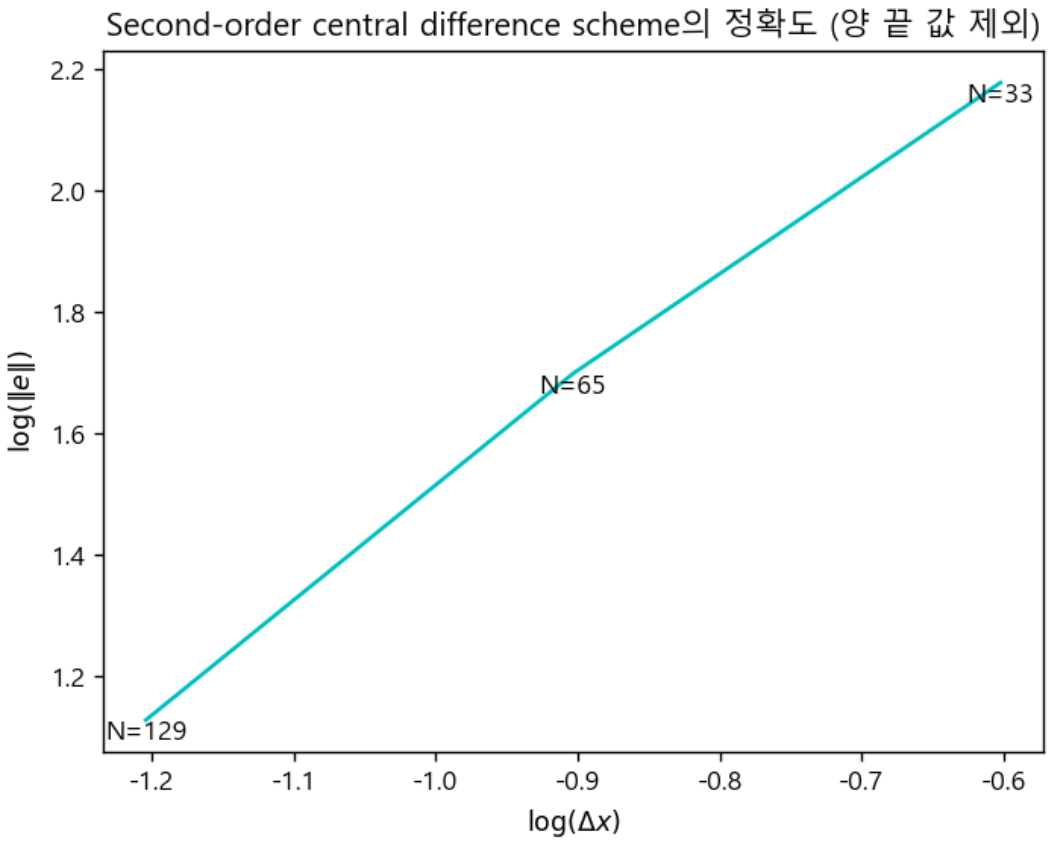
Order of accuracy: 1.6388

함수의 기울기가 유의미하지 않아 전진 차분, 후진 차분으로 만든 부분을 빼고 다시 error을 살펴보았다.

```
error33=[(exact_fff(xlist33[i+1])-fff_N33[i+1])**2 for i in range(31)]
error65=[(exact_fff(xlist65[i+1])-fff_N65[i+1])**2 for i in range(63)]
error129=[(exact_fff(xlist129[i+1])-fff_N129[i+1])**2 for i in range(127)]
error=[np.log10(math.sqrt(sum(error33)*8/(N33-1))),
       np.log10(math.sqrt(sum(error65)*8/(N65-1))),
       np.log10(math.sqrt(sum(error129)*8/(N129-1)))]
```

```
plt.plot(log_h, error, 'c', label='정확도')
plt.text(log_h[0], error[0], 'N=33', fontsize=10, ha='center', va='top')
plt.text(log_h[1], error[1], 'N=65', fontsize=10, ha='center', va='top')
plt.text(log_h[2], error[2], 'N=129', fontsize=10, ha='center', va='top')
plt.title("Second-order central difference scheme의 정확도 (양 끝 값 제외)")
plt.xlabel('$\log(\Delta x)$')
plt.ylabel('$\log(\Vert e \Vert)$')
plt.show()

from scipy.stats import linregress

print(f'order of accuracy: {linregress(log_h, error).slope}')
# order of accuracy: 1.7441
```

결과



Order of accuracy: 1.7441

함수의 기울기가 2와 유사함을 확인하였다. 이를 통해 오차가 Δx^2에 비례하여 감소함을 알 수 있다.

2)

2) (Baseball dynamics) 야구공은 날아가는 도중 다음과 같은 힘을 받는다. 중력(gravity)에 의한 힘, 유동 저항에 의한 항력(drag force), 그리고 공을 Figure 1(a)에서 볼 수 있는 바와 같이 공을 휘게 하는 Magnus force. 좌표축 (x,y,z)는 각각 투수에서 포수까지의 거리 축, 수평축, 그리고 수직축을 의미한다. 이 때, 야구공의 움직임에 대한 방정식은 다음과 같다.

$$\frac{dx}{dt} = v_x \tag{1}$$

$$\frac{dy}{dt} = v_y \tag{2}$$

$$\frac{dz}{dt} = v_z \tag{3}$$

$$\frac{dv_x}{dt} = -F(V)Vv_x + B\omega(v_z \sin\phi - v_y \cos\phi) \tag{4}$$

$$\frac{dv_y}{dt} = -F(V)Vv_y + B\omega v_x \cos\phi \tag{5}$$

$$\frac{dv_z}{dt} = -g - F(V)Vv_z - B\omega v_x \sin\phi \tag{6}$$

이 때, v_x, v_y, v_z 는 야구공의 각 속도 벡터를 의미하며, $V = \sqrt{v_x^2 + v_y^2 + v_z^2}$ 로 공의 속력을 의미한다. B는 Magnus force의 크기를 정의하는 특정 수이고, ω는 야구공의 회전율(rotation rate) 그리고 ϕ는 z축에 대한 ω의 방향(각도)를 나타낸다. G는 중력가속도로 9.81m/s^2 을 의미한다. 본 시뮬레이션에서 $B=4.1\times10^{-4}$ 그리고 $\omega = 1800rpm$ 으로 상정하여 문제를 풀도록 한다. 공에 가해지는 항력 $F(V)$ 은 다음과 같이 가정된다.

$$F(V) = 0.0039 + \frac{0.0058}{1 + exp[(V-35)/5]}$$

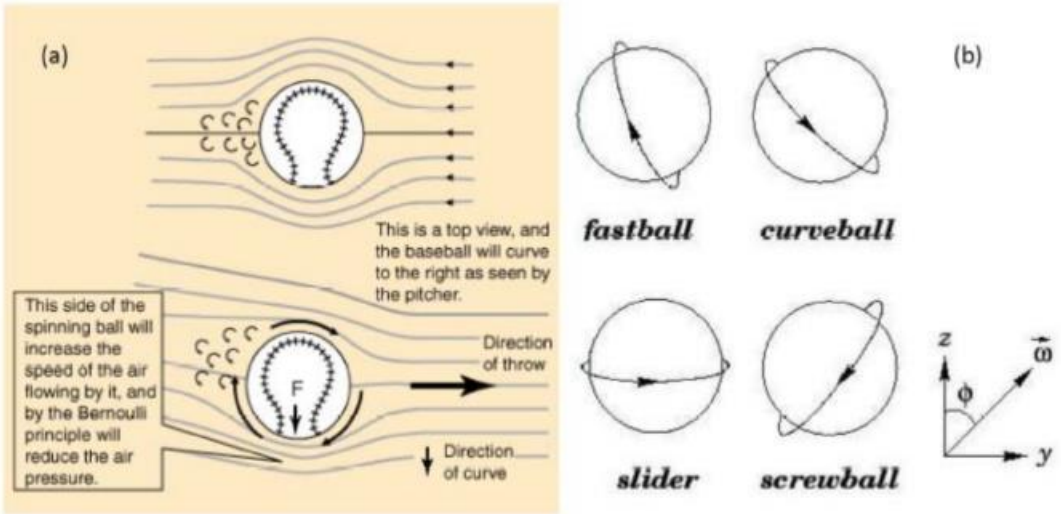


Figure 1: (a) Physics of baseball and (b) rotation direction for four pitches

공통적으로 아래의 환경에서 코딩하였다.

```
import numpy as np
import matplotlib.pyplot as plt
from mpl_toolkits.mplot3d import Axes3D
```

2-1

(1) 4th order RK method를 사용하여 위 6개 방정식을 계산하는 코드를 만드시오. t=0일 때의 초기조건은 $x(0)=0, y(0)=0, z(0)=h, v_x=v_0cos\theta, v_y=0, v_z=v_0sin\theta$ 로 주어지며, v_0는 투구의 초기 속도를 의미하며, θ는 투구의 상승각도를 의미하고 h는 지상에서 공을 놓는 위치까지의 수직 높이를 의미한다. 본 문제에서는 공이 포수의 위치인 x(t) =18.39m 까지 도달하도록 방정식을 계산하도록 한다. 코드를 만들어 나가는 과정에 대해서 서술하시오.

가장 먼저 상수를 정의한다. v0, h, phi는 아직 주어지지 않았으므로 0을 넣었다.

```
g=9.81
B=4.1e-4
w=1800*2*np.pi/60
theta=np.radians(1)
v0=0
h=0
phi=0
```

초기 조건을 설정한다. 식으로 주어진 6개의 변수의 초기 조건을 설정한 후, 넘파이 S로 묶었다.

```
x0=0
y0=0
z0=h
vx0=v0*np.cos(theta)
vy0=0
vz0=v0*np.sin(theta)
S=np.array([x0, y0, z0, vx0, vy0, vz0])
```

주어진 함수식인 F(V)를 정의한다.

```
def F(V):
    return 0.0039+0.0058/(1+np.exp((V-35)/5))
```

함수 deriv(S)를 정의한다. 먼저 dS/dt를 6개의 0이 있는 넘파이 배열로 정의한 후, 각각의 0를 문제에서 주어진 6개의 식에 대응되도록 하였다. 변수 6개의 초기 조건을 묶어 주었던 S를 deriv(S)의 미지수로 이용함으로써, S를 변화시키며 deriv(S)에 넣어주면 S 속 x, y, z, vx, vy, vz가 주어진 6개의 식에 순서대로 대응되도록 하였다.

```
def deriv(S):
    x, y, z, vx, vy, vz=S
    V=np.sqrt(vx**2+vy**2+vz**2)
    dS_dt=np.zeros(6)
    dS_dt[0]=vx
    dS_dt[1]=vy
    dS_dt[2]=vz
    dS_dt[3]=-F(V)*vx+B*w*(vz*np.sin(phi)-vy*np.cos(phi))
    dS_dt[4]=-F(V)*vy+B*w*vx*np.sin(phi)
    dS_dt[5]=-g-F(V)*V*vz-B*w*vx*np.sin(phi)
    return dS_dt
```

시간을 설정하고, every_S를 정의하여 각 시간에 따른 S를 보존한다.

```
dt=0.001
every_S=[S.copy()]
every_t=[0]
```

S와 deriv를 4th order RK method의 형식으로 연결하고, 반복 횟수를 정의한다. 시간에 따른 6개의 변수값은 묶여 every_S에 들어가게 된다. 또한 every_t에 멈추는 순간까지의 t를 저장하여 every_S와 같은 번째를 뽑으면 해당 시간에서의 6개의 변수값을 알 수 있도록 하였다. S[0]가 18.39보다 커지는 순간에 멈추도록 함으로써, 포수 위치에 가장 가까운 때의 값을 마지막으로 반복문이 끝나도록 하였다.

```
def RK4(S):
    while S[0]<18.39:
        k1=deriv(S)
        k2=deriv(S+dt*k1/2)
        k3=deriv(S+dt*k2/2)
        k4=deriv(S+dt*k3)
        S+=(k1+2*k2+2*k3+k4)*dt/6
        every_S.append(S.copy())
        every_t.append(every_t[-1] + dt)
    return every_S
```

마지막으로 이렇게 모인 every_S 속 값을 변수에 따라 묶어주면 6개의 변수 각각의 시간에 따른 값들이 모이게 된다.

```
every_S=np.array(every_S)
x=every_S[:,0]
y=every_S[:,1]
z=every_S[:,2]
vx=every_S[:,3]
vy=every_S[:,4]
vz=every_S[:,5]
```

(2) 방정식을 Figure 1(b)에서 보인 4가지 투구에 대해서 계산하고, 야구공의 trajectories를 각 투구에 대하여 plot 하시오. 상승각도 $\theta = 1^{\circ}$ 이며, fastball은 $v_0 = 40m/s$, 나머지는 $v_0 = 30m/s$ 로 각각 주어진다. 회전 방향(각도) ϕ 는 fastball, curveball, slider 그리고 screwball에 대하여 각각 $225^{\circ}, 45^{\circ}, 0^{\circ}$ 그리고 135° 로 주어진다. 수직 높이 h=1.7m로 주어진다.

코드의 큰 틀은 2-1과 같다. 다만, 마지막에 every_S으로부터 얻어야 할 값은 x, y, z 뿐이기 때문에 이것만을 모으도록 하였고, 이 x, y, z를 모아 궤적을 그려주는 함수 xyz_3d를 정의하였다.

또한 포수에게 가장 가까운 두 좌표, 즉 18.39의 직전과 직후의 (x, y, z) 값을 나타내어 공이 어느 좌표에 도착하는지 알 수 있게 하였다.

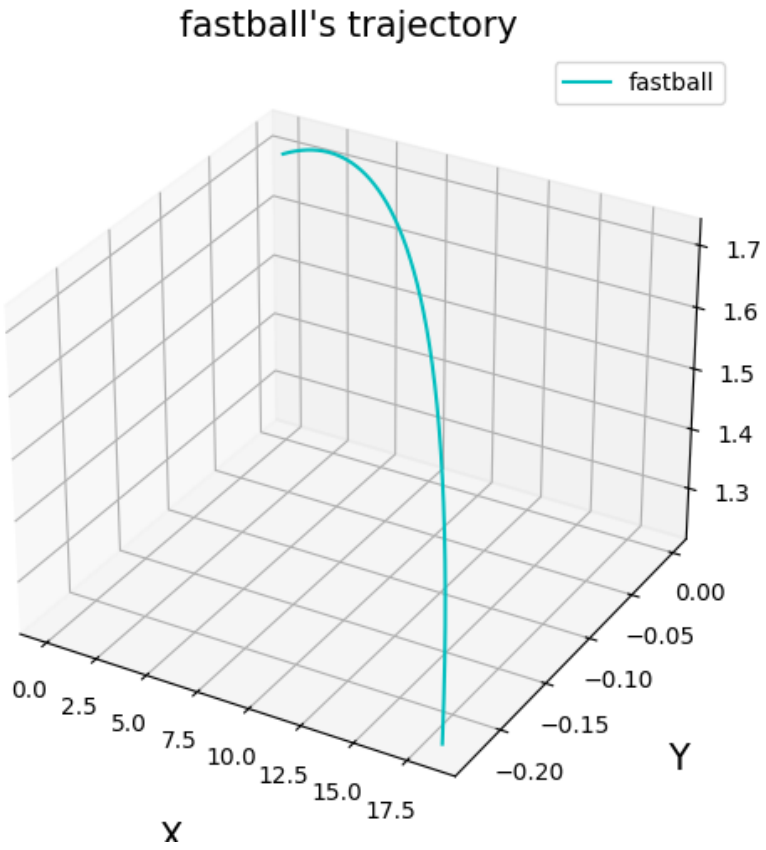
- Fastball의 경우

```
phi=np.radians(225)
S=np.array([0, 0, 1.7, 40*np.cos(theta), 0, 40*np.sin(theta)])
every_S=[S.copy()]
RK4(S)
every_S_fastball=np.array(every_S)
x_fastball=every_S_fastball[:,0]
y_fastball=every_S_fastball[:,1]
z_fastball=every_S_fastball[:,2]
print("포수에게 가장 가까운 두 좌표 (fastball)")
print(f'({x_fastball[-2]}, {y_fastball[-2]}, {z_fastball[-2]})')   # :18.3868 -0.2311 1.2242
print(f'({x_fastball[-1]}, {y_fastball[-1]}, {z_fastball[-1]})')   # :18.3908 -0.2312 1.2239
```

```
포수에게 가장 가까운 두 좌표 (fastball)
(18.386815150873545, -0.23114303731344046, 1.2242056750183636)
(18.390805828529494, -0.23124340114303116, 1.2239347693008211)
```

Fast ball의 그래프 코드와 결과

```
def xyz_3d(x, y, z , title):
    fig = plt.figure(figsize=(9, 6))
    ax = fig.add_subplot(111, projection='3d')
    ax.plot(x, y, z, c='c', label="fastball")
    ax.set_xlabel('X', fontsize=15, labelpad=20)
    ax.set_ylabel('Y', fontsize=15, labelpad=20)
    ax.set_zlabel('Z', fontsize=15, labelpad=20)
    ax.set_title(title, fontsize=15)
    ax.legend()
    plt.show()

xyz_3d(x_fastball, y_fastball, z_fastball, "fastball's trajectory" )
```



- Curveball의 경우

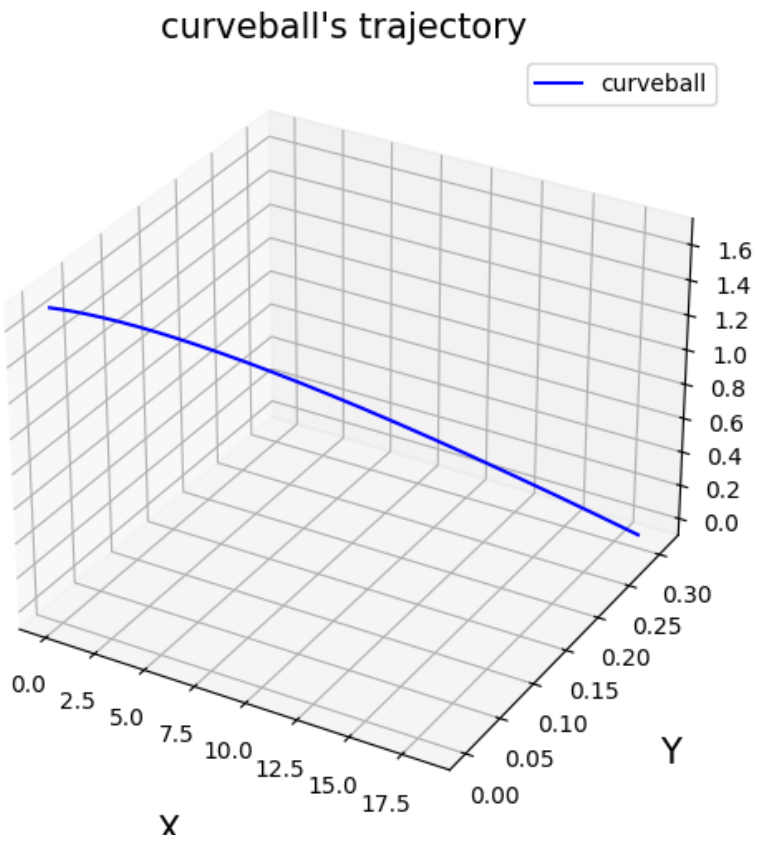
```
phi=np.radians(45)
S=np.array([0, 0, 1.7, 30*np.cos(theta), 0, 30*np.sin(theta)])
every_S=[S.copy()]
RK4(S)
every_S_curveball=np.array(every_S)
x_curveball=every_S_curveball[:,0]
y_curveball=every_S_curveball[:,1]
z_curveball=every_S_curveball[:,2]

print("포수에게 가장 가까운 두 좌표 (curveball)")
print(f'({x_curveball[-2]}, {y_curveball[-2]}, {z_curveball[-2]})')   # :18.3883 0.3091 -0.0629
print(f'({x_curveball[-1]}, {y_curveball[-1]}, {z_curveball[-1]})')   # :18.3914 0.3092 -0.0634
```

```
포수에게 가장 가까운 두 좌표 (curveball)
(18.38839016922089, 0.3090891867101659, -0.06288822692425174)
(18.391363455249117, 0.309189432148646, -0.06349693327914861)
```

Curveball의 그래프 코드와 결과

```
def xyz_3d(x, y, z , title):
    fig = plt.figure(figsize=(9, 6))
    ax = fig.add_subplot(111, projection='3d')
    ax.plot(x, y, z, c='b', label="curveball")
    ax.set_xlabel('X', fontsize=15, labelpad=20)
    ax.set_ylabel('Y', fontsize=15, labelpad=20)
    ax.set_zlabel('Z', fontsize=15, labelpad=20)
    ax.set_title(title, fontsize=15)
```

```
    ax.legend()
    plt.show()

xyz_3d(x_curveball, y_curveball, z_curveball, "curveball's trajectory" )
```



- Slider의 경우

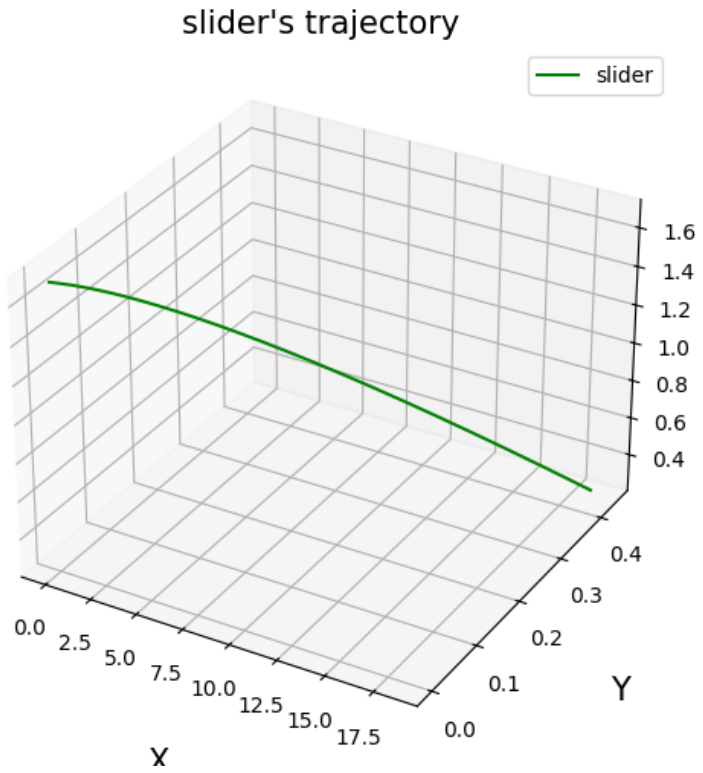
```
phi=np.radians(0)
S=np.array([0, 0, 1.7, 30*np.cos(theta), 0, 30*np.sin(theta)])
every_S=[S.copy()]
RK4(S)
every_S_slider=np.array(every_S)
x_slider=every_S_slider[:,0]
y_slider=every_S_slider[:,1]
z_slider=every_S_slider[:,2]
print("포수에게 가장 가까운 두 좌표 (slider)")
print(f'({x_slider[-2]}, {y_slider[-2]}, {z_slider[-2]})')   # :18.3884 0.3091 -0.0629
print(f'({x_slider[-1]}, {y_slider[-1]}, {z_slider[-1]})')   # :18.3914 0.3092 -0.0635
```

```
포수에게 가장 가까운 두 좌표 (slider)
(18.38822474992457, 0.4365643406411487, 0.23432701971786293)
(18.391205974354023, 0.4367061078758308, 0.23381202832839101)
```

Slider의 그래프 코드와 결과

```
def xyz_3d(x, y, z , title):
    fig = plt.figure(figsize=(9, 6))
```

```
    ax = fig.add_subplot(111, projection='3d')
    ax.plot(x, y, z, c='g', label="slider")
    ax.set_xlabel('X', fontsize=15, labelpad=20)
    ax.set_ylabel('Y', fontsize=15, labelpad=20)
    ax.set_zlabel('Z', fontsize=15, labelpad=20)
    ax.set_title(title, fontsize=15)
    ax.legend()
    plt.show()

xyz_3d(x_slider, y_slider, z_slider, "slider's trajectory" )
```



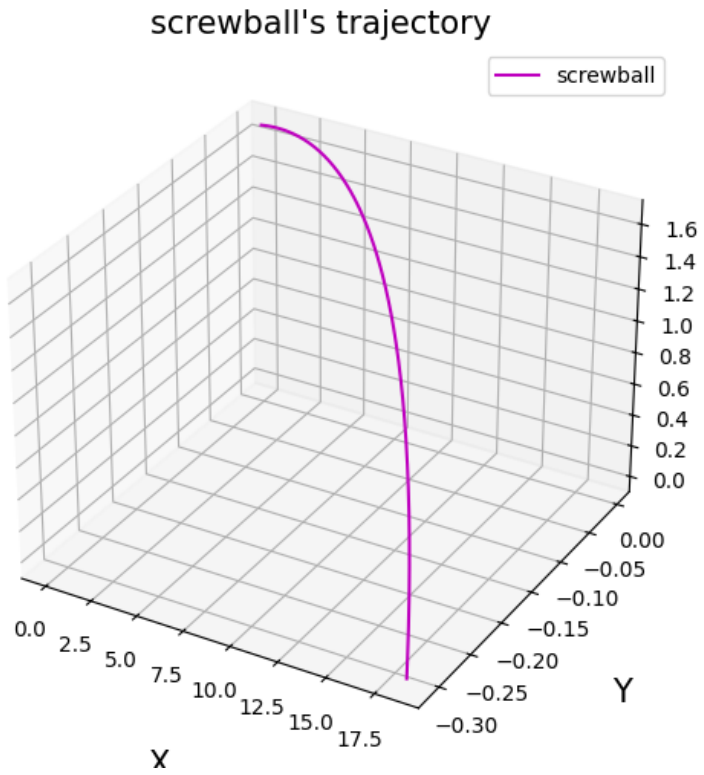
- Screwball의 경우

```
phi=np.radians(135)
S=np.array([0, 0, 1.7, 30*np.cos(theta), 0, 30*np.sin(theta)])
every_S=[S.copy()]
RK4(S)
every_S_screwball=np.array(every_S)
x_screwball=every_S_screwball[:,0]
y_screwball=every_S_screwball[:,1]
z_screwball=every_S_screwball[:,2]
print("포수에게 가장 가까운 두 좌표 (screwball)")
print(f'({x_screwball[-2]}, {y_screwball[-2]}, {z_screwball[-2]})')   # :18.3892 0.0 0.2354
print(f'({x_screwball[-1]}, {y_screwball[-1]}, {z_screwball[-1]})')   # :18.3921 0.0 0.2348
```

```
포수에게 가장 가까운 두 좌표 (screwball)
(18.38839016922089, -0.3090891867101658, -0.06288822692425174)
(18.391363455249117, -0.3091894321486459, -0.06349693327914861)
```

Screwball의 그래프 코드와 결과

```
def xyz_3d(x, y, z , title):
    fig = plt.figure(figsize=(9, 6))
    ax = fig.add_subplot(111, projection='3d')
    ax.plot(x, y, z, c='m', label="screwball")
    ax.set_xlabel('X', fontsize=15, labelpad=20)
    ax.set_ylabel('Y', fontsize=15, labelpad=20)
    ax.set_zlabel('Z', fontsize=15, labelpad=20)
    ax.set_title(title, fontsize=15)
    ax.legend()
    plt.show()

xyz_3d(x_screwball, y_screwball, z_screwball, "screwball's trajectory" )
```



마지막으로 네 가지 공의 궤적을 전부 함께 살펴보면 이와 같다.