Week 2 report

김민규

실행 환경: Google colab

사용 언어: Python

주어진 함수 $f(x) = sin\big((4-x)(4+x)\big), 0 \le x \le 8$ 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)

[0, 8]의 구간을 33개의 uniform nodes를 사용하므로 각 점사이의 거리 h = 0.25임을 알 수 있다. $x_0 = 0, x_1 = x_0 + h, x_2 = x_0 + 2h, x_3 = x_0 + 3h$라고 하자. 각 점에서 Taylor 전개식을 살펴보면 다음과 같다.

$$f(x_1) = f(x_0 + h) = f(x_0) + hf'(x_0) + \frac{h^2}{2}f''(x_0) + \frac{h^3}{6}f'''(x_0) + \frac{h^4}{24}f^{(4)}(x_0) + \cdots (1)$$

$$f(x_2) = f(x_0 + 2h)$$
$$= f(x_0) + 2hf'(x_0) + \frac{(2h)^2}{2}f''(x_0) + \frac{(2h)^3}{6}f'''(x_0) + \frac{(2h)^4}{24}f^{(4)}(x_0) + \cdots (2)$$

$$f(x_3) = f(x_0 + 3h)$$
$$= f(x_0) + 3hf'(x_0) + \frac{(3h)^2}{2}f''(x_0) + \frac{(3h)^3}{6}f'''(x_0) + \frac{(3h)^4}{24}f^{(4)}(x_0) + \cdots (3)$$

위의 식에서 $f'(x_0), f'''(x_0)$를 소거하여 정리하면 다음과 같다.

$$f''(x_0) = \frac{2f(x_0) - 5f(x_1) + 4f(x_2) - f(x_3)}{h^2} + O(h^2)$$

위와 같이 $x_{33} = 8$에 대해 다음과 같이 정리할 수 있다.

$$f''(x_{33}) = \frac{2f(x_{33}) - 5f(x_{32}) + 4f(x_{31}) - f(x_{30})}{h^2} + O(h^2)$$

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

Second-order central difference scheme을 사용하면 한 점에서 f''값을 다음과 같이 구할 수 있다.

$$f''(x_i) = \frac{f(x_{i-1}) - 2f(x_i) + f(x_{i+1})}{h^2} + O(h^2)$$

위의 공식을 이용하면 다음과 같이 코드를 작성할 수 있다.

```
#주어진 함수와 이계도함수
def f(x):

  return np.sin((4-x)*(4+x))
```

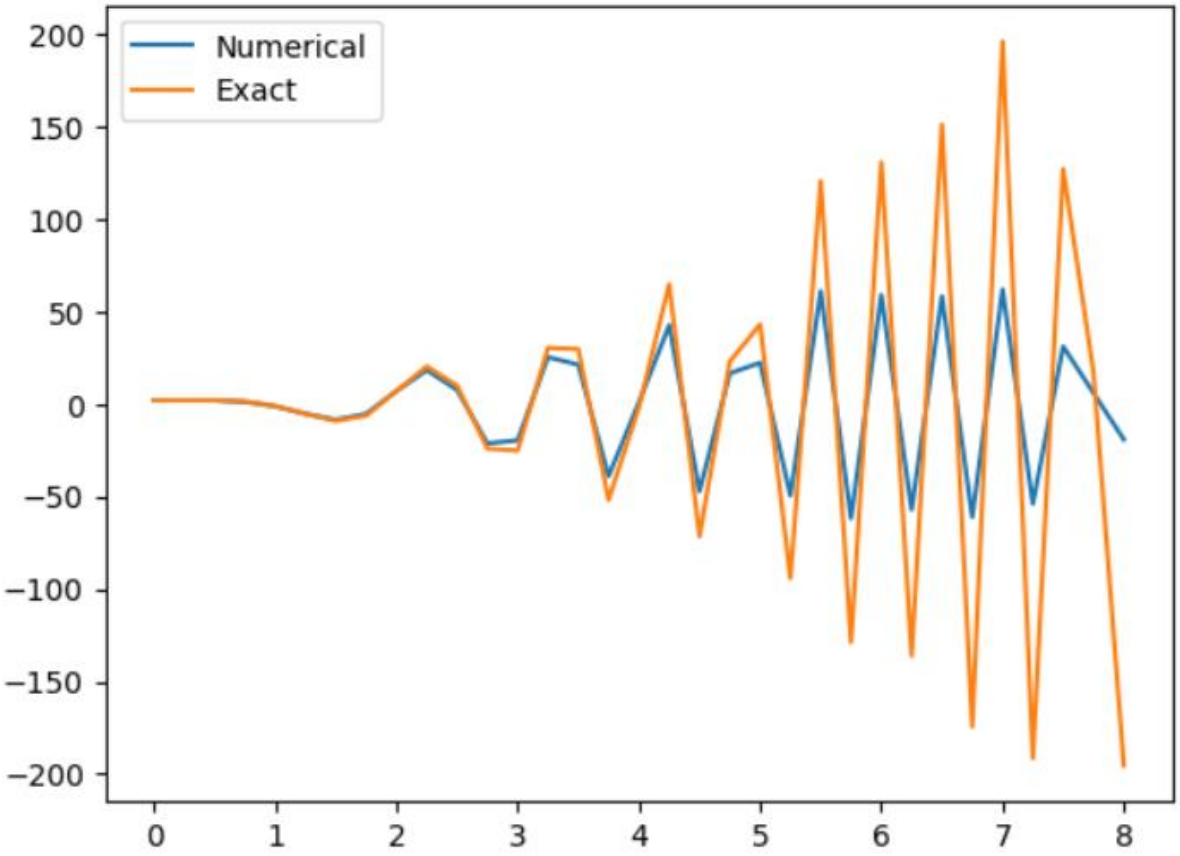
```
#second-order central difference shceme
for i in range(1, N-1):
  f_dd[i] = (f(x_points[i-1]) - 2*f(x_points[i]) + f(x_points[i+1])) /(h**2)
```

따라서 exact solution과 numerical 하게 구한 f''를 그리는 코드는 다음과 같다.

```
import numpy as np
import matplotlib.pyplot as plt

#주어진 함수와 이계도함수
def f(x):

  return np.sin((4-x)*(4+x))

def f_dd_real(x):

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

# 0부터 8까지 uniform node 생성
x_points = np.linspace(0, 8, 33)
N = len(x_points)
h = x_points[1] - x_points[0]

#f'' 근사값 계산
f_dd = np.zeros(N)

#second-order one-sided difference shceme
f_dd[0] = (2*f(x_points[0])-5*f(x_points[1])+4*f(x_points[2])-f(x_points[3]))/(h**2)
f_dd[-1] = (2*f(x_points[-1])-5*f(x_points[-2])+4*f(x_points[-3])-f(x_points[-4]))/(h**2)
```

```
#second-order central difference shceme
for i in range(1, N-1):
  f_dd[i] = (f(x_points[i-1]) - 2*f(x_points[i]) + f(x_points[i+1])) /(h**2)

#실제 node별 f''값
f_dd_true = f_dd_real(x_points)

#결과 비교 시각화
plt.plot(x_points, f_dd,label='Numerical')
plt.plot(x_points, f_dd_true, label ='Exact')
plt.legend()
```

그리고 결과는 다음과 같다.



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.

```
# error 계산 함수
def error(exact, numerical):

  return np.sqrt(np.mean((exact[2:-2] - numerical[2:-2])**2))
```

Exact solution과 numerical solution의 L2 norm error를 계산하는 함수이다. 구간의 양 끝 점에서 변동성이 크기 때문에 제외하여 계산했다.

```
grid_numbers = [33, 65, 129]
errors = []
delta = []

# grid 생성
for i in grid_numbers:
  x = np.linspace(0, 8, i)
  h = x[1] - x[0]
  delta.append(h)

# error 계산
  exact = f_dd_exact(x)
  numerical = f_dd_numerical(x, h)
  errors.append(error(exact, numerical))
```

위와 같이 코드를 작성하여 격자 개수 별 error를 계산했다.

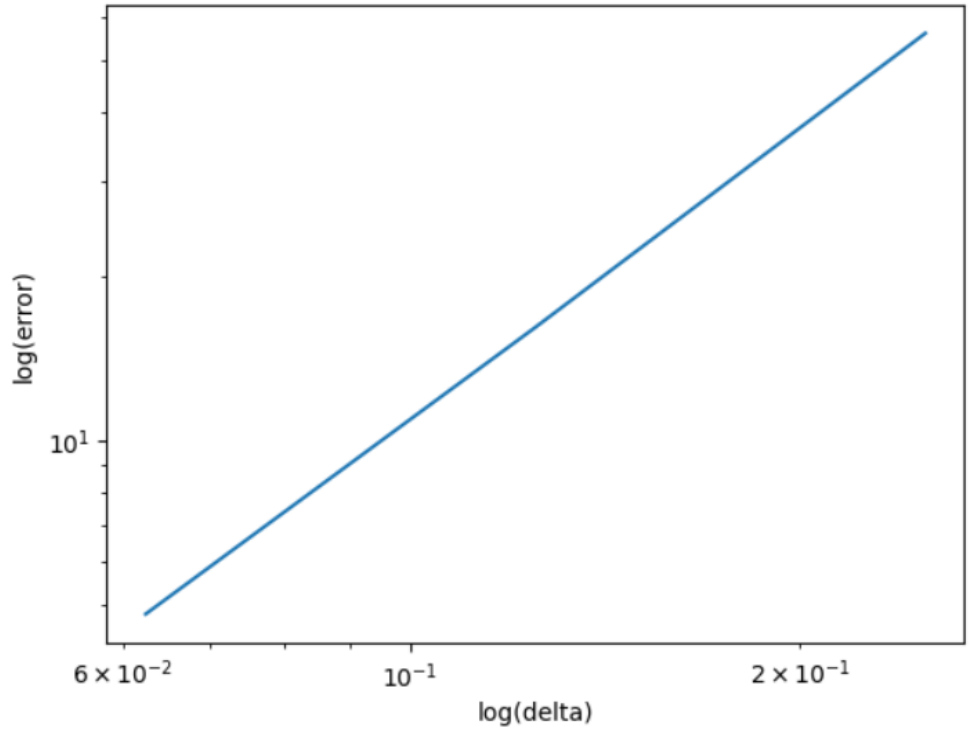
전체 코드는 다음과 같다.

```python
import numpy as np
import matplotlib.pyplot as plt

#주어진 함수와 이계도함수
def f(x):

  return np.sin((4-x)*(4+x))

def f_dd_exact(x):

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

#Second-order central difference scheme
def f_dd_numerical(x, h):
  N = len(x)
  f_dd = np.zeros_like(x)
  f_dd[0] = (2*f(x[0])-5*f(x[1])+4*f(x[2])-f(x[3]))/(h**2)
  f_dd[-1] = (2*f(x[-1])-5*f(x[-2])+4*f(x[-3])-f(x[-4]))/(h**2)
  for i in range(1, N-1):
    f_dd[i] = (f(x[i-1]) - 2*f(x[i]) + f(x[i+1])) /(h**2)

  return f_dd
```

```python
# error 계산 함수
def error(exact, numerical):

  return np.sqrt(np.mean((exact[2:-2] - numerical[2:-2])**2))

grid_numbers = [33, 65, 129]
errors = []
delta = []

# grid 생성
for i in grid_numbers:
  x = np.linspace(0, 8, i)
  h = x[1] - x[0]
  delta.append(h)

# error 계산
  exact = f_dd_exact(x)
  numerical = f_dd_numerical(x, h)
  errors.append(error(exact, numerical))

print(errors)

#시각화
plt.loglog(delta,errors)
plt.xlabel(r'log(delta)')
plt.ylabel(r'log(error)')
```

결과는 다음과 같이 log-log관계가 선형적으로 나왔다.

```
[np.float64(55.99024326147189), np.float64(16.149632313648144), np.float64(4.800814931915463)]
Text(0, 0.5, 'log(error)')
```



1) (Runge-Kutta Methods) Initial-value problem $x' = t + 2xt$ with $x(0) = 0$ 을 주어진 구간 [0,2]에서 Runge-Kutta 공식을 사용하여 계산하시오.

(1) Find x(t) using the second-order Runge-Kutta method with h=0.01

Second-order Runge-kutta method를 다음과 같은 코드로 구현할 수 있다.

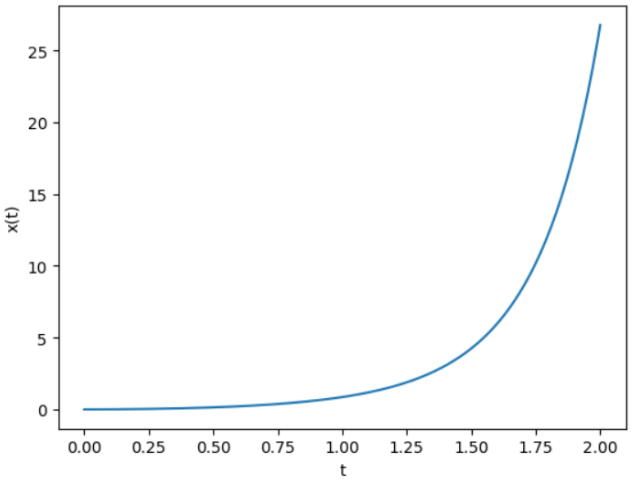
```
#RK2 알고리즘
for n in range(N-1):
  k1 = func(x[n], t[n])
  k2 = func(x[n] + h * k1 / 2, t[n] + h/2)
  x[n+1] = x[n] + h * k2
```

전체 코드는 다음과 같다

```
import numpy as np
import matplotlib.pyplot as plt

#미분 방정식
def func(x, t):

  return t + 2*x*t

# time nodes 생성
t_0 = 0
t_end = 2
h = 0.01
t = np.arange(t_0, t_end+h, h)
N = len(t)
```

```
#x 초기값
x = np.zeros(N)
x[0] = 0

#RK2 알고리즘
for n in range(N-1):
  k1 = func(x[n], t[n])
  k2 = func(x[n] + h * k1 / 2, t[n] + h/2)
  x[n+1] = x[n] + h * k2

#시각화
plt.plot(t, x)
plt.xlabel('t')
plt.ylabel('x(t)')
```

결과는 다음과 같다.

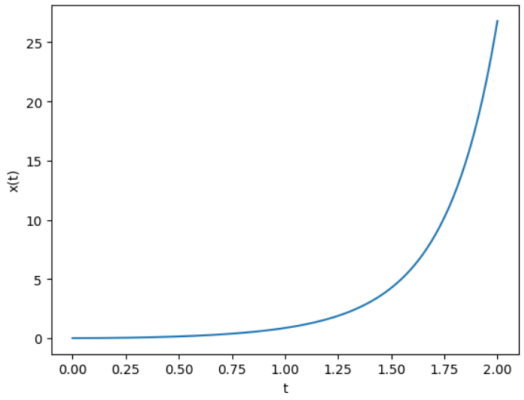


(2) Find x(t) using the fourth-order Runge-Kutta method with h=0.01

Fourth-order Runge-Kutta method는 다음과 같이 구현할 수 있다.

```
#RK4 알고리즘
for n in range(N-1):
  k1 = func(x[n], t[n])
  k2 = func(x[n] + h * k1 / 2, t[n] + h / 2)
  k3 = func(x[n] + h * k2 / 2, t[n] + h / 2)
  k4 = func(x[n] + h * k3, t[n] + h)
  x[n+1] = x[n] + h * (k1 + 2*k2 + 2*k3 + k4) / 6
```

나머지 코드는 (1)번 문제와 동일하고 결과는 다음과 같다.



(3) (1)과 (2)의 solution을 exact solution $\frac{1}{2}\left(e^{t^2}-1\right)$ 과 비교하여 각 RK method에 대한 order of accuracy를 분석하시오.

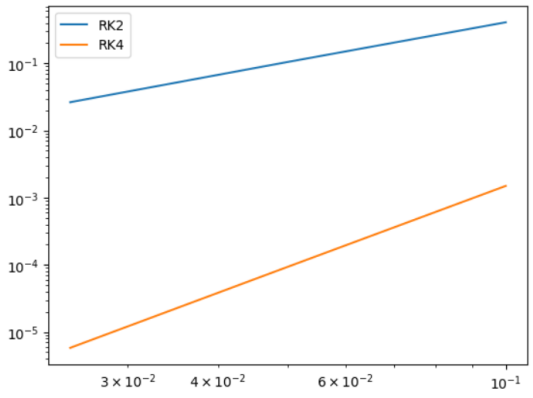
각 method에 대해 h를 0.1, 0.05, 0.025로 바꾸어가며 h와 norm2 error를 log-log 관계로 나타내어 order of accuracy를 분석했다. 전체 코드와 결과는 다음과 같다.

```
import numpy as np
import matplotlib.pyplot as plt

# 미분 방정식
def func(x, t):

  return t + 2*x*t

# exact solution
def x_exact(t):

  return 0.5*(np.exp(t**2)-1)

#RK2 알고리즘
def rk2(t):
  N = len(t)
  x = np.zeros(N)
  x[0] = x0
  for n in range(N-1):
    k1 = func(x[n], t[n])
    x_mid = x[n] + h * k1 / 2
    k2 = func(x_mid, t[n] + h/2)
    x[n+1] = x[n] + h*k2

  return x
```

```
#RK4 알고리즘
def rk4(t):
  N = len(t)
  x = np.zeros(N)
  x[0] = x0
  for n in range(N-1):
    k1 = func(x[n], t[n])
    k2 = func(x[n] + h * k1 / 2, t[n] + h / 2)
    k3 = func(x[n] + h * k2 / 2, t[n] + h / 2)
    k4 = func(x[n] + h * k3, t[n] + h)
    x[n+1] = x[n] + h * (k1 + 2*k2 + 2*k3 + k4) / 6

  return x

# error 함수
def error(exact, numerical):

  return np.sqrt(np.mean((exact - numerical)**2))

# 초기 조건
x0 = 0

# time nodes 생성
t_0 = 0
t_end = 2
h_lst = [0.1, 0.05, 0.025]
rk2_errors=[]
rk4_errors=[]
```

```
# error 계산
for h in h_lst:
  t = np.arange(t_0, t_end +h, h)
  rk2_errors.append(error(x_exact(t), rk2(t)))
  rk4_errors.append(error(x_exact(t), rk4(t)))

#시각화
plt.loglog(h_lst, rk2_errors, label = 'RK2')
plt.loglog(h_lst, rk4_errors, label = 'RK4')
plt.legend()

#order of accuracy
rk2 = np.polyfit(np.log(h_lst), np.log(rk2_errors), 1)[0]
rk4 = np.polyfit(np.log(h_lst), np.log(rk4_errors), 1)[0]

print(f"RK2의 order of accuracy: {rk2:.1f}")
print(f"RK4의 order of accuracy: {rk4:.1f}")
```

```
RK2의 order of accuracy: 2.0
RK4의 order of accuracy: 4.0
```



(4) step size h의 영향을 4th order RK method를 사용하여 분석하시오. (Hint : t=2에서의 error 분석을 진행하면 되며, 다른 step size h=0.01,0.05,0.1에 대한 분석을 진행하도록 하시오.

t=2일 때 error를 분석하는 코드를 작성하였다. 나머지는 전 문제와 같다.

```
# error 함수
def error(exact, numerical):

  return np.sqrt(np.mean((exact[-1] - numerical[-1])**2))

# time nodes 생성
t_0 = 0
t_end = 2
h_lst = [0.01, 0.05, 0.1]
rk2_errors = []

for h in h_lst:
  t = np.arange(t_0, t_end +h, h)
  rk2_errors.append(error(x_exact(t), rk4(t)))

for i in range(len(h_lst)):
  print(f"h = {h_lst[i]}: error at t=2: {rk2_errors[i]:.5e}")
```

```
h = 0.01: error at t=2: 7.55805e-07
h = 0.05: error at t=2: 4.23879e-04
h = 0.1: error at t=2: 5.92067e-03
```

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.

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

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

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

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

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

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

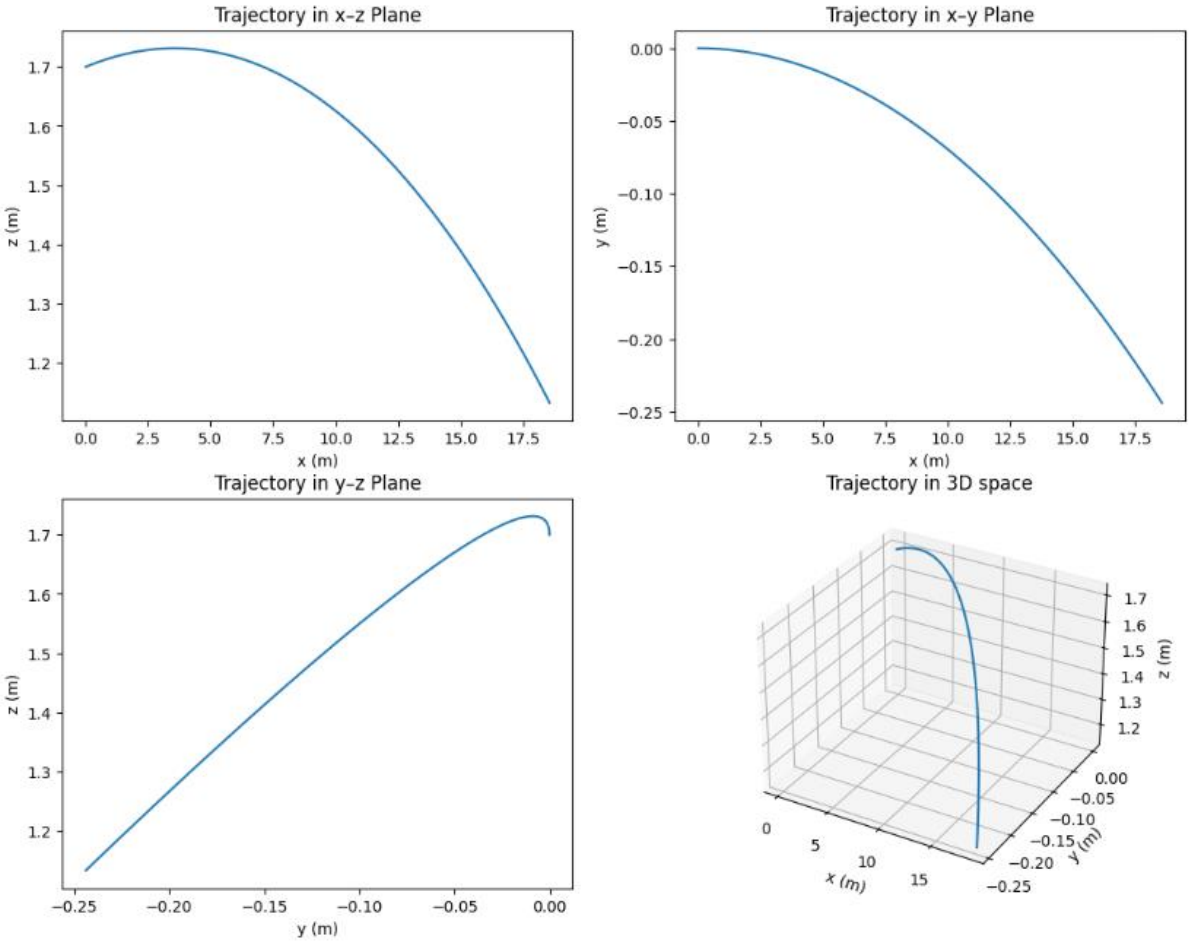
전체 코드는 rk4 method를 이용하여 다음과 같다.

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

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

# 미분 방정식
def states(Y):
    x, y, z, vx, vy, vz = Y
    V = np.sqrt(vx ** 2 + vy ** 2 + vz ** 2)
    dx = vx
    dy = vy
    dz = vz
    dvx = -F(V) * V * vx + B * w * (vz * np.sin(phi) - vy * np.cos(phi))
    dvy = -F(V) * V * vy + B * w * vx * np.cos(phi)
    dvz = -g - F(V) * V * vz - B * w * vx * np.sin(phi)
    return np.array([dx, dy, dz, dvx, dvy, dvz])

# RK4 알고리즘
def rk4(Y):
    y = np.array(Y).reshape(-1, 1)
    while y[0, -1] < 18.39:  # 종료 조건: 공이 포수에게 도달할 때까지
        k1 = states(y[:, -1])
        k2 = states(y[:, -1] + dt * k1 / 2)
        k3 = states(y[:, -1] + dt * k2 / 2)
        k4 = states(y[:, -1] + dt * k3)
        y_new = y[:, -1] + dt * (k1 + 2 * k2 + 2 * k3 + k4) / 6
        y = np.hstack((y, y_new.reshape(-1, 1)))
    return y
```

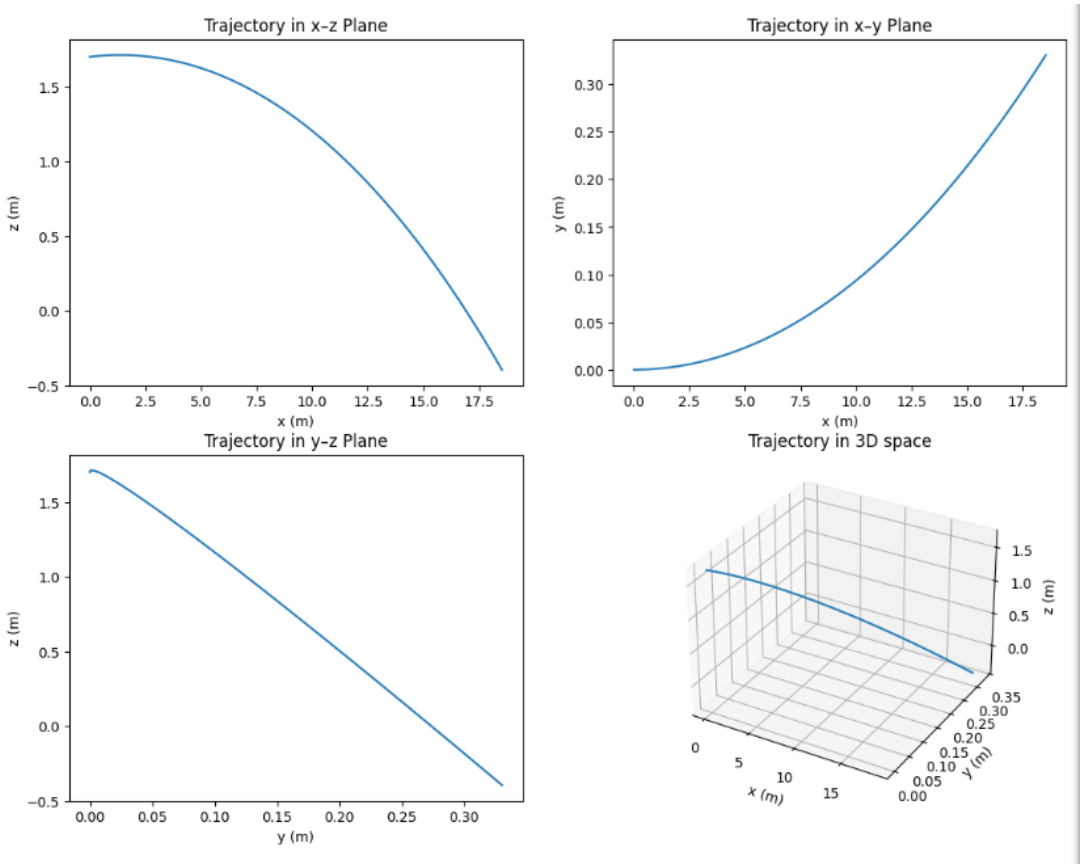
```
# 파라미터 설정
g = 9.81
B = 4.1e-4
w = 1800 * 2 * np.pi / 60
theta = np.radians(1)
h = 1.7
dt = 0.01
#fastball
v0 = 40
phi = np.radians(225)

# 초기 조건
y0 = [0, 0, h, v0 * np.cos(theta), 0, v0 * np.sin(theta)]

result = rk4(y0)
x = result[0]
y = result[1]
z = result[2]

fig = plt.figure(figsize = (13, 10))
#xz plane
plt.subplot(2, 2, 1)
plt.plot(x, z)
plt.xlabel('x (m)')
plt.ylabel('z (m)')
plt.title('Trajectory in x-z Plane')
```

```
#xy plane
plt.subplot(2, 2, 2)
plt.plot(x, y)
plt.xlabel('x (m)')
plt.ylabel('y (m)')
plt.title('Trajectory in x-y Plane')

#yz plane
plt.subplot(2, 2, 3)
plt.plot(y, z)
plt.xlabel('y (m)')
plt.ylabel('z (m)')
plt.title('Trajectory in y-z Plane')

#3D trajectory
ax = fig.add_subplot(2, 2, 4, projection = '3d')
ax.plot(x, y, z)
ax.set_xlabel('x (m)')
ax.set_ylabel('y (m)')
ax.set_zlabel('z (m)')
plt.title('Trajectory in 3D space')

plt.show()
```

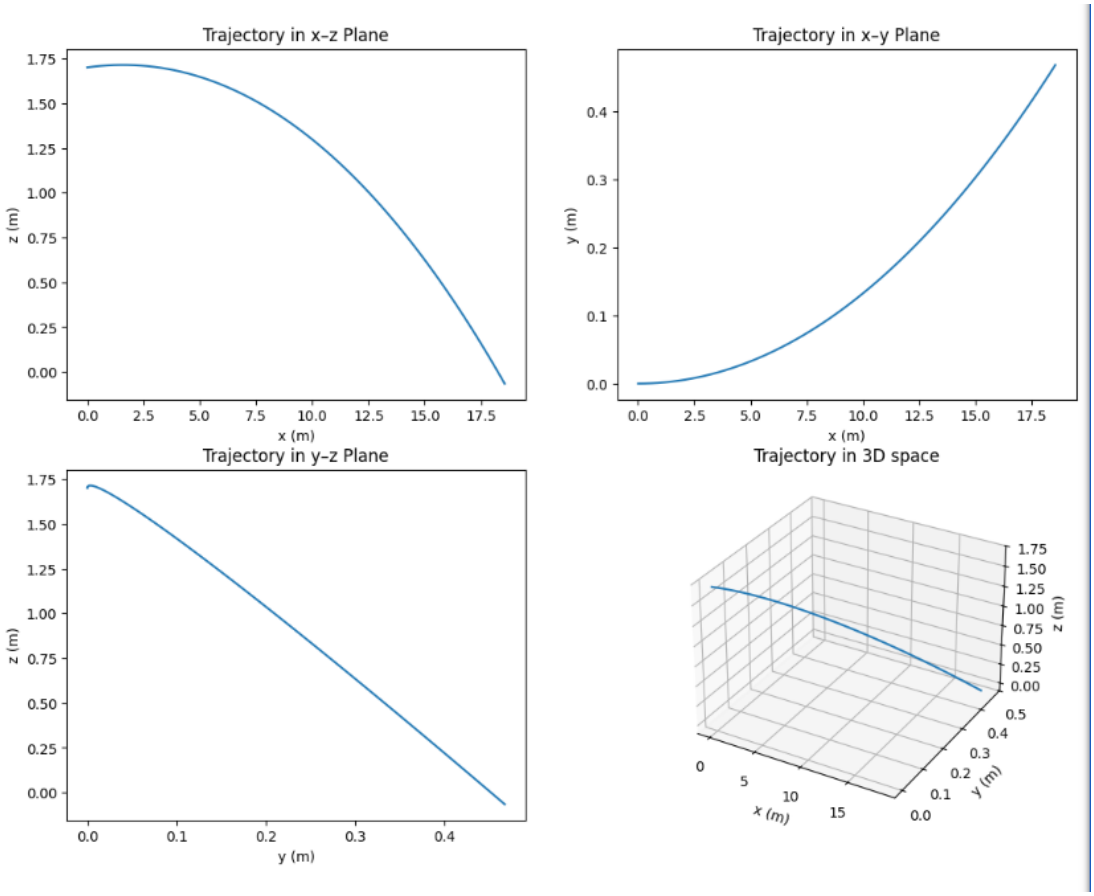
각 4가지 투구별로 파라미터를 다르게 설정하면 fastball, curveball, slider, screwball에 대해 궤적을 그릴 수 있고 결과는 다음과 같다.


Trajectory in x-z Plane
z (m)
x (m)
Trajectory in x-y Plane
y (m)
x (m)
Trajectory in y-z Plane
z (m)
y (m)
Trajectory in 3D space
x (m)
y (m)
z (m)

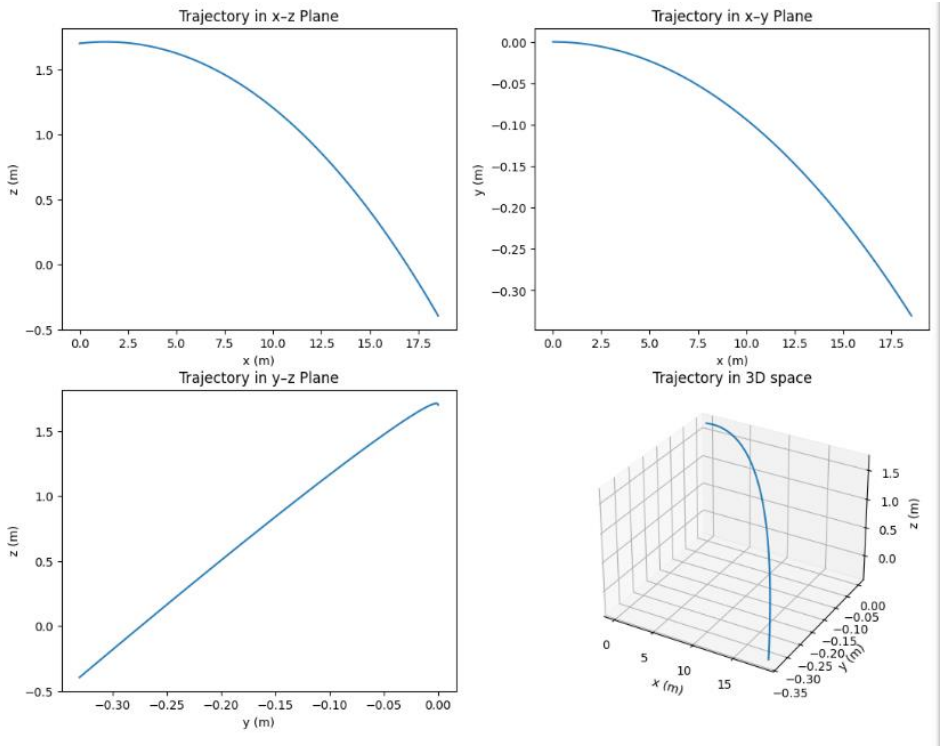
<fastball>


Trajectory in x–z Plane
z (m)
x (m)
Trajectory in x–y Plane
y (m)
x (m)
Trajectory in y–z Plane
z (m)
y (m)
Trajectory in 3D space
x (m)
y (m)
z (m)

<curveball>



<slider>



<screwball>