WEEK 2

2020136006 정성원

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)

Seond-order one-sided difference scheme

Taylor expansion

$$f(x+h) \simeq f(x) + f'(x)\cdot h + \frac{1}{2!}f''(x)\cdot h^2 + \frac{1}{3!}f'''(x)\cdot h^3 + \frac{1}{4!}f''''(x)\cdot h^4 + \frac{1}{5!}f'''''(x)\cdot h^5 + \cdots$$

$$f(x+2h) \simeq f(x) + 2f'(x)h + \frac{2^2}{2!}f''(x)\cdot h^2 + \frac{2^3}{3!}f'''(x)\cdot h^3 + \frac{2^4}{4!}f''''(x)\cdot h^4 + \frac{2^5}{5!}f'''''(x)\cdot h^5 + \cdots$$

$$4f(x+h) - f(x+2h) = 3f(x) + 2f'(x)\cdot h - \frac{2}{3}f'''(x)\cdot h^3 + \cdots$$

$$f'(x) = \frac{4f(x+h) - f(x+2h) - 3f(x)}{2h} \underbrace{- \frac{1}{3}f'''(x)\cdot h^2 + \cdots}_{O(h^2)}$$

$$f(x+3h) \simeq f(x) + 3f'(x)\cdot h + \frac{3^2}{2!}f''(x)\cdot h^2 + \frac{3^3}{3!}\cdot f'''(x)\cdot h^3 + \frac{3^4}{4!}f''''(x)h^4 + \frac{3^5}{5!}f'''''(x)\cdot h^5$$

$$Af(x) + Bf(x+h) + Cf(x+2h) + Df(x+3h) \simeq h^2 f''(x)$$

$$(A+B+C+D)f(x) + h\left(B+2C+3D\right)f'(x) + \frac{h^2}{2!}\left(B+4C+9D\right)f''(x)$$

$$+ \frac{h^3}{3!}\left(B+8C+8|D\right)f'''(x) \simeq h^2 f''(x)$$

$$\begin{aligned} A+B+C+D &= 0 \\ B+2C+3D &= 0 \\ \frac{1}{2}\left(B+4C+9D\right) &= 1 \\ \frac{1}{3!}\left(B+8C+8|D\right) &= 0 \end{aligned} \Rightarrow \begin{aligned} A &= 2 \\ B &= -5 \\ C &= 4 \\ D &= -1 \end{aligned}$$

$$\therefore f''(x) = \frac{2f(x) - 5f(x+h) + 4f(x+2h) - f(x+3h)}{h^2}$$

해당 방법은 작은 경계에서 사용하는 조건이고 큰 경계에서 사용하는 조건은 h대신 -h를 대입하면 된다.

```
f2_0 = (2*np.sin(16) - 5*np.sin(16-0.25**2) + 4*np.sin(16-0.5**2) - np.sin(16-0.75**2)) / (0.25**2)
print(f2_0)
```
✓ 0.0s
```
2.0248034695313226
```
```
f2_8 = (2*np.sin(16-64) - 5*np.sin(16-(8-0.25)**2) + 4*np.sin(16-(8-0.5)**2) - np.sin(16-(8-0.75)**2)) / (0.25**2)
print(f2_8)
```
✓ 0.0s
```
-19.16647580126187
```

해당 코드는 위에서 구한 second-order one-sided difference scheme 식에 f(x)=sin((4-x)(4+x))를 0과 8 에서의 f''을 구하는 과정이다. 격자점이 33개 이므로 h= 0.25이다.

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

```
def f(x):
    return np.sin(16-x**2)
```
✓ 0.0s

```
def f2(x):
    return -2*(2*(x**2)*np.sin(16-x**2)+np.cos(16-x**2))
```
✓ 0.0s

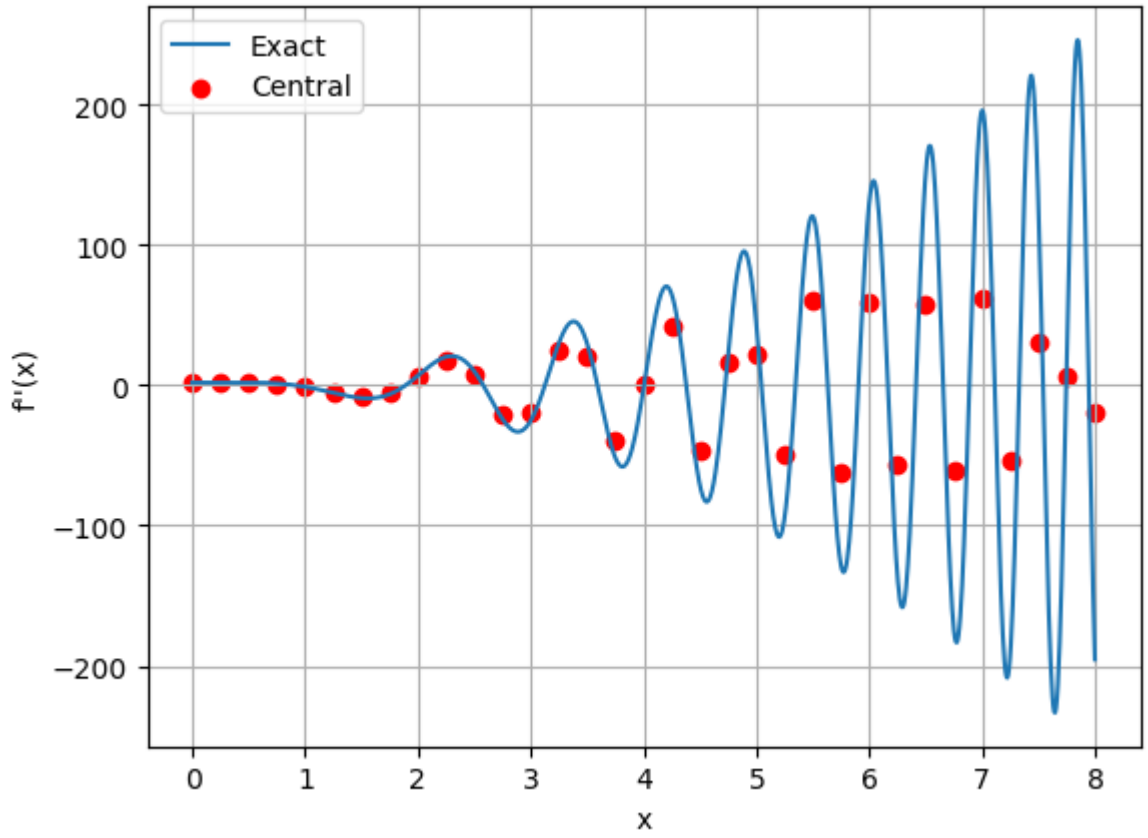
주어진 함수 f(x) = sin(16-x^2)이므로 def 함수를 이용하여 결과 값을 반환하는 함수를 만들었고 실제 exact solution인 -2(2x^2sin(16-x^2)+cos(16-x^2))를 def 함수를 이용하여 f2(x) 함수를 만들어 결과 값을 반환하는 함수를 만들었다.

```
h = 0.25
x = np.linspace(0, 8, 33)
x_real = np.linspace(0, 8, 1000)
f2_exact = f2(x_real)
f2_central = (f(x[2:])+f(x[:-2])-2*f(x[1:-1])) / (h**2)
f2_central_33 = np.insert(f2_central, 0, f2_0)
f2_central_33 = np.append(f2_central_33, f2_8)
```

h=0.25이고 해당 좌표를 만들기 위해 numpy의 linspace 함수를 사용하여 33개의 지점을 만들어 x에 저장하였다. 실제 exact solution을 보이기 위해 지점을 1000개로 쪼개 x_real 변수에 저장하였다. Exact solution은 f2함수를 이용하여 구한 뒤 f2_exact 변수에 저장하

였다. Central difference scheme은 $f''(x) = \frac{f(x+h)+f(x-h)-2f(x)}{h^2}$ 이므로 이를 이용하여 f2_central에 해당 값의 결과를 저장하였다. 문제 1에서 구한 f2_0과 f2_8을 각각 f2_central의 앞뒤에 넣기 위해 insert 함수와 append 함수를 사용하였다.

```
plt.plot(x_real, f2_exact, label='Exact')
plt.scatter(x, f2_central_33, label='Central', color='re
plt.legend()
plt.xlabel('x')
plt.ylabel('f\'\'(x)')
plt.grid()
```



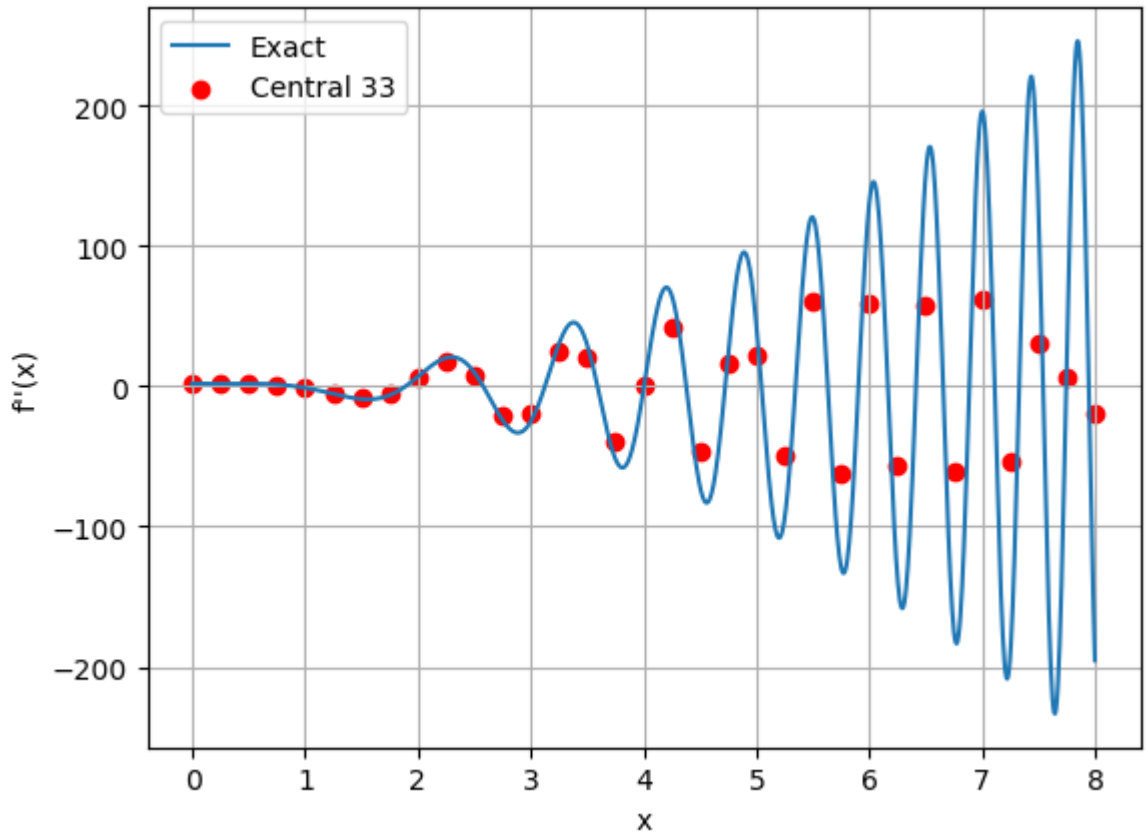
시각화 하면 위와 같다.

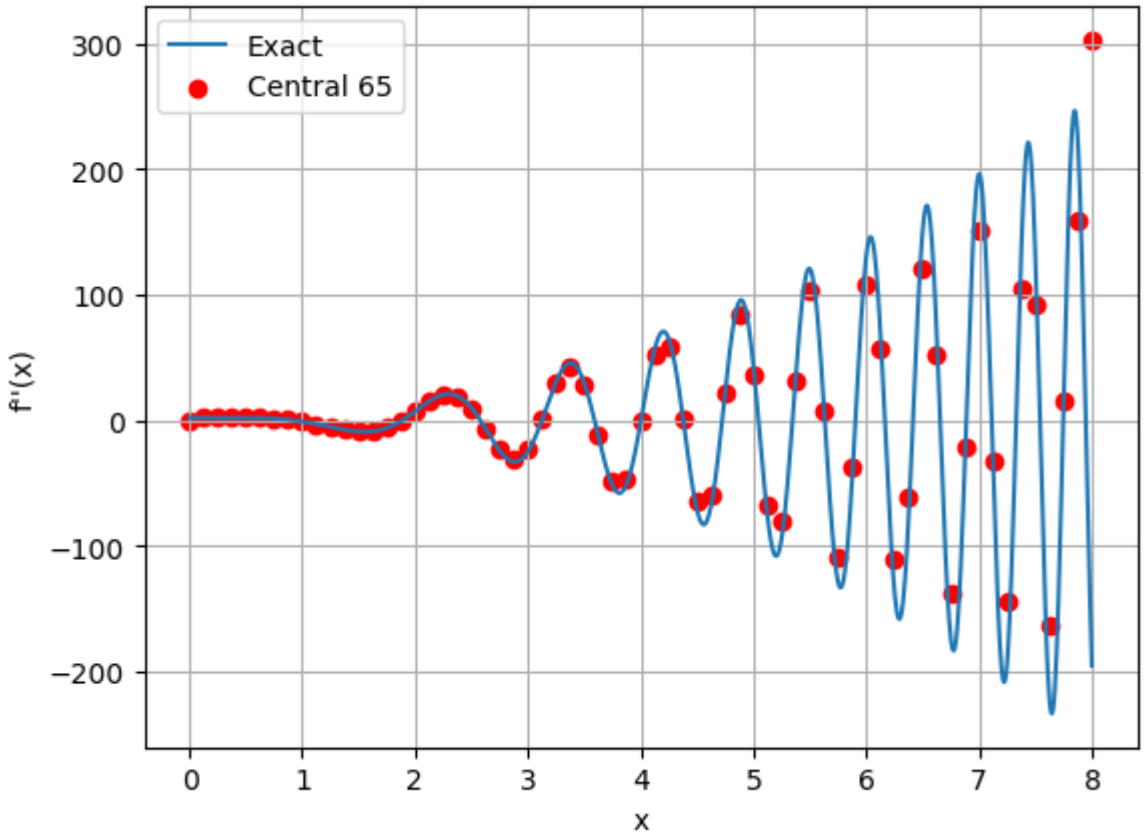
3. Second-order central difference scheme을 사용하여 격자 개수를 바꿔가며 (33,65,129) 정확도를 분석하시오. 정확도 분석은 x축을 $log(\Delta x)$, y축을 $log||e||$ 를 그리도록 한다. 본 방법에서 $||e||$ 는 L_2 $norm\ error$ 를 의미한다.

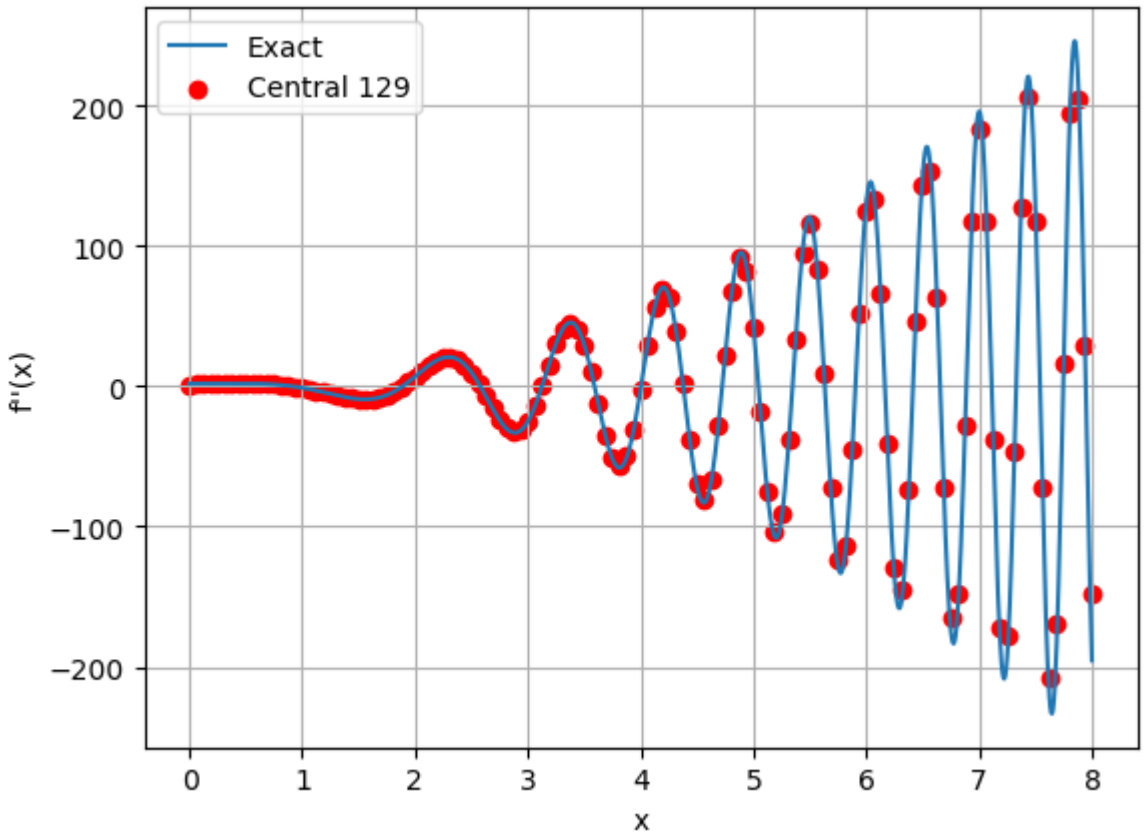
```
x_65 = np.linspace(0, 8, 65)
h_65 = x_65[1] - x_65[0]
f2_central_65 = (f(x_65[2:])+f(x_65[:-2])-2*f(x_65[1:-1])) / (h_65**2)
f2_0 = (2*np.sin(16) - 5*np.sin(16-h_65**2) + 4*np.sin(16-2*h_65**2) - np.sin(16-3*h_65**2)) / (h_65**2)
f2_8 = (2*np.sin(16-64) - 5*np.sin(16-(8-h_65)**2) + 4*np.sin(16-(8-2*h_65)**2) - np.sin(16-(8-3*h_65)**2)) / (h_65**2)
f2_central_65 = np.insert(f2_central_65, 0, f2_0)
f2_central_65 = np.append(f2_central_65, f2_8)


x_129 = np.linspace(0, 8, 129)
h_129 = x_129[1] - x_129[0]
f2_central_129 = (f(x_129[2:])+f(x_129[:-2])-2*f(x_129[1:-1])) / (h_129**2)
f2_0 = (2*np.sin(16) - 5*np.sin(16-h_129**2) + 4*np.sin(16-2*h_129**2) - np.sin(16-3*h_129**2)) / (h_129**2)
f2_8 = (2*np.sin(16-64) - 5*np.sin(16-(8-h_129)**2) + 4*np.sin(16-(8-2*h_129)**2) - np.sin(16-(8-3*h_129)**2)) / (h_129**2)
f2_central_129 = np.insert(f2_central_129, 0, f2_0)
f2_central_129 = np.append(f2_central_129, f2_8)
```

2 번 문제와 같은 방법으로 계산을 진행하여 격자 개수가 65, 129 개인 경우의 결과를 각각 f2_central_65 와 f2_central_129 에 저장하였다. 이를 확인해보기 위해 그림을 그렸다.




Exact
Central 65
300
200
100
0
−100
−200
f'(x)
0
1
2
3
4
5
6
7
8
x

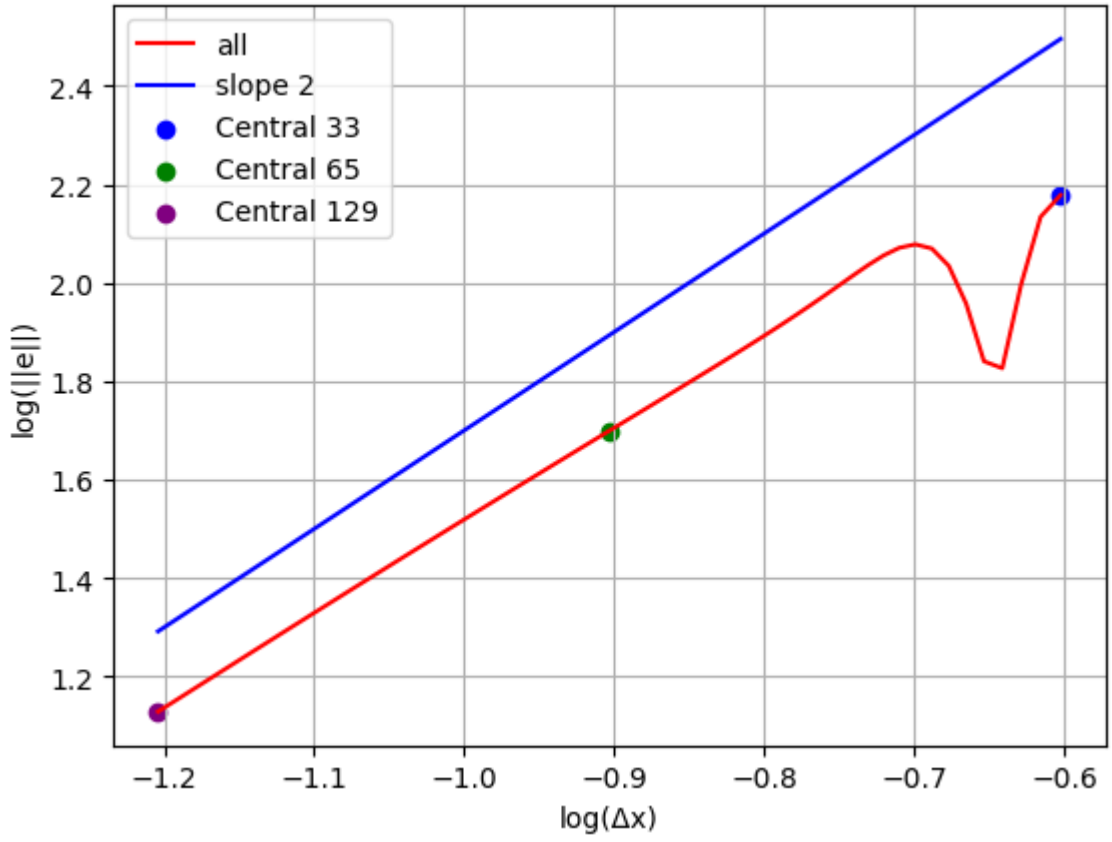

Exact
Central 129
200
100
0
−100
−200
f'(x)
0
1
2
3
4
5
6
7
8
x

```
error_33 = (np.sum((f2_central_33[1:-1] - f2(x)[1:-1])**2)*h)**0.5
error_65 = (np.sum((f2_central_65[1:-1] - f2(x_65)[1:-1])**2)*h_65)**0.5
error_129 = (np.sum((f2_central_129[1:-1] - f2(x_129)[1:-1])**2)*h_129)**0.5
```

$||e|| = (\Sigma(f_{exact} - f_{central})^2 \times h)^{\frac{1}{2}}$로 계산하여 각각 error_33, error_65, error_129 저장하였다.

```
error_list = []
h_list = []
for a in range(33, 130):
    x_mid = np.linspace(0, 8, a)
    h_mid = x_mid[1] - x_mid[0]
    f2_central_mid = (f(x_mid[2:])+f(x_mid[:-2])-2*f(x_mid[1:-1])) / (h_mid**2)
    error_mid = (np.sum((f2_central_mid - f2(x_mid)[1:-1])**2)*h_mid)**0.5
    error_list.append(error_mid)
    h_list.append(x_mid[1] - x_mid[0])
```

연속적으로 오류를 계산하여 그리기 위해 for 문을 사용하여 격자 점을 33 개에서 129 개까지로 나눴을 때의 error 를 계산하여 error list 에 저장하였다.



그림을 확인해보면 실제로 기울기가 2 인 그래프와 기울기를 비교했을 때 기울기가 비슷하다는 것을 확인할 수 있다.

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

```
h = 0.01
x_2 = np.zeros(201)
t = np.arange(0, 2 + h, h)
for i in range(200):
    k1 = f(t[i], x_2[i])
    k2 = f(t[i] + h, x_2[i] + h * k1)
    x_2[i + 1] = x_2[i] + (k1 + k2) * h / 2
```

h=0.01 이면 t 의 격자수는 201 개 이므로 결과를 저장할 x_2 array 를 만들었다. 또한, t 의 구간이 0 부터 2 이므로 np.arange 를 만들었다.

$$k_1 = f(t_i, x_i)$$

$$k_2 = f(t_i + h, x_i + h \times k_1)$$

$$x_{i+1} = x_i + (k_1 + k_2) \times \frac{h}{2}$$

위는 second-order Runge-Kutta method 이므로 이를 코드로 나타내었다.

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

```
h = 0.01
x_4 = np.zeros(201)
t = np.arange(0, 2 + h, h)
for i in range(200):
    k1 = f(t[i], x_4[i])
    k2 = f(t[i] + 1/2 * h, x_4[i] + 1/2 * h * k1)
    k3 = f(t[i] + 1/2 * h, x_4[i] + 1/2 * h * k2)
    k4 = f(t[i] + h, x_4[i] + h * k3)
    x_4[i + 1] = x_4[i] + (k1 + 2*k2 + 2*k3 + k4) * h / 6
```

1 과 같은 방법으로 제작하고 fourth order Runge-Kutta method 를 아래와 같은 식으로 제작하면 된다.

$$k_1 = f(t_i, x_i)$$

$$k_2 = f(t_i + \frac{1}{2}h, x_i + \frac{1}{2}h \times k_1)$$

$$k_3 = f(t_i + \frac{1}{2}h, x_i + \frac{1}{2}h \times k_2)$$

$$k_4 = f(t_i + h, x_i + h \times k_3)$$

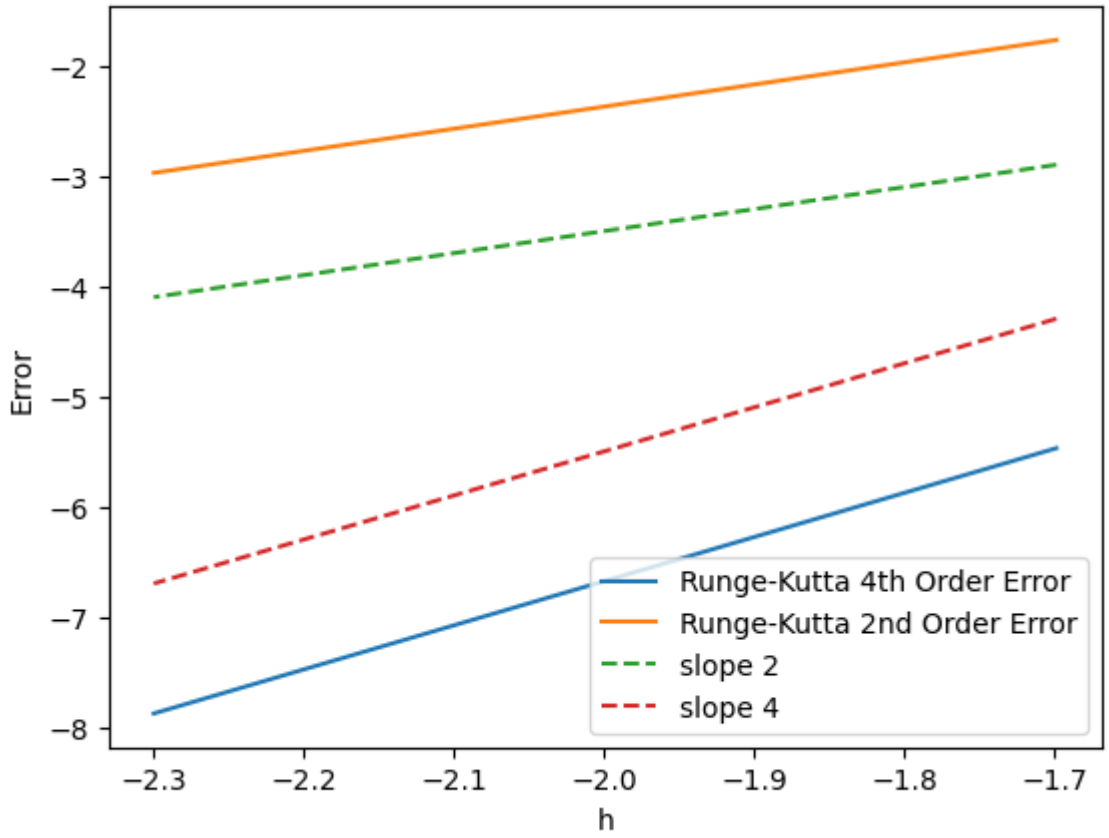
$$x_{i+1} = x_i + (k_1 + 2k_2 + 2k_3 + k_4) \times h/6$$

(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.

```
error_2_list = []
h_list = []
# a는 구간 개수
for a in range(100, 400):
    h = 2 / a
    x_2 = np.zeros(a + 1)
    t = np.linspace(0, 2, a + 1)
    for i in range(a):
        k1 = f(t[i], x_2[i])
        k2 = f(t[i] + h, x_2[i] + h * k1)
        x_2[i + 1] = x_2[i] + (k1 + k2) * h / 2
    error_2 = (np.sum(( x_2 - (np.exp(t**2)/2 - 1/2))**2)*h)**0.5
    error_2_list.append(error_2)
    h_list.append(h)
```

```
error_4_list = []
for a in range(100, 400):
    h = 2 / a
    x_4 = np.zeros(a + 1)
    t = np.linspace(0, 2, a + 1)
    for i in range(a):
        k1 = f(t[i], x_4[i])
        k2 = f(t[i] + 1/2 * h, x_4[i] + 1/2 * h * k1)
        k3 = f(t[i] + 1/2 * h, x_4[i] + 1/2 * h * k2)
        k4 = f(t[i] + h, x_4[i] + h * k3)
        x_4[i + 1] = x_4[i] + (k1 + 2*k2 + 2*k3 + k4) * h / 6
    error_4 = (np.sum(( x_4 - (np.exp(t**2)/2 - 1/2))**2)*h)**0.5
    error_4_list.append(error_4)
```

(1) 과 (2)에서 구한 식을 이용하여 구간의 개수를 for 문을 통해 여러 개로 만들어서 결과로 도출하였다. 과제 2 와 같은 방법으로 error 를 구하였다.



해당 결과를 확인해보면 second order Runge-Kutta method 와 fourth order Runge-Kutta method 의 기울기를 보면 각각 2 와 4 로 나타나기 때문에 각각 $\sim O(h^2)$와 $\sim O(h^4)$인 것을 알 수 있다.

(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.

```
error_list = []
for h in [0.01, 0.05, 0.1]:
    x_4 = np.zeros(int(2/h) + 1)
    t = np.arange(0, 2 + h, h)
    for i in range(len(t) - 1):
        k1 = f(t[i], x_4[i])
        k2 = f(t[i] + 1/2 * h, x_4[i] + 1/2 * h * k1)
        k3 = f(t[i] + 1/2 * h, x_4[i] + 1/2 * h * k2)
        k4 = f(t[i] + h, x_4[i] + h * k3)
        x_4[i + 1] = x_4[i] + (k1 + 2*k2 + 2*k3 + k4) * h / 6
    error = abs(x_4[-1] - (np.exp(t[-1]**2)/2 - 1/2))
    print(f"Error for h={h}: {error}")
    error_list.append(error)

error_list = np.array(error_list)
h_list = np.array([0.01, 0.05, 0.1])
print(error_list[1:]/error_list[:-1])
print(h_list[1:]**4/h_list[:-1]**4)
```

여기서는 error 를 exact solution 과 RK4 의 차이의 절댓값이다. h 에 따라서 해당 결과를 도출을 여 error_list 에 넣었다. Error 의 비율을 계산하면 h^4 의 비율과 같다는 결과를 도출할 수 있다. 이를 보면 error 는 $\sim O(h^4)$인 것을 알 수 있다.

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.

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

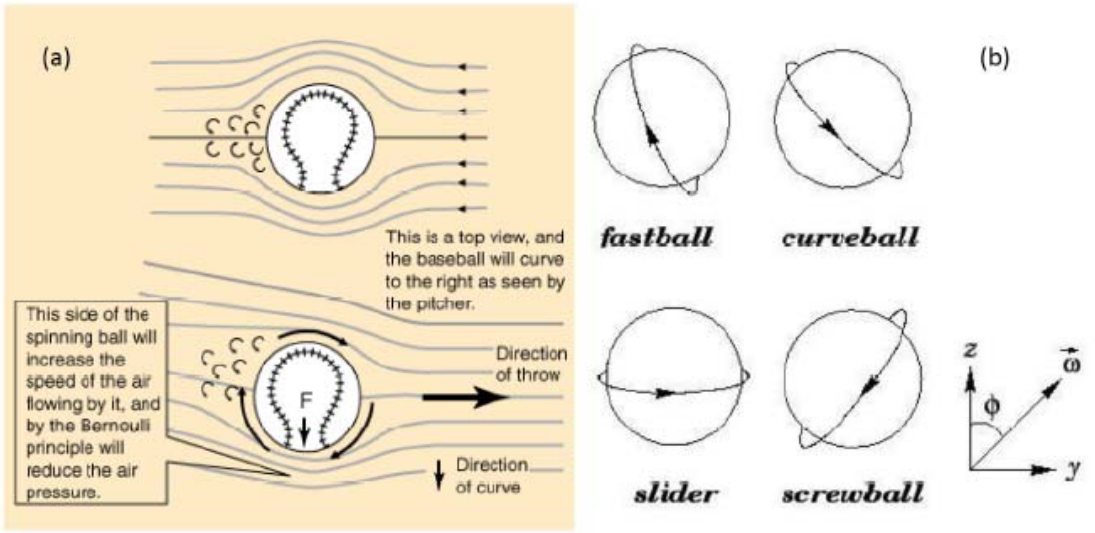


Figure 1: (a) Physics of baseball and (b) rotation direction for four pitches

(1) 4th order RK method를 사용하여 위 6개 방정식을 계산하는 코드를 만드시오. t=0일 때의 초기조건은 $x(0) = 0, y(0) = 0, z(0) = h, v_x = v_0 cos\theta, v_y = 0, v_z = v_0 sin\theta$ 로 주어지며, v_0는 투구의 초기속도를 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.

```
B = 4.1 * 10**-4
W = 1800 * 2*np.pi / 60  # Convert RPM to rad/s
g = 9.81  # Acceleration due to gravity
```
[10] ✓ 0.0s

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

def u_t(u,v,w,pi):
    return -F((u**2+v**2+w**2)**0.5) * (u**2+v**2+w**2)**0.5 * u + B * W *(w*np.sin(pi) - v*np.cos(pi))

def v_t(u,v,w,pi):
    return -F((u**2+v**2+w**2)**0.5) * (u**2+v**2+w**2)**0.5 * v + B * W *u*np.cos(pi)

def w_t(u,v,w,pi):
    return -g -F((u**2+v**2+w**2)**0.5) * (u**2+v**2+w**2)**0.5 * w - B * W *u*np.sin(pi)
```
[11] ✓ 0.0s

각자의 변수를 정의하고 주어진 함수들을 F, u_t, v_t, w_t 에 정의하였다.

```
# 속도 RK
def RK4_u(u,v,w,pi,t_h):
    u_k1 = u_t(u,v,w,pi)
    v_k1 = v_t(u,v,w,pi)
    w_k1 = w_t(u,v,w,pi)

    u_k2 = u_t(u+t_h/2*u_k1,v+t_h/2*v_k1,w+t_h/2*w_k1,pi)
    v_k2 = v_t(u+t_h/2*u_k1,v+t_h/2*v_k1,w+t_h/2*w_k1,pi)
    w_k2 = w_t(u+t_h/2*u_k1,v+t_h/2*v_k1,w+t_h/2*w_k1,pi)

    u_k3 = u_t(u+t_h/2*u_k2,v+t_h/2*v_k2,w+t_h/2*w_k2,pi)
    v_k3 = v_t(u+t_h/2*u_k2,v+t_h/2*v_k2,w+t_h/2*w_k2,pi)
    w_k3 = w_t(u+t_h/2*u_k2,v+t_h/2*v_k2,w+t_h/2*w_k2,pi)

    u_k4 = u_t(u+t_h/2*u_k3,v+t_h/2*v_k3,w+t_h/2*w_k3,pi)
    v_k4 = v_t(u+t_h/2*u_k3,v+t_h/2*v_k3,w+t_h/2*w_k3,pi)
    w_k4 = w_t(u+t_h/2*u_k3,v+t_h/2*v_k3,w+t_h/2*w_k3,pi)
```

RK4_u 함수는 RK4 를 이용하여 해당함수를 사용하면 u,v,w 를 반환한다.

```
def RK4_x(x,y,z,u,v,w,pi,t_h):
    x_k1 = u
    y_k1 = v
    z_k1 = w

    x_k2 = RK4_u(u,v,w,pi,t_h/2)[0]
    y_k2 = RK4_u(u,v,w,pi,t_h/2)[1]
    z_k2 = RK4_u(u,v,w,pi,t_h/2)[2]

    x_k3 = x_k2
    y_k3 = y_k2
    z_k3 = z_k2

    x_k4 = RK4_u(u,v,w,pi,t_h)[0]
    y_k4 = RK4_u(u,v,w,pi,t_h)[1]
    z_k4 = RK4_u(u,v,w,pi,t_h)[2]
```

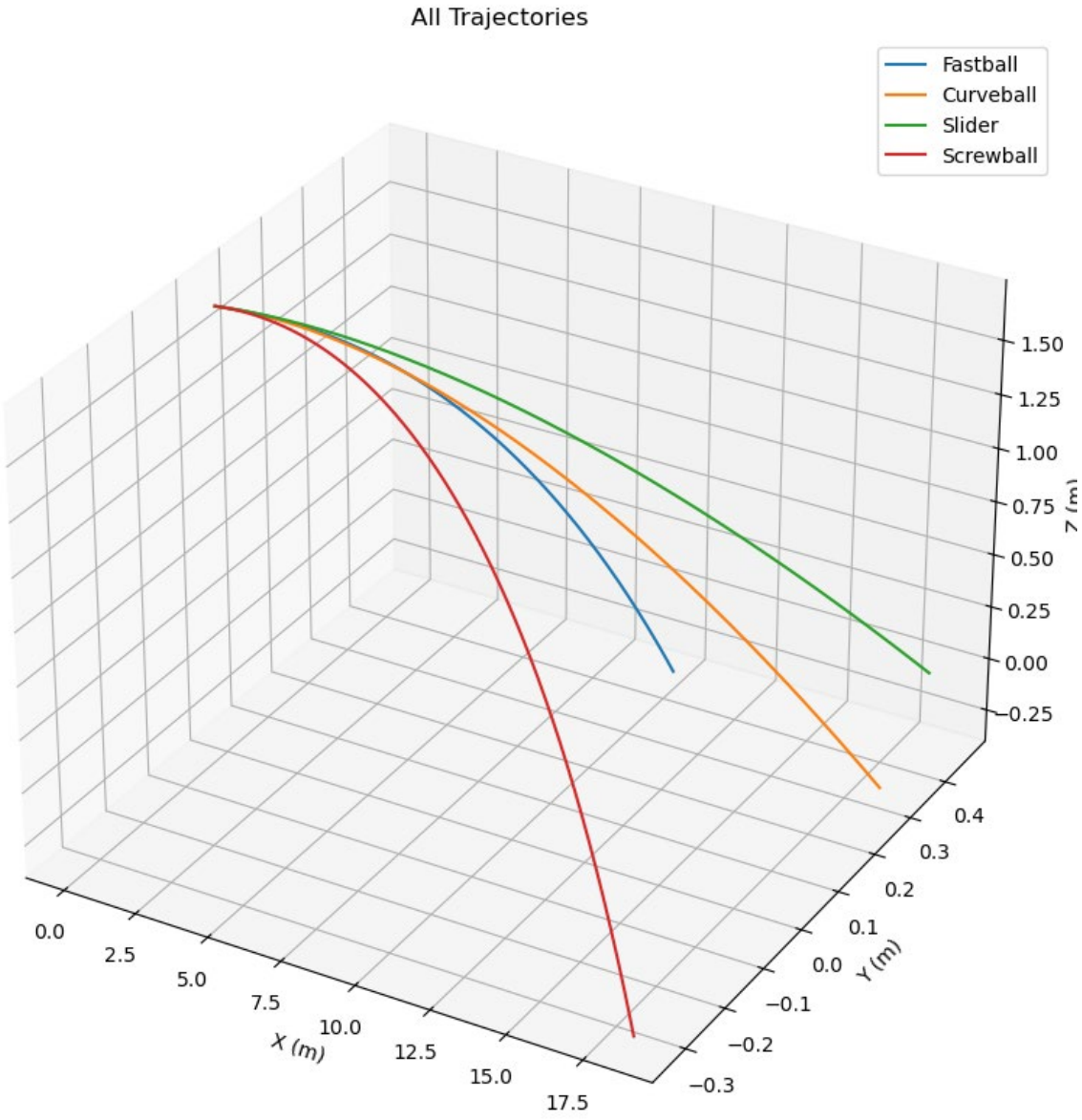
RK4_x 함수는 RK4 를 이용하여 해당함수를 사용하면 x,y,z 를 반환한다.

k2 와 k3 를 구할 때 같은 시간 step 후에 구하기 때문에 같은 같을 사용하여도 상관없기 때문에 같은 값을 사용하였다.

(2) 방정식을 Figure 1(b)에서 보인 4가지 투구에 대해서 계산하고, 야구공의 trajectories를 각 투구에 대하여 plot 하시오. 상승각도 $\theta = 1^{\circ}$ 이며, fastball은 $\mathbf{v_0} = 40m/s$, 나머지는 $\mathbf{v_0} = 30m/s$ 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.

```
# Fastball
x_fast = [0]
y_fast = [0]
z_fast = [1.7]
u_fast = [40*np.cos(np.deg2rad(1))]
v_fast = [0]
w_fast = [40*np.sin(np.deg2rad(1))]
while x_fast[len(x_fast)-1] < 18.39:
    x_m,y_m,z_m,u_m,v_m,w_m=RK4_x(x_fast[len(x_fast)-1], y_fast[len(y_fast)-1],
    x_fast.append(x_m)
    y_fast.append(y_m)
    z_fast.append(z_m)
    u_fast.append(u_m)
    v_fast.append(v_m)
    w_fast.append(w_m)
```

문제에서 주어진 상승각도, v0, 회전 방향 위치 등을 넣고 while 문을 사용하여 x 좌표가 18.39 보다 작을 때만 while 문이 구동되도록 하였다. 모든 구종에 대해 값을 넣으면 아래와 같은 그래프와 같은 결과가 나온다.


All Trajectories
Fastball
Curveball
Slider
Screwball
X (m)
Y (m)
Z (m)
0.0
2.5
5.0
7.5
10.0
12.5
15.0
17.5
−0.3
−0.2
−0.1
0.0
0.1
0.2
0.3
0.4
−0.25
0.00
0.25
0.50
0.75
1.00
1.25
1.50