Lecture 8, 9

# ODE and Programming

**Jung-Il Choi**

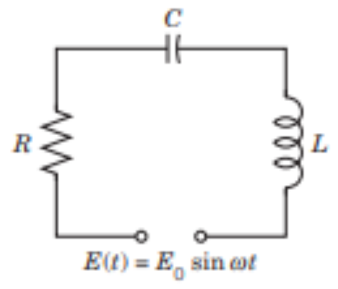
School of Mathematics and Computing (Computational Science and Engineering)

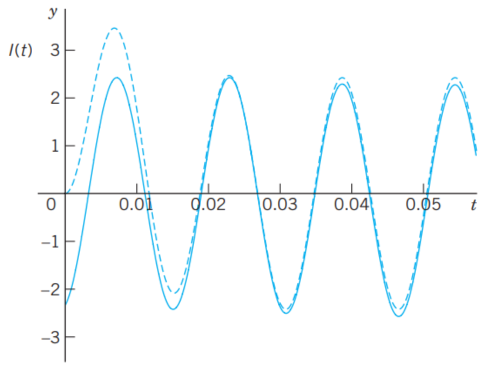
YONSEI UNIVERSITY

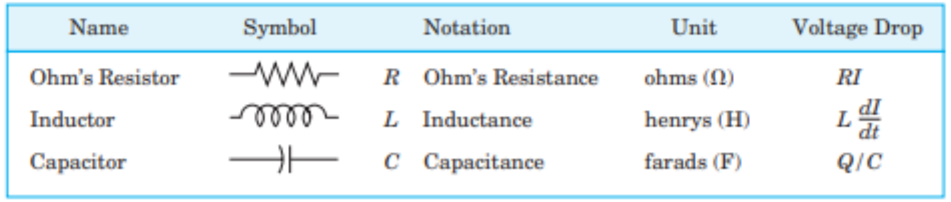
# Initial Value Problem (IVP)

- **Initial Value Problem**
  - Finding a solution to an ODE given an initial value is called **initial value problem**.
    - ➔ Ex) RLC circuit model

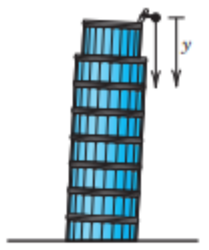
$$LI'' + RI' + \frac{1}{C}I = E'(t), \qquad I(0) = I_0, Q(0) = Q_0$$







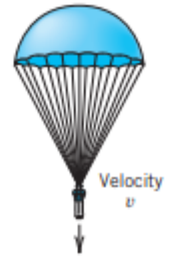
| Name | Symbol | Notation | | Unit | Voltage Drop |
|---|---|---|---|---|---|
| Ohm's Resistor | –WW– | $R$ | Ohm's Resistance | ohms (Ω) | $RI$ |
| Inductor | –ꝏ– | $L$ | Inductance | henrys (H) | $L\frac{dI}{dt}$ |
| Capacitor | –‖– | $C$ | Capacitance | farads (F) | $Q/C$ |



Falling stone

$y'' = g = const.$

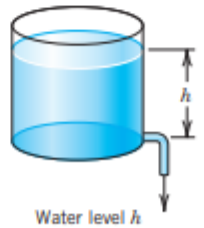
(Sec. 1.1)



Parachutist

$mv' = mg - bv^2$

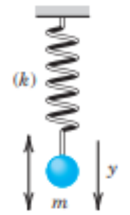
(Sec. 1.2)



Outflowing water

$h' = -k\sqrt{h}$

(Sec. 1.3)

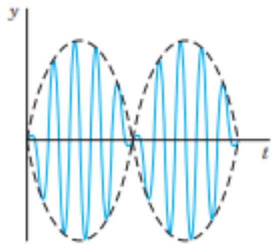


Displacement $y$

Vibrating mass on a spring

$my'' + ky = 0$

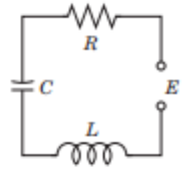
(Secs. 2.4, 2.8)



Beats of a vibrating system

$y'' + \omega_0^2 y = \cos\ \omega t, \quad \omega_0 \approx \omega$

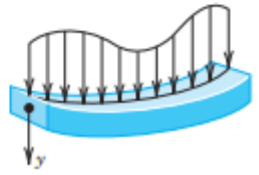
(Sec. 2.8)



Current $I$ in an $RLC$ circuit

$LI'' + RI' + \frac{1}{C}I = E'$

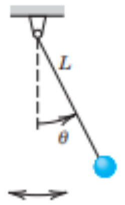
(Sec. 2.9)



Deformation of a beam

$EIy^{iv} = f(x)$

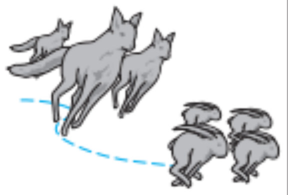
(Sec. 3.3)



Pendulum

$L\theta'' + g \sin\theta = 0$

(Sec. 4.5)



Lotka–Volterra predator–prey model

$y_1' = ay_1 - by_1y_2$

$y_2' = ky_1y_2 - ly_2$

(Sec. 4.5)

# Explicit or Forward Euler

- **Explicit Euler method**
  - Consider the first-order ODE

$$\frac{dy}{dt} = f(y,t) \qquad y(0) = y_0. \qquad 0 < t \le t_f.$$

  - From Taylor-series method

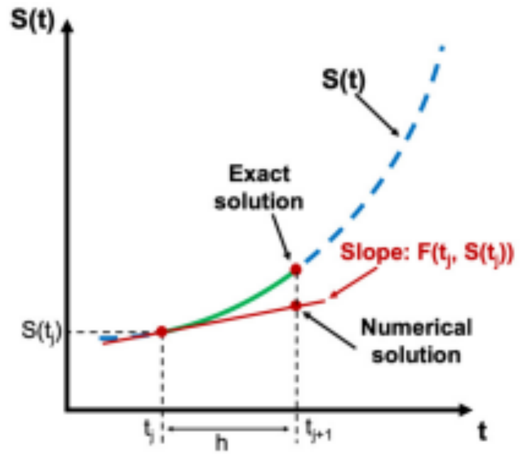
$$\boxed{y_{n+1} = y_n + h y'_n} + \frac{h^2}{2} y''_n + \frac{h^3}{6} y'''_n + \cdots \qquad h = \Delta t$$

$$y'_n = f(y_n, t_n)$$

  - Explicit Euler method; first order accurate;

$$y_{n+1} = y_n + h f(y_n, t_n).$$

y’으로 구할 수는 있겠지만, 오차가 더 생김
descrete point의 함숫값을 추정하는 문제로 바뀜



# Explicit or Forward Euler

- **sample code**

$$y' + 0.5y = 0$$

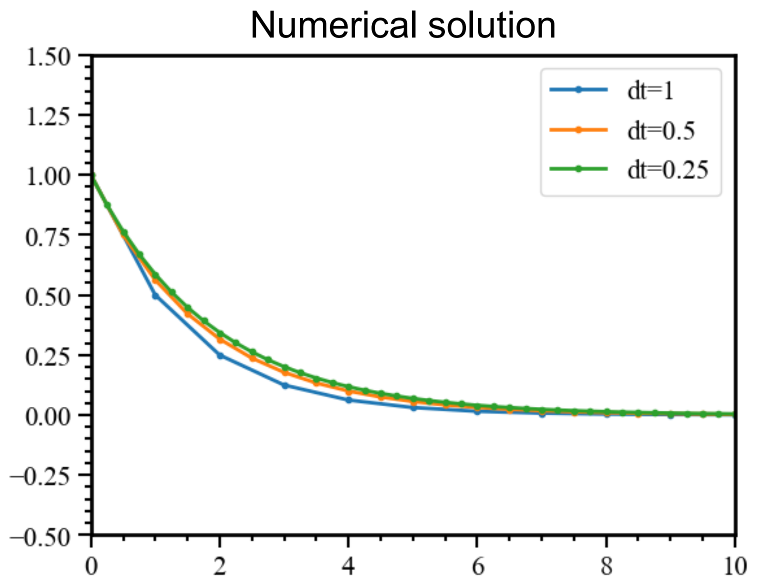
$$y(0) = 1 \qquad 0 \le t \le 10$$

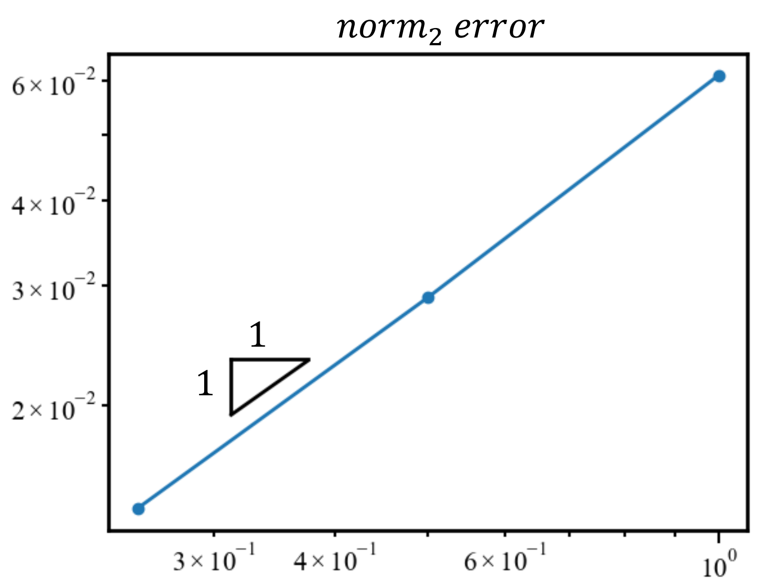
$$\xrightarrow[\text{Explicit Euler}]{} \qquad y_{n+1} = y_n - 0.5hy_n$$

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

```python
def Euler(t0, y0, tmax, dt):
    n = int((tmax-t0)/dt) + 1

    t = np.arange(t0, tmax+dt, dt)
    y = np.zeros(n)

    y[0] = y0

    for i in range(n-1):
        y[i+1] = y[i]*(1-0.5*dt)

    return t,y
```

```python
t1,y1 = Euler(t0=0, y0=1, tmax=10, dt=1)
t2,y2 = Euler(t0=0, y0=1, tmax=10, dt=0.5)
t3,y3 = Euler(t0=0, y0=1, tmax=10, dt=0.25)
```





## Numerical Stability

- **Stable numerical scheme**
  - Numerical solution does not grow unbounded(blow up) with any choice of parameters such as the step size.
  - We will have to see what the cost is for such robustness.

- **Unstable numerical scheme**
  - Numerical solution blows up with any choice of parameters.
  - Clearly, no matter how accurate they may be, such numerical schemes would not be useful.

- **Conditionally stable numerical scheme**
  - With certain choices of parameters the numerical solution remains bounded.
  - Hopefully, the cost of the calculation does not become prohibitively large.

## Stability Analysis

원래 문제가 unstable한 것으로 stable한 numerical scheme은 없어
우리가 하는건, stable한 것으로 stable하게 만들려고 함

- **Stability Analysis**
  - Consider two-dimensional Taylor series expansion

$$\boxed{\begin{aligned} f(y,t) &= f(y_0,t_0) + (t-t_0)\frac{\partial f}{\partial t}(y_0,t_0) + (y-y_0)\frac{\partial f}{\partial y}(y_0,t_0) \\ &+ \frac{1}{2!}\left[(t-t_0)^2\frac{\partial^2 f}{\partial t^2} + 2(t-t_0)(y-y_0)\frac{\partial^2 f}{\partial t\partial y} + (y-y_0)^2\frac{\partial^2 f}{\partial y^2}\right] + \cdots.\end{aligned}}$$

  - Collecting only the linear terms

$$y' = \lambda y + \alpha_1 + \alpha_2 t + \cdots$$

After discarding non-linear terms(high-order terms)

  - For example,

$$\lambda = \frac{\partial f}{\partial y}(y_0, t_0).$$

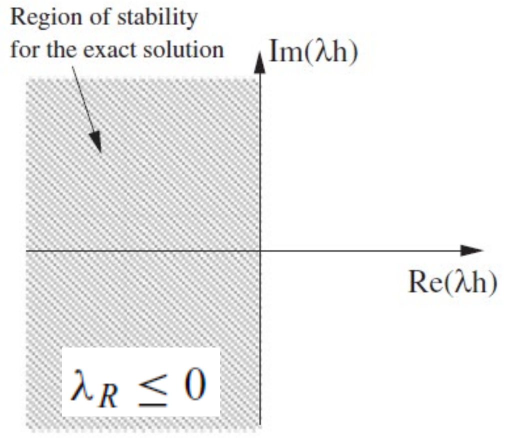
  - Model problem,

$$y' = \lambda y.$$

$$\lambda = \lambda_R + i\lambda_I$$

Stable conditions

the real part $\lambda_R \le 0$



# Stability Analysis

- **From Euler method**

$$y_{n+1} = y_n + h\underline{f(y_n, t_n)},$$

- **Applying the model problem,**

$$y' = \lambda y$$

$$y_{n+1} = y_n + \lambda h y_n = y_n(1 + \lambda h).$$

$$\underline{y_n = y_0(1 + \lambda h)^n.}$$

For complex $\lambda$, $$y_n = y_0(1 + \lambda_R h + i\lambda_I h)^n = y_0\sigma^n$$

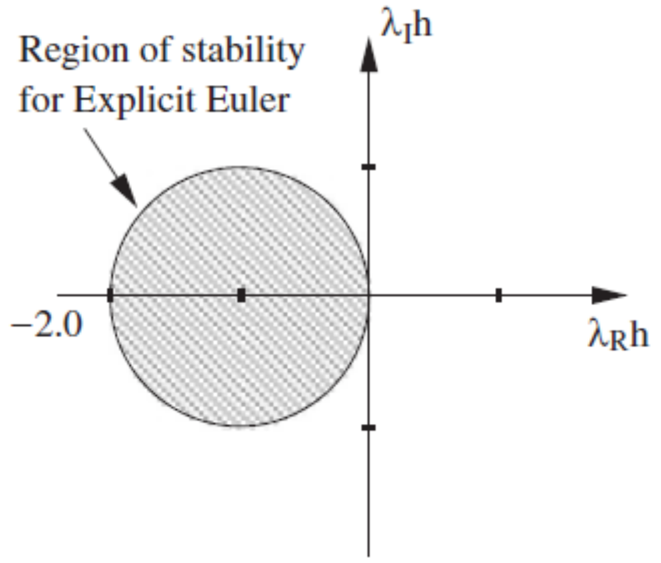
Amplification factor $\longrightarrow$ $$\underline{\sigma = (1 + \lambda_R h + i\lambda_I h)}$$

Stability conditions; $$\underline{|\sigma| \le 1.}$$

$$|\sigma|^2 = (1 + \lambda_R h)^2 + \lambda_I^2 h^2 = 1$$

If $\lambda$ is *real* negative

$$h \le \frac{2}{|\lambda|}$$

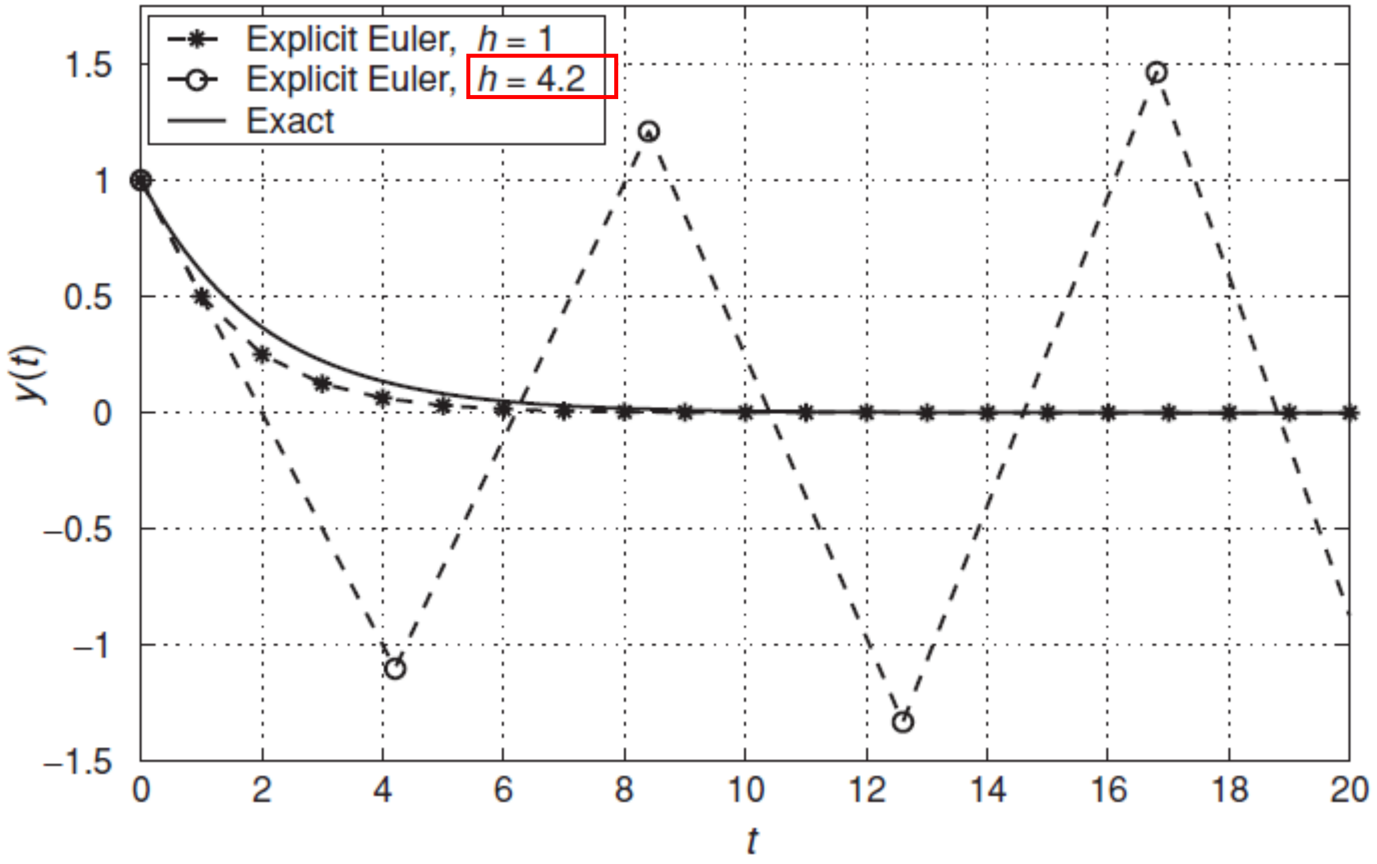


# Stability Analysis

- **Example**

$$y' + 0.5y = 0$$
$$y(0) = 1 \qquad 0 \le t \le 20.$$

$$\xrightarrow[\text{Explicit Euler}]{} \qquad y_{n+1} = y_n - 0.5hy_n$$



# Implicit or Backward Euler

- **Explicit (Forward) formula**

$$y_{n+1} = y_n + hf(y_n, t_n),$$

- **Implicit (Backward) formula**

$$y_{n+1} = y_n + hf(y_{n+1}, t_{n+1}).$$

root finding 해서, y(n+1) 만들면 됨.
(ex, newton method)
매우 iterative함. 좋은 method지만 너무 복잡해
자연계 system은 언젠간 0로 가게 되어있음.
Forward로 풀어도 되나, linear일때만 stable함

- **Applying the model,** $y' = \lambda y$

$$y_{n+1} = y_n + \lambda h y_{n+1}.$$

$$y_{n+1} = (1-\lambda h)^{-1} y_n \qquad y_n = \sigma^n y_0,$$

$$\sigma = \frac{1}{1-\lambda h}. \quad \sigma = \frac{1}{(1-\lambda_R h) - i\lambda_I h} = \frac{1}{Ae^{i\theta}}, \quad \left(A = \sqrt{(1-\lambda_R h)^2 + \lambda_I^2 h^2}, \qquad \theta = -\tan^{-1}\frac{\lambda_I h}{1-\lambda_R h}.\right)$$

➔ For stability, $|\sigma| = \frac{|e^{-i\theta}|}{A} = \frac{1}{A} \le 1.$ This is always true because $\lambda_R$ is negative and hence $A > 1$.

- Backward Euler scheme is unconditionally stable!

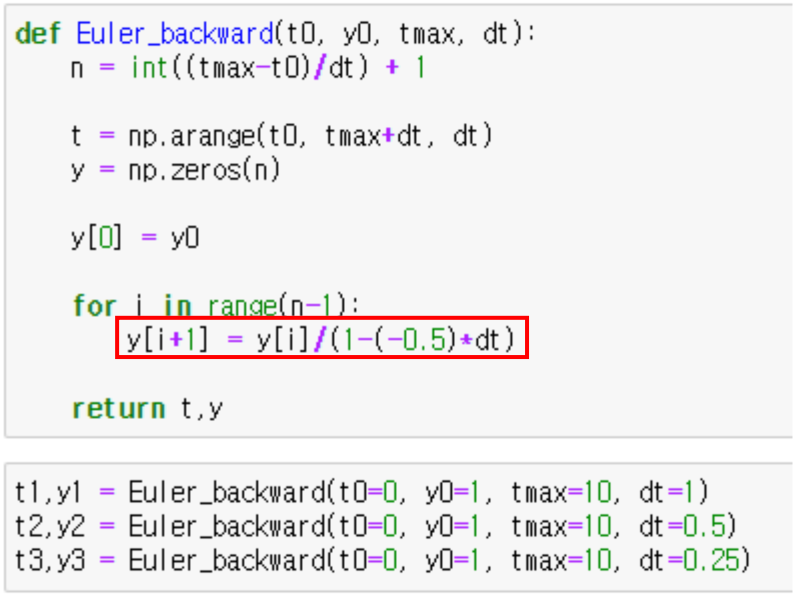
## Implicit or Backward Euler

- **Sample code**

$$y' + 0.5y = 0$$
$$y(0) = 1 \qquad 0 \le t \le 10$$

$$\xrightarrow[\text{Implicit Euler}]{}$$

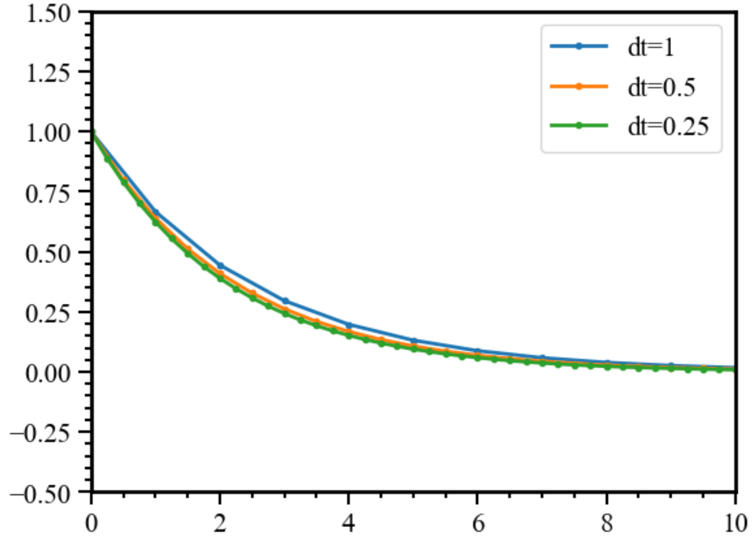
$$y_{n+1} = \frac{y_n}{(1 + 0.5h)}$$



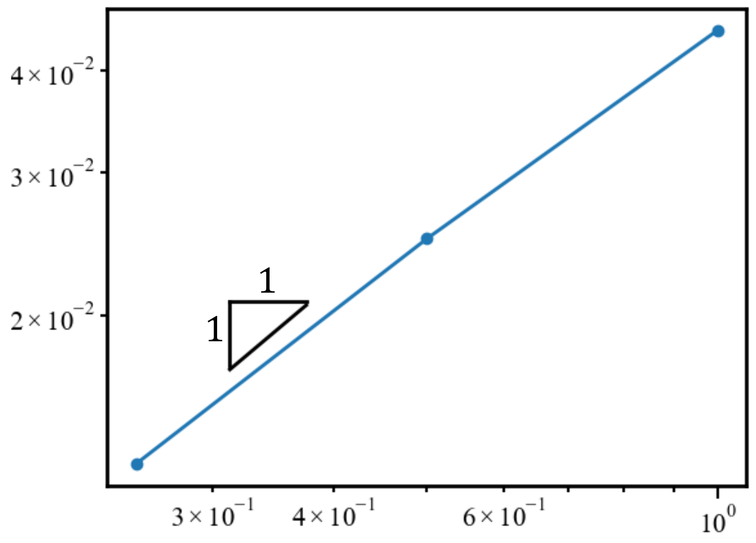
```
def Euler_backward(t0, y0, tmax, dt):
    n = int((tmax-t0)/dt) + 1

    t = np.arange(t0, tmax+dt, dt)
    y = np.zeros(n)

    y[0] = y0

    for i in range(n-1):
        y[i+1] = y[i]/(1-(-0.5)*dt)

    return t,y
```

```
t1,y1 = Euler_backward(t0=0, y0=1, tmax=10, dt=1)
t2,y2 = Euler_backward(t0=0, y0=1, tmax=10, dt=0.5)
t3,y3 = Euler_backward(t0=0, y0=1, tmax=10, dt=0.25)
```

Numerical solution



$norm_2\ error$



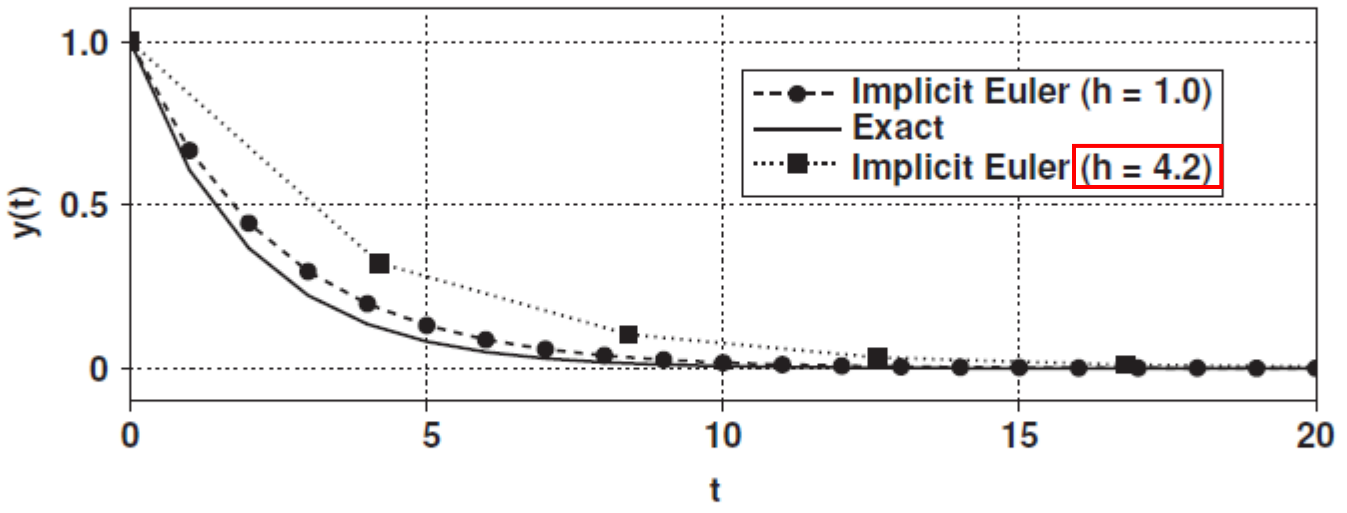
# Implicit or Backward Euler

- **Example**

$$y' + 0.5y = 0$$
$$y(0) = 1 \qquad 0 \le t \le 20.$$

$$\xrightarrow[\text{Implicit Euler}]{}$$

$$y_{n+1} = \frac{y_n}{(1 + 0.5h)}$$



# Trapezoid method 적분이었다 생각

## • Trapezoid method

- From first-order ODE $\frac{dy}{dt} = f(y,t) \qquad y(0) = y_0.$

$$y(t) = y_n + \int_{t_n}^{t} f(y,t')\,dt'.$$

$$y_{n+1} = y_n + \int_{t_n}^{t_{n+1}} f(y,t')\,dt'. \quad \text{At } t = t_{n+1}$$

$$\longrightarrow \boxed{y_{n+1} = y_n + \frac{h}{2}[f(y_{n+1},t_{n+1}) + f(y_n,t_n)].}$$

- Applying the model, $y' = \lambda y.$

$$y_{n+1} - y_n = \frac{h}{2}[\lambda y_{n+1} + \lambda y_n]$$

$$y_{n+1} = \boxed{\frac{1+\frac{\lambda h}{2}}{1-\frac{\lambda h}{2}}} y_n.$$

lambda < 0
==> 전체는 < 1

Exact solution $y(t) = y_0 e^{\lambda t} = y_0 e^{\lambda n h} = y_0 (e^{\lambda h})^n.$

numerical solution $y_n = \sigma^n y_0,$

$$\longrightarrow \quad e^{\lambda h} = 1 + \lambda h + \frac{\lambda^2 h^2}{2} + \boxed{\frac{\lambda^3 h^3}{6}} + \cdots.$$

$$\sigma = \frac{1+\frac{\lambda h}{2}}{1-\frac{\lambda h}{2}} = 1 + \lambda h + \frac{\lambda^2 h^2}{2} + \boxed{\frac{\lambda^3 h^3}{4}} + \cdots$$

Second order accuracy

# Trapezoid method

- **Trapezoid method**
  - Amplification factor

$$\sigma = \frac{1 + \frac{\lambda_R h}{2} + i\frac{\lambda_I h}{2}}{1 - \frac{\lambda_R h}{2} - i\frac{\lambda_I h}{2}}.$$

$$\sigma = \frac{A}{B} e^{i(\theta-\alpha)}$$

$$A = \sqrt{\left(1 + \frac{\lambda_R h}{2}\right)^2 + \frac{\lambda_I^2 h^2}{4}} \quad B = \sqrt{\left(1 - \frac{\lambda_R h}{2}\right)^2 + \frac{\lambda_I^2 h^2}{4}}.$$

$$|\sigma| = \frac{A}{B}.$$

$\lambda_R < 0$, and for these cases $A < B$, $|\sigma| < 1$.

➔ Unconditionally stable, second-order accurate!

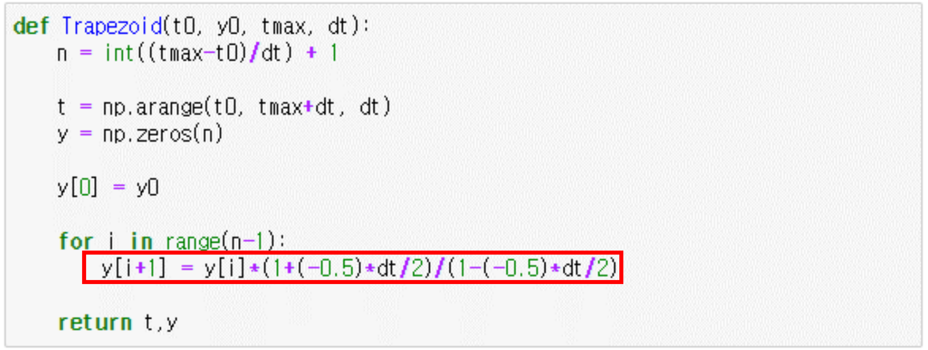
## Trapezoid method

- **Sample code**

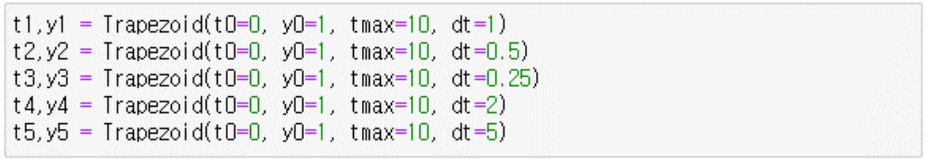
$$y' + 0.5y = 0$$
$$y(0) = 1 \qquad 0 \le t \le 10$$

$$\xrightarrow[\text{Trapezoid}]{}$$

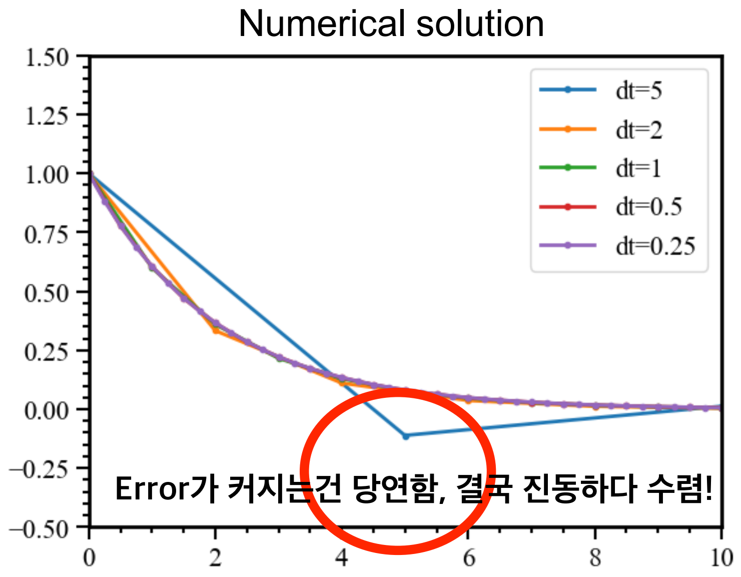
$$y_{n+1} = \frac{1 + \frac{(-0.5)h}{2}}{1 - \frac{(-0.5)h}{2}} y_n$$

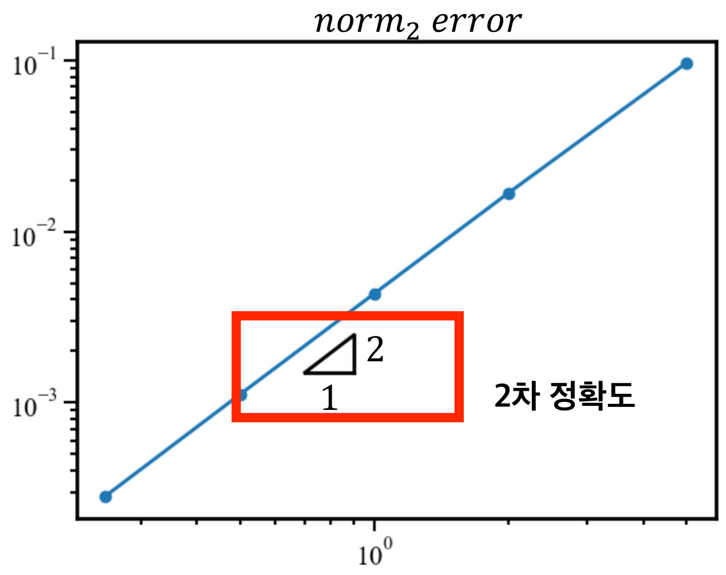


```python
def Trapezoid(t0, y0, tmax, dt):
    n = int((tmax-t0)/dt) + 1

    t = np.arange(t0, tmax+dt, dt)
    y = np.zeros(n)

    y[0] = y0

    for i in range(n-1):
        y[i+1] = y[i]*(1+(-0.5)*dt/2)/(1-(-0.5)*dt/2)

    return t,y
```



```python
t1,y1 = Trapezoid(t0=0, y0=1, tmax=10, dt=1)
t2,y2 = Trapezoid(t0=0, y0=1, tmax=10, dt=0.5)
t3,y3 = Trapezoid(t0=0, y0=1, tmax=10, dt=0.25)
t4,y4 = Trapezoid(t0=0, y0=1, tmax=10, dt=2)
t5,y5 = Trapezoid(t0=0, y0=1, tmax=10, dt=5)
```





# Linearization for implicit methods

$$y' = f(y, t).$$

Applying the trapezoidal method to this equation yields

$$y_{n+1} = y_n + \frac{h}{2}\left[\bigcirc\!\!\!\!\!\!\; f(y_{n+1}, t_{n+1}) + f(y_n, t_n)\right] + O(h^3).$$

$\boxed{\text{Consider the Taylor series expansion of } f(y_{n+1}, t_{n+1})\text{:}}$

$$f(y_{n+1}, t_{n+1}) = f(y_n, t_{n+1}) + (y_{n+1} - y_n)\left.\frac{\partial f}{\partial y}\right|_{(y_n, t_{n+1})}$$

$$+ \frac{1}{2}\cancel{f}(y_{n+1} - y_n)^2 \left.\frac{\partial^2 f}{\partial y^2}\right|_{(y_n, t_{n+1})} + \cdots .$$

$\boxed{y_{n+1} - y_n = O(h).}$

이미 h만큼 error가 존재하니까,
2차미분 term은 O(h^2)일 거야

$$y_{n+1} = y_n + \frac{h}{2}\left[f(y_n, t_{n+1}) + (y_{n+1} - y_n)\left.\frac{\partial f}{\partial y}\right|_{(y_n, t_{n+1})} + f(y_n, t_n)\right] + O(h^3).$$

이건 꼭 손으로 계산해봐야 함

$$\boxed{y_{n+1} = y_n + \frac{h}{2}\,\frac{f(y_n, t_{n+1}) + f(y_n, t_n)}{1 - \frac{h}{2}\frac{\partial f}{\partial y}\Big|_{(y_n, t_{n+1})}}.}$$

the solution can proceed without iteration

Global second-order accuracy… Unconditionally stable but linearization may lead to some loss of total stability of non-linear function $f$.

# Linearization for implicit methods

- **Example** $y' + y(1-y) = 0 \qquad y(0) = \frac{1}{2}$

  - Applying the trapezoidal method to this equation yields

$$y_{n+1} = y_n + \frac{h}{2}\left[f\left(y_{n+1}, t_{n+1}\right) + f\left(y_n, t_n\right)\right] + O(h^3).$$

$$y_{n+1} = y_n + \frac{h}{2}\left[y_{n+1}(y_{n+1} - 1) + y_n(y_n - 1)\right].$$

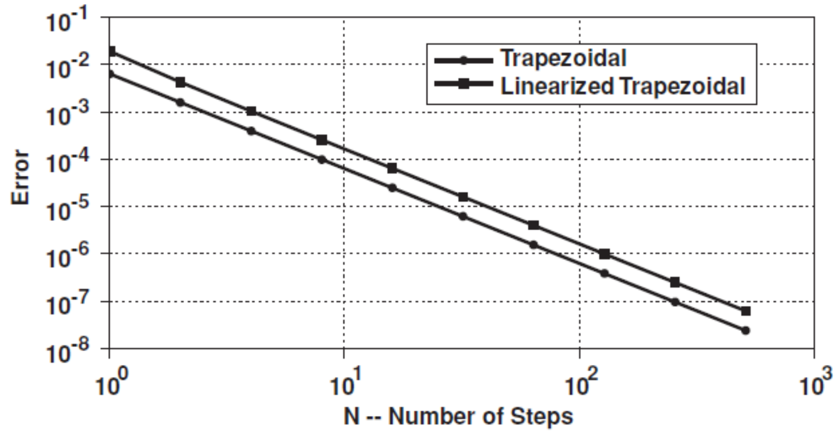
y(n+1)에 대한 2차 방정식의 근을 구하는 방식도 가능!

  - Linearization iterative scheme보다 computing cost측면에서는 더 나음

$$y_{n+1} = y_n + \frac{h}{2}\left[f(y_n, t_{n+1}) + (y_{n+1} - y_n)\frac{\partial f}{\partial y}\bigg|_{(y_n, t_{n+1})} + f(y_n, t_n)\right]$$

$$y_{n+1} = y_n + \frac{h y_n(y_n - 1)}{1 - h\left(y_n - \frac{1}{2}\right)}.$$

어떻게 하든 2차 정확도

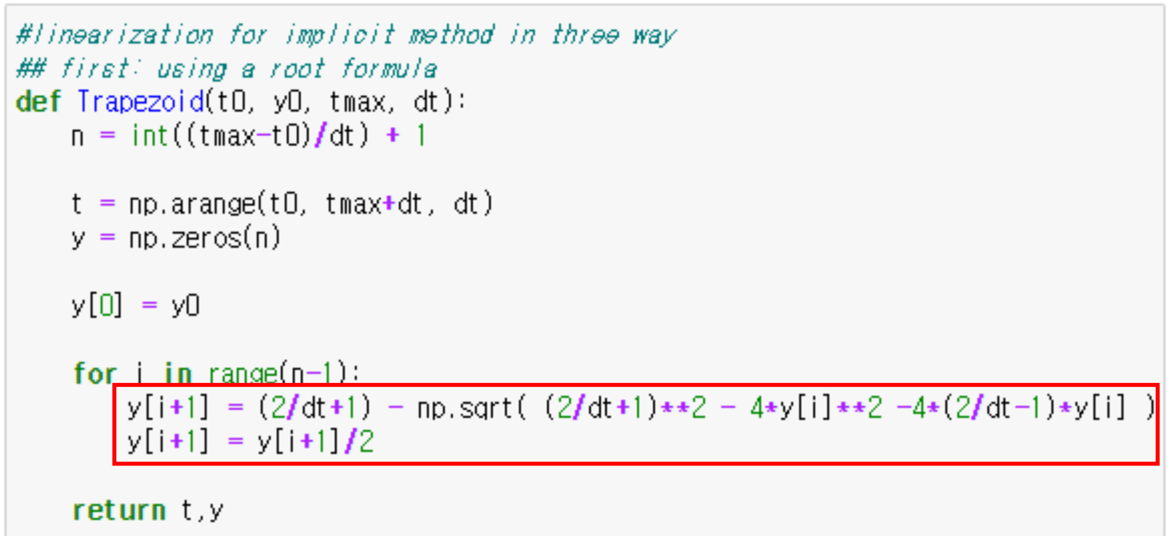


# Linearization for implicit methods

- **Sample code**

$$y' + y(1-y) = 0 \qquad y(0) = \frac{1}{2}$$

- Type 1 (Using a root formula)
- From trapezoid method $y_{n+1} = y_n + \frac{h}{2}\left[y_{n+1}(y_{n+1}-1) + y_n(y_n-1)\right].$



```
#linearization for implicit method in three way
## first: using a root formula
def Trapezoid(t0, y0, tmax, dt):
    n = int((tmax-t0)/dt) + 1

    t = np.arange(t0, tmax+dt, dt)
    y = np.zeros(n)

    y[0] = y0

    for i in range(n-1):
        y[i+1] = (2/dt+1) - np.sqrt( (2/dt+1)**2 - 4*y[i]**2 -4*(2/dt-1)*y[i] )
        y[i+1] = y[i+1]/2

    return t,y
```

$$\frac{2}{h}y_{n+1} = \frac{2}{h}y_n + y_{n+1}^2 - y_{n+1} + y_n^2 - y_n$$

$$y_{n+1}^2 - \left(1+\frac{2}{h}\right)y_{n+1} + y_n^2 + \left(\frac{2}{h}-1\right)y_n = 0$$

From a root formula

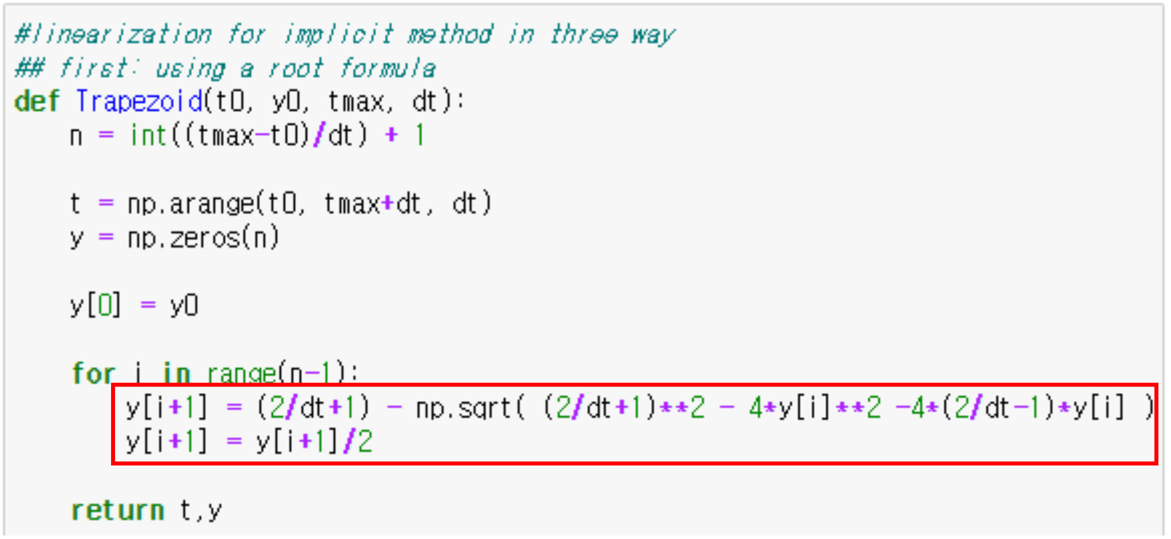
$$y_{n+1} = \frac{\left(1+\frac{2}{h}\right) \pm \sqrt{\left(1+\frac{2}{h}\right)^2 - 4\left(y_n^2 + \left(\frac{2}{h}-1\right)y_n\right)}}{2}$$

# Linearization for implicit methods

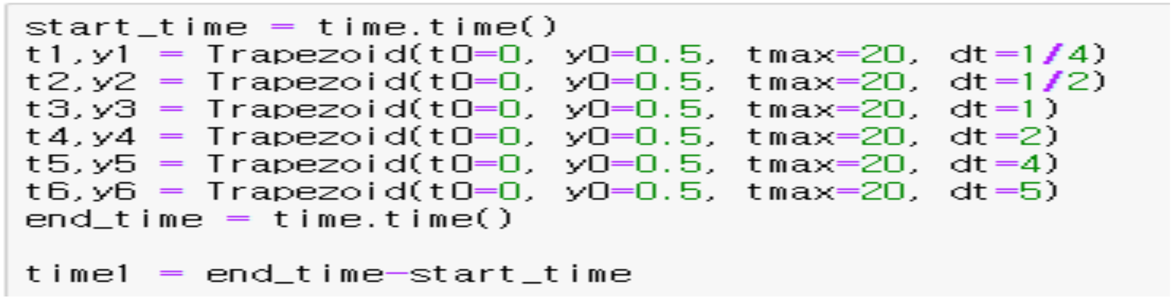
- **Sample code**

$$y' + y(1-y) = 0 \qquad y(0) = \frac{1}{2}$$

- Type 1 (Using a root formula)

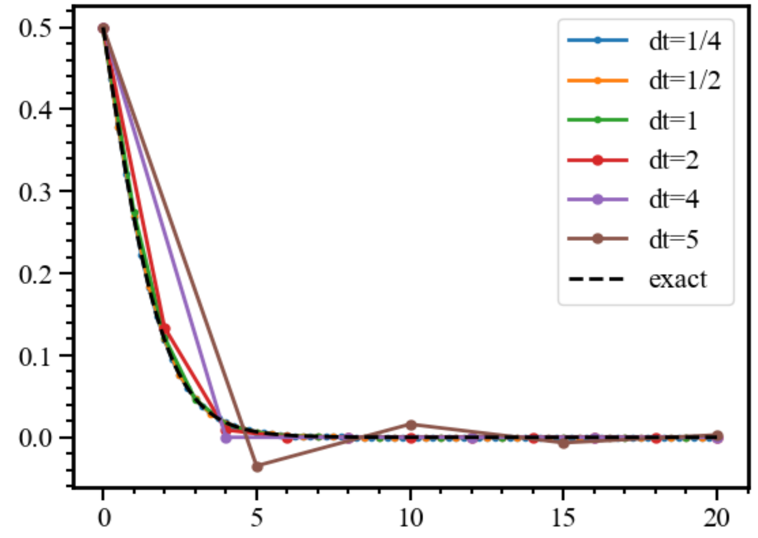


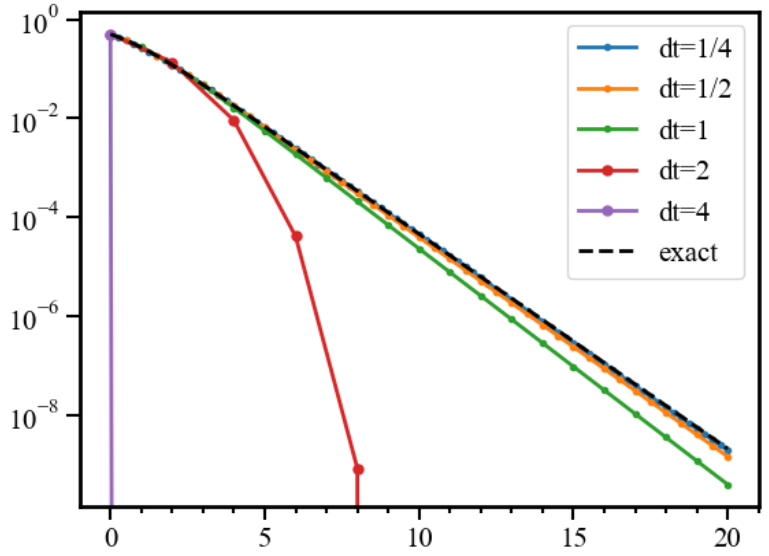
```python
#linearization for implicit method in three way
## first: using a root formula
def Trapezoid(t0, y0, tmax, dt):
    n = int((tmax-t0)/dt) + 1

    t = np.arange(t0, tmax+dt, dt)
    y = np.zeros(n)

    y[0] = y0

    for i in range(n-1):
        y[i+1] = (2/dt+1) - np.sqrt( (2/dt+1)**2 - 4*y[i]**2 -4*(2/dt-1)*y[i] )
        y[i+1] = y[i+1]/2

    return t,y
```



```python
start_time = time.time()
t1,y1 = Trapezoid(t0=0, y0=0.5, tmax=20, dt=1/4)
t2,y2 = Trapezoid(t0=0, y0=0.5, tmax=20, dt=1/2)
t3,y3 = Trapezoid(t0=0, y0=0.5, tmax=20, dt=1)
t4,y4 = Trapezoid(t0=0, y0=0.5, tmax=20, dt=2)
t5,y5 = Trapezoid(t0=0, y0=0.5, tmax=20, dt=4)
t6,y6 = Trapezoid(t0=0, y0=0.5, tmax=20, dt=5)
end_time = time.time()

time1 = end_time-start_time
```

Numerical solution



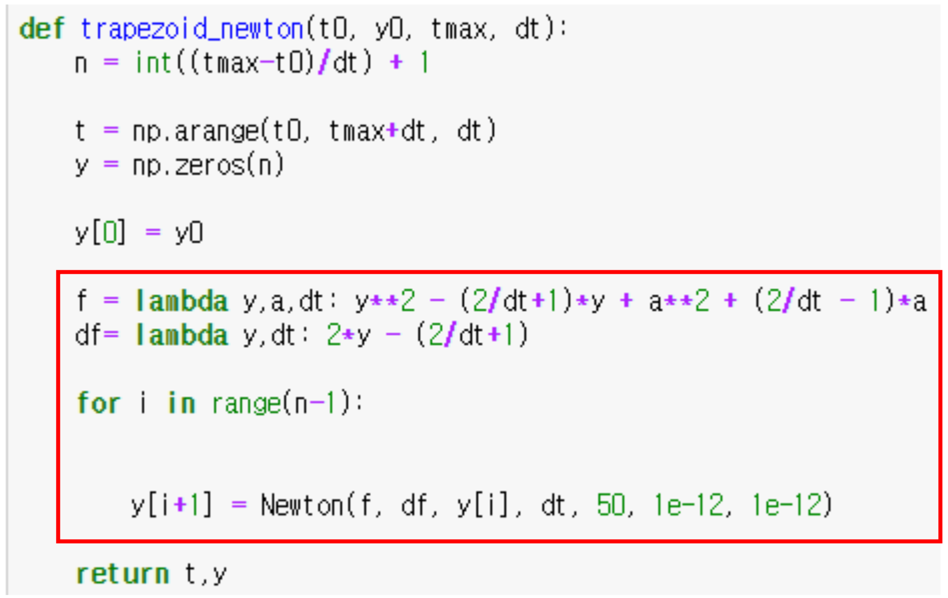


# Linearization for implicit methods

- **Sample code**

$$y' + y(1-y) = 0 \qquad y(0) = \frac{1}{2}$$

- Type 2 (Using Newton method)
- From trapezoid method $y_{n+1} = y_n + \frac{h}{2}\left[y_{n+1}(y_{n+1}-1) + y_n(y_n-1)\right].$



```python
def trapezoid_newton(t0, y0, tmax, dt):
    n = int((tmax-t0)/dt) + 1

    t = np.arange(t0, tmax+dt, dt)
    y = np.zeros(n)

    y[0] = y0

    f = lambda y,a,dt: y**2 - (2/dt+1)*y + a**2 + (2/dt - 1)*a
    df= lambda y,dt: 2*y - (2/dt+1)

    for i in range(n-1):


        y[i+1] = Newton(f, df, y[i], dt, 50, 1e-12, 1e-12)

    return t,y
```

$$\frac{2}{h}y_{n+1} = \frac{2}{h}y_n + y_{n+1}^2 - y_{n+1} + y_n^2 - y_n$$

$$F(y_{n+1}) = y_{n+1}^2 - \left(1+\frac{2}{h}\right)y_{n+1} + y_n^2 + \left(\frac{2}{h}-1\right)y_n = 0$$

Find $y_{n+1}$ such as $F(y_{n+1}) = 0$ using newton method.

$$y_{n+1}^{k+1} = y_{n+1}^k - F'\left(y_{n+1}^k\right)^{-1}F\left(y_{n+1}^k\right)$$

➔ We learn newton method code lecture 4

# Linearization for implicit methods

- **Sample code**

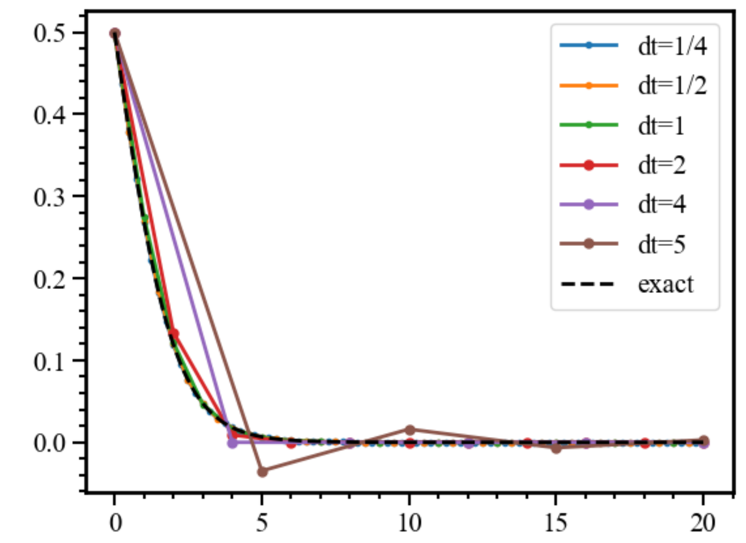
$$y' + y(1-y) = 0 \qquad y(0) = \frac{1}{2}$$

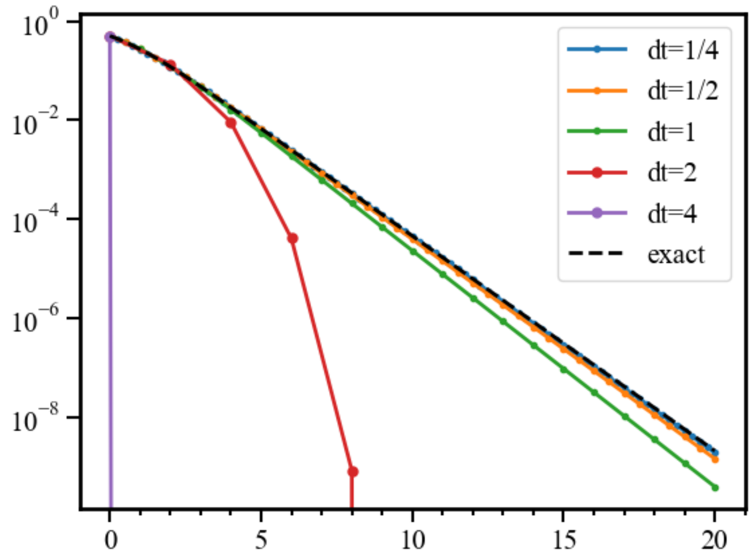
  - Type 2 (Using Newton method)

```
def trapezoid_newton(t0, y0, tmax, dt):
    n = int((tmax-t0)/dt) + 1

    t = np.arange(t0, tmax+dt, dt)
    y = np.zeros(n)

    y[0] = y0

    f = lambda y,a,dt: y**2 - (2/dt+1)*y + a**2 + (2/dt - 1)*a
    df= lambda y,dt: 2*y - (2/dt+1)

    for i in range(n-1):


        y[i+1] = Newton(f, df, y[i], dt, 50, 1e-12, 1e-12)

    return t,y
```

```
start_time = time.time()
t1,y1 = trapezoid_newton(t0=0, y0=0.5, tmax=20, dt=1/4)
t2,y2 = trapezoid_newton(t0=0, y0=0.5, tmax=20, dt=1/2)
t3,y3 = trapezoid_newton(t0=0, y0=0.5, tmax=20, dt=1)
t4,y4 = trapezoid_newton(t0=0, y0=0.5, tmax=20, dt=2)
t5,y5 = trapezoid_newton(t0=0, y0=0.5, tmax=20, dt=4)
t6,y6 = trapezoid_newton(t0=0, y0=0.5, tmax=20, dt=5)
end_time = time.time()

time1 = end_time-start_time
```

Numerical solution



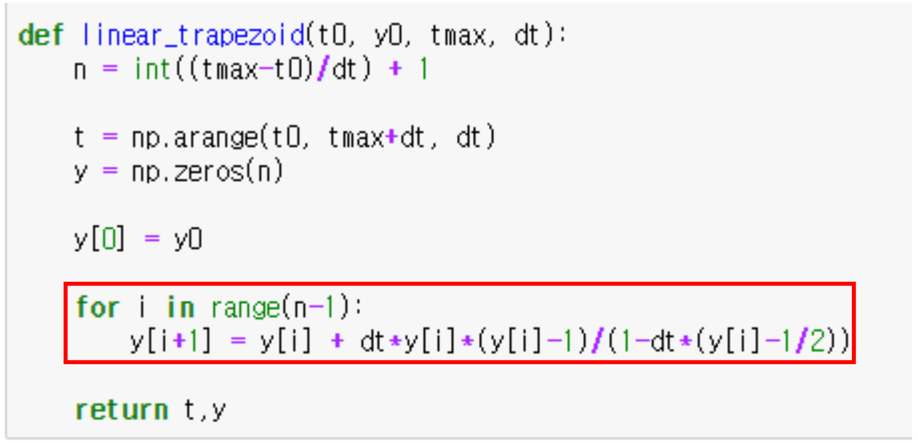


## Linearization for implicit methods

- **Sample code**

$$y' + y(1-y) = 0 \qquad y(0) = \frac{1}{2}$$

- ● Type 3 (Linearization trapezoid method)
- ● From trapezoid method $y_{n+1} = y_n + \frac{h}{2}\left[y_{n+1}(y_{n+1}-1) + y_n(y_n-1)\right].$



```
def linear_trapezoid(t0, y0, tmax, dt):
    n = int((tmax-t0)/dt) + 1

    t = np.arange(t0, tmax+dt, dt)
    y = np.zeros(n)

    y[0] = y0

    for i in range(n-1):
        y[i+1] = y[i] + dt*y[i]*(y[i]-1)/(1-dt*(y[i]-1/2))

    return t,y
```

$$y_{n+1} = y_n + \frac{h}{2}\left[f(y_n, t_{n+1}) + (y_{n+1} - y_n)\frac{\partial f}{\partial y}\bigg|_{(y_n, t_{n+1})} + f(y_n, t_n)\right]$$

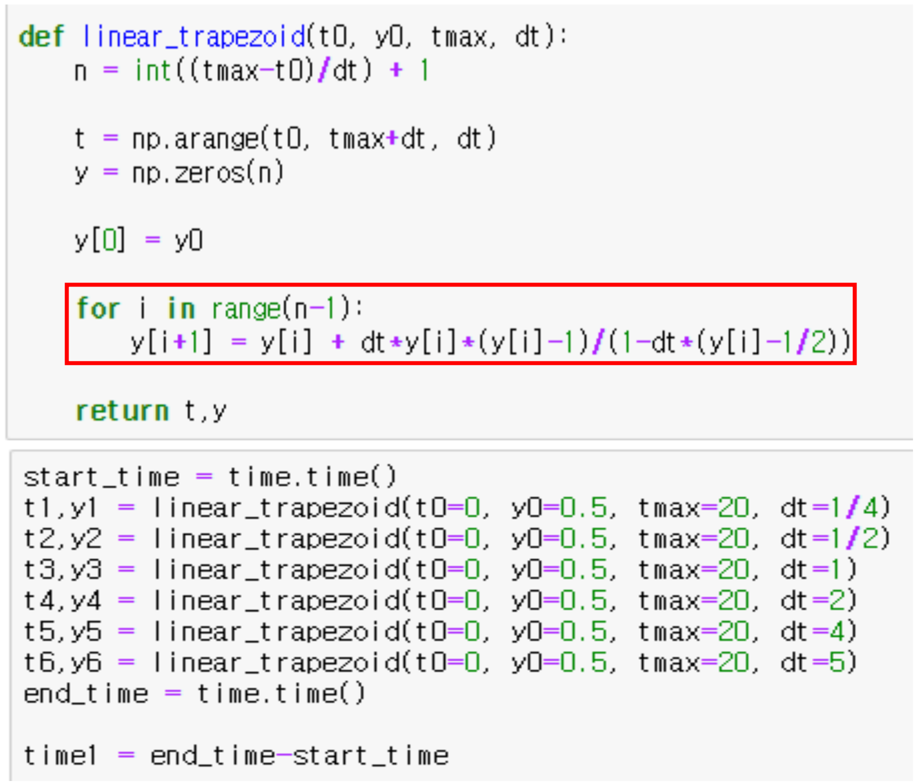
$$y_{n+1} = y_n + \frac{h y_n(y_n-1)}{1-h\left(y_n - \frac{1}{2}\right)}.$$

# Linearization for implicit methods

- **Sample code**

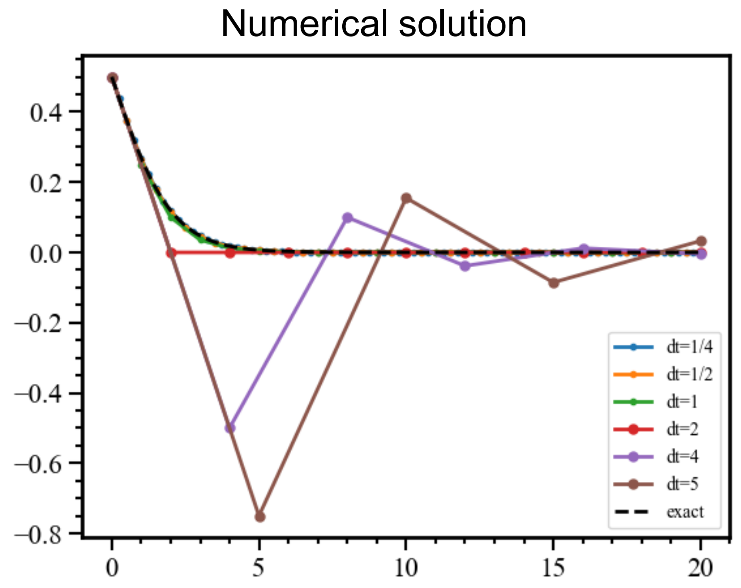
$$y' + y(1-y) = 0 \qquad y(0) = \frac{1}{2}$$

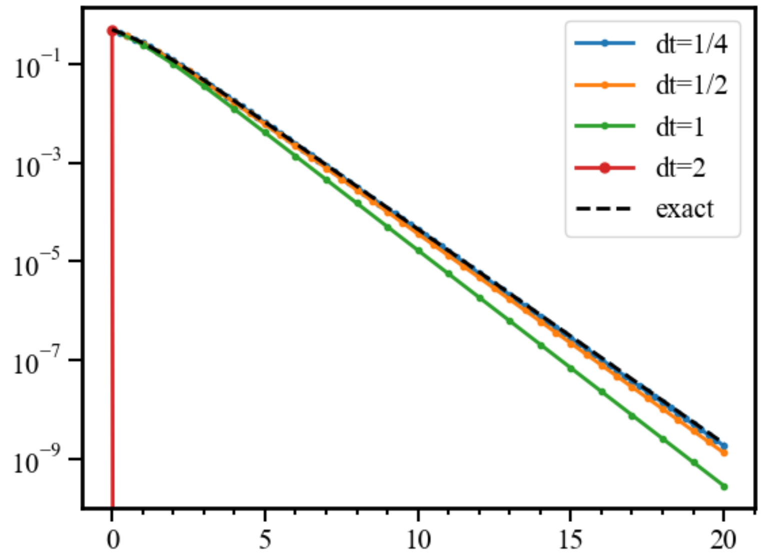
- Type 3 (Linearization trapezoid method)



```
def linear_trapezoid(t0, y0, tmax, dt):
    n = int((tmax-t0)/dt) + 1

    t = np.arange(t0, tmax+dt, dt)
    y = np.zeros(n)

    y[0] = y0

    for i in range(n-1):
        y[i+1] = y[i] + dt*y[i]*(y[i]-1)/(1-dt*(y[i]-1/2))

    return t,y
```

```
start_time = time.time()
t1,y1 = linear_trapezoid(t0=0, y0=0.5, tmax=20, dt=1/4)
t2,y2 = linear_trapezoid(t0=0, y0=0.5, tmax=20, dt=1/2)
t3,y3 = linear_trapezoid(t0=0, y0=0.5, tmax=20, dt=1)
t4,y4 = linear_trapezoid(t0=0, y0=0.5, tmax=20, dt=2)
t5,y5 = linear_trapezoid(t0=0, y0=0.5, tmax=20, dt=4)
t6,y6 = linear_trapezoid(t0=0, y0=0.5, tmax=20, dt=5)
end_time = time.time()

time1 = end_time-start_time
```



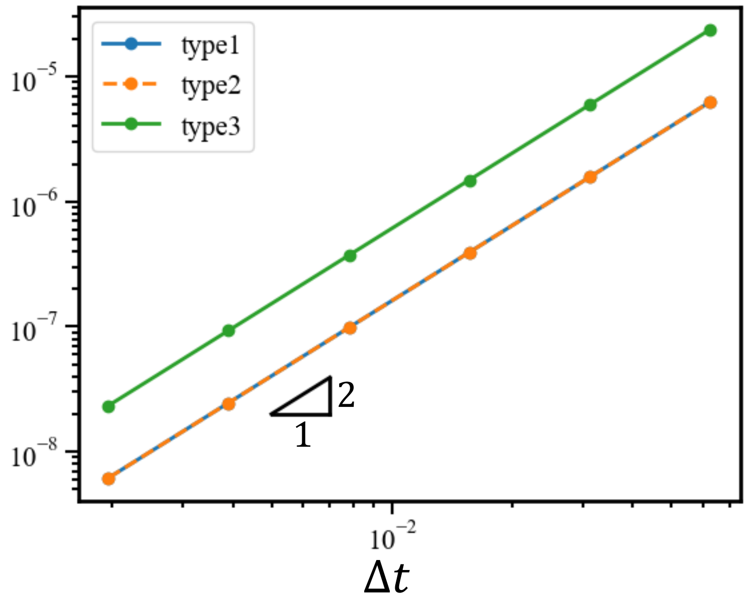


# Linearization for implicit methods

- **Compare Three methods**

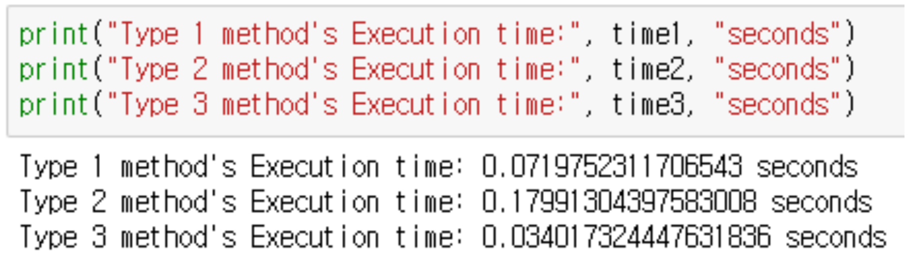
$$y' + y(1-y) = 0 \qquad y(0) = \frac{1}{2}$$

- Type 1 (Using a root formula)
- Type 2 (Using Newton method)
- Type 3 (Linearization trapezoid method)



➔ Types 1 and 2 are more accurate than type 3,
but all three types have second-order accuracy.

➔ Type 3 is about 5 times faster to calculate than type 2,
and it is also twice as fast as type 1,
although it is difficult to use the root formula and
not suitable for nonlinear equations that cannot be solved
manually.



```
print("Type 1 method's Execution time:", time1, "seconds")
print("Type 2 method's Execution time:", time2, "seconds")
print("Type 3 method's Execution time:", time3, "seconds")
```

```
Type 1 method's Execution time: 0.0719752311706543 seconds
Type 2 method's Execution time: 0.17991304397583008 seconds
Type 3 method's Execution time: 0.034017324447631836 seconds
```

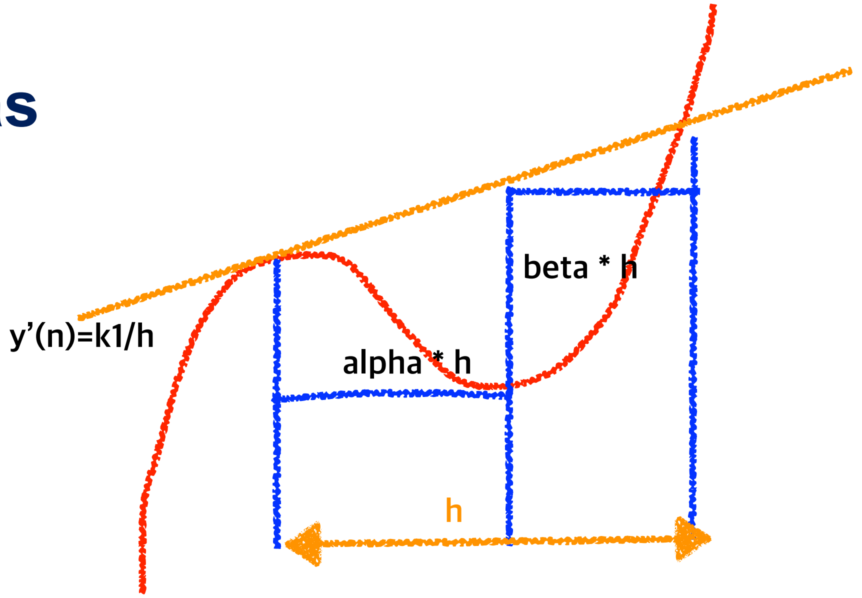
# Runge-Kutta method

- **Second-order (two stage) Runge-Kutta formulas**

$$y_{n+1} = y_n + \gamma_1 k_1 + \gamma_2 k_2,$$

$$k_1 = hf(y_n, t_n)$$

$$k_2 = hf(y_n + \beta k_1, t_n + \alpha h),$$



y* = y(n) + beta*h*y'(n)
k1 = h*y'(n) 라고 가정해서
다음걸 선택해보자

- consider Taylor series expansion

y(n+1)을 explicit하게!

$$y_{n+1} = y_n + hy'_n + \frac{h^2}{2} y''_n + \cdots.$$

$$y'_n = f(y_n, t_n),$$

$$y'' = f_t + ff_y,$$ y'' = ft + fy*y'

$$y_{n+1} = y_n + hf(y_n, t_n) + \frac{h^2}{2}(f_{t_n} + f_n f_{y_n}) + \cdots. \quad \ldots(1)$$

- Taylor series expansion for $k_2$

$$k_2 = h[f(y_n, t_n) + \beta k_1 f_{y_n} + \alpha h f_{t_n} + \boxed{O(h^2)}].$$

$$\boxed{y_{n+1} = y_n + (\gamma_1 + \gamma_2)hf_n + \gamma_2 \beta h^2 f_n f_{y_n} + \gamma_2 \alpha h^2 f_{t_n}} + \cdots. \quad \ldots(2)$$

# Runge-Kutta method

- Second-order (two stage) Runge-Kutta formulas

(1) $$y_{n+1} = y_n + hf(y_n, t_n) + \frac{h^2}{2}(f_{t_n} + f_n f_{y_n}) + \cdots.$$

(2) $$y_{n+1} = y_n + (\gamma_1 + \gamma_2)hf_n + \gamma_2 \beta h^2 f_n f_{y_n} + \gamma_2 \alpha h^2 f_{t_n} + \cdots.$$

$$\begin{aligned} \gamma_1 + \gamma_2 &= 1 \\ \gamma_2 \alpha &= \frac{1}{2} \\ \gamma_2 \beta &= \frac{1}{2}. \end{aligned} \longrightarrow \quad \gamma_2 = \frac{1}{2\alpha} \qquad \beta = \alpha \qquad \gamma_1 = 1 - \frac{1}{2\alpha}.$$

$$\boxed{\begin{aligned} k_1 &= hf(y_n, t_n) \\ k_2 &= hf(y_n + \alpha k_1, t_n + \alpha h) \\ y_{n+1} &= y_n + \left(1 - \frac{1}{2\alpha}\right) k_1 + \frac{1}{2\alpha} k_2. \end{aligned}}$$

alpha는 t에 대한 변수 이므로
항상 alpha의 bound 를 정해줄 수 있음
—> 따라서 alpha에 대해서 식을 정의함
0 < alpha <= 1

cf) alpha가 1보다 크다면...?????

alpha = 1 :
Trapezoid method linearlization하는 모양이 됨

# Runge-Kutta method

- **Second-order (two stage) Runge-Kutta formulas**
  - Popular form of the second-order Runge-Kutta method
    - ➔ Predictor-corrector scheme $(\alpha = 1/2)$

$$\boxed{\begin{aligned} y^*_{n+1/2} &= y_n + \frac{h}{2} f(y_n, t_n) \\ y_{n+1} &= y_n + h f(y^*_{n+1/2}, t_{n+1/2}). \end{aligned}}$$

step 2단계라는 걸 강조하기 위해서

  - Accuracy of RK2 with model problem $y' = \lambda y$

$$k_1 = \lambda h y_n$$

$$k_2 = h(\lambda y_n + \alpha\lambda^2 h y_n) = \lambda h\,(1 + \alpha h\lambda)\, y_n$$

$$y_{n+1} = y_n + \left(1 - \frac{1}{2\alpha}\right)\lambda h y_n + \frac{1}{2\alpha}\lambda h\,(1 + \alpha\lambda h)\, y_n$$

$$= y_n \underline{\left(1 + \lambda h + \frac{\lambda^2 h^2}{2}\right)}.$$

$$e^{\lambda h} = \underline{1 + \lambda h + \frac{\lambda^2 h^2}{2}} + \boxed{\frac{\lambda^3 h^3}{6} + \cdots}.$$

# Runge-Kutta method

- **Fourth-order Runge-Kutta formula**
  - Fourth-order formula (most popular)

$$y_{n+1} = y_n + \frac{1}{6}k_1 + \frac{1}{3}(k_2 + k_3) + \frac{1}{6}k_4,$$

weight sum = 1

$$k_1 = hf(y_n, t_n)$$

$$\boxed{\begin{aligned} k_2 &= hf\left(y_n + \frac{1}{2}k_1, t_n + \frac{h}{2}\right) \\ k_3 &= hf\left(y_n + \frac{1}{2}k_2, t_n + \frac{h}{2}\right) \end{aligned}}$$

RK2 관점

$$k_4 = hf(y_n + k_3, t_n + h).$$

  - Accuracy of $\underline{\text{RK4}}$ with model problem $y' = \lambda y$

4차 정확도

$$y_{n+1} = \left(1 + \lambda h + \frac{\lambda^2 h^2}{2} + \frac{\lambda^3 h^3}{6} + \frac{\lambda^4 h^4}{24}\right) y_n,$$

$$\lambda h + \frac{\lambda^2 h^2}{2} + \frac{\lambda^3 h^3}{6} + \frac{\lambda^4 h^4}{24} + 1 - e^{i\theta} = 0,$$

## Runge-Kutta method

- **Stability of Runge-Kutta formulas**

원래, 고차정확도로 갈수록 stability는 나빠지지만,
RK의 경우 고차 정확도로 갈수록 stability가 더 높아짐

$$\sigma = \left(1 + \lambda h + \frac{\lambda^2 h^2}{2}\right) \qquad |\sigma| \leq 1$$

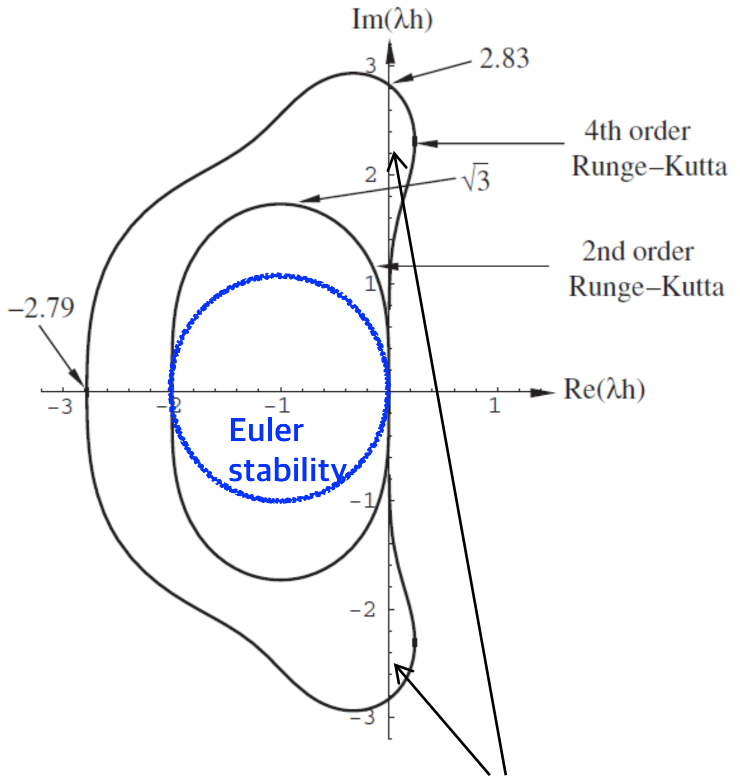
unstable for purely imaginary $\lambda$.

$$|\sigma| = \sqrt{1 + \frac{\omega^4 h^4}{4}} > 1,$$

for small values of $\omega h$, this method is less unstable than explicit Euler.

Explicit Euler

$$|\sigma|^2 = (1 + \lambda_R h)^2 + \lambda_I^2 h^2$$



two small stable regions corresponding to positive $Re(\lambda)$,

# Runge-Kutta method

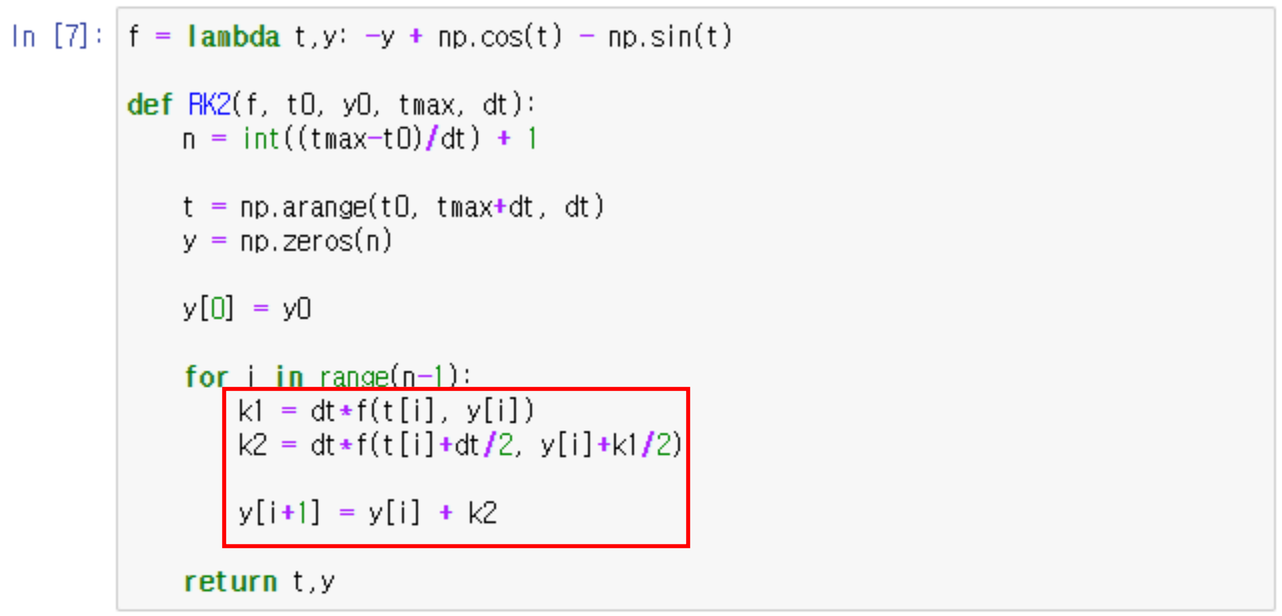
$$k_1 = hf(y_n, t_n)$$

$$k_2 = hf(y_n + \alpha k_1, t_n + \alpha h)$$

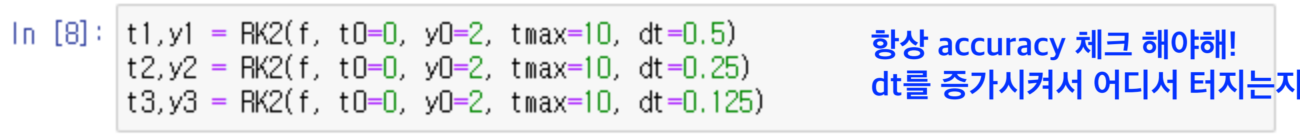
$$y_{n+1} = y_n + \left(1 - \frac{1}{2\alpha}\right) k_1 + \frac{1}{2\alpha} k_2.$$

- **Sample code**

$$\frac{dy}{dt} = -y + \cos(t) - \sin(t) \quad , y(0) = 2$$

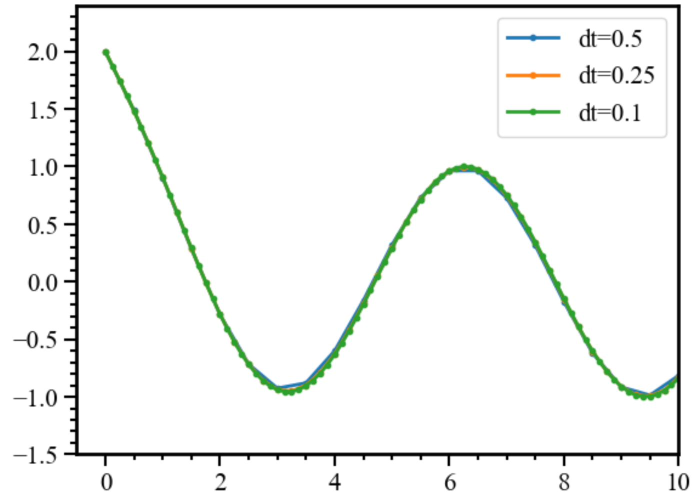


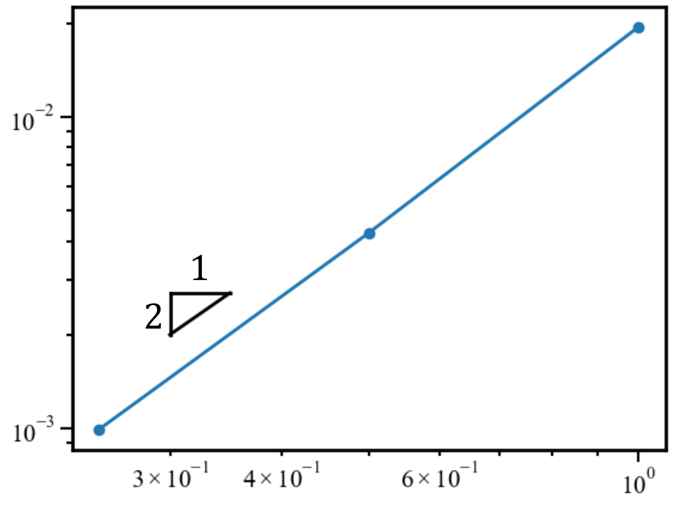
```
In [7]: f = lambda t,y: -y + np.cos(t) - np.sin(t)

        def RK2(f, t0, y0, tmax, dt):
            n = int((tmax-t0)/dt) + 1

            t = np.arange(t0, tmax+dt, dt)
            y = np.zeros(n)

            y[0] = y0

            for i in range(n-1):
                k1 = dt*f(t[i], y[i])
                k2 = dt*f(t[i]+dt/2, y[i]+k1/2)

                y[i+1] = y[i] + k2

            return t,y
```



```
In [8]: t1,y1 = RK2(f, t0=0, y0=2, tmax=10, dt=0.5)
        t2,y2 = RK2(f, t0=0, y0=2, tmax=10, dt=0.25)
        t3,y3 = RK2(f, t0=0, y0=2, tmax=10, dt=0.125)
```

항상 accuracy 체크 해야해!
dt를 증가시켜서 어디서 터지는지 확인!





$$y_{n+1} = y_n + \frac{1}{6}k_1 + \frac{1}{3}(k_2 + k_3) + \frac{1}{6}k_4,$$

$$k_1 = hf(y_n, t_n)$$

$$k_2 = hf\left(y_n + \frac{1}{2}k_1, t_n + \frac{h}{2}\right)$$

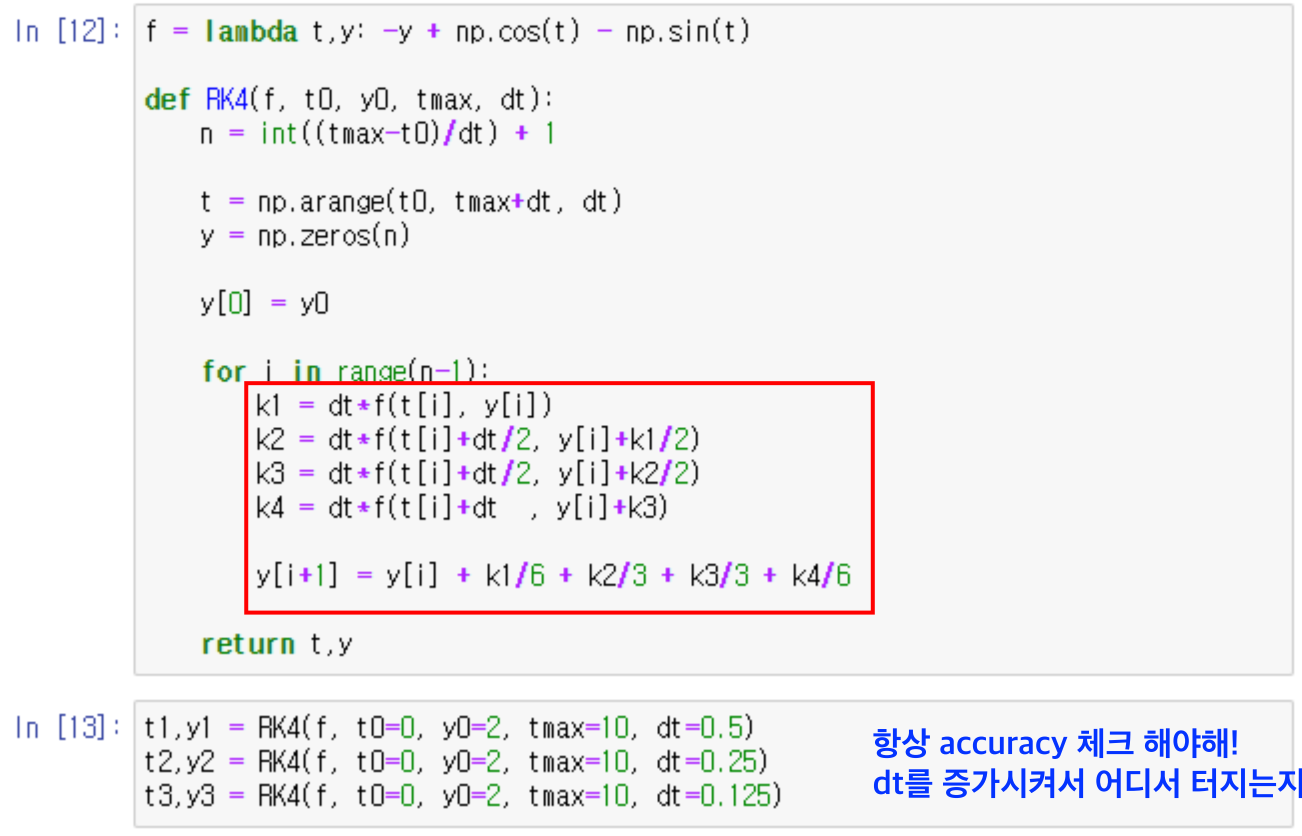
$$k_3 = hf\left(y_n + \frac{1}{2}k_2, t_n + \frac{h}{2}\right)$$

$$k_4 = hf(y_n + k_3, t_n + h).$$

# Runge-Kutta method

- **Sample code**

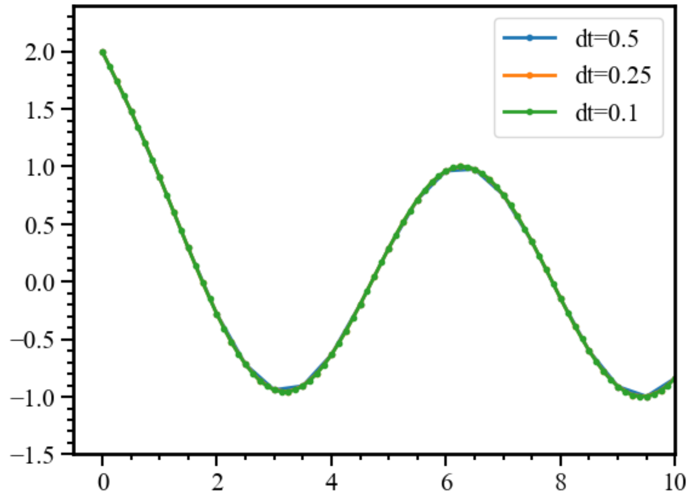
$$\frac{dy}{dt} = -y + \cos(t) - \sin(t) \quad , y(0) = 2$$

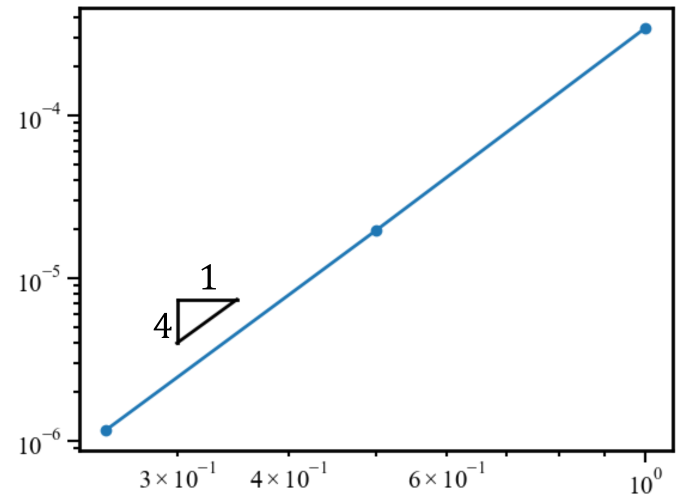


```
In [12]: f = lambda t,y: -y + np.cos(t) - np.sin(t)

         def RK4(f, t0, y0, tmax, dt):
             n = int((tmax-t0)/dt) + 1

             t = np.arange(t0, tmax+dt, dt)
             y = np.zeros(n)

             y[0] = y0

             for i in range(n-1):
                 k1 = dt*f(t[i], y[i])
                 k2 = dt*f(t[i]+dt/2, y[i]+k1/2)
                 k3 = dt*f(t[i]+dt/2, y[i]+k2/2)
                 k4 = dt*f(t[i]+dt  , y[i]+k3)

                 y[i+1] = y[i] + k1/6 + k2/3 + k3/3 + k4/6

             return t,y
```

```
In [13]: t1,y1 = RK4(f, t0=0, y0=2, tmax=10, dt=0.5)
         t2,y2 = RK4(f, t0=0, y0=2, tmax=10, dt=0.25)
         t3,y3 = RK4(f, t0=0, y0=2, tmax=10, dt=0.125)
```

항상 accuracy 체크 해야해!
dt를 증가시켜서 어디서 터지는지 확인!



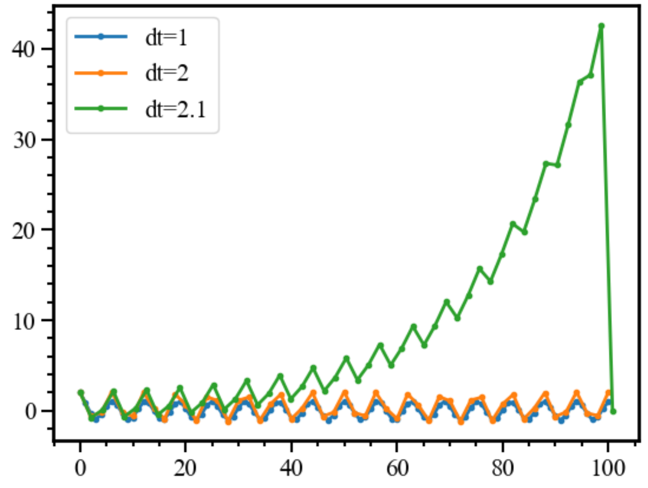


# Runge-Kutta method

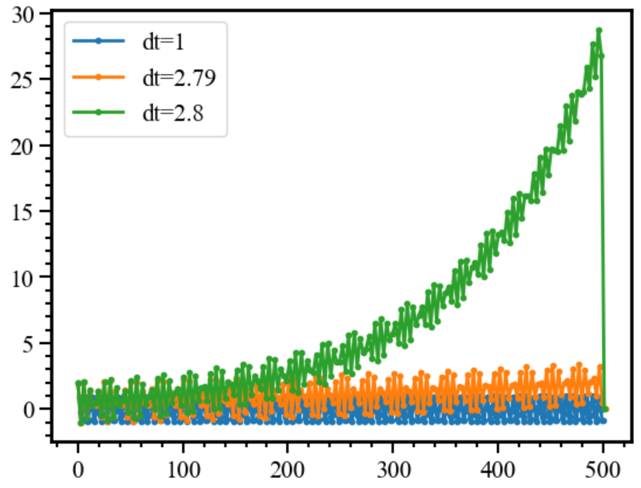
- **stability**

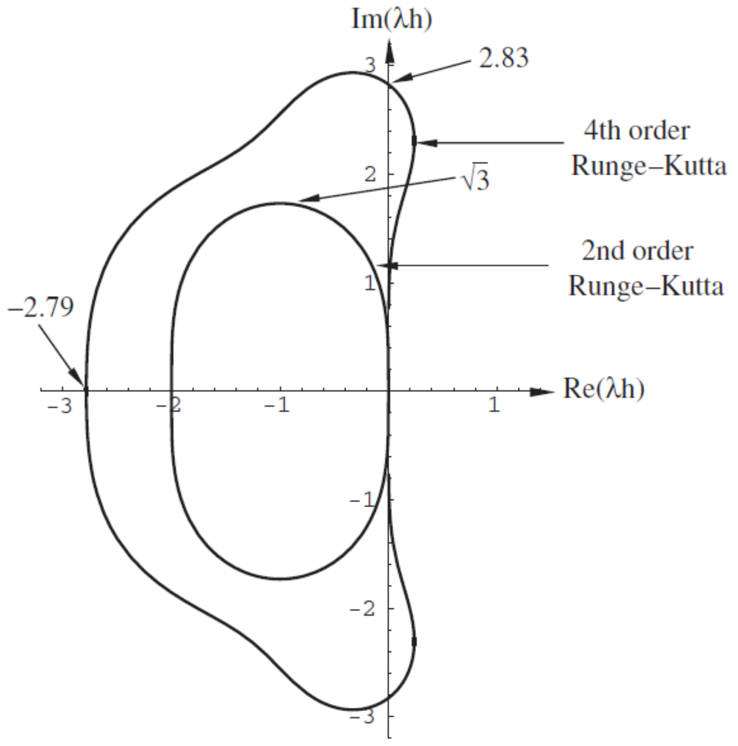
$$\frac{dy}{dt} = -y + \cos(t) - \sin(t) \quad , y(0) = 2$$

RK2



RK4





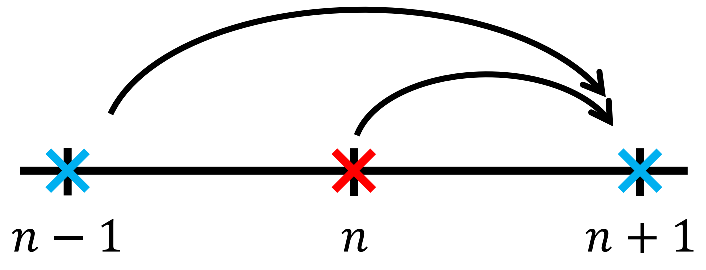
# Multi-step methods

현재까진,
과거 정보 하나로 미래 하나를 예측했음
(n)번째로, (n+1)번째 예측

- **Leapfrog method**
  - Second-order central difference formula for $y'$

$$y_{n+1} = y_{n-1} + 2hf(y_n, t_n) + O(h^3).$$

dt를 줄이거나 하면 error가 늘어날 수 있어서 잘 쓰지 않음



- **Adams-Bashforth method**
  - Second-order accurate globally

$$y_{n+1} = y_n + hy_n' + \frac{h^2}{2}y_n'' + \frac{h^3}{6}y_n''' + \cdots.$$

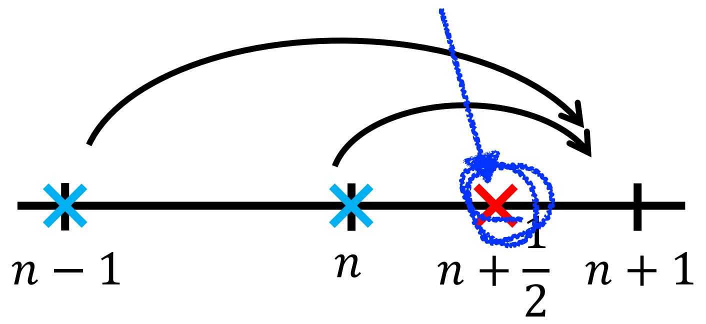
★

$$y_n'' = \frac{f(y_n, t_n) - f(y_{n-1}, t_{n-1})}{h} + O(h)$$

$$y_{n+1} = y_n + \boxed{\frac{3h}{2}f(y_n, t_n) - \frac{h}{2}f(y_{n-1}, t_{n-1})} + O(h^3).$$

t(n+1/2)에서의 값을 모르니까 interpolation 해야함
—> n-1 과 n의 외분으로 구함

t = t(n)  : Forward —> 1st
t = t(n+1) : Backward
t = t(n+1/2) : Trapezoid —> 2nd accuracy



## System of First-order ODEs

- **First-order ODE system**

$$\mathbf{y}' = \mathbf{f}(\mathbf{y}, t) \qquad \mathbf{y}(0) = \mathbf{y}_0$$

$$f_i(y_1, y_2, y_3, \ldots, y_m, t), i = 1, 2, \ldots, m$$

  - For example, application of the explicit Euler

$$y_i^{(n+1)} = y_i^{(n)} + h f_i(y_1^{(n)}, y_2^{(n)}, \ldots, y_m^{(n)}, t_n) \quad i = 1, 2, 3, \ldots, m.$$

$$\frac{d\mathbf{y}}{dt} = A\mathbf{y}$$

$$\mathbf{y}_{n+1} = \mathbf{y}_n + hA\mathbf{y}_n = (I + hA)\mathbf{y}_n \qquad \mathbf{y}_n = (I + hA)^n \mathbf{y}_0.$$

$|1 + \lambda_i h| \le 1.$ $\lambda_i$ are the eigenvalues of the matrix $A$

$$h \le \frac{2}{|\lambda|_{\max}}.$$

# System of First-order ODEs

- **Linearization** implicit

$$\frac{d\boldsymbol{u}}{dt} = \boldsymbol{f}(u_1, u_2, \ldots, u_m, t)$$

$$\boldsymbol{u}^{(n+1)} = \boldsymbol{u}^{(n)} + \frac{h}{2}\left[\boldsymbol{f}\left(\boldsymbol{u}^{(n+1)}, t_{n+1}\right) + \boldsymbol{f}\left(\boldsymbol{u}^{(n)}, t_n\right)\right].$$

$$f_i\left(\boldsymbol{u}^{(n+1)}, t_{n+1}\right) = f_i\left(\boldsymbol{u}^{(n)}, t_{n+1}\right) + \sum_{j=1}^{m}\left(u_j^{(n+1)} - u_j^{(n)}\right)\left.\frac{\partial f_i}{\partial u_j}\right|_{\boldsymbol{u}^{(n)}, t_{n+1}} + O(h^2)$$

$$i = 1, 2, \ldots, m.$$

$$\underline{\boldsymbol{f}\left(\boldsymbol{u}^{(n+1)}, t_{n+1}\right) = \boldsymbol{f}\left(\boldsymbol{u}^{(n)}, t_{n+1}\right) + A_n\left(\boldsymbol{u}^{(n+1)} - \boldsymbol{u}^{(n)}\right) + O(h^2)}$$

$$A_n = \begin{bmatrix} \frac{\partial f_1}{\partial u_1} & \frac{\partial f_1}{\partial u_2} & \cdots & \frac{\partial f_1}{\partial u_m} \\ \vdots & & & \\ \frac{\partial f_m}{\partial u_1} & \frac{\partial f_m}{\partial u_2} & \cdots & \frac{\partial f_m}{\partial u_m} \end{bmatrix}_{(\boldsymbol{u}^{(n)}, t_{n+1})}$$

Jacobian matrix

iterative 하게 풀자 : fast process를 듬성듬성 해서 slow process를 길게 error를 줄일 수 있어

$$\left(I - \frac{h}{2}A_n\right)\boldsymbol{u}^{(n+1)} = \left(I - \frac{h}{2}A_n\right)\boldsymbol{u}^{(n)} + \frac{h}{2}\left[\boldsymbol{f}\left(\boldsymbol{u}^{(n)}, t_n\right) + \boldsymbol{f}\left(\boldsymbol{u}^{(n)}, t_{n+1}\right)\right].$$

# Initial Value Problem (IVP) in general form

- **Initial Value Problem in general**
  - Consider the $n$th-order ordinary differential equation
  - For an $n$th order ODE the initial value is known value for the 0th to $(n-1)^{\text{th}}$ derivative at $t = 0$

$$F\left(t, f(t), \frac{df(t)}{dt}, \frac{d^2f(t)}{dt^2}, \dots, \frac{d^{n-1}f(t)}{dt^{n-1}}, \frac{d^nf(t)}{dt^n}\right) = 0 \qquad \begin{aligned} f(0) &= \alpha, \\ f'(0) &= \beta, \\ \vdots \quad & \quad \vdots \\ f^{n-1}(0) &= \gamma \end{aligned} \qquad 0 < t \le t_f.$$

  - Finding a solution to an ODE given an initial value is called **initial value problem**.
- **Reduction of Order**
  - Many numerical methods for solving IVP are designed specifically to solve first-order ODE.
  - To make these useful for solving higher-order ODE, we often reduce the order of the ODE.

$$S(t) = \begin{bmatrix} f(t) \\ f^{(1)}(t) \\ f^{(2)}(t) \\ f^{(3)}(t) \\ \vdots \\ f^{(n-1)}(t) \end{bmatrix}. \qquad \frac{dS(t)}{dt} = \begin{bmatrix} f^{(1)}(t) \\ f^{(2)}(t) \\ f^{(3)}(t) \\ f^{(4)}(t) \\ \vdots \\ f^{(n)}(t) \end{bmatrix} = \begin{bmatrix} f^{(1)}(t) \\ f^{(2)}(t) \\ f^{(3)}(t) \\ f^{(4)}(t) \\ \vdots \\ F\left(t, f(t), f^{(1)}(t), \dots, f^{(n-1)}(t)\right) \end{bmatrix} = \begin{bmatrix} S_2(t) \\ S_3(t) \\ S_4(t) \\ S_5(t) \\ \vdots \\ F\left(t, S_1(t), S_2(t), \dots, S_{n-1}(t)\right) \end{bmatrix}$$

# Boundary Value Problem (BVP)

- **Boundary Value Problem (BVP)**
  - At least second-order differential equation

공간에 대한 미분

$$y'' = f(x, y, \bigcirc{y'}) \qquad y(0) = y_0 \qquad y(L) = y_L$$

$$y'' = f(x, y, y') \qquad y(0) = y_0 \qquad y'(0) = y_p$$

y" 한정 : RK4쓰게 되면 이렇게 하면 좋을 수 있다

➔ Shooting method; an iterative technique which uses the standard methods for initial value problem.

➔ Direct methods; based on finite-differencing of the derivative in the differential equation and solving the resulting system of algebraic equations.

# Boundary Value Problem (BVP)

- **Shooting method**
  - Let’s reduce the second-order differential equation to two first-order equation

$$u = y \qquad v = y'$$

$$\begin{cases} u' = v \\ v' = f(x, u, v). \end{cases}$$

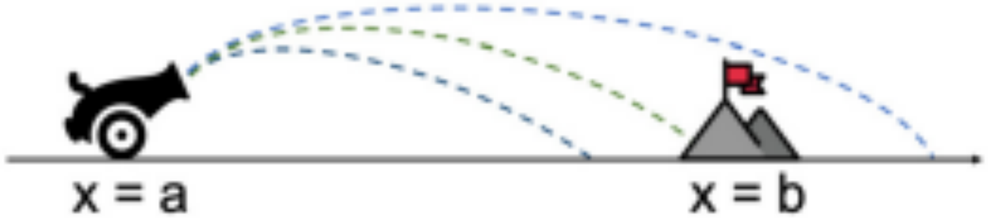
$$u(0) = y_0 \quad \text{and} \quad u(L) = y_L$$

v(0)가 없어!!!

  - We need one(initial) condition for each of unknowns!

1. Guess for $v(0)$ and integrate both equations to $x = L$.
2. Compare $u(L)$ with $y_L$
3. Do iterations to get satisfactorable solutions.

How to guess appropriate $v(0)$?



## 1. Shooting method

**[Detail Algorithm]**

*Pdf 참고*

**[Step]**

1. *Idea : BVP → IVP system*

$$\begin{array}{c} BVP \\ u'' = f(t,u,u') \\ u(a)=\alpha, \quad u(b)=\beta \end{array} \longrightarrow \begin{array}{c} IVP \\ \begin{bmatrix} u \\ u' \end{bmatrix}' = \begin{bmatrix} u' \\ u'' \end{bmatrix} \\ u(a)=\alpha, \quad u'(a)=\gamma \end{array}$$

*2. Find a numerical solution*

1. *IVP solver* $(u'(a)=\gamma$ *찾기)*
2. *Nonlinear equation solver* $(u(b)=\beta \Rightarrow g(\gamma)=u_\gamma(b)-u(b)=0)$

# Boundary Value Problem (BVP)

- **Shooting method** linear ODE 경우 shooting 2번이면 바로 나옴! cf) 푸코 진자 : theta가 작으면 linear, 아니면 nonlinear
  - Consider the general second-order linear equation

$$y''(x) + A(x)y'(x) + B(x)y(x) = f(x)$$

$$y(0) = y_0 \qquad y(L) = y_L.$$

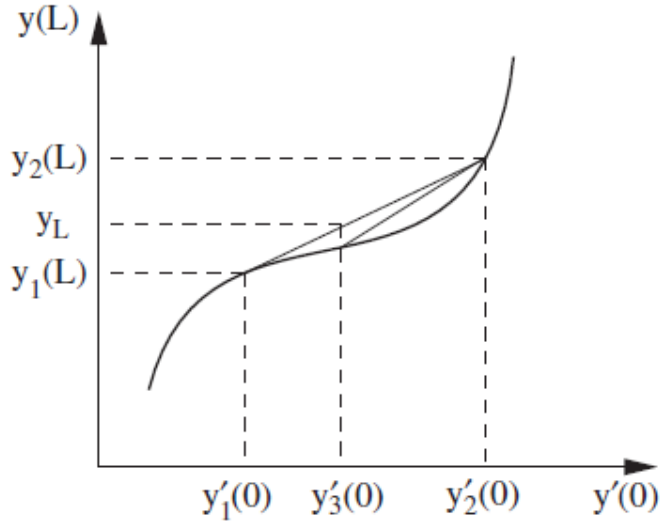
  - Let's denote two solutions of the equation as $y_1(x)$ and $y_2(x)$.
  - Using $y_1(0) = y_2(0) = y(0) = y_0$ and two different initial guesses for $y'(0)$

$$y(x) = c_1 y_1(x) + c_2 y_2(x) \qquad c_1 + c_2 = 1.$$

Next, we require that $y(L) = y_L$, which, in turn, requires that

$$c_1 y_1(L) + c_2 y_2(L) = y_L.$$

$$c_1 = \frac{y_L - y_2(L)}{y_1(L) - y_2(L)} \quad \text{and} \quad c_2 = \frac{y_1(L) - y_L}{y_1(L) - y_2(L)}.$$

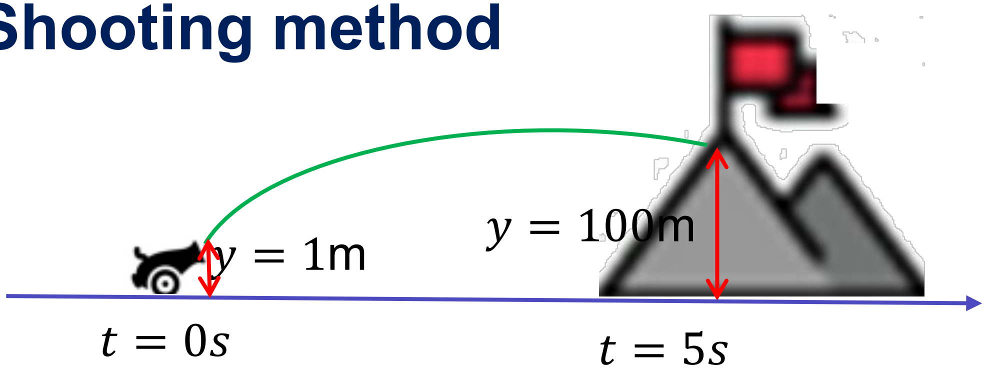


Non-linear case secant method로 root finding!!

$$y'(0) = y'_2(0) + m[y(L) - y_2(L)], \qquad m = \frac{y'_1(0) - y'_2(0)}{y_1(L) - y_2(L)}$$

# Boundary Value Problem (BVP)

- **Shooting method**



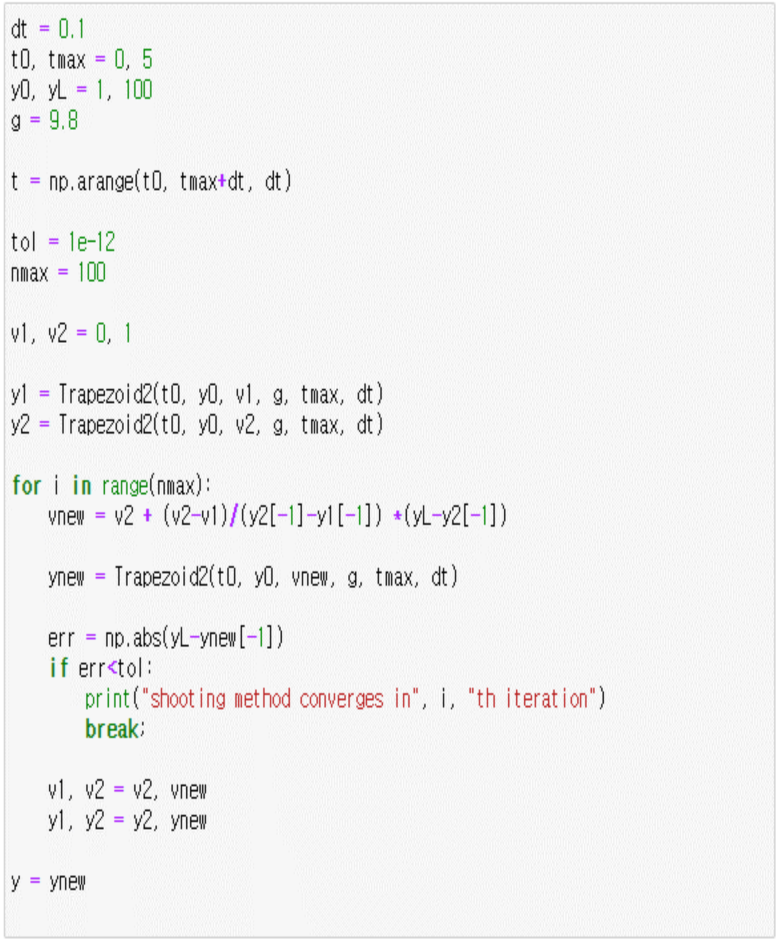
$$\frac{d^2y}{dt^2} = -g \longrightarrow \begin{array}{l} \dfrac{dy}{dt} = v \\ \dfrac{dv}{dt} = -g \end{array}$$

$$y(0) = 1, y(5) = 100$$

- **Step 1**: Choose $y'(0) = v1$, solve the initial value problem and calculate $y(L) = y1(L)$
- **Step 2**: Choose $y'(0) = v2$, solve the initial value problem and calculate $y(L) = y2(L)$
- **Step 3**: Calculate $vnew$, solve the initial value problem with $y'(0) = vnew$ and calculate $y(L) = ynew$
- **Step 4-1**: If $|ynew - y(L)| < tol$, adopt the solution from Step 3 as the numerical solution and stop the calculation.
- **Step 4-2**: If $|ynew - y(L)| > tol$, then change $\mathrm{v}2, y2$ to $\mathrm{v}1, y1$ and $vnew, ynew$ to $\mathrm{v}2, y2$ and go to Step 3

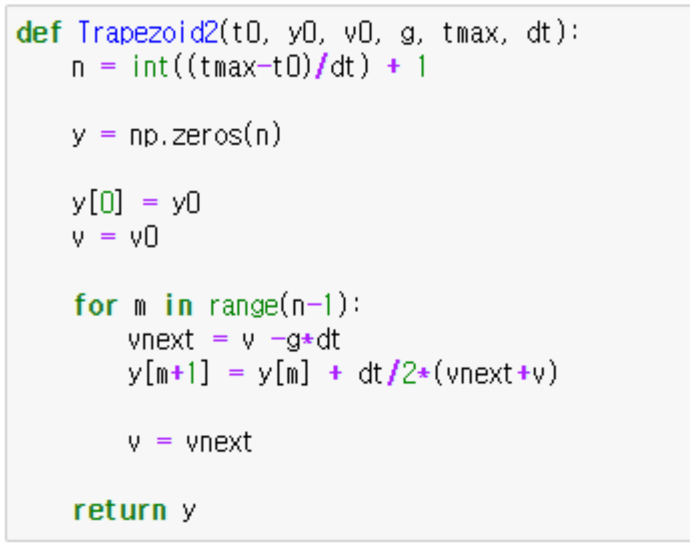
# Boundary Value Problem (BVP)

- **Shooting method**

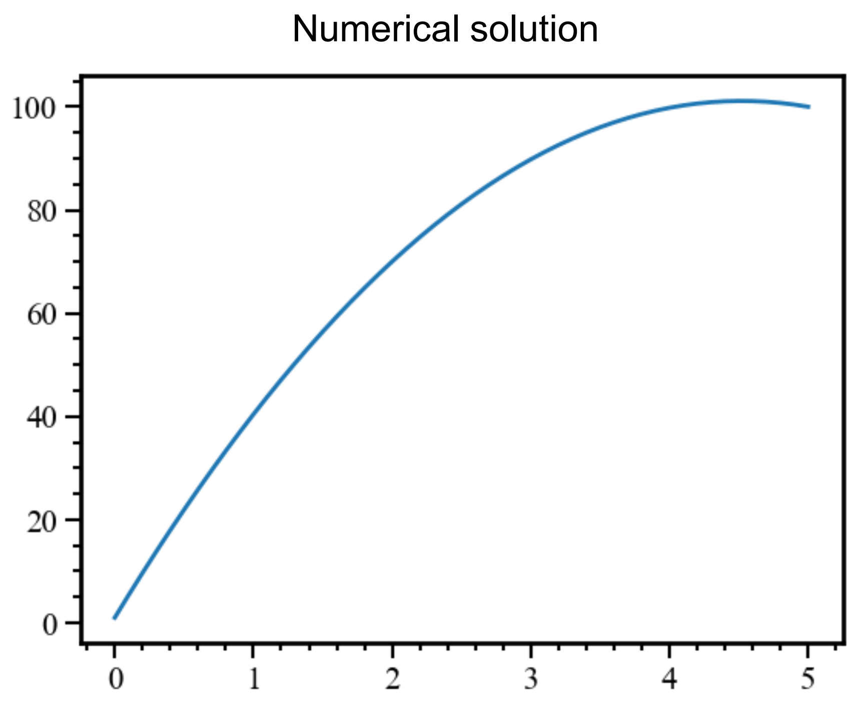


```
dt = 0.1
t0, tmax = 0, 5
y0, yL = 1, 100
g = 9.8

t = np.arange(t0, tmax+dt, dt)

tol = 1e-12
nmax = 100

v1, v2 = 0, 1

y1 = Trapezoid2(t0, y0, v1, g, tmax, dt)
y2 = Trapezoid2(t0, y0, v2, g, tmax, dt)

for i in range(nmax):
    vnew = v2 + (v2-v1)/(y2[-1]-y1[-1]) *(yL-y2[-1])

    ynew = Trapezoid2(t0, y0, vnew, g, tmax, dt)

    err = np.abs(yL-ynew[-1])
    if err<tol:
        print("shooting method converges in", i, "th iteration")
        break;

    v1, v2 = v2, vnew
    y1, y2 = y2, ynew

y = ynew
```

```
shooting method converges in 0 th iteration
```



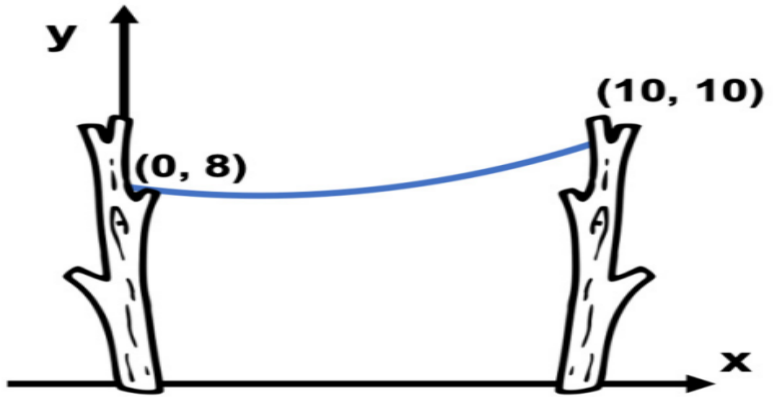
```
def Trapezoid2(t0, y0, v0, g, tmax, dt):
    n = int((tmax-t0)/dt) + 1

    y = np.zeros(n)

    y[0] = y0
    v = v0

    for m in range(n-1):
        vnext = v -g*dt
        y[m+1] = y[m] + dt/2*(vnext+v)

        v = vnext

    return y
```



# Boundary Value Problem (BVP)

- **Shooting method**
  - Non-linear case

$$\frac{d^2y}{d^2x} = C\sqrt{1+\left(\frac{dy}{dx}\right)^2}\ , C = 0.1m^{-1}$$



Using linearized trapezoid method

$$\begin{bmatrix} y' \\ s' \end{bmatrix} = \begin{bmatrix} s \\ C\sqrt{1+s^2} \end{bmatrix}$$

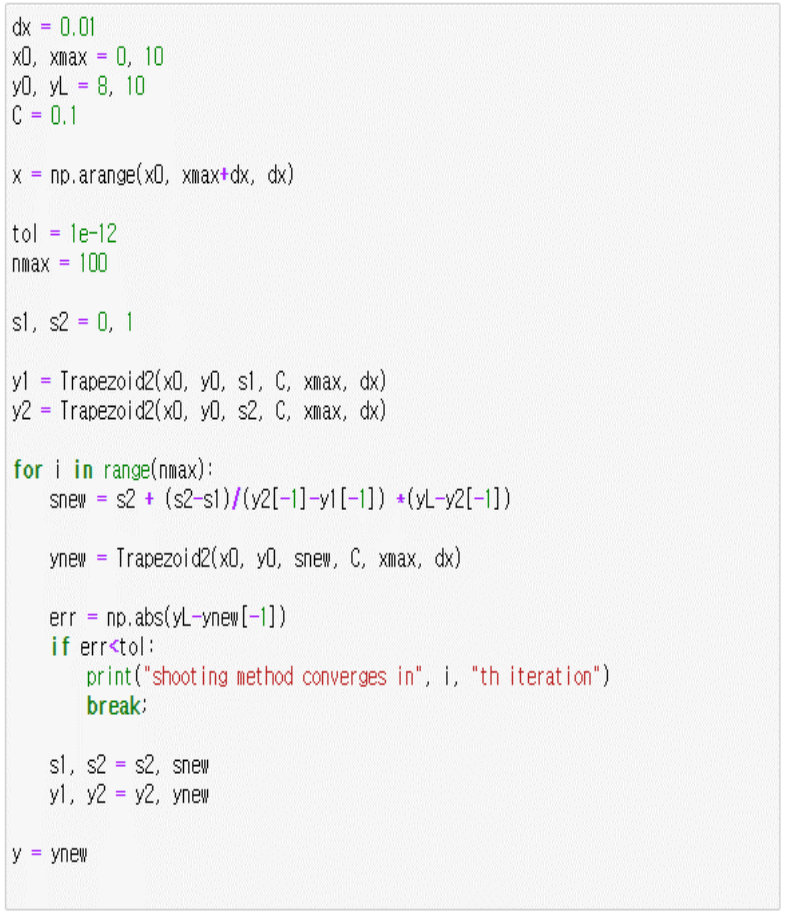
$$y_{n+1} = y_n + dx\frac{s_{n+1}+s_n}{2}$$

$$s_{n+1} = s_n + \frac{dx}{2}\left(C\sqrt{1+s_{n+1}^2} + C\sqrt{1+s_n^2}\right)$$

$$\approx s_n + \frac{Cdx\sqrt{1+s_n^2}}{1-\frac{Cdxs_n}{2\sqrt{1+s_n^2}}}$$

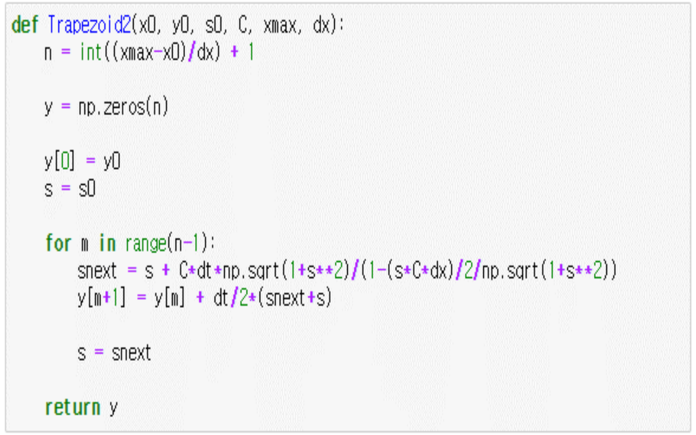
## Boundary Value Problem (BVP)

- **Shooting method**
  - Non-linear case



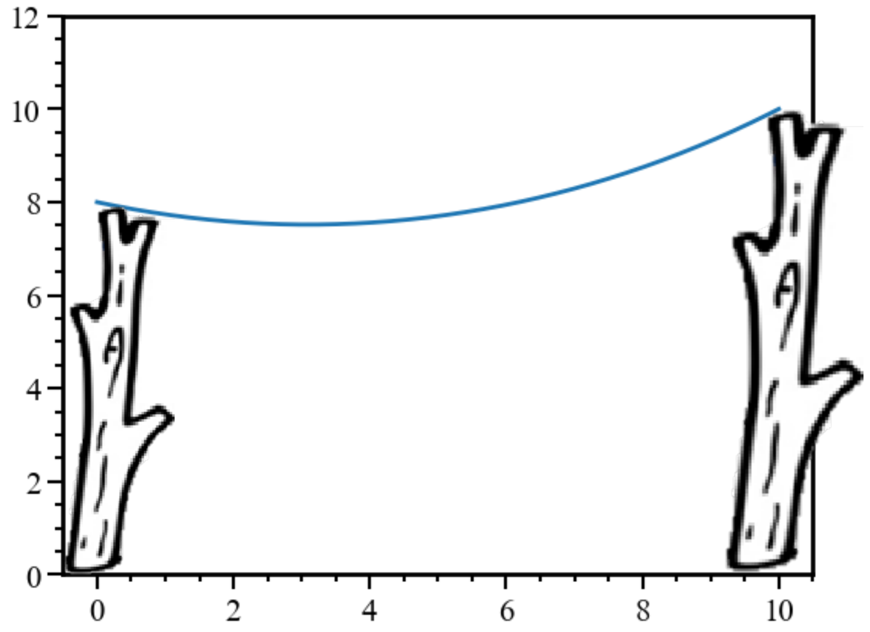
```
dx = 0.01
x0, xmax = 0, 10
y0, yL = 8, 10
C = 0.1

x = np.arange(x0, xmax+dx, dx)

tol = 1e-12
nmax = 100

s1, s2 = 0, 1

y1 = Trapezoid2(x0, y0, s1, C, xmax, dx)
y2 = Trapezoid2(x0, y0, s2, C, xmax, dx)

for i in range(nmax):
    snew = s2 + (s2-s1)/(y2[-1]-y1[-1]) *(yL-y2[-1])

    ynew = Trapezoid2(x0, y0, snew, C, xmax, dx)

    err = np.abs(yL-ynew[-1])
    if err<tol:
        print("shooting method converges in", i, "th iteration")
        break;

    s1, s2 = s2, snew
    y1, y2 = y2, ynew

y = ynew
```

```
shooting method converges in 5 th iteration
```



```
def Trapezoid2(x0, y0, s0, C, xmax, dx):
    n = int((xmax-x0)/dx) + 1

    y = np.zeros(n)

    y[0] = y0
    s = s0

    for m in range(n-1):
        snext = s + C*dt*np.sqrt(1+s**2)/(1-(s*C*dx)/2/np.sqrt(1+s**2))
        y[m+1] = y[m] + dt/2*(snext+s)

        s = snext

    return y
```

Numerical solution



# Boundary Value Problem (BVP)

- **Direct method (Finite difference method)** Linear 일 때, 품!

$$y''(x) + A(x)y'(x) + B(x)y(x) = f(x)$$

$$y(0) = y_0 \qquad y(L) = y_L.$$

Second-order approximation 공간에 대한 diffrence —> 중간에서 define

$$\frac{y_{j+1} - 2y_j + y_{j-1}}{h^2} + A_j \frac{y_{j+1} - y_{j-1}}{2h} + B_j y_j = f_j$$

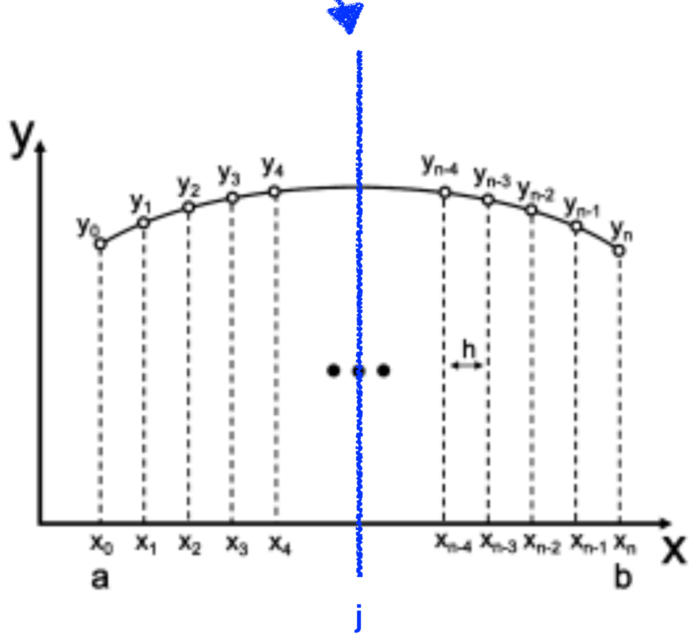
$$y_{(j=0)} = y_0 \qquad y_{(j=N)} = y_L$$ 차수를 다 맞춰야 하니까!

where a uniform grid, $x_j = x_{j-1} + h$, $j = 1, 2, \ldots, N-1$,

$$\alpha_j y_{j+1} + \beta_j y_j + \gamma_j y_{j-1} = f_j,$$ n-1개의 equation이 존재

$$\alpha_j = \left(\frac{1}{h^2} + \frac{A_j}{2h}\right) \qquad \beta_j = \left(B_j - \frac{2}{h^2}\right) \qquad \gamma_j = \left(\frac{1}{h^2} - \frac{A_j}{2h}\right)$$

$j = 1, 2, \ldots, N-1.$



# Boundary Value Problem (BVP)

- **Direct method (Finite difference method)**
  - Special treatment at the boundaries $j = 1$ and $j = N - 1$
    - ➔ $j = 1$ , $\alpha_1 y_2 + \beta_1 y_1 = f_1 - \gamma_1 \gamma_0$
    - ➔ $j = N - 1$ , $\gamma_{N-1} y_{N-2} + \beta_{N-1} y_{N-1} = f_{N-1} - \alpha_{N-1} y_N$

$$\begin{bmatrix} \beta_1 & \alpha_1 & & \\ \gamma_2 & \beta_2 & \alpha_2 & \\ & \ddots & \ddots & \ddots \\ & & \gamma_{N-1} & \beta_{N-1} \end{bmatrix} \begin{bmatrix} y_1 \\ y_2 \\ \vdots \\ y_{N-1} \end{bmatrix} = \begin{bmatrix} f_1 - \gamma_1 y_0 \\ f_2 \\ \vdots \\ f_{N-1} - \alpha_{N-1} y_N \end{bmatrix} .$$

TDMA : system of ODE 푸는 정도의 속도?

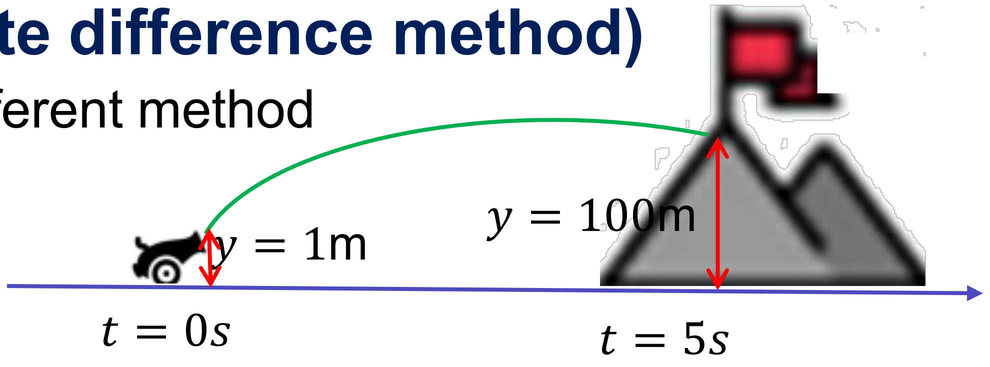
  - Implementation of mixed boundary condition

$$a y(0) + b y'(0) = g$$

$$y'(0) = \frac{-3y_0 + 4y_1 - y_2}{2h} + O(h^2)$$

## Boundary Value Problem (BVP)

- **Direct method (Finite difference method)**
  - Same problem with different method



$$\frac{d^2y}{dt^2} = -g$$

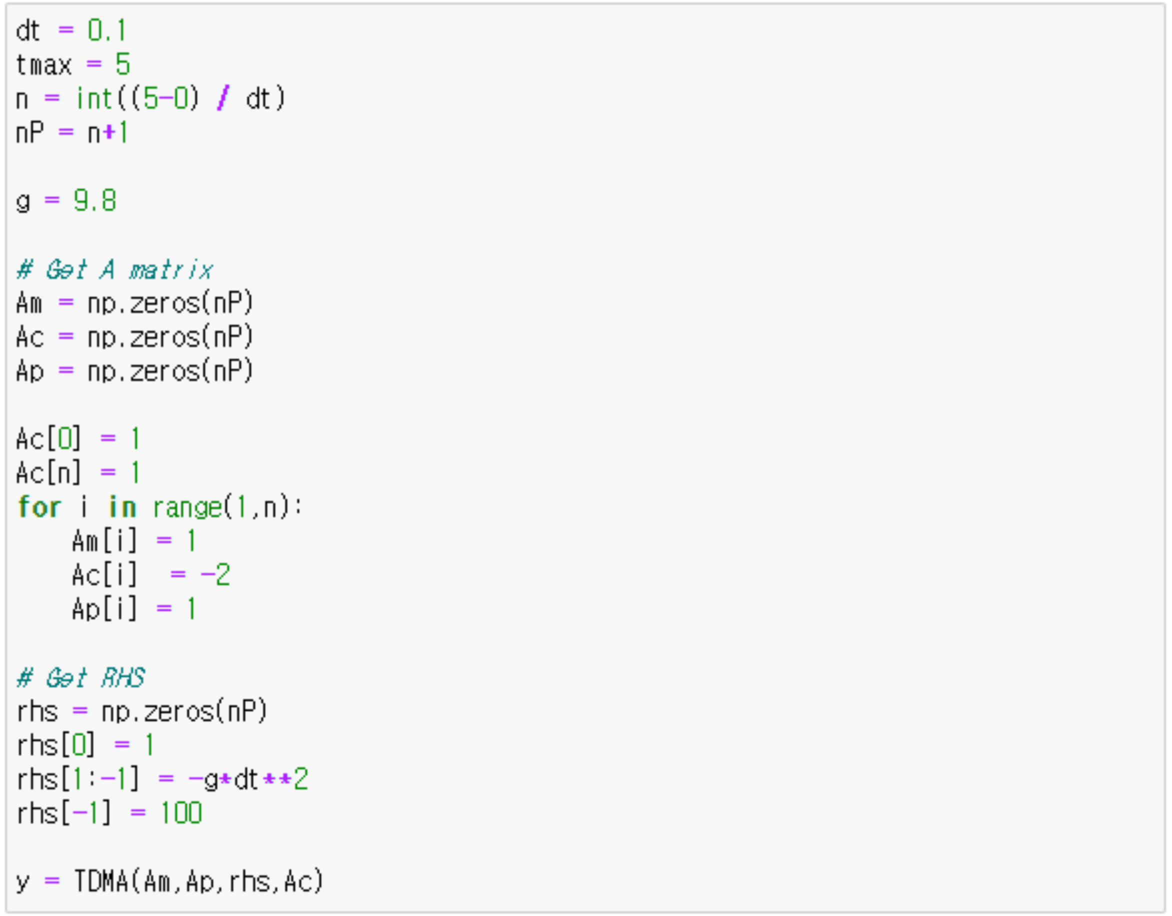
$$y(0) = 1, y(5) = 100$$

$$y_0 = 1,\ y_{i-1} - 2y_i + y_{i+1} = -gh^2,\ i = 1, 2, \ldots, n-1,\ y_n = 100$$

$$\begin{bmatrix} 1 & 0 & & & \\ 1 & -2 & 1 & & \\ & \ddots & \ddots & \ddots & \\ & & 1 & -2 & 1 \\ & & & & 1 \end{bmatrix} \begin{bmatrix} y_0 \\ y_1 \\ \vdots \\ y_{n-1} \\ y_n \end{bmatrix} = \begin{bmatrix} 1 \\ -gh^2 \\ \vdots \\ -gh^2 \\ 100 \end{bmatrix}$$

# Boundary Value Problem (BVP)

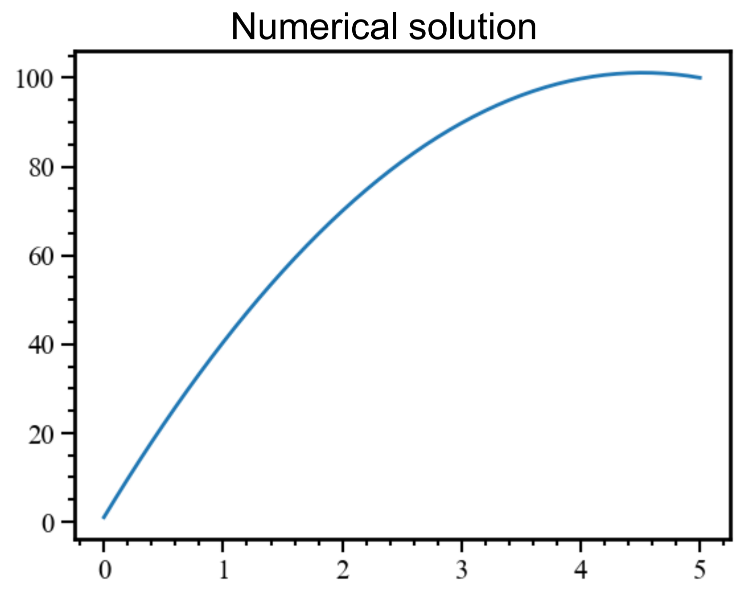
- **Direct method (Finite difference method)**



```
dt = 0.1
tmax = 5
n = int((5-0) / dt)
nP = n+1

g = 9.8

# Get A matrix
Am = np.zeros(nP)
Ac = np.zeros(nP)
Ap = np.zeros(nP)

Ac[0] = 1
Ac[n] = 1
for i in range(1,n):
    Am[i] = 1
    Ac[i]  = -2
    Ap[i] = 1

# Get RHS
rhs = np.zeros(nP)
rhs[0] = 1
rhs[1:-1] = -g*dt**2
rhs[-1] = 100

y = TDMA(Am,Ap,rhs,Ac)
```

➔ We learned TDMA in lecture6

$$\begin{bmatrix} 1 & 0 & & & \\ 1 & -2 & 1 & & \\ & \ddots & \ddots & \ddots & \\ & & 1 & -2 & 1 \\ & & & & 1 \end{bmatrix} \begin{bmatrix} y_0 \\ y_1 \\ \vdots \\ y_{n-1} \\ y_n \end{bmatrix} = \begin{bmatrix} 1 \\ -gh^2 \\ \vdots \\ -gh^2 \\ 100 \end{bmatrix}$$



Q&A Thanks for listening