수치해석
(2019학년도 1학기)

[5주/1차시 학습내용]: 이분법 (Bisection)

trial and error의 반복을 통해서 참 값에 근사하는 방법을 학습한다.

국민대학교 소프트웨어학부

Problem Statement

- 번지 점퍼 회사에서 문제가 발생했다.
- 번지 점퍼를 하는 사람들 중에 심각한 척추 부상을 당하는 경우가 다수 발생하였다.
- 그래서 번지 점퍼 회사는 여러분을 회사의 신입 사원으로 입사 시켜, 이러한 문제를 해결하도록 업무를 지시하였다.

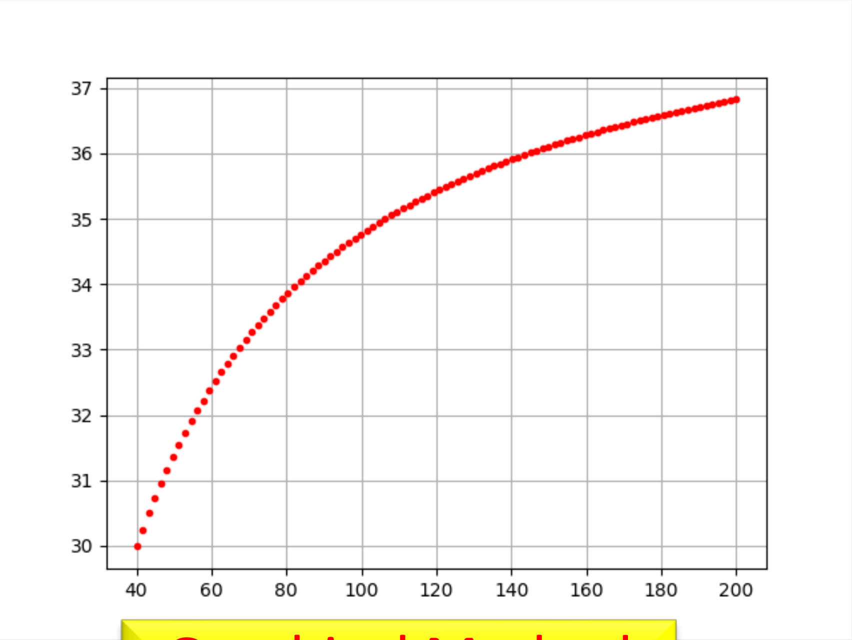
```
Problem of significant vertebrae injury if the free-fall velocity exceeds
36 m/s after 4 s of free fall.
```

수치 알고리즘의 **근사화 전략**

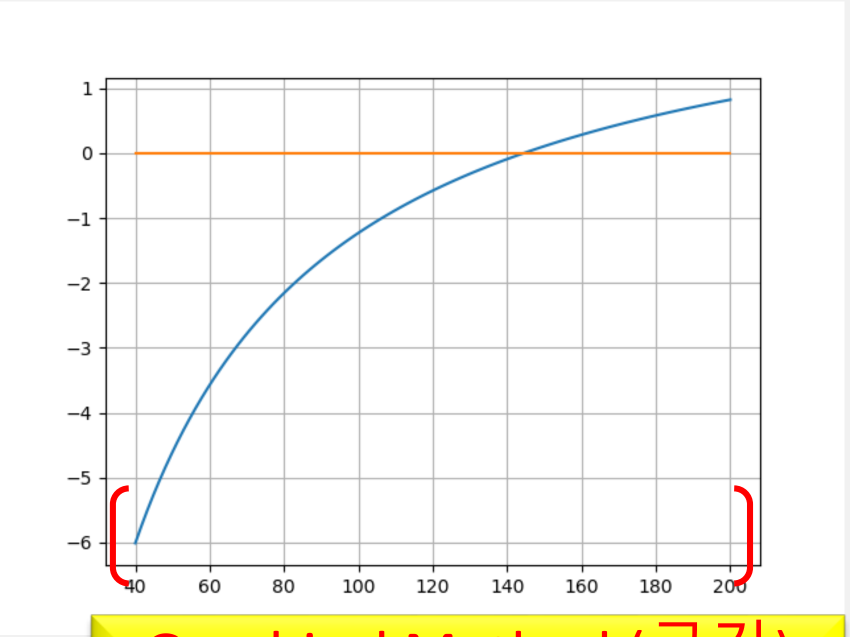
- **trial and error의 반복을 통해서 참 값에 근사한다.**
- **Bracketing Method (구간법)**
  - Graphical Method : 구간에 대한 정의 탄생
  - 증분법 : 증분점에 대한 fitting optimization 필요
  - 이분법 : 구간을 찾아내는 것이 아니라, 근을 찾아내려고 노력함
  - 가위치법 : 이분법과 같이 근을 찾아내려고 노력함
- **Open Method (개방법)**
  - 구간법에서 꼭 필요한 구간을 더 이상 사용하지 않음
  - 어떤 점도 근이 될 수 있음
  - 구간법보다 빠르게 근을 찾는 알고리즘을 제공함

Graphical Method $f(x) = 0$

- 구간에 대한 의미를 파악할 수 있다.
- 두 번 정도의 그래프를 그려서 0을 지나게 하였다
  - Trial and Error 방법을 통해 그래프가 0을 지나게 하였다
- 그래프가 0을 지나는 점을 가지는 구간이 중요하다



Graphical Method

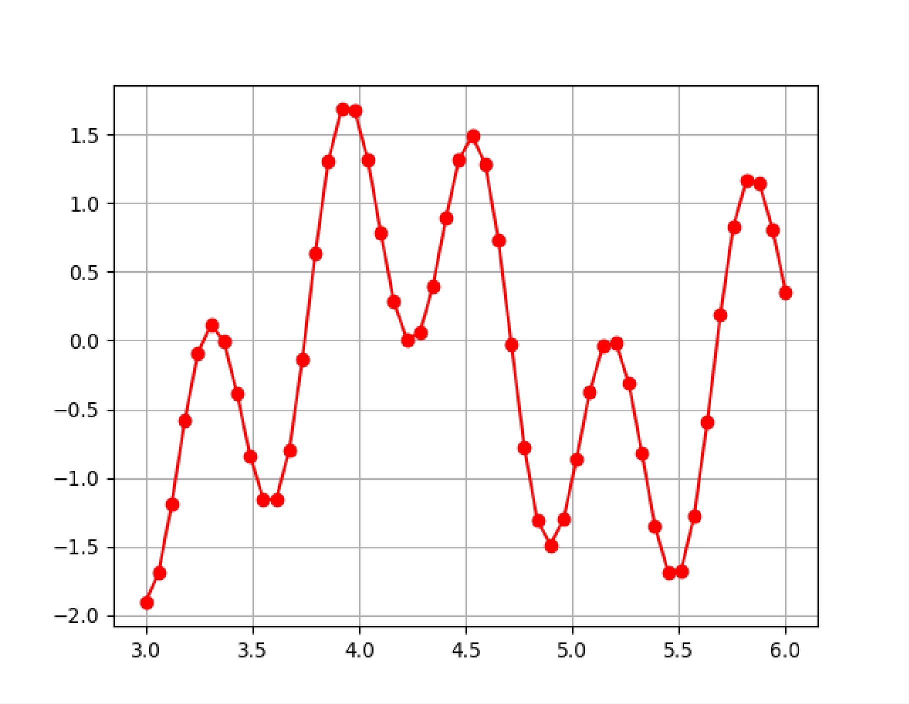


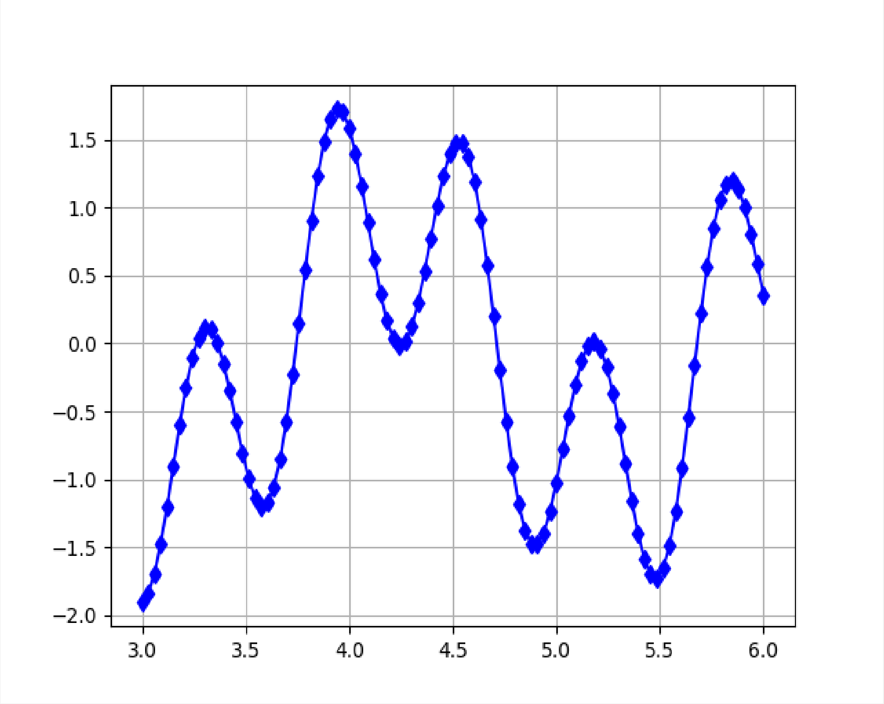
Graphical Method (구간)

증분법 (Fitting 전략) $f(x) = 0$

- Too Small number of incremental length (증분 수가 너무 적음)과 Too many number of incremental length (증분 수가 너무 많음) 사이의 선택이 있다.
- Under fitting: 데이터를 표현할 수 있는 증분 수가 너무 적어서, 존재하는 해를 제대로 구해내지 못하는 현상
  - Too Small number of incremental length (증분 수가 너무 적음)
- Normal fitting: 데이터를 표현할 수 있는 증분 수가 적절 하여, 존재하는 해를 모두 구하는 상황
- Over fitting: 존재하는 해를 모두 구하는 데 필요한 증분 수가 필요 이상으로 너무 많아, 컴퓨팅 리소스를 낭비하게 되는 상황
  - Too many number of incremental lengths (증분 수가 너무 많음)

증분법 (Fitting 전략)





근사화 전략 : Incremental search (증분법)

```
import numpy as np
def incsearch(func, xmin, xmax):
    x=np.arange(xmin, xmax+1)  #
    np.linspace(xmin, xmax, ns)
    f=func(x)
    nb=0
    xb=[]
    for k in np.arange(np.size(x)-1):
        if np.sign(f[k]) != np.sign(f[k+1]):
            nb=nb+1
            xb.append(x[k])
            xb.append(x[k+1])
    return nb, xb

if_name_== '_main_':
    g=9.81; m=68.1; cd=0.25; v=36; t=4;
    fp=lambda mp:
np.sqrt(g*np.asarray(mp)/cd)*np.tanh(np.sqrt(g*cd/np.asarray(mp))*t)-v
    nb, xb=incsearch(fp, 40, 200)
    print('number of brackets= ', nb)
    print('root interval=', xb)
```

```
number of brackets=  1
root interval= [142, 143]
```

Debugging in Incremental search (증분법)

- 증분법은 k가 102 정도 일 때 근사 구간 찾게 됨, 이분법은 5번 정도에서 근사근을 찾게 됨.
- 증분법은 이분법보다 계산량이 많다.

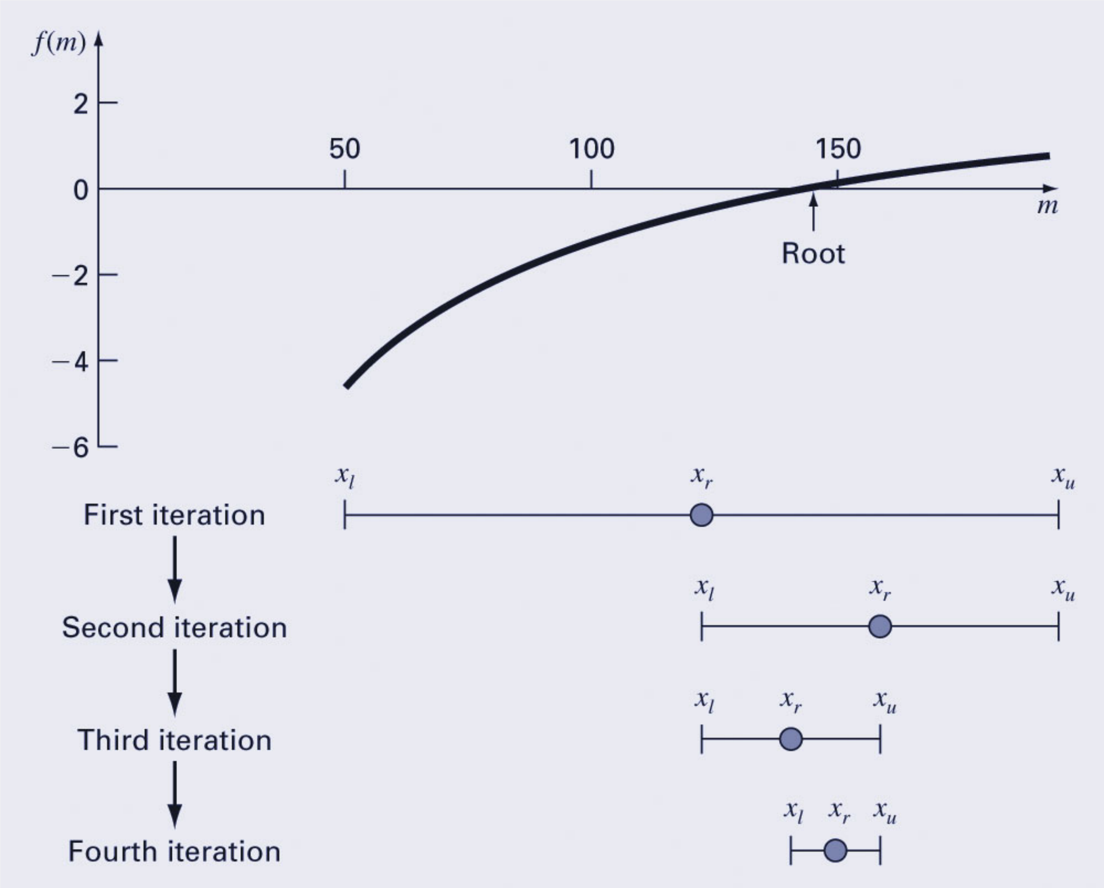
```
import numpy as np

def incsearch(func, xmin, xmax):   xmin: 40   xmax: 200
    x=np.arange(xmin, xmax+1)   x: [ 40   41   42   43   44   45   46   47   48
    #np.linspace(xmin, xmax, ns)
    f=func(x)   f: [-5.98701054e+00 -5.82380001e+00 -5.66627460e+00 -5.5
    nb=0   nb: 0
    xb=[]   xb: <class 'list'>: []

    for k in np.arange(np.size(x)-1):   k: 102
        if np.sign(f[k]) != np.sign(f[k+1]):
            nb=nb+1
            xb.append(x[k])
            xb.append(x[k+1])

    return nb, xb


g=9.81; cd=0.25; v=36; t=4;
```

Bisection (이분법) $f(x) = 0$

- 이분법부터는 근이 존재하는 구간을 찾아내는 것이 아니라, 이제는 근을 찾아낸다.
- https://github.com/SCKIMOSU/Numerical-Analysis/blob/master/bisection.py

Bisection (이분법)

- The *bisection method* is a variation of the incremental search method in which the interval is always divided in half.
- If a function changes sign over an interval, the function value at the midpoint is evaluated.
- The location of the root is then determined as lying within the subinterval where the sign change occurs.
- The absolute error is reduced by a factor of 2 for each iteration.



Bisection (이분법) 알고리즘

- 1: x_l 과 x_u 의 구간을 정한다
  - $x_l = 40$, $x_u = 200$
- 2: $f(x_l)$의 값과 $f(x_u)$의 값의 부호가 변하면 구간으로 정한다
- 3: 이분법에서는 구간이 정해지면, $x_r = \frac{(x_l+x_u)}{2}$ 을 새로운 근이라고 정한다 ($x_r = 120$)
- 4. $f(x_l)$의 값과 $f(x_r)$의 값의 부호 변화를 체크한다
  - 변하지 않으면, $x_l = x_r$로 대체함
  - 변하면, $x_u = x_r$로 대체함
- 5. 단계 1로 간다

Bisection (이분법) 코드

```python
import numpy as np

def bisect(func, xl, xu):
    maxit=100
    es=1.0e-4

    test=func(xl)*func(xu)

    if test > 0:
        print('No sign change')
        return [], [], [], []

    iter=0
    xr=xl

    ea=100

    while (1):
        xrold=xr
        xr=np.float((xl+xu)/2)

        iter=iter+1
```

Bisection (이분법) 코드

```
if xr != 0:
        ea=np.float(np.abs(np.float(xr)-np.float(xrold))/np.float(xr))*100

    test=func(xl)*func(xr)

    if test > 0:
        xl=xr
    elif test < 0:
        xu=xr
    else:
        ea=0

    if np.int(ea<es) | np.int(iter >= maxit):
        break

root=xr
fx=func(xr)

return root, fx, ea, iter
```

Bisection (이분법) 코드

```
if __name__== '__main__':
    fm=lambda m: np.sqrt(9.81*m/0.25)*np.tanh(np.sqrt(9.81*0.25/m)*4)-36
    root, fx, ea, iter=bisect(fm, 40, 200)

    print('root = ', root, '(Bisection)')
    print('f(root) = ', fx, '(must be zero, Bisection)')
    print('estimated error= ', ea, '(must be zero error, Bisection)')
    print('iterated number to find root =', iter, '(Bisection)')
```

```
root = 142.73765563964844(Bisection)
f(root) = 4.608913576288077e-07(must be zero, Bisection)
estimated error = 5.3450468252827136e-05(must be zero error, Bisection)
iterated number to find root = 21(Bisection)
```

fsolve 실제 근 구해 보기

- fsolve () 함수
  - $f(x) = 0$ 으로 만드는 실제 근 x 는 scipy 라이브러리 함수

```
import numpy as np
from scipy.optimize import fsolve

fm=lambda m: np.sqrt(9.81*m/0.25)*np.tanh(np.sqrt(9.81*0.25/m)*4)-36
m=fsolve(fm, 1)
print("Real Root= ", m)
```

```
Real Root=  [142.73763311]
```

https://github.com/SCKIMOSU/Numerical-Analysis/blob/master/real_root.py

np.matrix() 메소드와 np.array() 메소드에 대해

- 행렬 연산 중 내적, 외적 연산은 np.matrix() 메소드로 수행
- 행렬 연산 중 element by element 연산은 np.array() 메소드로 수행 : 수치해석에서 많이 사용

```
a=np.matrix([1,2,3])
b=np.matrix([4,5,6])

np.dot(a, b)

   np.dot(a, b)
ValueError: shapes (1,3) and (1,3) not aligned: 3 (dim 1) != 1 (dim 0)


b.transpose()

bt=b.transpose()
matrix([[4],
        [5],
        [6]])
np.dot(a, bt)

matrix([[32]])


a1=np.array([1,2,3])
b1=np.array([4,5,6])
np.dot(a1, b1)
32
```

유용한 함수 np.where()

- np.where(v>36)
- np.where(m==102.0)