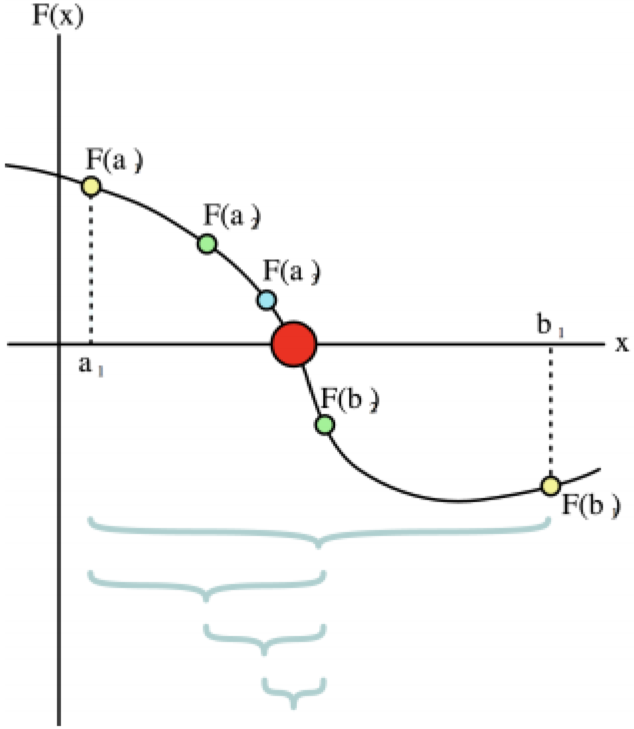
Root Finding

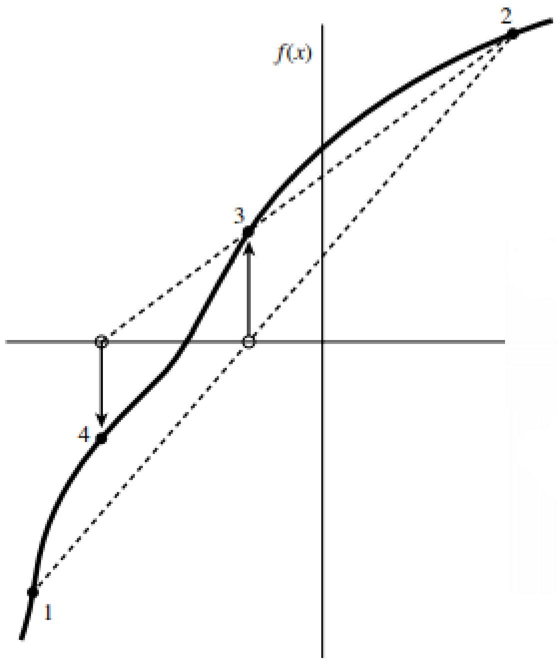
2017135002/최성윤

기본원리

Bisection Method



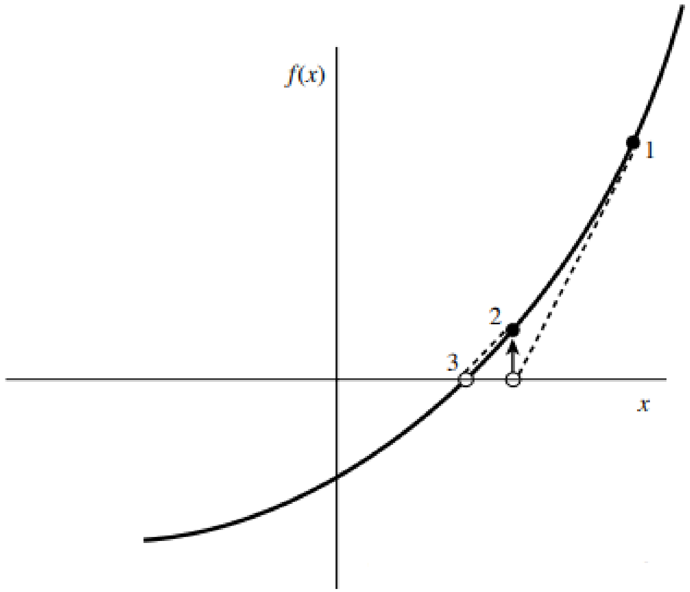
Secant Method



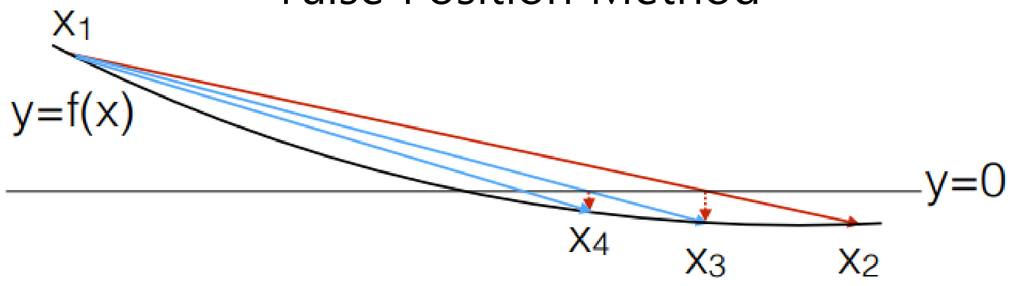
$$x_{n+1} = x_n - f(x_n) \cdot \frac{x_n - x_{n-1}}{f(x_n) - f(x_{n-1})}$$

Newton-Raphson Method

$$x_{n+1} = x_n - \frac{f(x_n)}{f'(x_n)}$$



False Position Method



실행

Bisection Method

```python
def bis(a,b):
    c=(a+b)/2
    N=1 # c=(a+b)/2 으로 이미 한번 bisection 한번 실행한것
    while not fun(c)< 10e-12 or not fun(c)>  -10e-12:
        N+=1
        if fun(a)*fun(c)<0:
            b=c
            c=(a+b)/2
            continue
        if fun(b)*fun(c)<0:
            a=c
            c=(a+b)/2
            continue
    return c, N
```

False Position Method

```python
def false(a,b):
    c=a-fun(a)*(b-a)/(fun(b)-fun(a))
    N=1
    while not fun(c) < 10e-12 or not fun(c) > -10e-12:
        a=c
        c=a-fun(a)*(b-a)/(fun(b)-fun(a))
        N+=1
    return c, N
```

Secant Method

```python
def secant(a,b):
    c=a-fun(a)*(b-a)/(fun(b)-fun(a))
    N=1
    while not fun(c) < 10e-12 or not fun(c) > -10e-12:
        N+=1
        a=b
        b=c
        c=a-fun(a)*(b-a)/(fun(b)-fun(a))
    return c, N
```

Newton-Raphson Method

```python
def newton(a):
    b=a-fun(a)/derfun(a)
    N=1
    while not fun(b) < 10e-12 or not fun(b) > -10e-12:
        a=b
        b=a-fun(a)/derfun(a)
        N+=1
    return b, N
```

결과

Bisection Method

```
(1.448426645384643, 40)
```

Secant Method

```
(1.448426645384929, 7)
```

False Position Method

```
(1.4484266453846186, 74)
```

Newton-Raphson Method

```
(1.4484266453853585, 7)
```

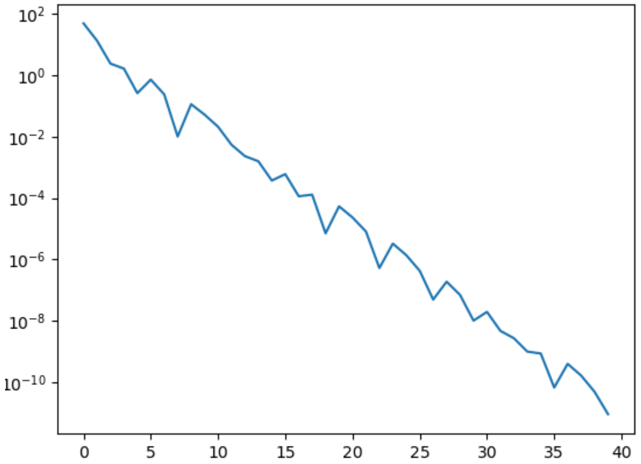
4가지 방법 모두 결과값은
동일한 값으로 convergence 된다.

하지만 step size에서 큰 차이를 보인다.
(convergence rate 의 차이?)

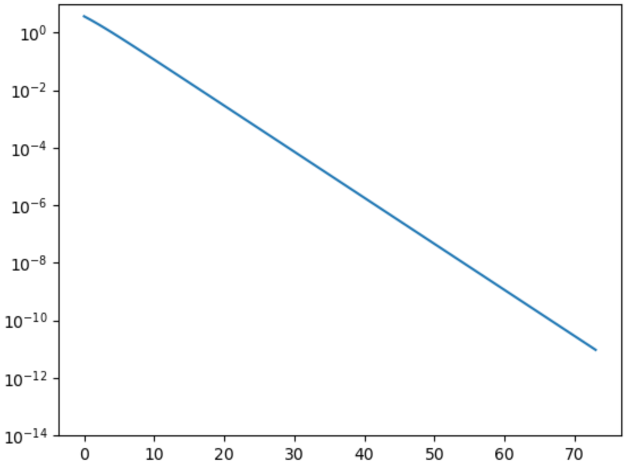
토의사항

Convergence rate 비교

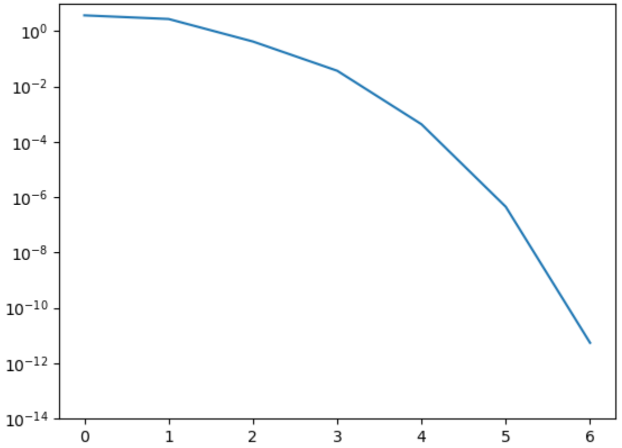
Bisection Method



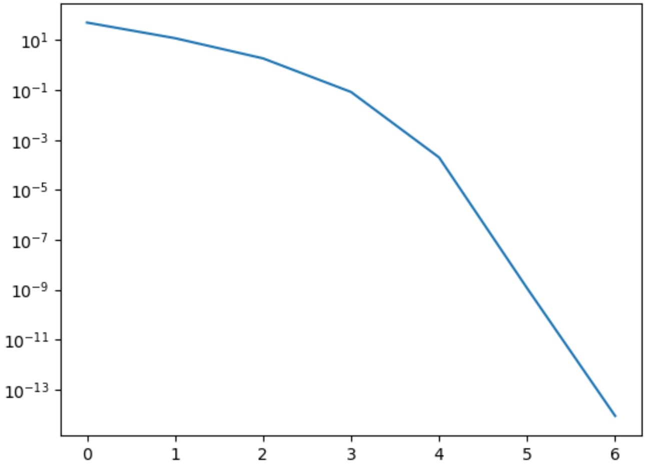
False Position Method



Secant Method



Newton-Raphson Method



Bisection 와 False Position 의 경우
Linear 한 speed를 보이며

Secant와 Newton의 경우
quadratic 한 speed를 보인다

토의사항

Convergence rate 비교

$$\lim_{k\to\infty} \frac{|x_{k+1} - L|}{|x_k - L|} = \mu$$

을 통하여 convergence rate를 더 정밀하게 비교해 보았다

Bisection Method

```
0.415445041401801, 0.2964606057995671, 0.1865465639982566]
```

위의 값은 $\frac{|x_{k+1} - L|}{|x_k - L|}$ 에서 k가 끝지점 부근에서의 값이다.

0~1의 값을 가지므로 linear 하다

False Position Method

```
0.6918521289196434, 0.6919264732384214, 0.6917567391587446]
```

동일하게 0~1의 값을 가지므로 linear 하다

토의사항

Convergence rate 비교

Secant Method

```
0.0010484159140869082, 1.2218281902317726e-05]
```

Step이 끝날 때즘에 거의 0에 가까운 값을 가진다.
Superlinearly에 가깝다

Newton-Raphson Method

```
0.0023897246030961366, 5.759273810260851e-06, 7.656018166199904e-06]
```

Step이 끝날 때즘에 거의 0에 가까운 값을 가진다.
Superlinearly에 가깝다

더 자세한 비교를 위하여

$$\lim_{k\to\infty} \frac{|x_{k+1} - L|}{|x_k - L|^q} < M$$

를 통하여 비교해보았다.

$$M = \frac{f''(\alpha)}{2f'(\alpha)},$$

Secant Method

```
[1.75433421193888]
```

Newton-Raphson Method

```
[2.29922318833455]
```

Newton의 경우 q가 2에 가까운 값을 가져
quadratic convergence 에 가까움을 확인할수 있다.

하지만 Secant의 경우 q가 2에 미치지 못하여 (golden ration에 근사)
quadratic convergence 하지 않음을 확인할수 있다.

기본원리

Broyden Mehtod

$$\mathbf{J}_n = \mathbf{J}_{n-1} + \frac{\Delta\mathbf{f}_n - \mathbf{J}_{n-1}\Delta\mathbf{x}_n}{\|\Delta\mathbf{x}_n\|^2}\Delta\mathbf{x}_n^{\mathrm{T}}$$

$$\mathbf{x}_{n+1} = \mathbf{x}_n - \mathbf{J}_n^{-1}\mathbf{f}(\mathbf{x}_n)$$

Newton-Raphson in Multiple Dimension

$$\mathbf{J}_n\Delta\mathbf{x}_n \simeq \Delta\mathbf{f}_n$$

$$\mathbf{x}_{n+1} = \mathbf{x}_n - \mathbf{J}_n^{-1}\mathbf{f}(\mathbf{x}_n)$$

실행

Newton-Raphson in Multiple Dimension

```python
def nr(a,b):
    X1=[]
    X2=[]
    N=[]
    for i in range(1000):
        T=[[a[i]],[b[i]]]
        n=0
        try:
            while n<100:
                np.seterr(over='raise')
                n+=1
                if abs(fun1(T[0][0], T[1][0])) < 10e-12 and abs(fun2(T[0][0], T[1][0])) < 10e-12:
                    N.append(n)
                    X1.append(T[0][0])
                    X2.append(T[1][0])
                    break
                else:
                    T=T-np.dot(invjac(T[0][0],T[1][0]),[[fun1(T[0][0],T[1][0])],[fun2(T[0][0],T[1][0])]])
```

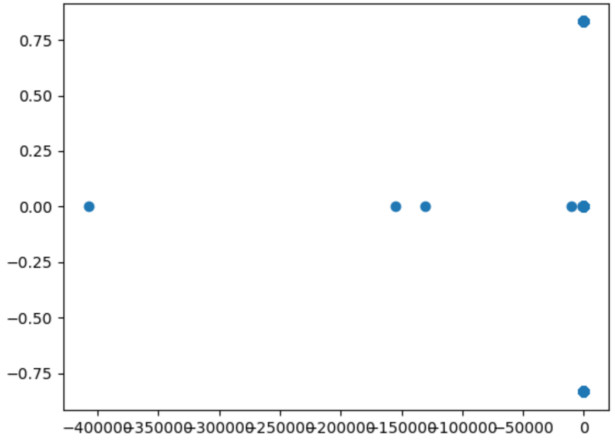
Broyden Mehtod

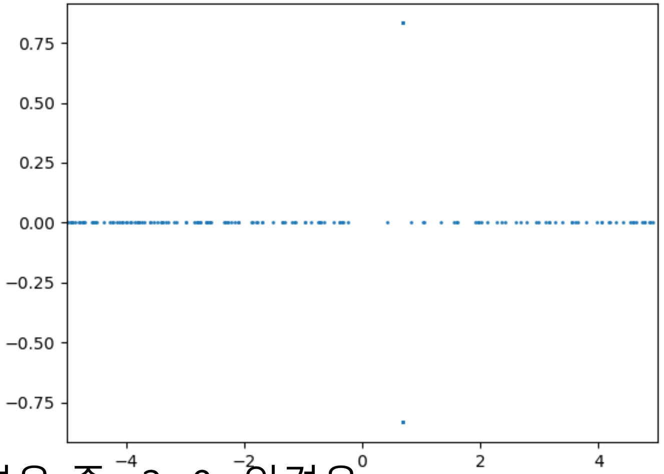
```python
def bro(a,b):
    X1=[]
    X2=[]
    N=[]
    for i in range(1000):
        T =[[a[i]],[b[i]]]
        J=jac(T[0][0],T[1][0])  # 첫 자코비안 저장
        Tn=T-np.dot(np.linalg.inv(J),[[fun1(T[0][0], T[1][0])],[fun2(T[0][0], T[1][0])]]) #그다음 X2 생성
        n=1
        try:
            while n<100:
                np.seterr(over='raise')
                n+=1
                if abs(fun1(Tn[0][0], Tn[1][0])) < 10e-12 and abs(fun2(Tn[0][0], Tn[1][0])) < 10e-12:
                    N.append(n)
                    X1.append(Tn[0][0])
                    X2.append(Tn[1][0])
                    break
                else:
                    df=[fun1(Tn[0][0], Tn[1][0])-fun1(T[0][0], T[1][0]), fun2(Tn[0][0], Tn[1][0])-fun2(T[0][0], T[1][0])]
                    dx=[Tn[0][0]-T[0][0],Tn[1][0]-T[1][0]] # a,b의 변화량
                    sizedx=(Tn[0][0]-T[0][0])**2+(Tn[1][0]-T[1][0])**2  # 변화량 크기의 제곱
                    J=J+ np.outer((df-np.dot(J, dx)),dx)/sizedx   # 새로운 자코비안 저장
                    T=Tn
                    Tn=T-np.dot(np.linalg.inv(J),[[fun1(T[0][0], T[1][0])],[fun2(T[0][0], T[1][0])]])
        except:
            pass
    return X1,X2,N
```

Trivial solution (x2=0) 인 경우로 인하여
x1의 overflow를 무시하기 위하여
try, except 문 사용

결과

Newton-Raphson in Multiple Dimension





Trivial 인 경우 즉 x2=0 인경우
x1은 무한정으로 작아질수 있다.
-> 범위 -5~5로 조정

solution

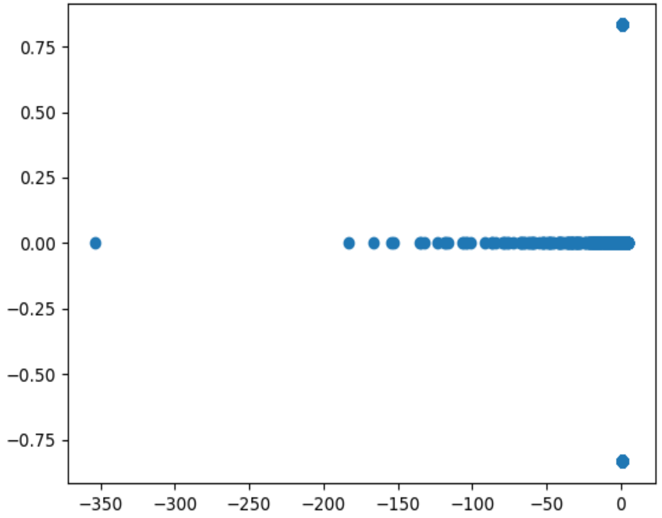
X1=0.69314718 , X2=0.83255461

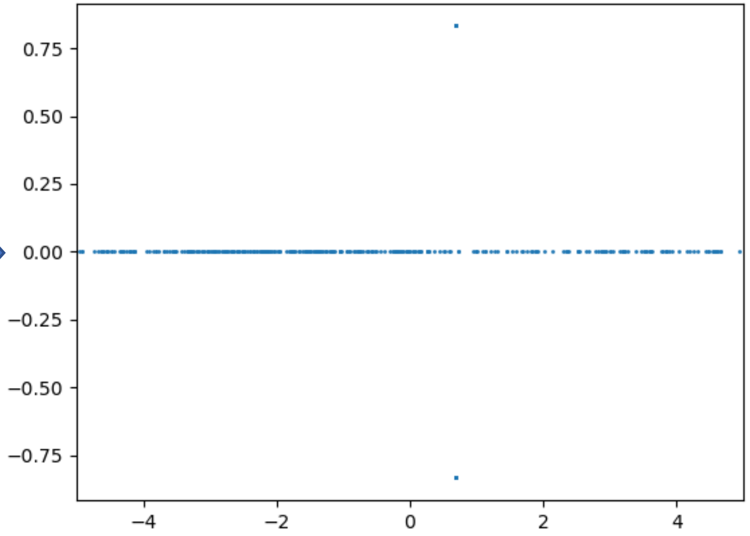
X1=0.69314718 , X2=-0.83255461

X2=0

N(step) : 약 10

Broyden Mehtod





solution

X1=0.69314718 , X2=0.83255461

X1=0.69314718 , X2=-0.83255461

X2=0

N(step) : 약 15

Solution의 값은 동일하지만
Broyden 경우가 step이 더 소요된다

Reference

THE ORDER OF CONVERGENCE FOR THE SECANT METHOD/Grinsphan