Lecture 4

프로그래밍....디버깅. 의사 코드

Jung-Il Choi
School of Mathematics and Computing (Computational Science and Engineering)
YONSEI UNIVERSITY

1. 프로그래밍

- **좋은 프로그램이란?** 답 나왔을 때, 재검토하라!
  - 정확성: 수치적인 결과를 다루기 때문에, 프로그램이 정확한 값을 계산해야 한다.
    Validation vs. Verification
  - 성능: 대량의 데이터나 복잡한 계산을 다루기 때문에, 프로그램이 빠르고 효율적으로 처리할 수 있어야 한다.
  - 재사용성: 다양한 문제에 동일한 알고리즘이 사용되기 때문에, 프로그램이 모듈화되어 다른 문제에서도 쉽게 재사용될 수 있어야 한다.
  - 확장성: 문제가 복잡해질수록 더 많은 계산 자원이 필요하게 되므로, 프로그램이 대규모 분산 시스템에서도 동작할 수 있도록 설계되어야 한다.
  - 유지보수성: 복잡한 수식과 알고리즘이 사용되기 때문에, 코드의 가독성과 유지보수성이 높아야 다른 연구자나 개발자가 이해하고 수정할 수 있다.

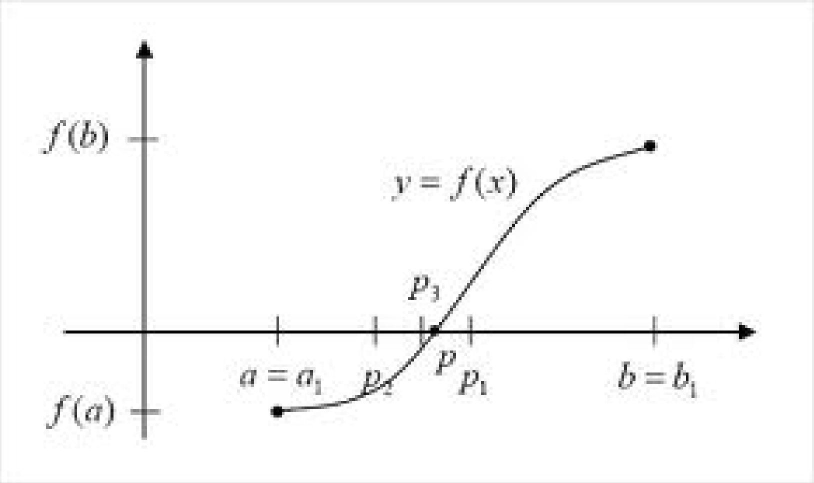
1. 프로그래밍

- **좋은 프로그래밍 방법** 알고리즘 이해가 첫번째
  - 프로그램에 대한 계획을 정확히 세우고 작성할 것
    - ➔ 문제에 대한 정확한 이해와 알고리즘 구현의 구체적인 계획이 없으면 프로그램 작성 이후 오류를 수정하는 데에 더 많은 시간과 노력이 들어가게 된다.
    - ➔ 계획하는 데에는 해결하려는 문제의 개요부터 시작해 모듈을 어떻게 구성할 것인가, 어떤 순서로 프로그램이 실행되는가를 포함한다.
    - ➔ 프로그램 계획은 전체적인 것부터 시작하고, 코드의 작성은 구체적인 함수부터 시작해 전체 프로그램을 완성하는 것이 좋다.
    - ➔ 프로그램 계획에 사용되는 방법에 의사코드(pseudo code) 작성이 있다.

      ✓ 의사코드(Pseudo code)
      - 실제 프로그래밍 언어가 아닌, 프로그램의 구조와 흐름을 설명하기 위한 비공식적인 코드 표현 방식.
      - 실제 코드 작성 전에 프로그램의 구조와 흐름을 설계하고 검증하는 데 도움이 된다.
      - 작성할 코드를 더 쉽게 이해하고 유지보수가 가능하다.
      - 프로그램 설계 과정에서 발생하는 오류를 미리 파악할 수 있다.

1. 프로그래밍

- 의사코드의 예시) bisection method




```
program Test_Bisection
integer n, nmax ← 20
real a, b, ε ← ½10^-6
external function f, g
a ← 0.0
b ← 1.0
call Bisection(f, a, b, nmax, ε)
a ← 0.5
b ← 2.0
call Bisection(g, a, b, nmax, ε)
end program Test_Bisection

real function f(x)
real x
f ← x^3 − 3x + 1
end function f

real function g(x)
real x
g ← x^3 − 2 sin x
end function g
```

```
procedure Bisection(f, a, b, nmax, ε)
integer n, nmax;    real a, b, c, fa, fb, fc, error
fa ← f(a)
fb ← f(b)
if sign(fa) = sign(fb) then
    output a, b, fa, fb
    output "function has same signs at a and b"
    return
end if
error ← b − a
for n = 0 to nmax do
    error ← error/2
    c ← a + error
    fc ← f(c)
    output n, c, fc, error
    if |error| < ε then
        output "convergence"
        return
    end if
    if sign(fa) ≠ sign(fc) then
        b ← c
        fb ← fc
    else
        a ← c
        fa ← fc
    end if
end for
end procedure Bisection
```

1. 프로그래밍

- ## 좋은 프로그래밍 방법
  - 오류가 있는지 모듈별로 확인하며 작성할 것
    - ➔ 모듈마다 예외까지 포함해 오류가 있는 지 충분히 확인하고 다음 모듈을 작성하는 것이 좋다.
    - ➔ 현재 모듈이 오류가 생겼을 때 호출한 모듈이 오류가 있다면 오류를 찾는 것으로도 많은 시간을 소비하게 된다.
  - 코드를 항상 보기 좋게 작성할 것
    - ➔ 다음은 가독성을 높이는 몇가지 예시이다.
      - ✓ 명령어(instruction)을 적게 사용

```
In [1]: y = x**2 + 2*x + 1
```

```
In [2]: y = x**2
        y = y + 2*x
        y = y + 1
```

      - ✓ n을 사용하여 가독성을 높이고 수정을 쉽게 함

```
In [4]: import numpy as np

        n = 10
        s = 0
        a = np.random.rand(n)
        for i in range(n):
            s = s + a[i]
```

```
In [3]: import numpy as np

        s = 0
        a = np.random.rand(10)
        for i in range(10):
            s = s + a[i]
```

      - ✓ 기본함수가 있다면 적극사용

```
In [5]: s = sum(np.random.rand(10))
```

2. 디버깅

- **에러의 종류**
  - 기본 에러의 종류로는 Syntax error, runtime error와 logical error가 있다.
  - Syntax error: 프로그램 코드를 작성하면서 문법적으로 잘못된 부분이 있는 경우 발생하는 오류

```
In [6]: 1 = x

  File "C:₩Users₩MPMC₩AppData₩Local₩Temp₩ipykernel_3528₩3270958869.py", line 1
    1 = x
    ^
SyntaxError: cannot assign to literal
```

```
In [7]: print("Hello, world!)

  File "C:₩Users₩MPMC₩AppData₩Local₩Temp₩ipykernel_3528₩3760663132.py", line 1
    print("Hello, world!)
                         ^
SyntaxError: EOL while scanning string literal
```

```
In [8]: if x in range(10)
            print(x)

  File "C:₩Users₩MPMC₩AppData₩Local₩Temp₩ipykernel_3528₩657364628.py", line 1
    if x in range(10)
                     ^
SyntaxError: invalid syntax
```

2. 디버깅

- **에러의 종류**
  - runtime error: 코드가 실행될 때 발생하는 에러

➔ 자주 발생하는 runtime error에는 다음과 같은 경우가 있다.

1. TypeError: 변수 또는 인수의 형(type)이 올바르지 않은 경우
2. NameError: 정의되지 않은 변수를 참조하려고 하는 경우
3. ZeroDivisionError: 0으로 나누는 경우

➔ TypeError의 예시

```
In [10]: num = 123
         str = '456'
         result = num + str
         print(result)

---------------------------------------------
---------------------------------
TypeError                                 Traceb
ack (most recent call last)
~\AppData\Local\Temp\ipykernel_3528\1388497
210.py in <module>
      1 num = 123
      2 str = '456'
----> 3 result = num + str
      4 print(result)

TypeError: unsupported operand type(s) for +: 'i
nt' and 'str'
```

2. 디버깅

- **에러의 종류**
  - runtime error: 코드가 실행될 때 발생하는 에러

    ➔ 자주 발생하는 runtime error에는 다음과 같은 경우가 있다.

    1. TypeError: 변수 또는 인수의 형(type)이 올바르지 않은 경우
    2. NameError: 정의되지 않은 변수를 참조하려고 하는 경우
    3. ZeroDivisionError: 0으로 나누는 경우

    ➔ NameError의 예시

```
In [12]: print(A)

---------------------------------------------------------------------------
NameError                                 Traceback (most recent call last)
~\AppData\Local\Temp\ipykernel_3528\212384544.py in <module>
----> 1 print(A)

NameError: name 'A' is not defined
```

2. 디버깅

• 에러의 종류

- runtime error: 코드가 실행될 때 발생하는 에러

➔ 자주 발생하는 runtime error에는 다음과 같은 경우가 있다.

1. TypeError: 변수 또는 인수의 형(type)이 올바르지 않은 경우
2. NameError: 정의되지 않은 변수를 참조하려고 하는 경우
3. ZeroDivisionError: 0으로 나누는 경우

➔ ZeroDivisionError의 예시

```
In [14]: a = 1
         b = 0
         c = a/b

---------------------------------------------------------------------------
ZeroDivisionError                         Traceback (most recent call last)
~₩AppData₩Local₩Temp₩ipykernel_3528₩2602038102.py in <module>
      1 a = 1
      2 b = 0
----> 3 c = a/b

ZeroDivisionError: division by zero
```

2. 디버깅

- **에러의 종류**
  - logical error: 프로그램의 알고리즘 자체가 오류를 포함하고 있어 올바르지 않은 결과가 나타나는 오류

```
In [27]: def factorial_wth_err(n):
             out = 0
             for i in range(1,n+1):
                 out = out*i

             return out

In [28]: def factorial(n):
             out = 1
             for i in range(1,n+1):
                 out = out*i

             return out

In [29]: print(factorial_wth_err(4))
         print(factorial(4))

         0
         24
```

2. 디버깅

- **Try/except**
  - 파이썬에서 예외 처리를 가능하게 해주는 구문
  - 예외가 발생할 가능성이 있는 코드블록을 ‘try’블록에 넣고, 해당 예외가 발생했을 때 실행할 코드를 ‘except’블록에 넣어 구현한다.
  - 다양한 오류가 발생했을 때 예외 처리를 가능하게 해준다.

```
In [22]: try:
             numer = int(input("Enter numerator: "))
             denom = int(input("Enter denominator: "))
             result = numer / denom
             print(result)
         except ZeroDivisionError:
             print("Denominator cannot be zero.")

         Enter numerator: 12
         Enter denominator: 3
         4.0
```

```
In [23]: try:
             numer = int(input("Enter numerator: "))
             denom = int(input("Enter denominator: "))
             result = numer / denom
             print(result)
         except ZeroDivisionError:
             print("Denominator cannot be zero.")

         Enter numerator: 12
         Enter denominator: 0
         Denominator cannot be zero.
```

2. 디버깅

- **Type checking**
  - 변수나 함수의 인자, 반환값 등에 사용되는 데이터 타입을 검사하는 과정
  - 파이썬은 동적 타이핑(Dynamic Typing) 언어로 변수를 선언할 때 데이터 타입을 지정하지 않아도 된다. 따라서 코드를 작성하다 보면 변수나 함수에서 예상하지 않은 타입의 데이터가 발생할 수 있다. 이러한 문제를 예방하고 디버깅을 용이하게 하기 위해 수행하는 과정이다.
  - type()함수를 사용하여 직접 확인하거나 isinstance() 함수를 사용한다.

In [30]:
```
x = 5
y = "hello"
z = [1, 2, 3]

print(type(x))
print(type(y))
print(type(z))
```

```
<class 'int'>
<class 'str'>
<class 'list'>
```

In [31]:
```
def my_adder(a,b,c):
    # type check
    if isinstance(a,float) and isinstance(b,float) and isinstance(c,float):
        pass
    else:
        raise(TypeError("Inputs must be floats"))

    out = a + b + c
    return out
```

a,b,c가 float형이 아니면 계산하고 싶지 않을 때 다음과 같이 type check를 통해 예외 처리를 할 수 있다.

2. 디버깅

- **디버깅이란?**
  - 프로그램 내의 오류 또는 버그를 시스템적으로 제거하는 과정으로 오류가 발생한 원인을 찾아내고 수정하기 위해 데이터 또는 알고리즘을 수정해야 할 수도 있다.
  - 중단점 설정, 코드 단계별 실행, 문자열 출력 등의 다양한 기법과 도구를 활용할 수 있다.

  - 파이썬에서는 pdb(Python Debugger), ipdb(IPython Debugger)를 제공한다.
    ➔ http://docs.python.org/3/library/pdb.html

3. 디버깅

- **파이썬 디버깅 도구 사용 예제**
  - 실행 중 오류가 발생한 후 디버깅
    - ➔ Magic command인 %debug로 디버깅 도구를 활성화 시킬 수 있다.

```
In [31]: def my_adder(a,b,c):
             # type check
             if isinstance(a,float) and isinstance(b,float) and isinstance(c,float):
                 pass
             else:
                 raise(TypeError("Inputs must be floats"))

             out = a + b + c
             return out
```

```
In [33]: def square_number(x):
             sq = x**2
             sq += x

             return sq
```

```
In [34]: square_number("10")
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
~₩AppData₩Local₩Temp₩ipykernel_3528₩3473278975.py in <module>
----> 1 square_number("10")

~₩AppData₩Local₩Temp₩ipykernel_3528₩2802720476.py in square_number(x)
      1 def square_number(x):
----> 2     sq = x**2
      3     sq += x
      4
      5     return sq

TypeError: unsupported operand type(s) for ** or pow(): 'str' and 'int'
```

```
In [*]: %debug
```

```
> c:₩users₩mpmc₩appdata₩local₩temp₩ipykernel_3528₩2802720476.py(2)square_number()

ipdb>
```

• 파이썬 디버깅 도구 사용 예제

- 실행 중 오류가 발생한 후 디버깅

➔ %debug로 ipdb를 활성화 시킨 후 디버깅 명령어를 사용해 쉽게 오류를 찾아 수정할 수 있다..

```
In [36]: %debug

> c:\users\mpmc\appdata\local\temp\ipykernel_3528\2802720476.py(2)square_number()

ipdb> h

Documented commands (type help <topic>):
========================================
EOF    commands   enable    ll        pp       s                until
a      condition  exit      longlist  psource  skip_hidden      up
alias  cont       h         n         q        skip_predicates  w
args   context    help      next      quit     source           whatis
b      continue   ignore    p         r        step             where
break  d          interact  pdef      restart  tbreak
bt     debug      j         pdoc      return   u
c      disable    jump      pfile     retval   unalias
cl     display    l         pinfo     run      undisplay
clear  down       list      pinfo2    rv       unt

Miscellaneous help topics:
==========================
exec  pdb

ipdb> p x
'10'
ipdb> type(x)
<class 'str'>
ipdb> p locals()
{'x': '10'}
ipdb> q
```

- ✓ h : 도움말 함수 목록을 얻기
- ✓ p x : x의 값을 출력하기
- ✓ type(x) : x의 타입 가져오기
- ✓ p locals() : 모든 로컬 변수 출력하기
- ✓ q : 디버거(또는 프로그램) 종료하기

3. 디버깅

- **파이썬 디버깅 도구 사용 예제**
  - 실행 전 디버깅 도구 활성화

➔ %pdb on 명령어를 사용하면 디버깅 도구가 활성화 된다. (%pdb off로 비활성화)

```
In [37]: %pdb on

Automatic pdb calling has been turned ON
```

➔ 활성화 후 코드실행시 오류가 발생하면 이전과 동일한 방법으로 디버깅 도구를 사용할 수 있다.

```
In [*]: square_number("10")

---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
~\AppData\Local\Temp\ipykernel_3528\3473278975.py in <module>
----> 1 square_number("10")

~\AppData\Local\Temp\ipykernel_3528\2802720476.py in square_number(x)
      1 def square_number(x):
----> 2     sq = x**2
      3     sq += x
      4
      5     return sq

TypeError: unsupported operand type(s) for ** or pow(): 'str' and 'int'

> c:\users\mpmc\appdata\local\temp\ipykernel_3528\2802720476.py(2)square_number()

ipdb>
```

3. 디버깅

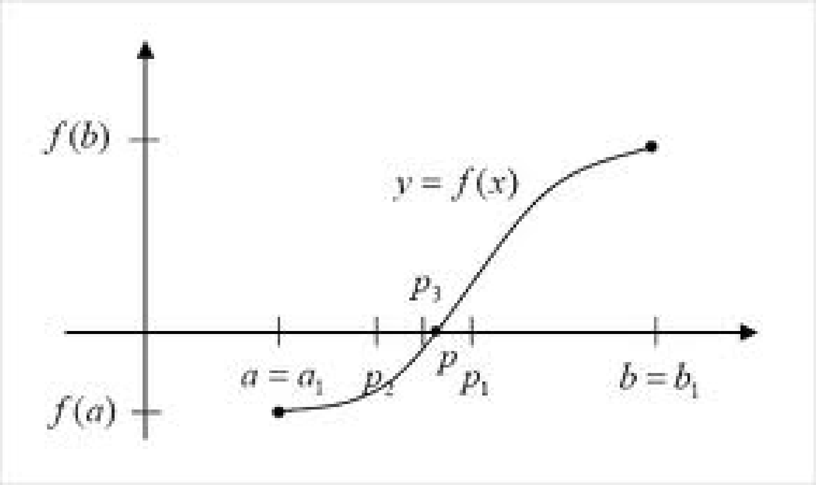
- **파이썬 디버깅 도구 사용 예제**
  - 중단점 설정(Breakpoint)
    - ➔ pdb 모듈의 set_trace()함수를 이용해 중단점을 설정할 수 있다.
    - ➔ 코드실행 시 중단점에서 디버깅도구가 실행되며 c(continue)를 입력할 때까지 디버깅 도구 명령어를 이용해 변수의 값 등을 확인할 수 있다.

```
In [44]: import pdb

         def square_number(x):

             sq = x**2

             pdb.set_trace()

             sq += x

             return sq
```

```
In [47]: square_number(3)

> c:\users\mpmc\appdata\local\temp\ipykernel_3528\1579418250.py(9)square_number()
ipdb> l                  현재 위치 확인
      4
      5     sq = x**2
      6
      7     pdb.set_trace()
      8
----> 9       sq += x
     10
     11     return sq

ipdb> p x                x값 출력
3
ipdb> p sq               sq값 출력
9
ipdb> c                  계속 실행
Out[47]: 12
```

3. Root finding

- **문제 정의(root finding)**
  - f(x) = 0 → y = f(x) and y = 0
- **Bisection method**
  - 기본 아이디어
    - ➔ 중간값 정리(intermediate-value theorem)




```
procedure Bisection(f, a, b, nmax, ε)
integer n, nmax;    real a, b, c, fa, fb, fc, error
fa ← f(a)
fb ← f(b)
if sign(fa) = sign(fb) then
    output a, b, fa, fb
    output "function has same signs at a and b"
    return
end if
error ← b − a
for n = 0 to nmax do
    error ← error/2
    c ← a + error
    fc ← f(c)
    output n, c, fc, error
    if |error| < ε then
        output "convergence"
        return
    end if
    if sign(fa) ≠ sign(fc) then
        b ← c
        fb ← fc
    else
        a ← c
        fa ← fc
    end if
end for
end procedure Bisection
```

```
program Test_Bisection
integer n, nmax ← 20
real a, b, ε ← ½10^−6
external function f, g
a ← 0.0
b ← 1.0
call Bisection(f, a, b, nmax, ε)
a ← 0.5
b ← 2.0
call Bisection(g, a, b, nmax, ε)
end program Test_Bisection

real function f(x)
real x
f ← x^3 − 3x + 1
end function f

real function g(x)
real x
g ← x^3 − 2 sin x
end function g
```

3. Root finding

- **Bisection method**

In [1]:
```python
import numpy as np

def Bisection(f, a, b, nmax, tol):

    # approximates a root, c, of f bounded
    # by a and b to within tolerance
    # | f(m) | < tol with m being the midpoint
    # between a and b. Recursive implementation.

    fa = f(a)
    fb = f(b)

    if np.sign(fa)==np.sign(fb):
        print('a= %0.1f b= %0.1f f(a)= % 1.2e f(b)= % 1.2e' % (a, b, fa, fb))
        print('function has same signs at a and b')
        return

    error = b - a

    for n in range(0,nmax):

        error = error/2
        c     = a + error
        fc    = f(c)

        print('n= %02d c= % 0.7f f(c)= % 1.2e error= % 1.2e' % (n, c, fc, error))

        if np.abs(error)<tol:
            print('convergence')
            return c
        if np.sign(fa)!=np.sign(fc):
            b  = c
            fb = fc
        else:
            a  = c
            fa = fc
```

3. Root finding

- **Bisection method**

```
In [2]: f = lambda x : x**3 - 3*x + 1
        g = lambda x : x**3 - 2*np.sin(x)
```

```
In [3]: nmax = 25
        tol = 1e-6/2.0

        a = 0.0
        b = 1.0
        Rt_f = Bisection(f, a, b, nmax, tol)
        print('root of function f = ', Rt_f,'\n')

        a = 0.5
        b = 2.0
        Rt_g = Bisection(g, a, b, nmax, tol)
        print('root of function g = ', Rt_g)
```

solution monitoring 중요
1) iteration
2) root (잘 수렴하느냐)
3) f(x) 값
4) eroor

수렴성 확인이 중요하다

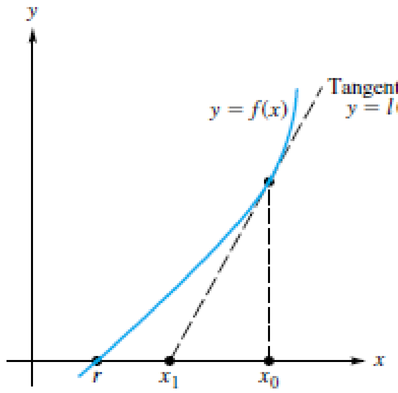
$$f(x) = x^3 - 3x + 1 = 0$$

```
n= 00 c=  0.5000000 f(c)= -3.75e-01 error=  5.00e-01
n= 01 c=  0.2500000 f(c)=  2.66e-01 error=  2.50e-01
n= 02 c=  0.3750000 f(c)= -7.23e-02 error=  1.25e-01
n= 03 c=  0.3125000 f(c)=  9.30e-02 error=  6.25e-02
n= 04 c=  0.3437500 f(c)=  9.37e-03 error=  3.12e-02
n= 05 c=  0.3593750 f(c)= -3.17e-02 error=  1.56e-02
n= 06 c=  0.3515625 f(c)= -1.12e-02 error=  7.81e-03
n= 07 c=  0.3476562 f(c)= -9.49e-04 error=  3.91e-03
n= 08 c=  0.3457031 f(c)=  4.21e-03 error=  1.95e-03
n= 09 c=  0.3466797 f(c)=  1.63e-03 error=  9.77e-04
n= 10 c=  0.3471680 f(c)=  3.39e-04 error=  4.88e-04
n= 11 c=  0.3474121 f(c)= -3.05e-04 error=  2.44e-04
n= 12 c=  0.3472900 f(c)=  1.67e-05 error=  1.22e-04
n= 13 c=  0.3473511 f(c)= -1.44e-04 error=  6.10e-05
n= 14 c=  0.3473206 f(c)= -6.38e-05 error=  3.05e-05
n= 15 c=  0.3473053 f(c)= -2.36e-05 error=  1.53e-05
n= 16 c=  0.3472977 f(c)= -3.46e-06 error=  7.63e-06
n= 17 c=  0.3472939 f(c)=  6.60e-06 error=  3.81e-06
n= 18 c=  0.3472958 f(c)=  1.57e-06 error=  1.91e-06
n= 19 c=  0.3472967 f(c)= -9.48e-07 error=  9.54e-07
n= 20 c=  0.3472962 f(c)=  3.10e-07 error=  4.77e-07
convergence
root of function f =  0.34729623794555664
```

$$g(x) = x^3 - 2\sin(x) = 0$$

```
n= 00 c=  1.2500000 f(c)=  5.52e-02 error=  7.50e-01
n= 01 c=  0.8750000 f(c)= -8.65e-01 error=  3.75e-01
n= 02 c=  1.0625000 f(c)= -5.48e-01 error=  1.88e-01
n= 03 c=  1.1562500 f(c)= -2.85e-01 error=  9.38e-02
n= 04 c=  1.2031250 f(c)= -1.25e-01 error=  4.69e-02
n= 05 c=  1.2265625 f(c)= -3.74e-02 error=  2.34e-02
n= 06 c=  1.2382812 f(c)=  8.26e-03 error=  1.17e-02
n= 07 c=  1.2324219 f(c)= -1.47e-02 error=  5.86e-03
n= 08 c=  1.2353516 f(c)= -3.27e-03 error=  2.93e-03
n= 09 c=  1.2368164 f(c)=  2.49e-03 error=  1.46e-03
n= 10 c=  1.2360840 f(c)= -3.92e-04 error=  7.32e-04
n= 11 c=  1.2364502 f(c)=  1.05e-03 error=  3.66e-04
n= 12 c=  1.2362671 f(c)=  3.27e-04 error=  1.83e-04
n= 13 c=  1.2361755 f(c)= -3.30e-05 error=  9.16e-05
n= 14 c=  1.2362213 f(c)=  1.47e-04 error=  4.58e-05
n= 15 c=  1.2361984 f(c)=  5.69e-05 error=  2.29e-05
n= 16 c=  1.2361870 f(c)=  1.20e-05 error=  1.14e-05
n= 17 c=  1.2361813 f(c)= -1.05e-05 error=  5.72e-06
n= 18 c=  1.2361841 f(c)=  7.54e-07 error=  2.86e-06
n= 19 c=  1.2361827 f(c)= -4.86e-06 error=  1.43e-06
n= 20 c=  1.2361834 f(c)= -2.05e-06 error=  7.15e-07
n= 21 c=  1.2361838 f(c)= -6.50e-07 error=  3.58e-07
convergence
root of function g =  1.236183762550354
```

3. Root finding



- **Newton's method**
- **기본 아이디어 : 기울기**

- 뉴턴의 방법에서는 함수 f가 미분 가능하다는 것을 가정한다. 이는 f의 그래프가 각 점에서 기울기와 고유한 접선을 가지고 있다는 것을 의미한다.

- 함수 f의 그래프에서 어떤 점 $(x_0, f(x_0))$에서는 그 점 근처의 곡선을 상당히 잘 근사하는 접선이 존재한다. 수학적으로, 이는 선형 함수 $l(x) = f'(x_0)(x - x_0) + f(x_0)$ 가 x_0 근처에서 주어진 함수 f에 근사된다는 것을 의미한다. x_0에서 두 함 수 l과 f는 일치한다. 따라서 l의 해를 f의 해로 근사할 수 있으며 l의 해는 다음과 같이 구할 수 있다.

$$x_1 = x_0 - \frac{f(x_0)}{f'(x_0)} \qquad x_2 = x_1 - \frac{f(x_1)}{f'(x_1)}, \qquad x_3 = x_2 - \frac{f(x_2)}{f'(x_2)},$$

3. Root finding

- **Newton's method**
- **해석(테일러 급수 전개(Taylor series expansion))**
  - x_0가 f의 해라고 가정을 하고 어느정도의 보정값 h를 더해줘야 정확하게 해를 구할 수 있을지, 즉, $f(x_0+h)=0$이 될지 생각해보자.
  - 만약 f가 충분히 좋은 함수라면, x_0에서의 테일러 급수를 다음과 같이 구할 수 있다 :

$$f(x_0)+hf'(x_0)+\frac{h^2}{2}f''(x_0)+\cdots=0$$

  - 위 식에서 h를 구하는 것은 쉽지 않다. h를 직접 구하는 대신 h의 근사값을 찾아보자. 두 번째 항까지 남기고 근사한 식은 다음과 같다 : $f(x_0)+hf'(x_0)=0$
  - 근사한 식으로부터 $h=-\frac{f(x_0)}{f'(x_0)}$ 이다.
  - $f(x_0+h)=0$을 정확히 만족하지는 않지만 새로운 근사값을 다음과 같이 얻을 수 있다 :

$$x_1=x_0+h=x_0-\frac{f(x_0)}{f'(x_0)}$$

3. Root finding

- **Newton's method**

procedure *Newton*(f, f', x, $nmax$, ε, δ)
integer n, $nmax$;   real x, fx, fp, ε, δ
external function f, f'
$fx \leftarrow f(x)$
output $0, x, fx$
for $n = 1$ to $nmax$ do
    $fp \leftarrow f'(x)$
    if $|fp| < \delta$ then
        output "small derivative"
        return
    end if
    $d \leftarrow fx/fp$
    $x \leftarrow x - d$
    $fx \leftarrow f(x)$
    output n, x, fx
    if $|d| < \varepsilon$ then
        output "convergence"
        return
    end if
end for
end procedure *Newton*

In [1]:
```
import numpy as np

def Newton(f, df, x, nmax, tol, delt):

    # output is an estimation of the root of f
    # using the Newton-Raphson method
    # recursive implementation

    fx  = f(x)
    print('%-4s %-20s %4s' %('n','x(n)','f(x(n))'))
    print('%-3d % 1.10e %2s % 1.3e' %(0, x,'', fx))

    for n in range(1,nmax+1):

        fp = df(x)

        if np.abs(fp)<delt:
            print('small derivative')
            return

        d  = fx/fp
        x  = x - d
        fx = f(x)

        print('%-3d % 1.10e %2s % 1.3e' %(n, x,'', fx))

        if np.abs(d)<tol:
            print('convergence')
            return x
```

3. Root finding

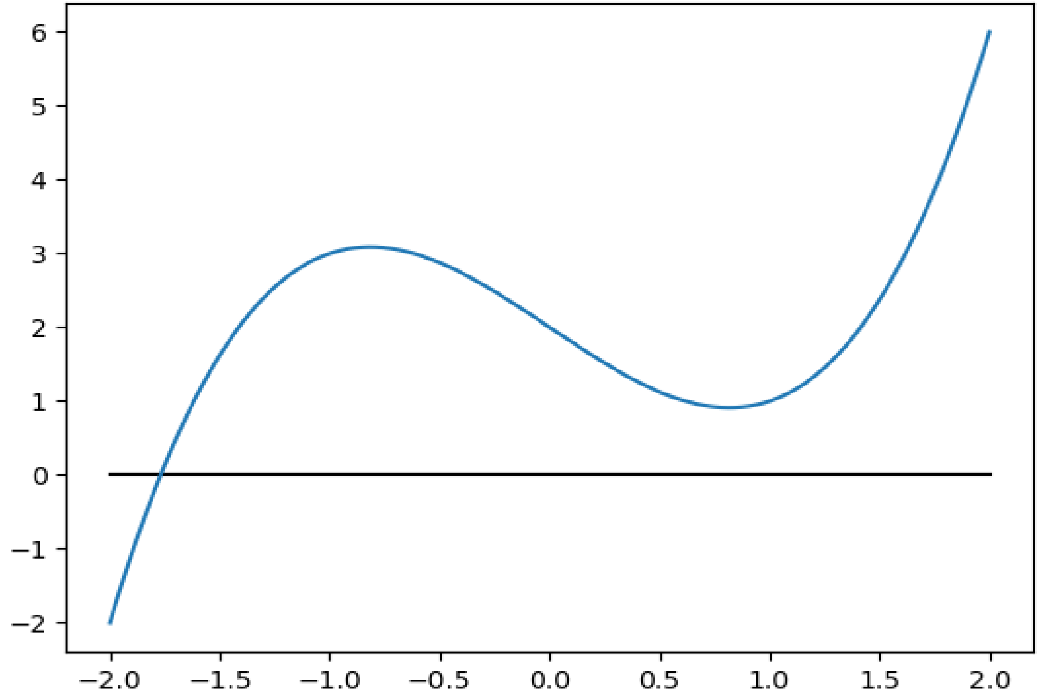
- **Newton's method**

```
In [2]: f  = lambda x : x**3 - 2*x + 2
        df = lambda x : 3*x**2 - 2
```

```
In [5]: nmax = 100
        tol  = 1e-12

        x0   = -1.5

        Rt_f = Newton(f, df, x0, nmax, tol, 1e-12)
        print('root of function f = ',Rt_f)
```

```
n    x(n)                     f(x(n))
0   -1.50000000000000e+00     1.625e+00
1   -1.84210526315789e+00    -5.667e-01
2   -1.77282691999216e+00    -2.619e-02
3   -1.76930129255045e+00    -6.607e-05
4   -1.76929235429601e+00    -4.241e-10
5   -1.76929235423863e+00     0.000e+00
6   -1.76929235423863e+00     0.000e+00
convergence
root of function f =  -1.7692923542386314
```

$f = x^3 - 2x + 2$



Pseudo code의 **if** $|d| < \varepsilon$ **then** 부분을 수정하면 설정한 tolerance에서 수렴할 수 있게 수정할 수 있다.

3. Root finding

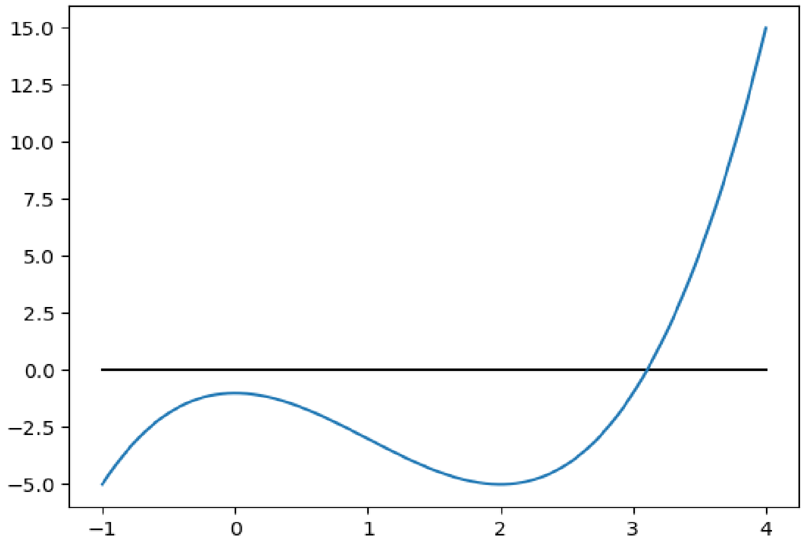
- **Newton's method**

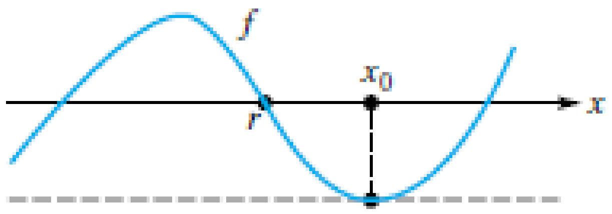
In [4]:
```
# flatpoint example

f  = lambda x : x**3 - 3*x**2 - 1
df = lambda x : 3*x**2 - 6*x
```

In [5]:
```
nmax = 100
tol  = 1e-12

x0   = 1

Rt_f = Newton(f, df, x0, nmax, tol, 1e-12)
print('root of function f = ',Rt_f)
```

```
n     x(n)                        f(x(n))
0     1.0000000000000e+00     -3.000e+00
1     0.0000000000000e+00     -1.000e+00
small derivative
root of function f =  None
```

$f = x^3 - 3x^2 - 1$





(b) Flat spot

3. Root finding

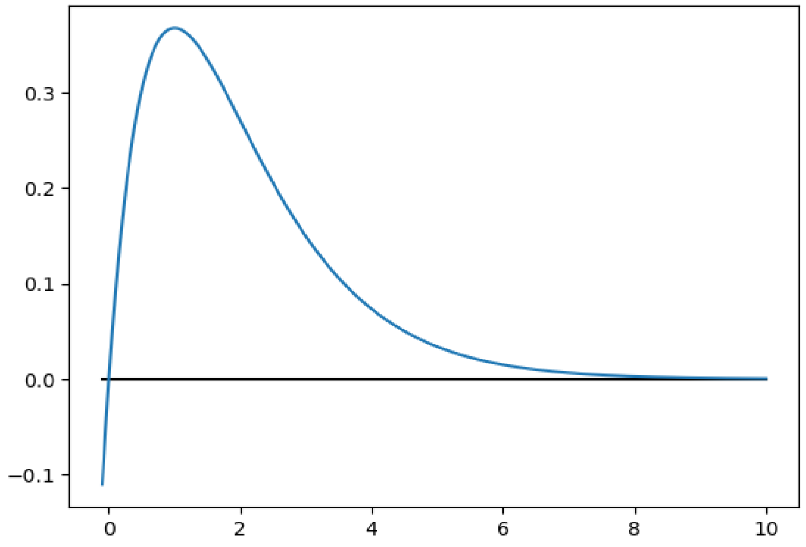
- **Newton's method**

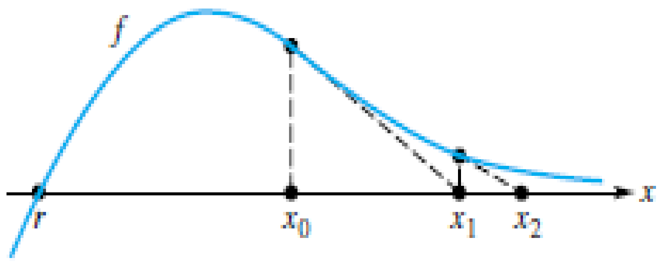
In [2]:
```python
# runaway example

f  = lambda x : x*np.exp(-x)
df = lambda x : (1.0-x)*np.exp(-x)
```

In [3]:
```python
nmax = 10
tol  = 1e-12

x0   = 4

Rt_f = Newton(f, df, x0, nmax, tol, 1e-12)
print('root of function f = ',Rt_f)
```

```
n    x(n)                    f(x(n))
0    4.0000000000000e+00     7.326e-02
1    5.3333333333333e+00     2.575e-02
2    6.5641025641026e+00     9.256e-03
3    7.7438260664067e+00     3.356e-03
4    8.8921098433240e+00     1.222e-03
5    1.0018818672757e+01     4.464e-04
6    1.1129697939353e+01     1.633e-04
7    1.2228417566136e+01     5.979e-05
8    1.3317477310673e+01     2.191e-05
9    1.4398662765680e+01     8.036e-06
10   1.5473297079379e+01     2.949e-06
root of function f =  None
```

$f = xe^{-x}$





(a) Runaway

3. Root finding

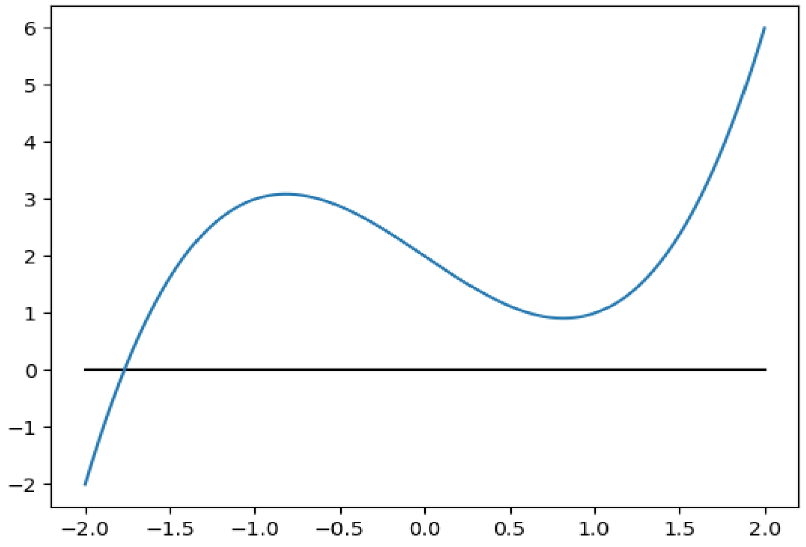
- **Newton's method**

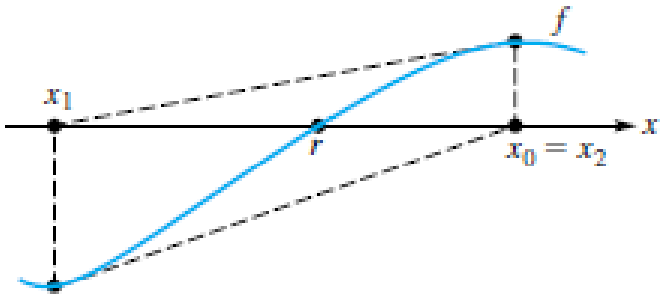
In [6]:
```python
# cycle example

f  = lambda x : x**3 - 2*x + 2
df = lambda x : 3*x**2 - 2
```

In [7]:
```python
nmax = 10
tol  = 1e-12

x0   = 0

Rt_f = Newton(f, df, x0, nmax, tol, 1e-12)
print('root of function f = ',Rt_f)
```

```
n     x(n)                    f(x(n))
0     0.0000000000000e+00     2.000e+00
1     1.0000000000000e+00     1.000e+00
2     0.0000000000000e+00     2.000e+00
3     1.0000000000000e+00     1.000e+00
4     0.0000000000000e+00     2.000e+00
5     1.0000000000000e+00     1.000e+00
6     0.0000000000000e+00     2.000e+00
7     1.0000000000000e+00     1.000e+00
8     0.0000000000000e+00     2.000e+00
9     1.0000000000000e+00     1.000e+00
10    0.0000000000000e+00     2.000e+00
root of function f =  None
```

$f = x^3 - 2x + 2$





(c) Cycle

3. Root finding

- **Newton's method with SciPy**
  - SciPy 모듈의 SciPy.optimize의 함수를 이용하여 해를 쉽게 구할 수 있다.

Root finding

Scalar functions

root_scalar(f[, args, method, bracket, ...])	Find a root of a scalar function.
brentq(f, a, b[, args, xtol, rtol, maxiter, ...])	Find a root of a function in a bracketing interval using Brent's method.
brenth(f, a, b[, args, xtol, rtol, maxiter, ...])	Find a root of a function in a bracketing interval using Brent's method with hyperbolic extrapolation.
ridder(f, a, b[, args, xtol, rtol, maxiter, ...])	Find a root of a function in an interval using Ridder's method.
bisect(f, a, b[, args, xtol, rtol, maxiter, ...])	Find root of a function within an interval using bisection.
newton(func, x0[, fprime, args, tol, ...])	Find a zero of a real or complex function using the Newton-Raphson (or secant or Halley's) method.
toms748(f, a, b[, args, k, xtol, rtol, ...])	Find a zero using TOMS Algorithm 748 method.
RootResults(root, iterations, ...)	Represents the root finding result.

Root finding

General nonlinear solvers:

fsolve(func, x0[, args, fprime, ...])	Find the roots of a function.
broyden1(F, xin[, iter, alpha, ...])	Find a root of a function, using Broyden's first Jacobian approximation.
broyden2(F, xin[, iter, alpha, ...])	Find a root of a function, using Broyden's second Jacobian approximation.

Large-scale nonlinear solvers:

newton_krylov(F, xin[, iter, rdiff, method, ...])	Find a root of a function, using Krylov approximation for inverse Jacobian.
anderson(F, xin[, iter, alpha, w0, M, ...])	Find a root of a function, using (extended) Anderson mixing.
BroydenFirst([alpha, reduction_method, max_rank])	Find a root of a function, using Broyden's first Jacobian approximation.
InverseJacobian(jacobian)	**Attributes:**
KrylovJacobian([rdiff, method, ...])	Find a root of a function, using Krylov approximation for inverse Jacobian.

Simple iteration solvers:

excitingmixing(F, xin[, iter, alpha, ...])	Find a root of a function, using a tuned diagonal Jacobian approximation.
linearmixing(F, xin[, iter, alpha, verbose, ...])	Find a root of a function, using a scalar Jacobian approximation.
diagbroyden(F, xin[, iter, alpha, verbose, ...])	Find a root of a function, using diagonal Broyden Jacobian approximation.

- https://docs.scipy.org/doc/scipy/reference/optimize.html

3. Root finding

- **Newton's method with SciPy**
  - $f = x^3 - 2x + 2$의 해를 구하는 경우, SciPy로 같은 결과를 얻을 수 있다.

scipy.optimize.newton

scipy.optimize.newton(*func, x0, fprime=None, args=(), tol=1.48e-08, maxiter=50, fprime2=None*) [source]

Find a zero using the Newton-Raphson or secant method.

Find a zero of the function *func* given a nearby starting point *x0*. The Newton-Raphson method is used if the derivative *fprime* of *func* is provided, otherwise the secant method is used. If the second order derivate *fprime2* of *func* is provided, parabolic Halley's method is used.

```
In [2]: f  = lambda x : x**3 - 2*x + 2
        df = lambda x : 3*x**2 - 2

In [5]: nmax = 100
        tol  = 1e-12

        x0   = -1.5

        Rt_f = Newton(f, df, x0, nmax, tol, 1e-12)
        print('root of function f = ',Rt_f)

        n    x(n)                      f(x(n))
        0   -1.500000000000000e+00     1.625e+00
        1   -1.842105263157895e+00    -5.667e-01
        2   -1.772826919992160e+00    -2.619e-02
        3   -1.769301292550450e+00    -6.607e-05
        4   -1.769292354296010e+00    -4.241e-10
        5   -1.769292354238630e+00     0.000e+00
        6   -1.769292354238630e+00     0.000e+00
        convergence
        root of function f =  -1.7692923542386314
```

```
In [6]: # cycle example

        f  = lambda x : x**3 - 2*x + 2
        df = lambda x : 3*x**2 - 2

In [7]: nmax = 10
        tol  = 1e-12

        x0   = 0

        Rt_f = Newton(f, df, x0, nmax, tol, 1e-12)
        print('root of function f = ',Rt_f)

        n    x(n)                      f(x(n))
        0    0.000000000000000e+00     2.000e+00
        1    1.000000000000000e+00     1.000e+00
        2    0.000000000000000e+00     2.000e+00
        3    1.000000000000000e+00     1.000e+00
        4    0.000000000000000e+00     2.000e+00
        5    1.000000000000000e+00     1.000e+00
        6    0.000000000000000e+00     2.000e+00
        7    1.000000000000000e+00     1.000e+00
        8    0.000000000000000e+00     2.000e+00
        9    1.000000000000000e+00     1.000e+00
        10   0.000000000000000e+00     2.000e+00
        root of function f =  None
```

```
In [1]: import numpy as np
        f  = lambda x : x**3 - 2*x + 2
        df = lambda x : 3*x**2 - 2

In [2]: import scipy.optimize as op

In [4]: a = op.newton(f, fprime=df, x0=-1.5, tol=1e-12, full_output=True)
        print(a)

        (-1.7692923542386314,       converged: True
                   flag: 'converged'
         function_calls: 11
             iterations: 5
                   root: -1.7692923542386314)
```

3. Root finding

- **Newton's method (Systems of Nonlinear Equations)**

$$\begin{cases} f_1(x_1, x_2, \ldots, x_N) &= 0 \\ f_2(x_1, x_2, \ldots, x_N) &= 0 \\ &\vdots \\ f_N(x_1, x_2, \ldots, x_N) &= 0 \end{cases} \longrightarrow \quad \mathbf{F}(\mathbf{X}) = \mathbf{0} \qquad \begin{array}{l} \mathbf{F} = [f_1, f_2, \ldots, f_N]^T \\ \mathbf{X} = [x_1, x_2, \ldots, x_N]^T \end{array}$$

$$\mathbf{X}^{(k+1)} = \mathbf{X}^{(k)} - \left[\mathbf{F}'\left(\mathbf{X}^{(k)}\right)\right]^{-1}\mathbf{F}\left(\mathbf{X}^{(k)}\right)$$

$\mathbf{F}'\left(\mathbf{X}^{(k)}\right)$ **is the Jacobian matrix**

Root finding

We illustrate the development of this procedure using three nonlinear equations

$$\begin{cases} f_1(x_1, x_2, x_3) = 0 \\ f_2(x_1, x_2, x_3) = 0 \\ f_3(x_1, x_2, x_3) = 0 \end{cases} \quad (3)$$

Recall the Taylor expansion in three variables for $i = 1, 2, 3$:

$$\boxed{f_i(x_1 + h_1, x_2 + h_2, x_3 + h_3) = f_i(x_1, x_2, x_3) + h_1 \frac{\partial f_i}{\partial x_1} + h_2 \frac{\partial f_i}{\partial x_2} + h_3 \frac{\partial f_i}{\partial x_3} + \cdots \quad (4)}$$

where the partial derivatives are evaluated at the point (x_1, x_2, x_3). Here only the linear terms in step sizes h_i are shown. Suppose that the vector $\mathbf{X}^{(0)} = \left(x_1^{(0)}, x_2^{(0)}, x_3^{(0)}\right)^T$ is an approximate solution to (3). Let $\mathbf{H} = [h_1, h_2, h_3]^T$ be a computed **correction** to the initial guess so that $\mathbf{X}^{(0)} + \mathbf{H} = \left[x_1^{(0)} + h_1, x_2^{(0)} + h_2, x_3^{(0)} + h_3\right]^T$ is a better approximate solution. Discarding the higher-order terms in the Taylor expansion (4), we have in vector notation

$$\mathbf{0} \approx \mathbf{F}\left(\mathbf{X}^{(0)} + \mathbf{H}\right) \approx \mathbf{F}\left(\mathbf{X}^{(0)}\right) + \mathbf{F}'\left(\mathbf{X}^{(0)}\right)\mathbf{H} \quad (5)$$

$$\mathbf{F}'\left(\mathbf{X}^{(0)}\right) = \begin{bmatrix} \frac{\partial f_1}{\partial x_1} & \frac{\partial f_1}{\partial x_2} & \frac{\partial f_1}{\partial x_3} \\ \frac{\partial f_2}{\partial x_1} & \frac{\partial f_2}{\partial x_2} & \frac{\partial f_2}{\partial x_3} \\ \frac{\partial f_3}{\partial x_1} & \frac{\partial f_3}{\partial x_2} & \frac{\partial f_3}{\partial x_3} \end{bmatrix}$$

$$\mathbf{H} \approx -\left[\mathbf{F}'\left(\mathbf{X}^{(0)}\right)\right]^{-1}\mathbf{F}\left(\mathbf{X}^{(0)}\right)$$

$$\mathbf{X}^{(k+1)} = \mathbf{X}^{(k)} - \left[\mathbf{F}'\left(\mathbf{X}^{(k)}\right)\right]^{-1}\mathbf{F}\left(\mathbf{X}^{(k)}\right)$$

3X3는 손으로 쉽게 구할 수 있음
근데 100X100은..???????
—> hard
(5)으로 생각하자
Ax = b를 이용하면 H 바로 구할 수 있음

$$\left[\mathbf{F}'\left(\mathbf{X}^{(k)}\right)\right]\mathbf{H}^{(k)} = -\mathbf{F}\left(\mathbf{X}^{(k)}\right)$$

$$\mathbf{X}^{(k+1)} = \mathbf{X}^{(k)} + \mathbf{H}^{(k)}$$

Q&A Thanks for listening