Homework #1

Implement Four root finding methods & Disscusion

GIST AI

20231126 황산하

*코드는 Python으로 작성했고 Report 이외에 스크립트도 Report와 함께 압축하여 제출하였습니다.

INDEX

1. Comparison of Four root finding methods

인수분해를 통해 간단하게 구할 수 있는 함수(F1)와 **해를 구하는게 좀 더 복잡한 임의의 함수(F2)**, **지수함수와 같은 다른 복합적인 함수로 표현된 함수(F3)** 로 각각의 성능과 특징을 분석했다. 사용한 함수는 다음과 같다.

$$F_1(x) = x^3 - 3x^2 + 3x + 1$$

$$F_2(x) = x^5 - 4x^4 + 3x^3 + 9x - 5$$

$$F_3(x) = e^x - x^e + 3x^3 - 2x - 5$$

또한 엄밀한 비교를 위해 HyperParameter는 최대한 일치시켰다. 시작점을 같게 하고 (Bisection method 같은 경우는 f(a)부터 탐색하기 때문에 이를 시작점으로 생각했다.) Tolerance 변수는 1e-6으로 주어 비슷한 지점까지 iteration이 반복하도록 했다.

1) Bisection Method

Bisection Method는 범위가 반씩 줄어들기 때문에, 3개의 함수 모두 아무리 범위를 크게 줘도 실제 iteration에는 차이가 크게 나지 않았다. 1000이상 범위가 차이나도 iteration은 2~4번 정도 차이가 났고, 이 정도는 Unimodal Root finding에서는 차이가 크게 체감되지 않는다. 따라서 범위를 좁게 주었을 때 그 안에 해가 없다면 오히려 해를 못찾을 수 있기 때문에 범위를 최대한 넓게 주는게 유리하다고 생각된다. 또한 F2, F3 함수의 경우, 범위를 어떻게 잡든 해를 중간에 건너 뛰는 바람에 아주 정확한 해를 찾지는 못했는데, 그럼에도 해에 수렴했기 때문에 최대한 해를 탐색하며 찾아보려는 수치적 기법으로서는 탁월한 방법이라고 생각된다.

2) Newton Method

Newton Method는 한 점에서의 기울기를 이용해서 해를 근사하는 방법인데, 이 역시도 해로 잘 수렴하는 모습을 보여주었다. 하지만 시작점이 해로부터 멀어질수록 Bisection Method에 비해서 iteration이 크게 증가했다. Bisecton Method는 (-1000, 1000) 범위에서 32번 iteration을 하는 반면, Newton Method는 49번을 탐색하고서야 알고리즘이 종료되었다. 또한 Newton Method를 구현하는 과정에서 f(x)를 df(x) 값으로 나누는 과정이 있는데, 미분값이 0으로 수렴하게 되면 값을 못구하는 경우가 생겼다. 따라서 이 경우를 방지하기 위해 h라는 작은 변수를 더해서 나누어주었다. Newton Method의 장점은 Bisection Method와는 달리 초기 값을 어떻게 주더라도 해의 근방으로 수렴할 수 있다는 점이었다.

3) Secant Method

Secant Method는 초기의 두점 사이의 기울기를 통해 해를 찾아가는 방법이다. 여러 실험을 해본 결과, 초기 두 점을 어떻게 주냐에 따라서 성능이 확실하게 달라지는 것을 확인했다. 초기 두점이 명확하게 해를 찾아가는 길에 있을 경우, 단 두 번의 iteration으로도 해를 찾을 수 있었고, 두 점이 매우 나쁜 경우, 100번을 반복해도 엉뚱한 해를 찾았다. 그래서 앞선 Bisection Method와는 달리 두 점 사이의 거리(interval)이 수렴속도에 영향을 미치지 않는 것을 확인했다. 하지만 이 Secant Method를 통해 찾은 해가 과연 실제 해를 보장하는지는 값을 대입해서 0에 근사한 값이 나오는지 확인해서 아니면 다시 시도하는 방식으로 해를 보장하는 트릭을 사용한다면 좀 더 신뢰할 수 있는 가장 빠른 알고리즘이 될 것이다.

4) Regular Falsi Method

Regular Falsi Method는 Secant와 Bisection Method의 방법을 합쳤다. 그래서 Secant Method에 비해서 발산하는 경우는 없으나 범위를 너무 크게 주게 되면 함수의 모양에 따라 수렴하는 시간이 오래걸릴 수도 있다. 실제로 Secant Method에서 발산하는 상황에서 같은 파라미터를 줬을 때 5000번을 넘게 알고리즘이 돌아갔지만, 조금씩 실제 해에 가까워졌다. 따라서 느리더라도 해를 정확하게 찾아야 하는 경우에 이 방법을 쓰면 도움이 될 것같고 해가 어디쯤 위치해 있는지 예상할 수 있다면 범위를 좁게 주는게 유리해 보인다. 이에 대한 이야기는 Chapter 2에 조금 더 자세히 다룰 예정이다.

2. Comparison of Secant and Regular Falsi with a given Equation

Homework 1에서는 비슷한 두가지 방법 Secant와 Regular Falsi를 주어진 함수에 대해서 비교하는 과제가 지정되어있었다. $x^3 = -cos(x)$가 주어진 방정식이다. 해당 방정식을 풀기위해 Cos(x)를 왼쪽으로 넘겨 새로운 함수 f(x)를 만들어 두가지 알고리즘에 넣어 결과를 확인했다.

```
------------------------Secant Method-----------------------
[Starting Point] - [X0] : -1, [X1] : 0

[Root Finding...] - [iter] : 1, [X] : -0.6850733573260451
[Root Finding...] - [iter] : 2, [X] : -1.252076488909229
[Root Finding...] - [iter] : 3, [X] : -0.8072055385060926
[Root Finding...] - [iter] : 4, [X] : -0.8477837694325691
[Root Finding...] - [iter] : 5, [X] : -0.8665281869207424
[Root Finding...] - [iter] : 6, [X] : -0.8654557261640932
[Root Finding...] - [iter] : 7, [X] : -0.8654740143806452
[Root Finding...] - [iter] : 8, [X] : -0.8654740331019471
break!

========================Root Found!=======================
[Terminal Result] - [iter] : 8, [X] : -0.8654740331019471, [time] : 0.00020 sec

------------------------Regular Falsi Method-----------------------
[Starting Point] - [a] : -1, [b] : 0

[Root Finding...] - [iter] : 1, [a] : -1, [b] : -0.6850733573260451
[Root Finding...] - [iter] : 2, [a] : -1, [b] : -0.8413551256656522
[Root Finding...] - [iter] : 3, [a] : -1, [b] : -0.8625474875571268
[Root Finding...] - [iter] : 4, [a] : -1, [b] : -0.8651234556846038
[Root Finding...] - [iter] : 5, [a] : -1, [b] : -0.8654321018259392
[Root Finding...] - [iter] : 6, [a] : -1, [b] : -0.8654690187887233
[Root Finding...] - [iter] : 7, [a] : -1, [b] : -0.8654734334829767
[Root Finding...] - [iter] : 8, [a] : -1, [b] : -0.8654739613985591

 ========================Root Found!=======================
[Terminal Result] - [iter] : 8, [a] : -1, [b] : -0.8654739613985591, [time] : 0.00007 sec

Secant : root - -0.865
Regular Falsi : root - -0.865
```

둘 다 해는 동일하게 -0.865로 찾은 걸로 보아 해로 수렴이 잘 된것으로 사료된다. 주목할만한 점은, Secant 방법이 수렴이 빠르게 되니까 당연히 Secant 방법이 더 빠를 줄 알았는데, 결과는 의외로 Regular Falsi 방법이 더 빠른 수렴결과를 보여주었다. iteration이 8로 똑같기 때문에 더 면밀하게 보기 위해 시간을 알고리즘이 동작하는 시간을 측정해서 비교했다. 물론 iteration도 같기도 하고 이 정도의 차이는 구현상 이슈일 수도 있다. 또한 다른 경우에 해만 보장된다면 Secant가 빠른 경우가 훨씬 많기 때문에 이번 과제의 결과로 Regular Falsi가 더 빠르다고 말할 수는 없다. 하지만 이번 결과로 인해 Regular Falsi 방법이라고 해서 모두 느린것은 아니고 만약 해의 범위가 적절하게 주어진다면 Secant보다도 빠르게 해로 수렴할 수 있다는 것을 확인할 수 있었다.

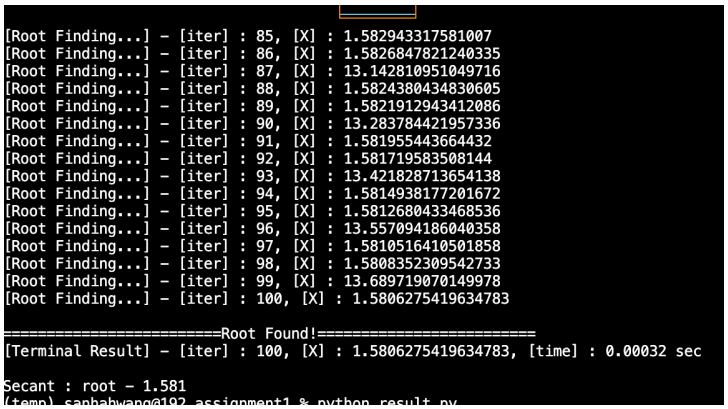
3. Appendix A : My Codes

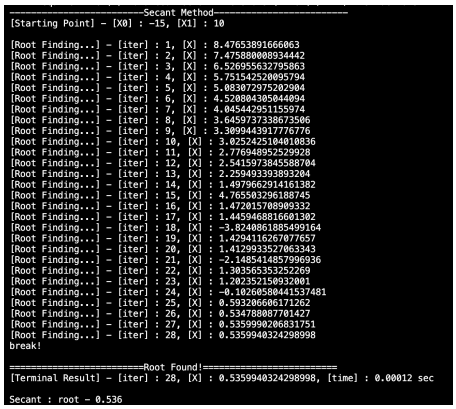
제 파이썬 스크립트는 크게 Methods를 구현한 rootfinding.py, 파라미터를 수정하며 결과를 확인한 result.py로 구성되어 있다. 실행을 위해, 스크립트가 존재하는 폴더로 디렉토리를 옮긴 이후에 'python result.py' 를 터미널에 입력하면 결과를 터미널에서 받아볼 수 있다. 가독성을 위해서 코드는 해당 보고서에 넣지 않고 따로 첨부했다.

4. Appendix B : Some results

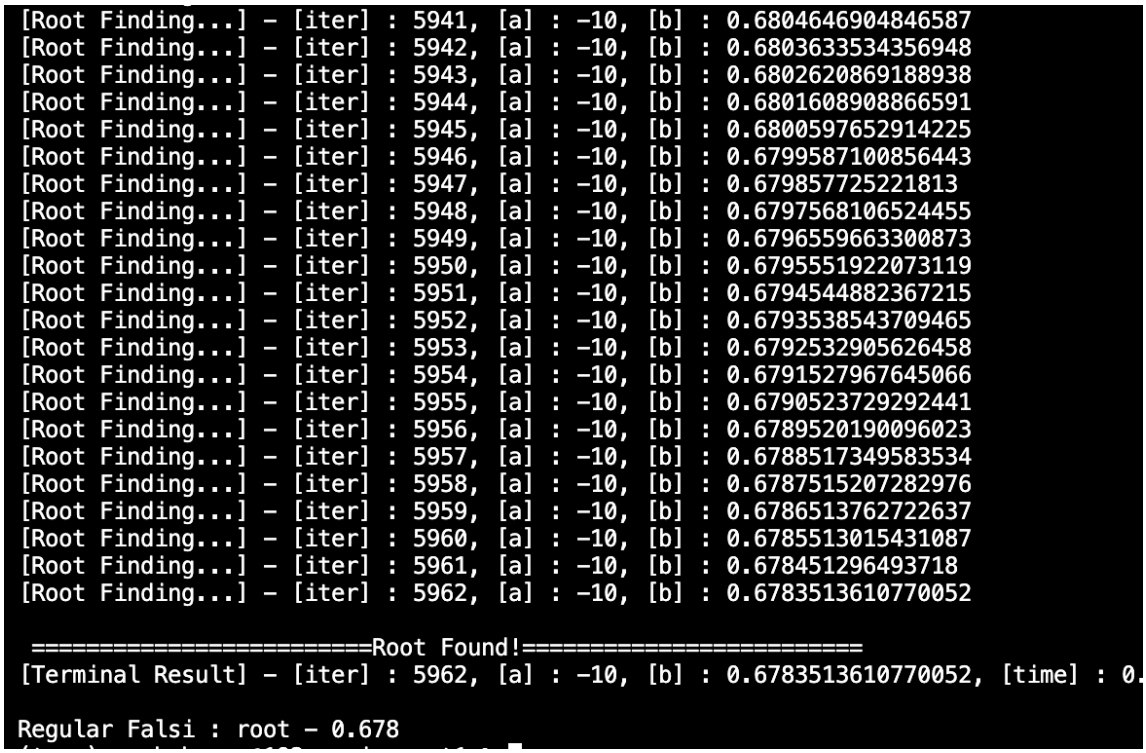
몇 가지 실험 결과를 그림으로 나타냈다.

1) Secant Method의 발산과 수렴 Case : F2함수의 해를 찾으려는 경우인데, 두 점을 각각 -10과 10으로 비교적 가까운 거리의 두점을 주었는데도 MAX Iter인 100까지 알고리즘이 동작하고 해를 찾지 못했다. 반면 조금 더 긴 거리의 두점 -15와 10을 주었을 때는 빠르게 해를 찾아가는 모습을 보였다.





2) Secant Method에서 발산했던 상황에서 같은 파라미터를 Regular Falsi Method에 줬을 때의 결과이다. 최대 iteration을 1000으로 설정하면 5962번 iteration에서 멈추고 해에 수렴하고 있는 모습을 보여준다.



3) Bisection Method 동작 결과 : (-100,100)에서 F1 함수를 찾을 경우 22 iteration만에 해로 수렴했다.

```
-------------------------Bisection Method-------------------------
[Starting Point] - [a] : -100, [b] : 100

[Root Finding...] - [iter] : 1, [a] : 0.0, [b] : 100
[Root Finding...] - [iter] : 2, [a] : 0.0, [b] : 50.0
[Root Finding...] - [iter] : 3, [a] : 0.0, [b] : 25.0
[Root Finding...] - [iter] : 4, [a] : 0.0, [b] : 12.5
[Root Finding...] - [iter] : 5, [a] : 0.0, [b] : 6.25
[Root Finding...] - [iter] : 6, [a] : 0.0, [b] : 3.125
[Root Finding...] - [iter] : 7, [a] : 0.0, [b] : 1.5625
[Root Finding...] - [iter] : 8, [a] : 0.78125, [b] : 1.5625
[Root Finding...] - [iter] : 9, [a] : 0.78125, [b] : 1.171875
[Root Finding...] - [iter] : 10, [a] : 0.9765625, [b] : 1.171875
[Root Finding...] - [iter] : 11, [a] : 0.9765625, [b] : 1.07421875
[Root Finding...] - [iter] : 12, [a] : 0.9765625, [b] : 1.025390625
[Root Finding...] - [iter] : 13, [a] : 0.9765625, [b] : 1.0009765625
[Root Finding...] - [iter] : 14, [a] : 0.98876953125, [b] : 1.0009765625
[Root Finding...] - [iter] : 15, [a] : 0.994873046875, [b] : 1.0009765625
[Root Finding...] - [iter] : 16, [a] : 0.9979248046875, [b] : 1.0009765625
[Root Finding...] - [iter] : 17, [a] : 0.99945068359375, [b] : 1.0009765625
[Root Finding...] - [iter] : 18, [a] : 0.99945068359375, [b] : 1.000213623046875
[Root Finding...] - [iter] : 19, [a] : 0.9998321533203125, [b] : 1.000213623046875
[Root Finding...] - [iter] : 20, [a] : 0.9998321533203125, [b] : 1.0000228881835938
[Root Finding...] - [iter] : 21, [a] : 0.9999275207519531, [b] : 1.0000228881835938
[Root Finding...] - [iter] : 22, [a] : 0.9999752044677734, [b] : 1.0000228881835938
0.0
=========================Root Found!=========================
[Terminal Result] - [iter] : 23, [a] : 0.9999752044677734, [b] : 1.0000228881835938, [time] : 0.000001 sec

Bisection : root - 1.000, [0.9999752044677734, 1.0000228881835938]
```