**고급소프트웨어실습1: 4주차 과제**

**20171672 이정원**

**프로그램 소개**

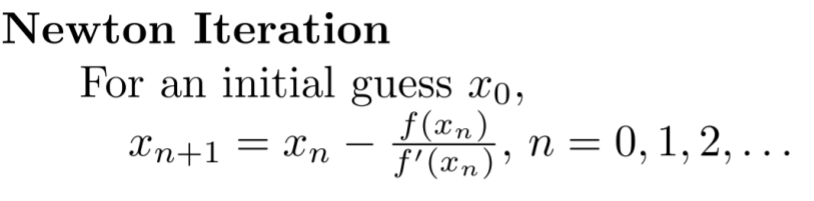
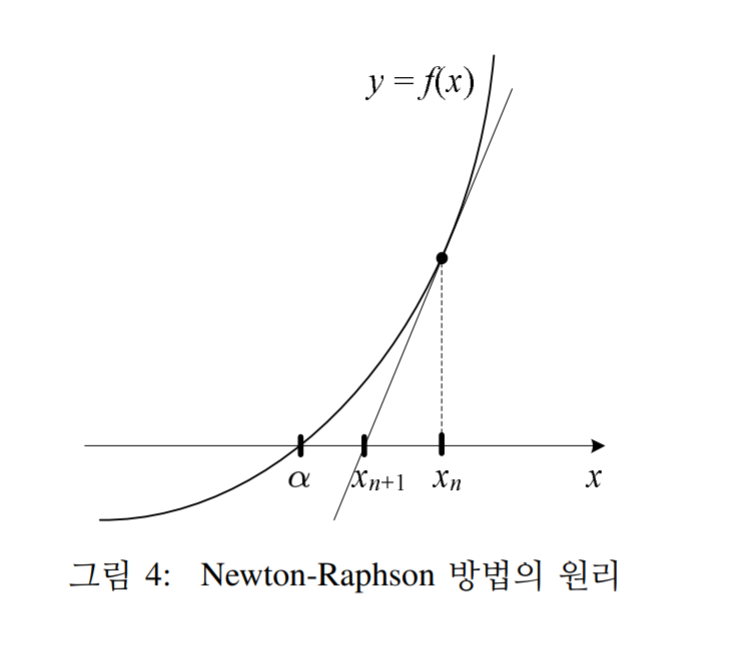
비선형 연립 방정식의 근을 구하기 위해 Newton-Raphson 방법, Secant 방법, Bisection 방법을 프로그램으로 구현한다.

**프로그램의 구동 방법**

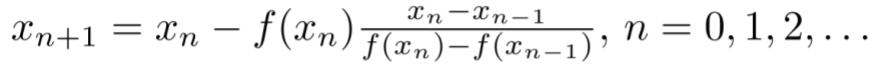
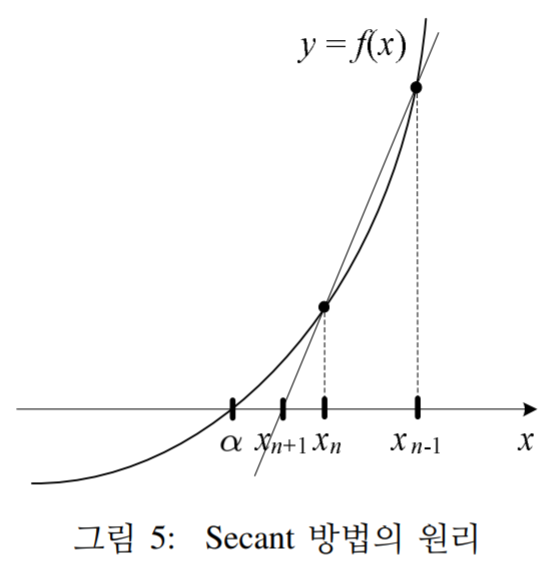
my\_solver.h와 function.cpp 에서 근을 구하려는 비선형 함수(그리고 필요한 경우에 도함수)를 특정한 변수에 입력하여 선언한다. main.cpp에서 \_f에 해당 함수를, \_fp에 해당 도함수를 할당한 후, Newton-Raphson 방법을 쓰려면 program1\_1을, Secant 방법을 쓰려면 program1\_2를, Bisection 방법을 쓰려면 program1\_3을 호출한다. program1\_1 호출 시 초기값 x0을, program1\_2 호출 시 초기값 x0과 x1을, program1\_3 호출 시 초기 구간의 경계값 a0, b0을 콘솔에 입력한다. 이후 콘솔에 반복문의 수행에 따라 결과값이 나타난다. 이 모든 과정은 result.txt에도 저장된다. 또한, 여러 근을 구해야 하는 함수인 경우 프로그램 함수는 근의 개수만큼 호출한다.

**실습 문제 1-1**

실습에서는 Newton-Raphson Method와 Secant Method을 다루었다. 두 방법의 원리는 다음과 같다.



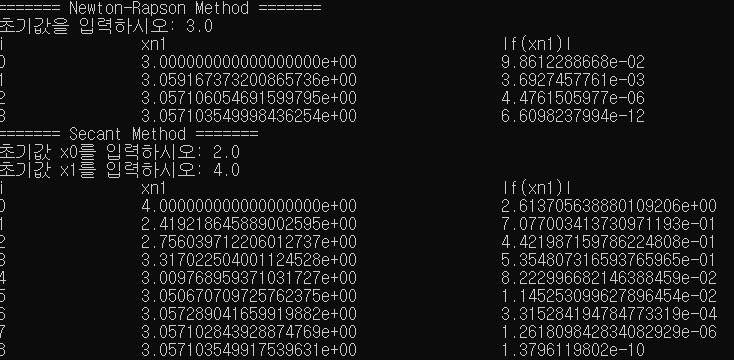
Newton-Raphson 방법의 원리와 프로그램에서 사용한 수식



Secant 방법의 원리와 프로그램에서 사용한 수식

Secant 방법은 도함수가 필요한 Newton-Raphson 방법과 달리 도함수를 원래 함수만을 이용해 근사하는 수식을 구현한 방법이다.

모든 실습 문제에 대하여 DELTA는 0.000001, EPSILON은 0.00001, Nmax는 50으로 설정하였다.

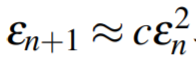


* **원하는 근을 정확히 찾고 있는지 분석하라. 과연 근이 맞는지 어떻게 확인할 수 있을까?**

Newton-Raphson Method을 쓸 때는 초기값 3.0을, Secant Method를 쓸 때는 초기값 2.0과 4.0으로 설정하였다. 반복문이 수행됨에 따라 |f(xn1)|이 0에 가까워지면 근이 맞는지 추정할 수 있다.

* **각 방법의 근의 수렴 속도가 과연 강의자료에서 설명한 속도로 수렴하는지 비교 분석한 다.**

이론적으로 Newton-Raphson 방법의 경우 반복문이 어느 정도 수행된 후에는 절대 오차가 대략적으로 다음과 같이 감소하게 된다.

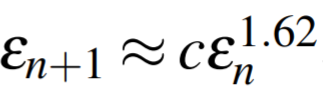


f1(x) Newton-Raphson 실행 결과에 따른 절대 오차

|  |  |
| --- | --- |
| 0 | 0.0571035 |
| 1 | 0.0020638 (여기서 c는 계산 결과 약 0.6329로 나오게 되었다.) |
| 2 | 0.0000025 (여기서 c는 계산 결과 약 0.5869로 나오게 되었다.) |

i번째에 따른 절대오차를 이용해 계산한 c의 값이 완벽히 같진 않지만 대략적으로 비슷하게 나와 이론에 어느 정도 일치하다는 것을 알 수 있었다.

이론적으로 Secant 방법의 경우 반복문이 어느 정도 수행된 후에는 절대 오차가 대략적으로 다음과 같이 감소하게 된다.



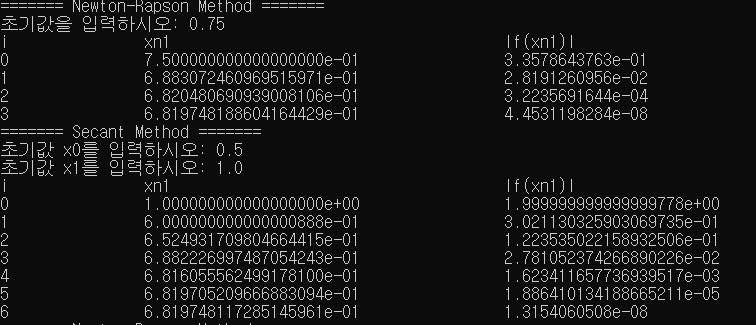
f1(x) Secant 실행 결과에 따른 절대 오차

|  |  |
| --- | --- |
| 0 | 0.9428965 |
| 1 | 0.6378849 (c ≈ 0.701633, 0과 1번째 오차에 대하여) |
| 2 | 0.3010638 (c ≈ 0.623700, 1과 2번째 오차에 대하여) |
| 3 | 0.259919 (c ≈ 1.181723, 2번째와 3번째 오차에 대하여) |
| 4 | 0.0473364 (c ≈ 0.419911, 3번째와 4번째 오차에 대하여) |
| 5 | 0.0064328 (c ≈ 0.900707, 4번째와 5번째 오차에 대하여) |
| 6 | 0.0001855 (c ≈ 0.658773, 5번째와 6번째 오차에 대하여) |
| 7 | 0.0000006561 (c ≈ 0.7282037, 6번째와 3번째 7차에 대하여) |

i번째에 따른 절대오차를 이용해 계산한 c의 값이 완벽히 같진 않지만 대략적으로 비슷하게 나와 이론에 어느 정도 일치하다는 것을 알 수 있었다.

이론적인 측면에서 수렴 속도란 반복문 수행을 통해 구한 수열에서 절대 오차가 존재하게 되는데, 오차 감소 과정을 보면 해당 방법의 수렴 속도를 비교할 수 있다. 이론적으로 Newton-Raphson의 수렴 속도는 Secant보다 빠르다.

실습에서의 수렴 속도는 원하는 정밀도를 가지는 근 추정값을 구하는 데 필요한 반복문의 수행 횟수를 의미한다고 가정하면, Newton-Raphson 방법의 i는 3번째까지, Secant 방법의 i는 8번째까지 출력된 것을 확인할 수 있다. 이를 통해 강의자료에서 설명한 것처럼 Newton-Raphson 방법이 Secant 방법보다 빠르게 근에 수렴함을 알 수 있다.

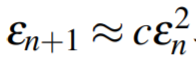


* **원하는 근을 정확히 찾고 있는지 분석하라. 과연 근이 맞는지 어떻게 확인할 수 있을까?**

Newton-Raphson Method을 쓸 때는 초기값 0.75을, Secant Method를 쓸 때는 초기값 0.5과 1.0으로 설정하였다. 반복문이 수행됨에 따라 |f(xn1)|이 0에 가까워지면 근이 맞는지 추정할 수 있다.

* **각 방법의 근의 수렴 속도가 과연 강의자료에서 설명한 속도로 수렴하는지 비교 분석한 다.**

이론적으로 Newton-Raphson 방법의 경우 반복문이 어느 정도 수행된 후에는 절대 오차가 대략적으로 다음과 같이 감소하게 된다.

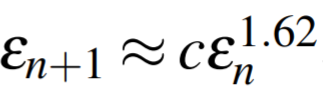


f2(x) Newton-Raphson 실행 결과에 따른 절대 오차

|  |  |
| --- | --- |
| 0 | 0.680252 |
| 1 | 0.063324 (c ≈ 0.476447, 0과 1번째 오차에 대하여) |
| 2 | 0.0007326 (c ≈ 0.182696, 1과 2번째 오차에 대하여) |

i번째에 따른 절대오차를 이용해 계산한 c의 값이 완벽히 같진 않지만 대략적으로 비슷하게 나와 이론에 어느 정도 일치하다는 것을 알 수 있었다.

이론적으로 Secant 방법의 경우 반복문이 어느 정도 수행된 후에는 절대 오차가 대략적으로 다음과 같이 감소하게 된다.



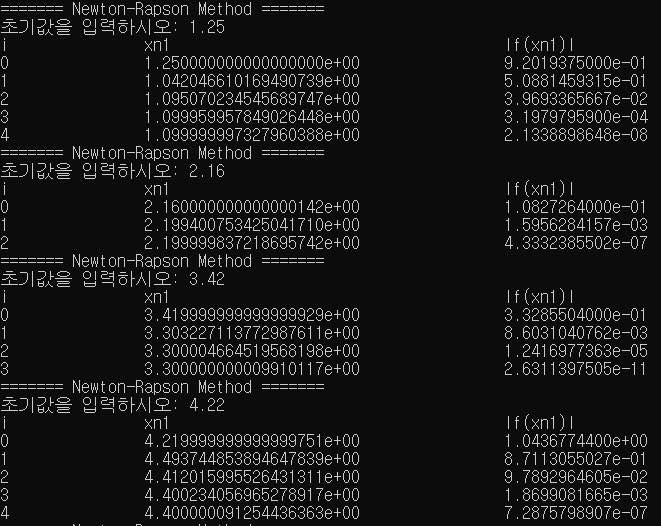
f2(x) Secant 실행 결과에 따른 절대 오차

|  |  |
| --- | --- |
| 0 | 5.819748 |
| 1 | 0.819748 (c ≈ 0.047264, 0과 1번째 오차에 대하여) |
| 2 | 0.2948163 (c ≈ 0.406807, 1과 2번째 오차에 대하여) |
| 3 | 0.0624789 (c ≈ 0.451918, 2번째와 3번째 오차에 대하여) |
| 4 | 0.00369244 (c ≈ 0.3297808, 3번째와 4번째 오차에 대하여) |
| 5 | 0.0000428 (c ≈ 0.3735906, 4번째와 5번째 오차에 대하여) |

i번째에 따른 절대오차를 이용해 계산한 c의 값이 완벽히 같진 않지만 대략적으로 비슷하게 나와 이론에 어느 정도 일치하다는 것을 알 수 있었다.

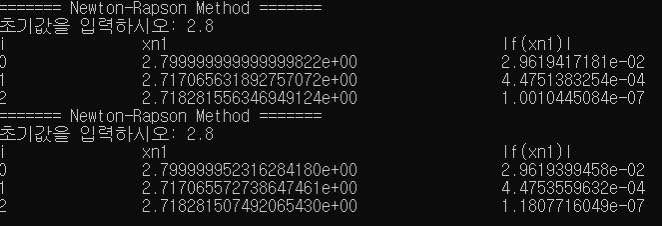
위에서 설명했던 것처럼 가정하면, Newton-Raphson 방법의 i는 3번째까지, Secant 방법의 i는 6번째까지 출력된 것을 확인할 수 있다. 이를 통해 강의자료에서 설명한 것처럼 Newton-Raphson 방법이 Secant 방법보다 빠르게 근에 수렴함을 알 수 있다.

**실습 문제 1-2**



이 방정식은 네 개의 서로 다른 실근을 가지고 있으며, 해당 실근의 범위를 알기 때문에 범위 내의 적절한 초기값을 설정하여 Newton-Raphson 프로그램을 호출한다. 실근이 4개이기 때문에 프로그램을 4번 호출하였다. 초기값은 각 구간의 중간값으로 설정하였다. (1.25, 2.16, 3.42, 4.22)

**실습 문제 1-4**



위는 double-precision, 아래는 single-precision 버전으로 계산한 반복문의 수행 결과이다. 초기값은 모두 2.8로 설정하였다.

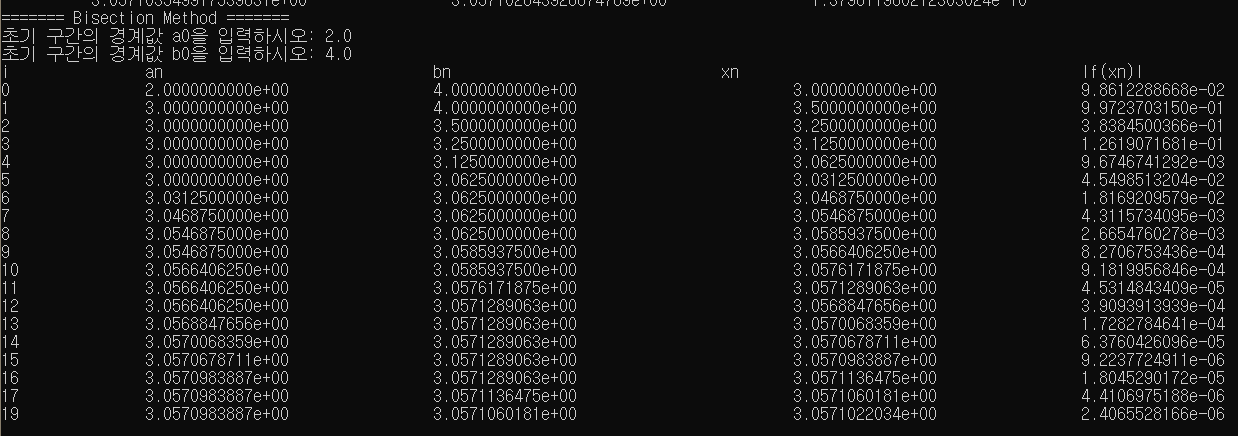
* **부동 소수점 연산의 정밀도가 다른 두 방법이 구한 근의 값이 정확한 근 e(2.718281828459045235360287471352...)와 비교하여 어떤 차이가 있는지?**

정확한 근과 비교했을 때 e와 더 근접한 근을 도출해낸 버전은 double-precision 버전이다. 이를 통해 single-precision 버전의 경우 double-precision 버전의 경우에 비해 정밀도는 떨어진다는 것을 확인할 수 있다.

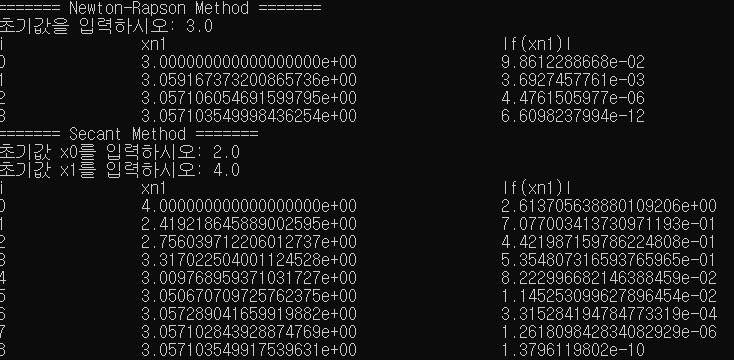
**과제 1 – Bisection**

비선형 연립 방정식의 근을 구하기 위해 초기 구간 경계값 a0, b0(f(a0)과 f(b0)이 서로 다른 부호를 가진)을 콘솔로 입력받은 후 a0과 b0의 중간값을 xn에 할당한다. 반복문 종료조건(세 가지 종료 조건과 \_f(xn)이 정확히 0이 되는 경우)을 검토한 후 \_f(an)과 \_f(xn)의 부호, \_f(bn)과 \_f(xn)의 부호를 비교한다. 전자의 부호가 반대면 다음 반복문의 구간은 [an,xn]이 될 것이고, 후자의 부호가 반대면 다음 반복문의 구간은 [xn, bn]이 될 것이다.

* **프로그램이 완성되면, 실습 시간에 사용한 세 개의 함수에 대해 적절한 초기 구간을 사용하여 올바르게 근에 수렴하는지에 대한 분석하라.**

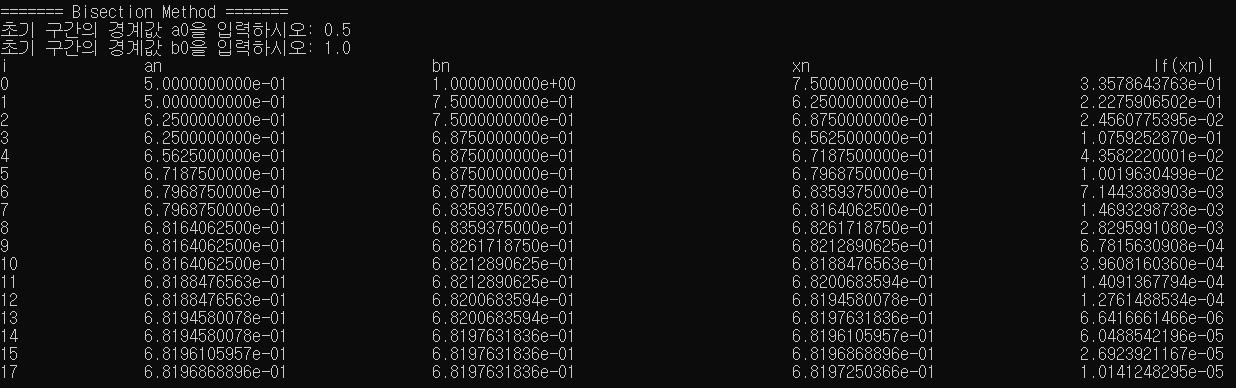


f1(x)- Bisection Method

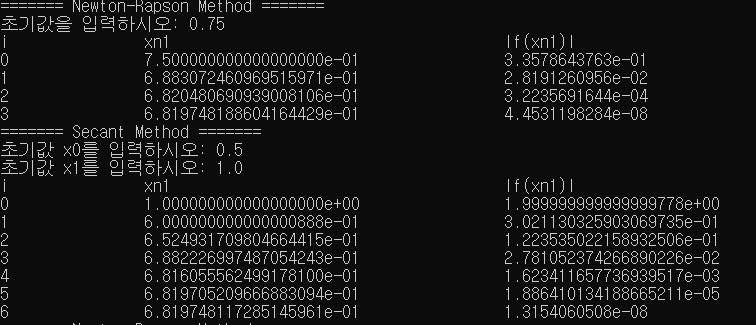


f1(x) – Newton Raphson Method & Secant Method

Bisection Method의 적절한 초기 구간 경계값을 설정하기 위해 함수에 값을 대입해 부호를 비교하여 설정하였다. 2.0과 4.0이 조건에 맞아 각각 a0과 b0에 할당하였다. 세 방법의 x의 값과 |f(x)|이 0에 가까이 가는지를 확인하여 올바르게 근에 수렴하는지 확인할 수 있었다.

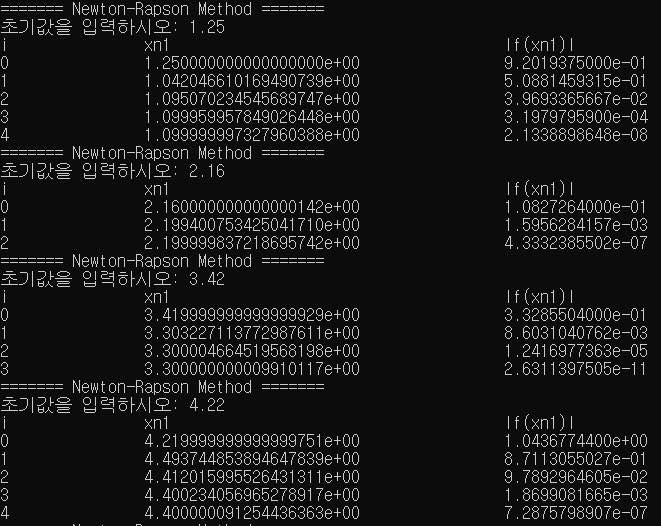


f2(x)- Bisection Method



f2(x) – Newton Raphson Method & Secant Method

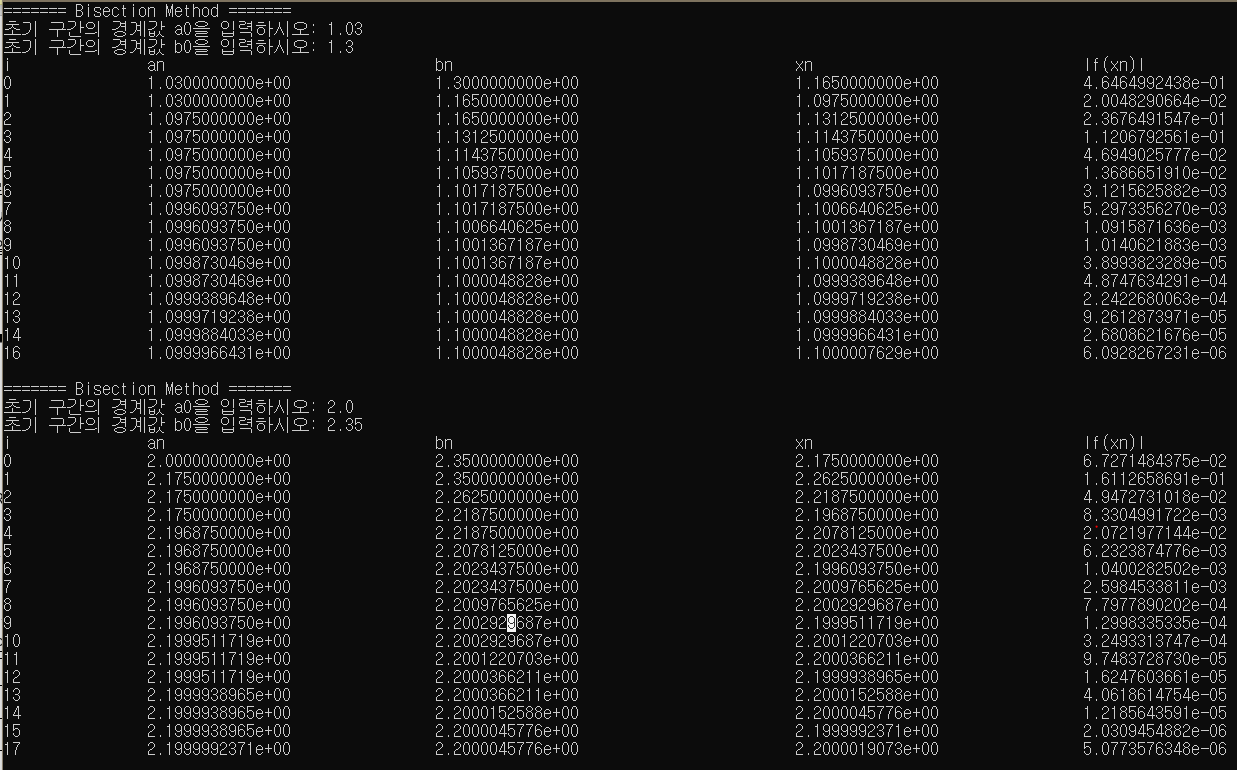
Bisection Method의 적절한 초기 구간 경계값은 0.5와 1.0이 조건에 맞아 각각 a0과 b0에 할당하였다. 세 방법의 x의 값과 |f(x)|이 0에 가까이 가는지를 확인하여 올바르게 근에 수렴하는지 확인할 수 있었다.

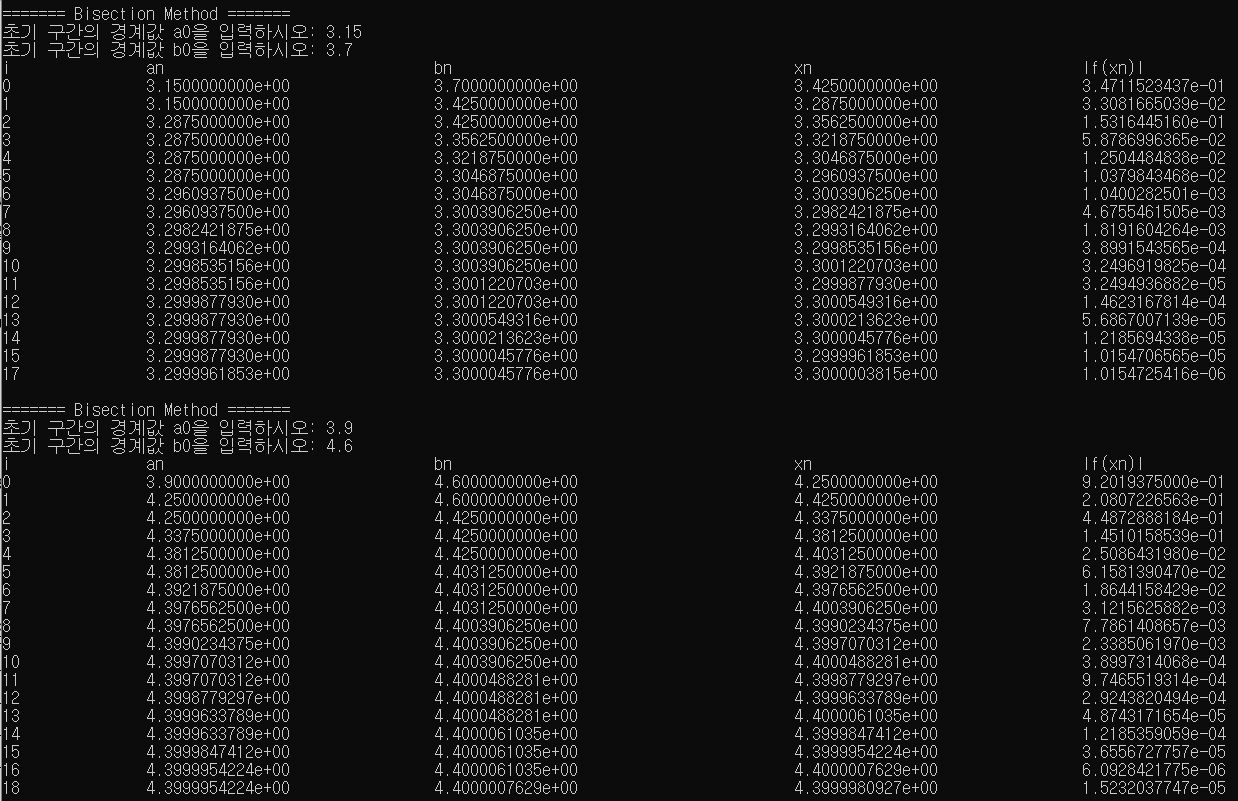


f3(x)- Newton-Raphson Method



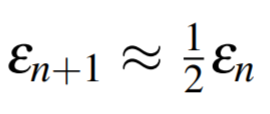
f3(x)- Secant Method





f3(x)- Bisection Method

f3(x)의 근을 Secant Method와 Bisection Method로 구하는 과정은 Newton-Raphson과 같이 함수를 4번 호출하여 각 근에 대한 초기값 두 개를 콘솔로 입력받아 진행하였다. Bisection Method의 적절한 초기 구간 경계값은 근이 존재하는 범위를 참고하였고, [1.03, 1.3], [2.0,2.35], [3.15, 3.7], [3.9, 4.6]이 조건에 맞아 각각 a0과 b0에 할당하였다. 세 방법의 x의 값과 |f(x)|이 0에 가까이 가는지를 확인하여 올바르게 근에 수렴하는지 확인할 수 있었다.

* **Bisection 수렴 속도는 선형적인, 즉 형태를 보인다. 위의 세 함수에 대한 방정식을 대상으로 Newton-Raphson 방법, Secant 방법, 그리고 Bisection 방법을 적용해 보고, 과연 각 방법이 이론적인 수렴 속도를 보이는지를 분석하라.**

*(f1(x) 함수와 f2(x)의 Newton-Raphson 방법과 Secant 방법의 이론적인 수렴 속도의 분석은 앞에서 이미 진행하였다.)*

f1(x) Bisection Method 실행 결과에 따른 절대 오차

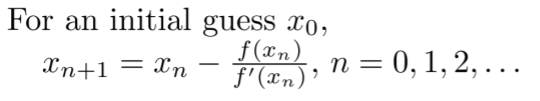
|  |  |
| --- | --- |
| 0 | 0.0571022 |
| 1 | 0.4428978 (c ≈ 7.756, 0과 1번째 오차에 대하여) |
| 2 | 0.1928978 (c ≈ 0.435535, 1번째와 2번째 오차에 대하여) |
| 3 | 0.0678978 (c ≈ 0.351988, 2번째와 3번째 오차에 대하여) |
| 4 | 0.053978 (c ≈ 0.794988, 3번째와 4번째 오차에 대하여) |
| 5 | 0.0258522 (c ≈ 0.478939, 4과 5번째 오차에 대하여) |
| 6 | 0.0102272 (c ≈ 0.395602, 5과 6번째 오차에 대하여) |
| ... | ... |

Bisection Method에서 c는 n+1번째 절대오차에서 n번째 절대오차를 나눈 값으로 계산하였다. i번째에 따른 절대오차를 이용해 계산한 c의 값이 이론처럼 완벽히 0.5가 되지는 않지만 대략적으로 비슷하게 나와 이론에 어느 정도 일치하다는 것을 알 수 있었다. f2(x), f3(x)에서도 마찬가지로 위 표와 같이 진행하여 확인하였다.

Bisection Method는 i가 19번째까지 반복문이 진행되었으므로 세 가지 방법 중 가장 속도가 오래 걸린다는 것을 확인할 수 있었다. 따라서, Newton-Raphson – Secant – Bisection 방법 순으로 속도가 빠르다는 것을 알 수 있다.

**과제 2 - Newton-Raphson 방법으로 방정식의 해를 풀이**

비선형 함수 f(x)에 대하여 f(x) = 0이란 방정식을 풀어 근을 찾기 위해 Newton-Raphson 방법을 이용하였다. 뉴턴 반복문은 다음과 같다.



초기값 x0에 대하여 위 수식을 반복하며, 반복문 종료 조건에 만족하면 반복을 종료한다. 반복문 종료 조건은 다음과 같다.

1. 현재 구한 xn+1에 대해 함수 값이 충분히 작은가? fabs(f(xn+1)) < DELTA
2. 충분히 많은 횟수만큼 반복문을 수행하였는가? i >= Nmax
3. 현재 구한 xn+1이 직전에 구한 xn에 비해 더 이상 의미 있는 진전을 하지 않는가? fabs(xn+1 – xn) < EPSILON

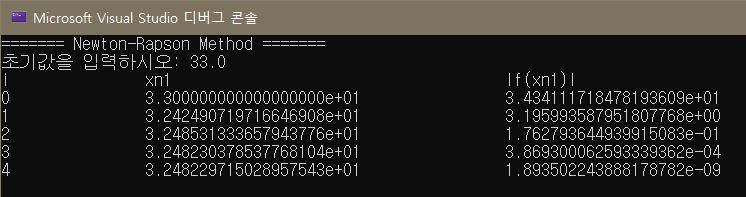
여기서 DELTA는 0.000001, EPSILON은 0.00001, Nmax는 50으로 설정하였다.

문제에서

이 변수들에 대하여 f(x)는 다음과 같이 설정하였다. (함수 이름: \_fA)

또한, f(x)의 도함수 f’(x)는 다음과 같이 설정하였다. (함수 이름: \_fpA)

main.cpp에서 \_f는 \_fA를, \_fp는 \_fpA를 할당하여 Newton-Raphson 방법을 구현한 함수 program1\_1(fp)를 호출하여 계산을 진행하였다. 초기값 x0은 33으로 설정하였다. 반복 결과는 다음과 같다.



위처럼 |f(xn1)|이 0에 가깝다는 것을 확인하여 근이 맞는지 확인할 수 있었다.