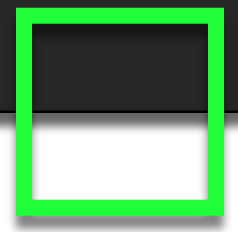
#수치 해석
#과제 1

Finding Roots

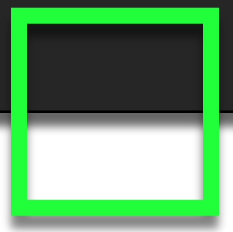
컴퓨터소프트웨어학부
2018008395 박정호



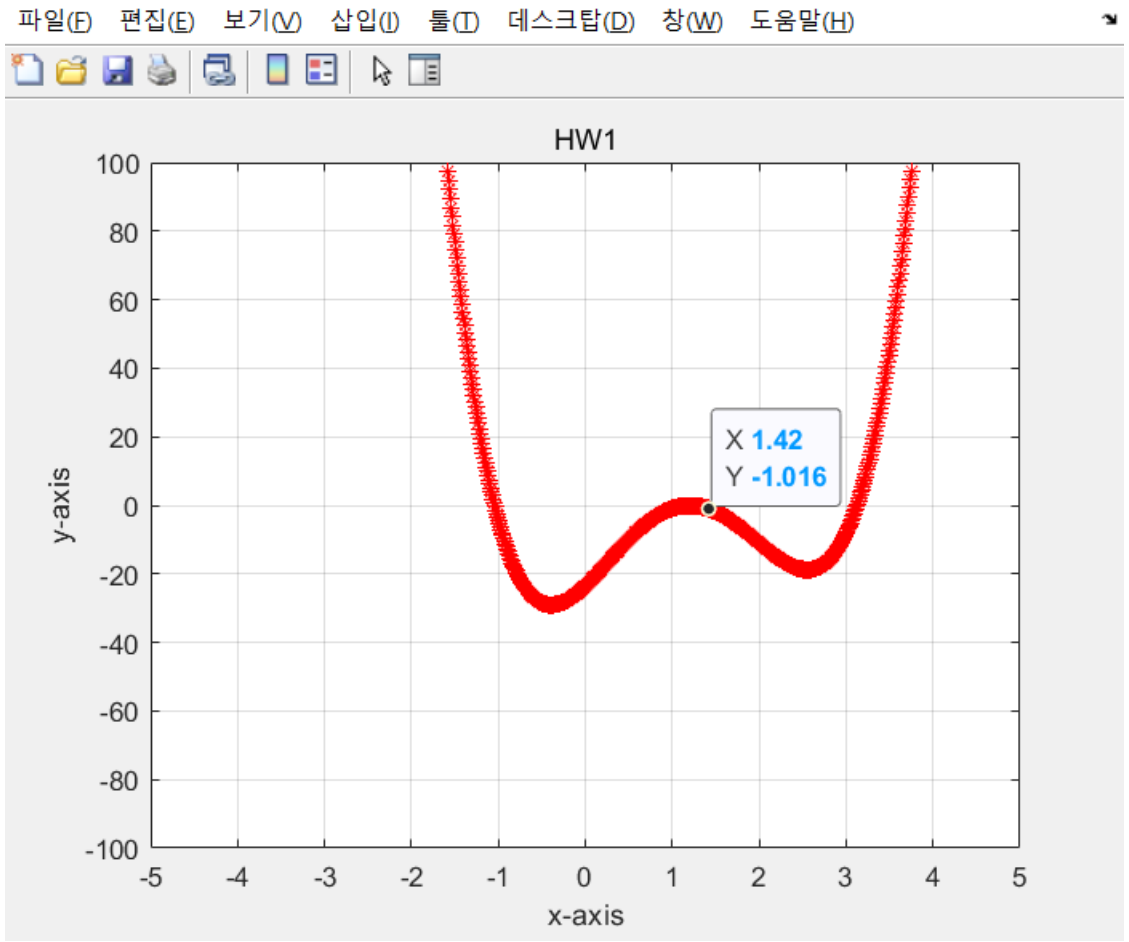
ALOHA
The algorithm club.

#1

Guessing



MATLAB을 이용한 함수 개형 파악



우선 함수 개형을 파악하기 위해 MATLAB을 사용했다.

우선 근은 대략적으로 3개 정도 있으리라 판단할 수 있었고, 그 범위는 rough하게 보아서 (-2, 4)라고 할 수 있을 것이다. 이 범위를 bisection method를 적용할 범위로 두었다.

또한 대략적으로 근의 위치를 -1, 1, 1.5, 3정도로 유추할 수 있는데 이 값을 Newton-Raphson metho의 input으로 쓸 것이다.

함수 개형 파악의 근거

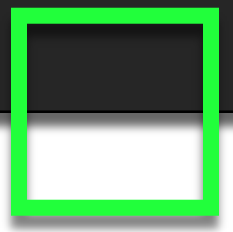
앞에서 구한 개형으로 필자가 Guess한 결과는 다음과 같다.

1. Bisection Method
   (-2, 4)의 구간 안에 함수의 근이 있다.
   조금 더 정확하게 구간을 나눠서 구할 수도 있겠지만, 필자는 이 구간을 일정한 길이로 나누어서 각 부분에 대해서 모두 Bisection Method를 적용했다. 정확하게는 각 부분 중 sign change가 발생하는 부분에 한해서 진행했는데 그 이유는 다음과 같다.
   우선 사람의 눈으로 판단하기에 한계가 있었다. 앞의 그래프에서 [1, 1.5]에 해당하는 구간에는 대략적으로 중근으로 추정되는 근이 있는데, 여기서 함수값이 양수인 부분을 찾기가 힘들었다. 따라서 그냥 전체 구간을 잘게 나누어서 그 양수 부분을 찾기로 했다.

2. Newton-Raphson Method
   -1, 1, 1.5, 3의 실수값이 함수의 근에 근접하다.
   위에서도 언급한 이유로 아마 initial value가 1인 결과와 1.5인 결과는 대략 같을 것으로 추정되지만, 아무튼 4차 함수이기에 4개의 initial value를 준비했다.

#2

구현



기본 함수

```
double bisection(double minus, double plus) {
    double ans = NAN;
    if (equal(f(minus), 0)) {
        return minus;
    }
    if (equal(f(plus), 0)) {
        return plus;
    }
    while (!equal(minus, plus)) {
        ans = (minus + plus) / 2;
        double val = f(ans);
        if (equal(val, 0)) {
            return ans;
        }
        else if (val > 0) {
            plus = ans;
        }
        else {
            minus = ans;
        }
    }
    return ans;
}
```

기본적으로 필요한 함수들이다.

우선 f는 주어진 함수, f_p는 그 도함수이다. 이 함수는 굳이 프로그램의 입력값으로 두지 않고 하드코딩으로 구현했다.

equal은 두 실수의 차이가 일정 범위 이하일 경우 같다고 하는 함수이다. 허용 오차 범위를 위해서 구현했다.

Bisection Method

```
double bisection(double minus, double plus) {
    double ans = NAN;
    if (equal(f(minus), 0)) {
        return minus;
    }
    if (equal(f(plus), 0)) {
        return plus;
    }
    while (!equal(minus, plus)) {
        ans = (minus + plus) / 2;
        double val = f(ans);
        if (equal(val, 0)) {
            return ans;
        }
        else if (val > 0) {
            plus = ans;
        }
        else {
            minus = ans;
        }
    }
    return ans;
}
```

Bisection Method를 구현한 함수이다.
인자로 주어진 minus와 plus는 각각 함수값이 음수인 x값과 양수인 x값이다.

기본적으로 minus와 plus의 함수값이 0에 가깝다면 추가 연산 없이 그 값을 return했다.

이후 코드가 Bisection Method의 연산인데, 기본적으로는 이분 탐색 알고리즘의 형태를 띄고 있다. 이는 각 구간을 반으로 나누는 연산을 반복하기에 이런 형태로 구현을 했다.

현재 구간의 중점을 구한 후, 그 x좌표의 함수값을 구해서 0이라면 바로 ans를 return하고, 그렇지 않다면 구간을 조정했다. 만약 함수값이 양수라면 plus를 현재 ans로 변경하고, 그렇지 않다면 음수라는 것이므로 minus를 ans로 변경했다.

Bisection Method - 결과

```
int main(void)
{
    cout << fixed;
    cout.precision(10);
    for (double i = -2.0001; i <= 4.0001; i += 0.0001) {
        double a = i, b = i + 0.0001;

        if (f(a) * f(b) > 0) {
            continue;
        }
        else if (f(a) > 0) {
            swap(a, b);
        }
        cout << bisection(a, b) << "\n";
    }
}
```

Bisection Method를 호출하는 main함수의 형태이다.

위에서 언급한 것처럼 -2에서 4까지의 구간을 0.0001의 길이로 나누는 방식을 사용했다. 여기서 각 부분의 양 끝값을 a, b로 두었는데, 만약 이 a, b의 함수값의 부호가 같다면, 두 함수값을 곱했을 때 그 값이 양수가 될 것이고, 이 경우에는 Bisection Method를 사용할 수 없기 때문에 사용하지 않았다.

또한, 필자가 구현한 bisection 함수는 함수값이 양수인 x값과 음수인 x값을 구분해서 받기 때문에, 혹시 a의 함수값이 양수일 경우, a와 b의 값을 바꾸어서 진행했다.

함수의 호출 결과이다.
다음과 같이 근이 구해진 것을 확인할 수 있다. 또한, 예상과 같이 두번째 근은 중근에 가까울 것으로 확인된다.

Newton-Raphson Method

```
double newton_raphson(double x) {
    double prev;

    do {
        prev = x;
        x = x - f(x) / f_p(x);
    } while (!equal(x, prev));
    return x;
}
```

Newton-Raphson Method를 구현한 함수와 이를 호출하는 main함수이다. 인자로 주어진 x는 함수로부터 guess된 initial value이다.

반복문을 사용해서 다음 x를 구하고 있는데, 여기서 위에서 구한 도함수가 같이 사용된다. 또한, 다음 x를 구하기 전에 prev라는 변수에 기존 x를 저장하고 있는데, 이는 이 반복문의 종료 조건인 이전 x와 다음 x의 차이가 특정 값 이하일 경우를 계산하기 위함이다.

main 함수는 앞에서 구한 initial value를 하나씩 인자로 사용해서 newton_raphson 함수를 호출한다.

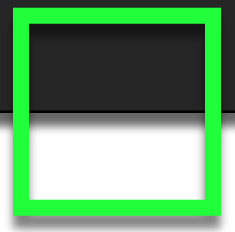
```
-1.0440000000
1.1999938814
1.2000086857
3.1240000000

C:₩Users₩pch68₩source₩repos₩Numerical
 0개).
이 창을 닫으려면 아무 키나 누르세요...
```

결과는 다음과 같다.
앞에서 구한 것과 거의 같은 값을 가지고 있고, 역시나 두번째 근은 중근에 가까울 것이라는 것을 알 수 있다.

#3

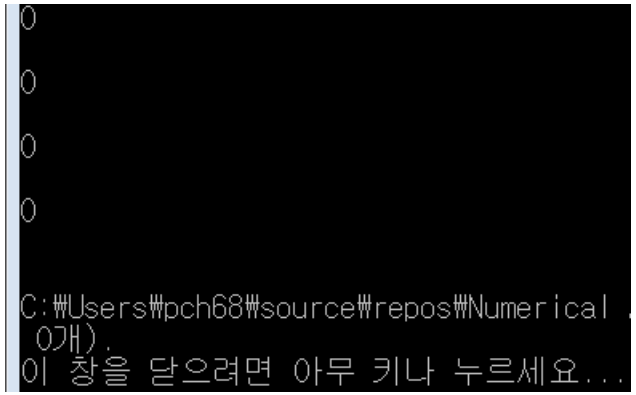
비교

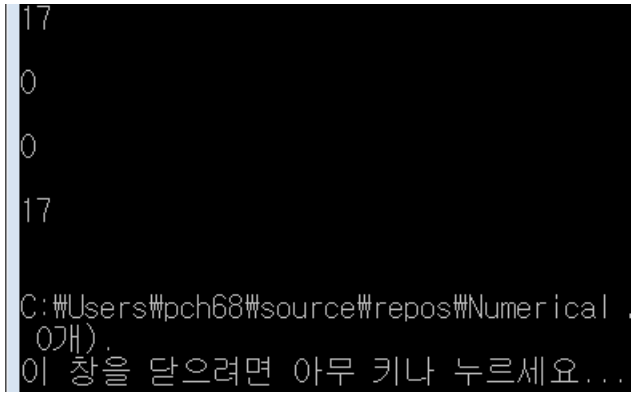


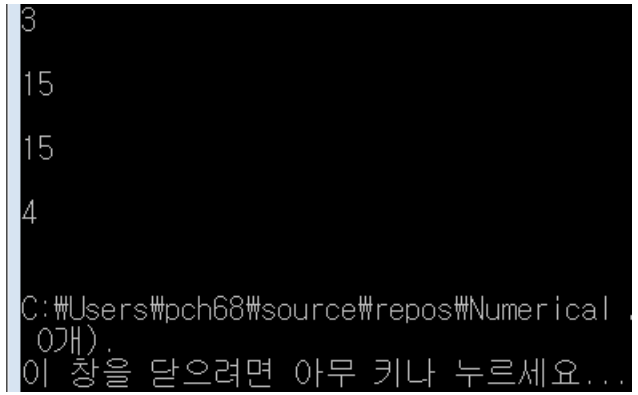
Iteration 횟수

Bisection Method는 구간을 어떻게 잡느냐에 따라 그 효율이 달라졌다.
아래의 첫번째 결과는 구간 길이를 0.0001로 했을 때, 두번째 결과는 0.1로 했을 때인데, 구간의 길이를 잘게 나눌 경우, 구간의 끝값이 함수의 근이 될 가능성이 높아져서 iteration의 횟수가 적어졌다. 하지만 이는 특수한 경우로, 더 높은 정확도를 요구하는 등의 경우에서는 여러 번의 iteration이 필요할 수 있다.

Newton-Raphson Method는 평균적으로 적은 iteration을 보여주었다. 하지만 이 역시 guess를 어느정도 정확하게 할 수 있느냐에 따라 iteration의 횟수가 달라질 것이다.







구현의 난이도

사실 두 방법 모두 그렇게 어려운 구현은 아니었다.

하지만 Bisection Method의 경우 기본적으로 이분 탐색의 형태를 띄고 있으며, 실수 정확도 문제로 인해 중근에 가까운 근의 경우 오차 범위나 구간 길이를 잘못 설정했을 때 찾지 못하는 경우가 많았다. 이 부분이 조금 번거로웠다.

Newton-Raphson Method는 구현 자체는 매우 간단했다. 도함수를 하드 코딩했기에 가능했다고 생각한다. 하지만, 만약 도함수를 구하기 어려운 함수형 같은 경우에는 오히려 Bisection Method가 더 구현이 간단할 수도 있다고 생각한다. 하지만 Bisection Method에서는 찾기 어려웠던 중근을 여기서는 별다른 처리 없이 찾을 수 있다는 점에서는 장점이 있다고 생각한다.