BISECTION/NEWTON-RAPSHON

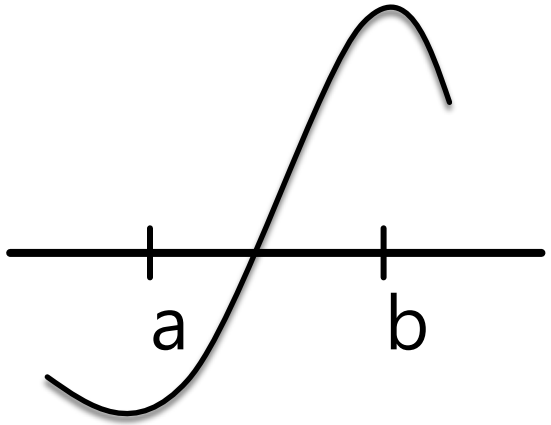
서왕규

2014004066

CONTENTS

01. Bracketing method : Bisection

1. 구간의 끝에서 부호가 변하면, 근이 적어도 한 개 존재한다.



$$f(a) * f(b) < 0$$

2. 구간을 절반으로 나누어 가며, 1번을 확인한다.

3. 종결 조건을 설정한다.

$$(New - old) / new * 100 < threshold$$

01. Bracketing method : Bisection

Implementation

```python
def upsilon(new, old) :
    return abs((new - old) / new) * 100

def func(point) :
    global eq
    result = 0
    for idx, coefficient in enumerate(eq) :
        result += pow(point, idx) * coefficient
    return result

def bisection(lower, upper, result) :
    center = (lower+upper)/2
    if func(lower) * func(upper) > 0 :
        return False
    elif func(lower) == 0 :
        result.append(lower)
        return [lower]
    elif func(upper) == 0 :
        result.append(upper)
        return [upper]
    elif upsilon(center, lower) < th :
        result.append(center)
        return [center]
    else :
        bisection(lower, center, result)
        bisection(center, upper, result)
        return result
```

Upsilon(신규값, 이전값) : 종결 조건을 계산

func(x값) : 함숫값을 계산

bisection(x값) : bracket을 두 구간으로 나누어가며 종결 조건이 만족할 때 까지 탐색

01. Bracketing method : Bisection Advanced

Implementation : parallel programming

```
elif args.parallel == "N" :
    t = time.time()
    for elem in bracket :
        resultmtmp.append(bisection.setAndCalculate(elem, thm, eq))
```

싱글 코어를 통해 선형 탐색

Implementation : parallel programming

```
if args.parallel == "Y" :
    processCount = multiprocessing.cpu_count()
    pool = multiprocessing.Pool(processes= processCount)
    func = partial(bisection.setAndCalculate, threshold = thm, equation= eq)
    t = time.time()
    resultmtmp = pool.map(func, bracket)
    pool.close()
    pool.join()
```

각 구간을 병렬 탐색

Result

```
PS C:\Users\tjdhk\OneDrive\바탕 화면\4-2\수치해석_녹화\HW1> python main.py -m bisection -p Y -e "[-23.4824832, 24.161472, 15.85272, -22.4, 5]"
Using time : 0.8119997978210449
Answer : -1.0435
Answer : 1.2
Answer : 3.1245000000000003
PS C:\Users\tjdhk\OneDrive\바탕 화면\4-2\수치해석_녹화\HW1> python main.py -m bisection -p N -e "[-23.4824832, 24.161472, 15.85272, -22.4, 5]"
Using time : 0.8810007572174072
Answer : -1.0435
Answer : 1.2
Answer : 3.1245000000000003
```

Test

과제의 방정식은 -5와 5 이상에서는 양수이고, 무한대로 발산한다는 것을 쉽게 파악할 수 있다. 병렬 수행의 벤치마크를 위해 식이 복잡하거나, 근이 모여있는 경우를 가정하여 탐색 범위, 구간 크기, 상대적 오차 제한을 통해 테스트를 진행한다. 5회의 테스트를 평균내어 산출한다.

단위 : 초

Search Range	Bracket Size	Threshold	Single core	6 core
[-10,10]	0.001	0.1	0.670996	0.461001
[-100,100]	0.001	0.1	0.802000	0.546997
[-1000,1000]	0.001	0.1	8.170002	3.501998
[-1000,1000]	0.001	0.01	8.453999	3.556000
[-1000,1000]	0.0001	0.01	8.572519	3.756516
[-1000,1000]	0.0001	1	8.540999	3.402998

탐색 범위가 넓을 경우, 병렬 프로그래밍의 스케쥴 오버헤드를 뛰어 넘는 속도 향상이 존재한다.

02. Newton-Raphson

1. 근과의 사이에 변곡점이 없는 적절한 초기값 X0을 설정한다.

2. 아래의 식을 통해 근사한다.

$$X(i+1) = X(i) - f(x(i))/f'(x(i))$$

3. 종결 조건을 설정한다.

$$(New - old) / new * 100 < threshold$$

02. Newton-Raphson

Implementation

```
def newton(old) :
    global th
    de = derivatedfunc(old)
    if de == 0 :
        return False

    new = old - (func(old)/de)
    count = 0
    flag = True
    while upsilon(new, old) > th :
        tmp = new
        de = derivatedfunc(new)
        if de == 0 :
            print("Wrong initial point")
            flag = False
            break
        new = new - (func(new)/de)
        if count > 1000 and abs(new - tmp) - abs(tmp - old) > 0 :
            print("Wrong initial point")
            flag = False
            break
        old = tmp
        count = count + 1
    if flag == False :
        return False
    return new
```

newton(이전 x값) : X(i)값을 통하여 X(i+1)을 구한다.

예외 처리 : 잘못된 Initial point를 잡았을 때를 방지하여, 예외를 설정한다.

```
seed = [-1,1,5]
```

적절한 초기값 3가지를 설정한다. 이 과제에서는, [-5, 5]의 밖의 범위에서 발산하는 것을 쉽게 알 수 있으므로, 그 내부의 3점을 설정한다.

02. Newton-Raphson

Implementation

```python
def upsilon(new, old) :
    if new == 0 :
        return abs(new - old)
    return abs((new - old) / new) * 100


def func(point) :
    global eq
    result = 0
    for idx, coefficient in enumerate(eq) :
        result += pow(point, idx) * coefficient
    return result

def differential(equation) :
    result = []
    for idx, coefficient in enumerate(equation) :
        if idx != 0 :
            result.append(idx * coefficient)
    return result

def derivatedfunc(point) :
    global eq
    result = 0
    for idx, coefficient in enumerate(dif) :
        result += pow(point, idx) * coefficient
    return result
```

Upsilon(신규값, 이전값) : 종결 조건을 계산

func(x값) : 함숫값을 계산

differential(수식 계수 배열) : 미분한 값의 계수 배열을 반환해준다.

Result

```
PS C:\Users\tjdhk\OneDrive\바탕 화면\4-2\수치해석_녹화\hw1> python main.py -m newton -p N -e "[-23.4824832, 24.161472, 15.85272, -22.4, 5]"
Using time : 0.20200037956237793
Answer : -1.0440000000437117
Answer : 1.199216871264406
Answer : 3.1240014706385564
```

02. Newton-Raphson

Implementation

```
def newton(old) :
    global th
    de = derivatedfunc(old)
    if de == 0 :
        return False

    new = old - (func(old)/de)
    count = 0
    flag = True
    while upsilon(new, old) > th :
        tmp = new
        de = derivatedfunc(new)
        if de == 0 :
            print("Wrong initial point")
            flag = False
            break
        new = new - (func(new)/de)
        if count > 1000 and abs(new - tmp) - abs(tmp - old) > 0 :
            print("Wrong initial point")
            flag = False
            break
        old = tmp
        count = count + 1
    if flag == False :
        return False
    return new
```

newton(이전 x값) : X(i)값을 통하여 X(i+1)을 구한다.

예외 처리 : 잘못된 Initial point를 잡았을 때를 방지하여, 예외를 설정한다.

```
seed = [-1,1,5]
```

적절한 초기값 3가지를 설정한다. 이 과제에서는, [-5, 5]의 밖의 범위에서 발산하는 것을 쉽게 알 수 있으므로, 그 내부의 3점을 설정한다.

Test

초기값 X(0)를 추정할 수 있는 경우와 알기 어려운 경우를 나누어 테스트 한다. 함수의 그래프를 예측하기 어려운 경우, 일정 범위를 0.01단위로 나누어 X(0)으로 설정하여 실행한다. Bisection의 Bracket size는 0.01로 설정한다.

Initial Value	Value	Threshold	Newton	Newton[6 core]	Bisection[6 core]
Predictable	(-1,1,5)	0.1	0.193999	-	0.461001
Unpredictable	[-100,100]	0.1	0.429547	0.491996	0.546997
Unpredictable	[-1000,1000]	0.1	3.124999	2.318998	3.501998
Unpredictable	[-5000,5000]	0.1	19.511795	13.097069	16.702539

Newton-Raphson는 그래프를 바탕으로 근의 위치를 추정할 수 있을 경우 효과적이다.

3. Advanced : Hybrid Algorithm

In [1], Bisection을 통해 2번 근사시킨 뒤, Newton-Rapson으로 근사한다.

아래의 논문에서 나온 제안에 추가로, 수식의 근의 위치를 예측하기 어려운 경우를 가정하여, [-5000, 5000] 구간을 탐색하여, 근이 있는 구간을 Bisection으로 2번 근사시킨 뒤, Newton Rapson으로 근사한다.

Initial Value	Value	Threshold	Hybrid	Newton[6 core]	Bisection[6 core]
Unpredictable	[-5000,5000]	0.01	1.910999	13.515673	17.120669

Result

```
PS C:\Users\tjdhk\OneDrive\바탕 화면\4-2\수치해석_녹화\hw1> python main.py -m hybrid -p Y -e "[-23.4824832, 24.161472, 15.85272, -22.4, 5]"
Using time : 1.9109992980957031
Answer : -1.0440000000014429
Answer : 1.2
Answer : 3.1240000000020864
```

[1] Parallel Hybrid Algorithm of Bisection and Newton-Raphson Methods to Find Non-Linear Equations Roots – [Khalid Ali Hussein1 ,Abed Ali H. Altaee2 ,Haider K. Hoomod3]

Source : https://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.1043.5769&rep=rep1&type=pdf

3. Advanced : Hybrid Algorithm

Implementation

```
elif args.methmod == "hybrid" :
    bracket = []
    diff = newton.differential(eq)
    for i in range(-500000,500000) :
        lower = i*0.001
        upper = i*0.001 + 0.001
        bracket.append((lower, upper))
    processCount = 1
    if args.parallel == "Y" :
        processCount = multiprocessing.cpu_count()
    pool = multiprocessing.Pool(processes= processCount)
    func = partial(bisection.setAndCalculate, threshold = 40, equation= eq)
    t = time.time()
    resultmtmp = pool.map(func, bracket)
    first = time.time()-t
    pool.close()
    pool.join()
    tmpseed = []
    seed = []
    for elem in resultmtmp :
        if len(elem) != 0 :
            for root in elem :
                seed.append(root)
    for idx, element in enumerate(tmpseed) :
        if element != False:
            seed.append(element)
    t = time.time()
    for elem in seed :
        resultm.append(newton.setAndCalculate(elem, thm, eq, diff))
    second = time.time()-t
    print(f'Using time : {first + second}')
```

4. Conclusion

초기값을 가지고 탐색할 경우 Newton-Rapson method가 Bisection method보다 좋은 성능을 지닌다.

탐색 범위가 넓을 때, Bisection 병렬 수행은 효과적이다.

[1]에서 제안된 Hybrid 방식을 토대로 근사 계산 iteration을 줄일 수 있다.
Hybrid 방식을 통해, 초기값에 대한 설정 없이 적은 실행시간안에 문제를 해결할 수 있다.