Hanbat National University

Course: Design of Coastal Engineering

Report I

November 29, 2022

Author	*Student ID*
Jonghyeok Kim	20201967

Report I

Jonghyeok Kim

November 29, 2022

Contents

1 Introduction

기본적으로 아래의 분산관계식(Dispersion Relationship)을 이용하여 본 보고서에 수록된 문제들을 해결할 수 있다.

$$\sigma^2 = gktanh(kh)$$

만약 kh의 값이 조건으로 제시가 되었다면, 위의 분산 관계식을 이용하여 σ^2으로부터 σ 값을 산정하고, 아래의 각진동수 σ에 대한 관계식으로부터,

$$\sigma = \frac{2\pi}{T}$$

주기 T를 산정해낼 수 있다.

반면에, kh가 제시되어 있지 않거나, 파수 k의 값을 모르는 경우에는 위의 분산관계식에 대해 양해법(Explicit Method)과 음해법(Implicit Method)을 이용하여 k값을 산정할 수 있다.

본 보고서에서 필자는 음해법으로서 이분법(Bisection Method)를 이용하였고, Boundary Points를 결정함에 있어서 양해법 중 하나인 아래의 Eckart 식을 이용하였다.

$$\sigma^2 = gk\sqrt{tanh(\frac{\sigma^2}{g}h)}$$

반드시는 아니나, 분산관계식에서 파수 k를 구함에 있어 음해법(Implicit Method)을 사용하는 경우에는 양해법(Explicit Method)을 이용하여 구한 k값을 초깃값으로 하는 경우가 일반적이다.

뒤에서도 살펴보겠지만, $\nu \leq \pi$의 수심범위에서 Hunt의 해가 가장 정확하다는 연구 결과로부터, 초깃값 산정의 방법에 있어서 아래의 Hunt의 식 사용을 고려해보는 것도 음해법을 이용한 해의 수렴속도를 증가시키는 한 가지 요인이 될 수 있을 것이다.

$$(kh)^2 = y^2 + \frac{y}{1 + \sum_{n=1}^{6} d_n y^n}$$

where,

$y = \frac{\sigma^2 h}{g}$

$d_1 = 0.6666666666$, $d_2 = 0.3555555555$, $d_3 = 0.1608465608$, $d_4 = 0.0632098765$, $d_5 = 0.0217540484$, $d_6 =$

0.0076507983

물론 양해법에는 Eckart의 방법과 Hunt의 방법 외에도 여러 개의 다른 양해법이 존재한다.

2 Questions

수심 1.0m인 수조에서 다음의 문제를 해결하시오. 파 속도는 음해법(Implicit Method)을 이용하여 소수점 둘째자리가 변하지 않을 때까지 구하시오.

2.1 Question #1

$kh = \pi/10$ 및 π를 만족하는 주기 T_1과 T_2를 구하시오.

2.2 Answer #1

먼저 고정값인 수심과 중력 가속도를 아래와 같이 정의한다.

```
h = 1.0
g = 9.81
```

2.2.1 $kh = \pi/10$

다음으로, 첫 번째 조건인 $kh = \pi/10$인 경우에 주기 T_1을 산정하기 위해 kh 값을 아래와 같이 정의한다.

```
kh = math.pi/10
```

마지막으로, 위에서 정의한 고정값과 조건식으로부터 아래와 같이 연산을 수행한다.

```
k = kh / h

sqr_sigma = g * k * np.tanh([kh])

sigma = math.sqrt(sqr_sigma)

T1 = 2 * math.pi / sigma
```

k는 주어진 $kh = \pi/10$이라는 식에서 kh를 h로 나눔으로써 산정할 수 있다.

즉, 아래의 코드가 이를 수행한다.

```
k = kh / h
```

또한, 다음의 분산관계식으로부터,

$$\sigma^2 = gktanh(kh)$$

σ^2을 산정할 수 있으며, 이를 코드로 보이면 다음과 같다.

```
sqr_sigma = g * k * np.tanh([kh])
```

σ^2의 제곱근인 σ를 아래의 코드로 산정할 수 있으며,

```
sigma = math.sqrt(sqr_sigma)
```

아래의 관계식으로부터,

$$\sigma = \frac{2\pi}{T}$$

주기 T_1의 값을 아래의 코드로부터 산정할 수 있다.

```
T1 = 2 * math.pi / sigma
```

위에 수록된 코드로부터 도출된 T1의 값을 아래의 Python 구문을 이용해 확인해보면,

```
print("When kh is pi/10: ", T1)
```

그 값은 아래와 같이 출력되는 것을 확인할 수 있다.

```
When kh is pi/10:  6.489022272321439
```

2.2.2 $kh = \pi$

마찬가지 방법으로, $kh = \pi$인 경우에 대해서 주기 T_2의 값을 산정해보자.

우선 kh를 아래와 같이 π로 정의한다.

```
kh = math.pi
```

위에서와 동일한 방식으로 T_2를 아래의 코드로부터 도출할 수 있다.

```
k = kh / h

sqr_sigma = g * k * np.tanh([kh])

sigma = math.sqrt(sqr_sigma)

T2 = 2 * math.pi / sigma
```

도출된 T_2를 아래의 Python 출력구문으로 확인해보면,

```
print("When kh is pi: ", T2)
```

그 결과는 다음과 같다.

```
When kh is pi:  1.1339174776189893
```

정리해보면, 조건으로 정의된 kh 값과 고정값인 수심 h로부터 파수 k를 산정한 다음, 아래의 분산관계식(Dispersion Relationship)으로부터,

$$\sigma^2 = gktanh(kh)$$

σ^2 값을 산정한 다음, 제곱근(Square Root)을 취해 σ 값을 산정한다.

이로부터 주기 T_1은 아래의 관계식을 이용하여,

$$T = \frac{2\pi}{\sigma}$$

산정할 수 있다.

2.3 Question #2

T_1보다 긴 주기를 갖는 파 5개를 임의로 선택하여 파 속도를 구하시오.

2.4 Answer #2

Question 1에서 산정한 T_1 값을 바탕으로, 아래와 같이 0.001에서 100까지 0.001 간격의 List를 생성한 다음,

```
periods = np.arange(0.001, 100, 0.001)
```

생성한 List에 반복문과 조건문을 조합하여 T_1의 값보다 큰 데이터 5개만을 별도의 리스트에 아래와 같이 저장한다.

```
long_T1 = [float(format(item, '.3f')) for i, item in enumerate(periods) if item > T1][:5]
```

위의 별도의 리스트의 결과값을 출력해보면 그 결과는 다음과 같다.

```
[6.49, 6.491, 6.492, 6.493, 6.494]
```

위에 출력된 데이터들은 주기 $T_1 = 6.489022272321439$보다 크므로 O.K이다.

생성한 5개의 주기 데이터들을 아래의 관계식으로부터,

$$\sigma = \frac{2\pi}{T}$$

다음의 코드로 σ 값을 산정한다.

```
long_sigma = [2 * math.pi / i for i in long_T1]
```

위에서 구한 σ 값으로부터 이를 두 번 곱하여 σ^2의 값을 아래의 코드로 산정한다.

```
long_sqr_sigma = [i * i for i in long_sigma]
```

2.4.1 Using the Eckart Method of Explicit Methods.

Implicit Method 즉 음해법을 이용하여 k값을 산정하기 위해서는 필수적인 것은 아니나, 먼저 Explicit Method 즉 양해법인 Eckart 식을 이용하여 초깃값을 산정해야 한다.

이를 위해 먼저 k값을 담을 List를 아래의 코드로 생성한다.

```
k_init = []
```

그 다음, 위에서 도출된 σ^2들이 5개 담긴 리스트로부터 반복문을 이용하여 한 개씩 값을 추출하여 아래의 Eckart 식을,

$$\sigma^2 = gk\sqrt{tanh(\frac{\sigma^2}{g}h)}$$

아래와 같이 k에 대한 식으로 정리한 다음,

$$k = \frac{\sigma^2}{g\sqrt{tanh(\frac{\sigma^2}{g}h)}}$$

위의 k에 대한 식을 아래와 같이 위에서 산정한 σ^2들에 각각 적용하면서, 초기 k값을 산정하게끔 한다.

```
for i in long_sqr_sigma:
    k_init.append(i / (g * math.sqrt(np.tanh([i / g * h]))))
```

이때의 양해법인 Eckart 식을 이용하여 산정한 초기 k값 5개를 출력해보면 다음과 같다.

```
[0.3095708159665305,
 0.30952283475267234,
 0.30947486854239015,
```

```
0.3094269173285474,
0.3093789811040123]
```

2.4.2 Using Bisection Method.

위에서 Eckart 식으로부터 산정한 초기 k값 5개를 바탕으로 수치해석기법 중 하나인 Bisection Method를 이용하여 k값을 산정한다.

이를 위해 먼저 아래와 같이 정의되는 분산관계식(Dispersion Relationship)에서,

$$\sigma^2 = gktanh(kh)$$

좌변이 0이 되도록 모든 항을 이항하여 정리하면 아래와 같다.

$$0 = \frac{gktanh(kh)}{\sigma^2} - 1$$

이를 Python의 함수(Function)를 이용하여 정의하면 다음과 같다.

```
def f(x, sqr_sigma):
    return g * (x / sqr_sigma) * np.tanh([x*h]) - 1
```

함수 f로 들어오는 인자 x 즉 k값과 sqr_sigma 즉 σ^2의 값을 받아 우변의 항을 계산하여 이를 반환한다.

오차(Error)를 0.5×10^{-6}으로 아래와 같이 정의한다.

```
error = 0.5 * 10**(-6)
```

그 다음, 위에서 T_1의 주기보다 긴 주기 T 5개에 대해 반복문을 통해 반복하면서, 각 주기에 대해 Eckart식에 의해 도출된 초기 k값을 바탕으로, $-0.1+k$에서 $+0.1+k$까지의 범위에 대해 Bisection Method를 아래의 코드로 수행한다.

```
for i, sqr_sigma in enumerate(long_sqr_sigma):
    a = -0.1 + k_init[i]
    b = 0.1 + k_init[i]

    while (b - a) / 2 > error:
        c = (b + a) / 2
        if f(c, sqr_sigma) == 0:
            break
        elif f(a, sqr_sigma)*f(c, sqr_sigma) > 0:
```

```
            a = c
        else:
            b = c
    c = (b + a) / 2
```

내부의 while 반복문의 조건인 $\frac{(b-a)}{2} > \epsilon$을 만족하지 않아 탈출하여 최종적으로 계산하게 되는 $(b+a)/2$의 값이 Bisection Method로 산정하게 되는 최종적인 k 값이 된다.

파속 C를 산정하기 전에, Bisection Method로부터 산정한 k 값을 담을 List를 아래와 같이 구성해준 다음,

```
k = []
```

Bisection Method로 산정한 k값을 저장하는 코드를 위의 Bisection Method의 구현부에 아래와 같이 정의한다.

```
for i, sqr_sigma in enumerate(short_sqr_sigma):
    a = -0.1 + k_init[i]
    b = 0.1 + k_init[i]

    while (b - a) / 2 > error:
        c = (b + a) / 2
        if f(c, sqr_sigma) == 0:
            break
        elif f(a, sqr_sigma)*f(c, sqr_sigma) > 0:
            a = c
        else:
            b = c
    c = (b + a) / 2

    k.append(c)
```

초기 k값과 Bisection Method로 구한 k값을 위의 코드에 출력문을 아래와 같이 추가구성하여,

```
error = 0.5 * 10**(-6)

for i, sqr_sigma in enumerate(long_sqr_sigma):
    a = -0.1 + k_init[i]
    b = 0.1 + k_init[i]

    while (b - a) / 2 > error:
        c = (b + a) / 2
        if f(c, sqr_sigma) == 0:
            break
        elif f(a, sqr_sigma)*f(c, sqr_sigma) > 0:
```

```
            a = c
        else:
            b = c
    c = (b + a) / 2

    k.append(c)

    # solution.append(f(c, sqr_sigma))
    print("k initial: ", k_init[i])
    print("k:", c)
    print("")
```

출력해보면 다음과 같다.

```
k initial:  0.3095708159665305
k: 0.3141106871823509

k initial:  0.30952283475267234
k: 0.31406041715013333

k initial:  0.30947486854239015
k: 0.3140101621214917

k initial:  0.3094269173285474
k: 0.31396068502874275

k initial:  0.3093789811040123
k: 0.3139104599858482
```

위의 출력문에서 k initial은 Eckart 식에 의해 도출된 초기 k 값이고, k는 Bisection Method에 의해 도출된 k 값이다.

5개의 경우에 대하여 두 k값의 차이에 대한 절댓값의 평균에 대한 백분율은 1.999%로 도출되었다.

Bisection Method로 도출해낸 5개의 k값을 바탕으로, 아래의 관계식을 이용하여,

$$L = \frac{2\pi}{k}$$

5개의 파장 L을 저장할 List를 아래와 같이 구성해준 다음,

```
L = []
```

파장 L을 아래와 같이 산정하여 이를 위에서 구성한 List L에 저장한다.

```
for i in range(0, len(k)):
    L.append(2 * math.pi / k[i])
```

5개의 L에 대해서 아래와 같이 정의되는 파속 C로부터,

$$C = \frac{L}{T}$$

위의 파장 L과 마찬가지로 5개의 파속 C를 저장할 List를 아래와 같이 구성해준 다음,

```
C = []
```

다음과 같이 코드를 작성하여 파속 C를 산정해낸다.

```
for i in range(0, len(k)):
    C.append(L[i] / long_T1[i])
```

산정해낸 5개의 파속 C 값을 출력해보면 아래와 같다.

```
[0.2991879757822466,
 0.3247919065561532,
 0.3511051069715095,
 0.3780254835712734,
 0.40544344210830563]
```

2.5 Question #3

T_2보다 짧은 주기를 갖는 파 5개를 임의로 선택하여 파 속도를 구하시오.

2.6 Answer #3

Question 2에서와 동일한 방식으로 T_2보다 짧은 주기를 갖는 파 5개를 임의로 선택하여 파 속도를 산정한다.

Question 1에서 산정한 T_2 값을 바탕으로, Question 2에서와 달리 T_2보다 0.1만큼 작은 값에서 T_2값까지 0.001 간격의 List를 생성한 다음,

```
periods = np.arange(T2-0.1, T2, 0.001)
```

위에서 생성한 periods List에 반복문과 조건문을 조합하여 T_2의 값보다 작은 데이터 5개만을 별도의 리스트에 아래와 같이 저장한다.

```
short_T2 = [float(format(item, '.3f')) for i, item in enumerate(periods) if item < T2][:5]
```

위의 별도의 리스트의 결과값을 출력해보면 그 결과는 다음과 같다.

```
[1.034, 1.035, 1.036, 1.037, 1.038]
```

위에 출력된 데이터들은 주기 $T_2 = 1.1339174776189893$보다 작으므로 O.K이다.

생성한 5개의 주기 데이터들을 아래의 관계식을 이용하여,

$$\sigma = \frac{2\pi}{T}$$

다음의 코드로 σ 값을 산정한다.

```
short_sigma = [2 * math.pi / i for i in short_T2]
```

위에서 구한 σ 값으로부터 이를 두 번 곱하여 σ^2의 값을 아래의 코드로 산정한다.

```
short_sqr_sigma = [i * i for i in short_sigma]
```

2.6.1 Using the Eckart Method of Explicit Methods.

Implicit Method 즉 음해법을 이용하여 k값을 산정하기 위해서 필수적인 것은 아니나, 먼저 Explicit Method 즉 양해법인 Eckart 식을 이용하여 k에 대한 초깃값을 산정해야 한다.

이를 위해 먼저 k값을 담을 List를 아래의 코드로 생성한다.

```
k_init = []
```

그 다음, 위에서 도출된 σ^2들이 5개 담긴 리스트로부터 반복문을 이용하여 한 개씩 값을 추출하여 아래의 Eckart 식을,

$$\sigma^2 = gk\sqrt{tanh(\frac{\sigma^2}{g}h)}$$

아래와 같이 k에 대한 식으로 정리한 다음,

$$k = \frac{\sigma^2}{g\sqrt{tanh(\frac{\sigma^2}{g}h)}}$$

위의 k에 대한 식을 아래와 같이 적용하면서, 초기 k값을 산정하게끔 한다.

```
for i in short_sqr_sigma:
    k_init.append(i / (g * math.sqrt(np.tanh([(i / g) * h]))))
```

이때의 양해법인 Eckart 식을 이용하여 산정한 초기 k값 5개를 출력해보면 다음과 같다.

```
[3.7660251921676675,
 3.7587809715925244,
 3.751558035416577,
 3.7443563038222476,
 3.7371756973804486]
```

2.6.2 Using Bisection Method.

위에서 Eckart 식으로부터 산정한 초기 k값 5개를 바탕으로 수치해석기법 중 하나인 Bisection Method를 이용하여 k값을 산정한다.

이를 위해 먼저 아래와 같이 정의되는 분산관계식(Dispersion Relationship)에서,

$$\sigma^2 = gktanh(kh)$$

좌변이 0이 되도록 모든 항을 이항하여 정리하면 아래와 같다.

$$0 = \frac{gktanh(kh)}{\sigma^2} - 1$$

이를 Python의 함수(Function)를 이용하여 정의하면 다음과 같다.

```
def f(x, sqr_sigma):
    return g * (x / sqr_sigma) * np.tanh([x*h]) - 1
```

함수 f로 들어오는 인자 x 즉 k값과 sqr_sigma 즉 σ^2의 값을 받아 우변의 항을 계산하여 이를 반환한다.

오차(Error)를 0.5×10^{-6}으로 아래와 같이 정의한다.

```
error = 0.5 * 10**(-6)
```

그 다음, 위에서 T_2의 주기보다 짧은 주기 T 5개에 대해 반복문을 통해 반복하면서, 각 주기에 대해 Eckart식에 의해 도출된 초기 k값을 바탕으로, $-0.1+k$에서 $+0.1+k$까지의 범위에 대해 Bisection Method를 아래의 코드로 수행한다.

```
for i, sqr_sigma in enumerate(short_sqr_sigma):
    a = -0.1 + k_init[i]
    b = 0.1 + k_init[i]

    while (b - a) / 2 > error:
```

```
        c = (b + a) / 2
        if f(c, sqr_sigma) == 0:
            break
        elif f(a, sqr_sigma)*f(c, sqr_sigma) > 0:
            a = c
        else:
            b = c
    c = (b + a) / 2
```

내부의 while 반복문의 조건인 $\frac{(b-a)}{2} > \epsilon$을 만족하지 않아 탈출하여 최종적으로 계산하게 되는 $(b+a)/2$의 값이 Bisection Method로 산정하게 되는 최종적인 k 값이 된다.

파속 C를 산정하기 위해서 Bisection Method로부터 산정한 k 값을 담을 List를 아래와 같이 구성해준 다음,

```
k = []
```

Bisection Method로 산정한 k값을 저장하는 코드를 위의 Bisection Method의 구현부에 아래와 같이 정의한다.

```
for i, sqr_sigma in enumerate(short_sqr_sigma):
    a = -0.1 + k_init[i]
    b = 0.1 + k_init[i]

    while (b - a) / 2 > error:
        c = (b + a) / 2
        if f(c, sqr_sigma) == 0:
            break
        elif f(a, sqr_sigma)*f(c, sqr_sigma) > 0:
            a = c
        else:
            b = c
    c = (b + a) / 2

    k.append(c)
```

초기 k값과 Bisection Method로 구한 k값을 위의 코드에 출력문을 아래와 같이 추가구성하여,

```
error = 0.5 * 10**(-6)

for i, sqr_sigma in enumerate(long_sqr_sigma):
    a = -0.1 + k_init[i]
    b = 0.1 + k_init[i]

    while (b - a) / 2 > error:
```

```
        c = (b + a) / 2
        if f(c, sqr_sigma) == 0:
            break
        elif f(a, sqr_sigma)*f(c, sqr_sigma) > 0:
            a = c
        else:
            b = c
    c = (b + a) / 2

    k.append(c)

    # solution.append(f(c, sqr_sigma))
    print("k initial: ", k_init[i])
    print("k:", c)
    print("")
```

출력해보면 다음과 같다.

```
k initial:  3.7660251921676675
k: 3.768018371488957

k initial:  3.7587809715925244
k: 3.7607993279157665

k initial:  3.751558035416577
k: 3.753601568741772

k initial:  3.7443563038222476
k: 3.746425014149396

k initial:  3.7371756973804486
k: 3.7392703476490037
```

위의 출력문에서 k initial은 Eckart 식에 의해 도출된 초기 k 값이고, k는 Bisection Method에 의해 도출된 k 값이다.

5개의 경우에 대하여 두 k값의 차이에 대한 절댓값의 평균에 대한 백분율은 0.04189%로 도출되었다.

Bisection Method로 도출해낸 5개의 k값을 바탕으로, 아래의 관계식을 이용하여,

$$L = \frac{2\pi}{k}$$

5개의 파장 L을 저장할 List를 아래와 같이 구성해준 다음,

```
L = []
```

파장 L을 아래와 같이 산정하여 이를 위에서 구성한 List L에 저장한다.

```
for i in range(0, len(k)):
    L.append(2 * math.pi / k[i])
```

5개의 L에 대해서 아래와 같이 정의되는 파속 C로부터,

$$C = \frac{L}{T}$$

위의 파장 L과 마찬가지로 5개의 파속 C를 저장할 List를 아래와 같이 구성해준 다음,

```
C = []
```

다음과 같이 코드를 작성하여 파속 C를 산정해낸다.

```
for i in range(0, len(k)):
    C.append(L[i] / long_T1[i])
```

산정해낸 5개의 파속 C 값을 출력해보면 아래와 같다.

```
[0.25693434422426026,
 0.2573878834782933,
 0.2578417174328627,
 0.2582958455488199,
 0.25875021449081254]
```

2.7 Question #4

$T_2 \leq T \leq T_1$ 구간에서 파 속도를 구하고 Eckart 및 Hunt의 식과 비교하시오.

2.8 Answer #4

먼저 아래의 코드로부터 $T_2 \leq T \leq T_1$ 구간에 있는 T값을 생성한다.

이때, T_2와 T_1 사이의 데이터들의 간격은 0.05로 하였으며, 이 경우 총 데이터의 갯수는 108개로 도출되었다.

```
T = np.arange(T2, T1, 0.05)
```

위의 주기 T값을 출력해보면 아래와 같다.

```
array([1.13391748, 1.18391748, 1.23391748, 1.28391748, 1.33391748,
       1.38391748, 1.43391748, 1.48391748, 1.53391748, 1.58391748,
       1.63391748, 1.68391748, 1.73391748, 1.78391748, 1.83391748,
       1.88391748, 1.93391748, 1.98391748, 2.03391748, 2.08391748,
       2.13391748, 2.18391748, 2.23391748, 2.28391748, 2.33391748,
       2.38391748, 2.43391748, 2.48391748, 2.53391748, 2.58391748,
       2.63391748, 2.68391748, 2.73391748, 2.78391748, 2.83391748,
       2.88391748, 2.93391748, 2.98391748, 3.03391748, 3.08391748,
       3.13391748, 3.18391748, 3.23391748, 3.28391748, 3.33391748,
       3.38391748, 3.43391748, 3.48391748, 3.53391748, 3.58391748,
       3.63391748, 3.68391748, 3.73391748, 3.78391748, 3.83391748,
       3.88391748, 3.93391748, 3.98391748, 4.03391748, 4.08391748,
       4.13391748, 4.18391748, 4.23391748, 4.28391748, 4.33391748,
       4.38391748, 4.43391748, 4.48391748, 4.53391748, 4.58391748,
       4.63391748, 4.68391748, 4.73391748, 4.78391748, 4.83391748,
       4.88391748, 4.93391748, 4.98391748, 5.03391748, 5.08391748,
       5.13391748, 5.18391748, 5.23391748, 5.28391748, 5.33391748,
       5.38391748, 5.43391748, 5.48391748, 5.53391748, 5.58391748,
       5.63391748, 5.68391748, 5.73391748, 5.78391748, 5.83391748,
       5.88391748, 5.93391748, 5.98391748, 6.03391748, 6.08391748,
       6.13391748, 6.18391748, 6.23391748, 6.28391748, 6.33391748,
       6.38391748, 6.43391748, 6.48391748])
```

위에서 산정한 주기 T를 바탕으로, 아래의 σ에 대한 관계식을 이용하여,

$$\sigma = \frac{2\pi}{T}$$

이를 제곱한 σ^2 값을 산정해낸다.

이때, 108개의 T로부터 산정하게 될 σ^2의 값을 저장할 리스트를 아래와 같이 정의한 다음,

```
added_sqr_sigma = []
```

아래의 코드로부터 총 108개 각각의 T에 대해서 σ^2의 값을 산정해낸다.

```
for i in T:
    added_sqr_sigma.append((2*math.pi/i) ** 2)
```

2.8.1 Using the Eckart Method of Explicit Methods.

Implicit Method 즉 음해법을 이용하여 k값을 산정하기 위해서 필수적인 것은 아니나, 먼저 Explicit Method 즉 양해법인 Eckart 식을 이용하여 k에 대한 초깃값을 산정해야 한다.

이를 위해 먼저 k값을 담을 List를 아래의 코드로 생성한다.

```
k_init = []
```

그 다음, 위에서 도출된 σ^2들이 108개 담긴 리스트로부터 반복문을 이용하여 한 개씩 값을 추출하여 아래의 Eckart 식을,

$$\sigma^2 = gk\sqrt{tanh(\frac{\sigma^2}{g}h)}$$

아래와 같이 k에 대한 식으로 정리한 다음,

$$k = \frac{\sigma^2}{g\sqrt{tanh(\frac{\sigma^2}{g}h)}}$$

위의 k에 대한 식을 아래와 같이 위에서 산정한 σ^2들에 각각 적용하면서, 초기 k값을 산정하게끔 한다.

```
for i in added_sqr_sigma:
    k_init.append(i / (g * math.sqrt(np.tanh([(i / g) * h]))))
```

이때의 양해법인 Eckart 식을 이용하여 산정한 초기 k값 108개를 출력해보면 다음과 같다.

```
[3.135870161236885,
 2.880321502961319,
 2.6565398217280007,
 2.459843722381739,
 2.286367761003905,
 2.1328854196396634,
 1.996679080573182,
 1.8754432647772303,
 1.7672113456359493,
 1.670298876681068,
 1.5832587953844968,
 1.5048452653040891,
 1.4339839524861973,
 1.3697472286504564,
 1.3113332551910493,
```

```
1.2580482044709902,
1.209291072547099,
1.1645406676354029,
1.1233444460913928,
1.0853089284301563,
1.050091472001765,
1.0173932105390635,
0.986952997617885,
0.958542213306537,
0.9319603122171266,
0.9070310075635305,
0.8835990001407303,
0.8615271736676672,
0.8406941888959165,
0.8209924184506829,
0.802326172687414,
0.7846101740517668,
0.7677682436480623,
0.7517321690682655,
0.7364407271175379,
0.7218388389918436,
0.707876838806769,
0.6945098392240203,
0.6816971803431773,
0.6694019500828079,
0.6575905660206002,
0.6462324101430793,
0.6352995092116726,
0.6247662545175011,
0.6146091557014666,
0.6048066240837825,
0.595338781599068,
0.5861872919873506,
0.5773352113628502,
0.5687668556839839,
0.5604676829903933,
0.5524241885650082,
0.5446238114288845,
0.5370548507902456,
0.5297063912522475,
0.5225682357410866,
```

```
0.5156308452510578,
0.5088852846193221,
0.5023231736432393,
0.4959366429395075,
0.4897182940190196,
0.48366116311598323,
0.47775868836588586,
0.4720046799755534,
0.46639329307085203,
0.46091900294445354,
0.4555765824582212,
0.4503610813828668,
0.4452678074820973,
0.440292309170013,
0.4354303595894052,
0.4306779419752245,
0.42603123618209826,
0.4214866062676737,
0.4170405890349343,
0.4126898834466975,
0.4084313408344043,
0.4042619558312086,
0.40017885796638075,
0.3961793038642717,
0.39226066999663034,
0.3884204459420129,
0.3846562281104369,
0.3809657138953759,
0.37734669621872613,
0.3737970584375337,
0.37031476958411835,
0.3668978799137715,
0.36354451673650723,
0.3602528805114065,
0.35702124118395967,
0.3538479347484916,
0.35073136001927646,
0.34766997559531965,
0.3446622970050335,
0.3417068940181608,
0.3388023881133244,
```

```
0.33594745009051347,
0.3331407978186657,
0.33038119410927225,
0.3276674447076417,
0.32499839639409356,
0.32237293518794474,
0.31978998464768676,
0.31724850426123585,
0.3147474879205983,
0.31228596247569634,
0.3098629863624799]
```

위의 Eckart로부터 구한 초기 k값을 담은 108개의 List의 이름을 아래의 코드를 통해 새로 명명한 다음,

```
eckart_k = k_init
```

Eckart에 의해 구한 k 값으로부터 도출하게될 파속 C를 저장할 List를 아래와 같이 정의한다.

```
Eckart_C = []
```

다음으로, 아래의 파속 C의 정의와,

$$C = \frac{L}{T}$$

파장 L의 관계식으로부터

$$L = \frac{2\pi}{k}$$

파속 C를 다음과 같이 산정할 수 있으므로,

$$C = \frac{L}{T} = \frac{\frac{2\pi}{k}}{T}$$

다음의 코드로 파속 C를 산정하여 이를 Eckart_C List에 저장한다.

```
for i in range(0, len(eckart_k)):
    Eckart_C.append((2 * math.pi / eckart_k[i])/T[i])
```

2.8.2 Using the Hunt Method of Explicit Methods.

다음으로, Explicit Method 즉 양해법 중 하나인 Hunt 식을 이용하여 k값을 산정해보자.

먼저 Hunt의 식은 아래와 같다.

$$(kh)^2 = y^2 + \frac{y}{1 + \sum_{n=1}^{6} d_n y^n}$$

where,

$y = \frac{\sigma^2 h}{g}$

$d_1 = 0.6666666666$, $d_2 = 0.3555555555$, $d_3 = 0.1608465608$, $d_4 = 0.0632098765$, $d_5 = 0.0217540484$, $d_6 = 0.0076507983$

위의 Hunt의 식의 좌변은 $(kh)^2$이므로, 아래와 같이 Python에서 108개의 $(kh)^2$을 담을 List인 sqr_kh를 정의한다.

```
sqr_kh = []
```

그 다음, 반복문을 통해 Summation할 수 있도록 d라는 List내에 순차적으로 d_1부터 d_6까지의 값을 아래와 같이 저장해준다.

```
d = [0.6666666666, 0.3555555555, 0.1608465608, 0.0632098765, 0.0217540484, 0.0076507983]
```

위에서 도출해낸 108개의 σ^2의 값들에 대해 반복문을 순회하면서,

아래와 같이 정의되는 y를,

$$y = \frac{\sigma^2 h}{g}$$

아래의 코드로 산정하고,

```
y = i*h/g
```

아래의 식을 계산하기 위해,

$$\sum_{n=1}^{6} d_n y^n$$

먼저 dny라는 변수를 다음과 같이 정의하고, 그 값을 0으로 초기화한 다음,

```
dny = 0
```

이전에 정의한 List d의 크기만큼 반복하면서 즉 6번 순회하며 d[0] d̃[5]의 값을 참조하고, y의 n 제곱승 역시 반복문 내에서 j의 인덱스에 1만큼 더하여 즉 0+1=1 5̃+1=6 제곱승을 하여 두 값을 서로 곱한 다음, 이를 dny에 더해줌으로써, 위의 Summation 식을 달성한다.

```
for j in range(0, len(d)):
        dny += d[j] * (y ** (j+1))
```

그리고 이를 위에서 정의한 List sqr_kh에 나머지 Hunt 식을 계산한 최종결과인 108개의 $(kh)^2$의 값을 아래와 같이 저장한다.

```
sqr_kh.append((y) ** 2 + (y) / (1 + dny))
```

위에서 설명한 코드들은 다음과 같이 하나의 반복문 내에서 구성되어야 한다.

```
for i in added_sqr_sigma:
    y = i*h/g
    dny = 0

    for j in range(0, len(d)):
        dny += d[j] * (y ** (j+1))

    sqr_kh.append((y) ** 2 + (y) / (1 + dny))
```

또한, Hunt 식에 의해 도출된 k값을 저장하기 위한 별도의 List를 아래와 같이 구성해준 다음,

```
hunt_k = []
```

다음 과정으로, 위에서 산정한 $(kh)^2$을 다시 반복문을 구성하여 이를 통해 순회시키며, 제곱근을 취해 kh의 값을 산정한 다음, 이를 h로 나누어 Hunt에 의한 파수 k값을 위에서 정의한 List hunt_k에 저장한다.

```
for i in sqr_kh:
    kh = math.sqrt(i)
    hunt_k.append(kh / h)
```

마찬가지로, Hunt에 의한 파속 C를 저장할 List를 아래와 같이 정의한 후,

```
Hunt_C = []
```

위에서와 마찬가지로 파속 C의 정의와 파장 L의 관계식으로부터, 아래와 같이 파속 C를 산정하고,

$$C = \frac{L}{T} = \frac{\frac{2\pi}{k}}{T}$$

이를 코드로 아래와 같이 구현해준다.

```
for i in range(0, len(hunt_k)):
    Hunt_C.append((2 * math.pi / hunt_k[i])/T[i])
```

2.8.3 Using Bisection Method.

이번에는 위에서 Eckart 식으로부터 산정한 초기 k값 108개를 바탕으로 수치해석기법 중 하나인 Bisection Method를 이용하여 k값을 산정한다.

이를 위해 먼저 아래와 같이 정의되는 분산관계식(Dispersion Relationship)에서,

$$\sigma^2 = gktanh(kh)$$

좌변이 0이 되도록 모든 항을 이항하여 정리하면 아래와 같다.

$$0 = \frac{gktanh(kh)}{\sigma^2} - 1$$

이를 Python의 함수(Function)를 이용하여 정의하면 다음과 같다.

```
def f(x, sqr_sigma):
    return g * (x / sqr_sigma) * np.tanh([x*h]) - 1
```

함수 f로 들어오는 인자 x 즉 k값과 sqr_sigma 즉 σ^2의 값을 받아 우변의 항을 계산하여 이를 반환한다.

오차(Error)를 0.5×10^{-6}으로 아래와 같이 정의한다.

```
error = 0.5 * 10**(-6)
```

그 다음, 위에서 T_2의 주기보다 짧은 주기 T 108개에 대해 반복문을 통해 반복하면서, 각 주기에 대해 Eckart 식에 의해 도출된 초기 k값을 바탕으로, $-0.1 + k$에서 $+0.1 + k$까지의 범위에 대해 Bisection Method를 아래의 코드로 수행한다.

```
for i, sqr_sigma in enumerate(short_sqr_sigma):
    a = -0.1 + k_init[i]
    b = 0.1 + k_init[i]

    while (b - a) / 2 > error:
        c = (b + a) / 2
        if f(c, sqr_sigma) == 0:
            break
        elif f(a, sqr_sigma)*f(c, sqr_sigma) > 0:
```

```
            a = c
        else:
            b = c
    c = (b + a) / 2
```

내부의 while 반복문의 조건인 $\frac{(b-a)}{2} > \epsilon$을 만족하지 않아 탈출하여 최종적으로 계산하게 되는 $(b+a)/2$의 값이 Bisection Method로 산정하게 되는 최종적인 k 값이 된다.

파속 C를 산정하기 전에, Bisection Method로부터 산정한 k 값을 담을 List를 아래와 같이 구성해준 다음,

```
k = []
```

Bisection Method로 산정한 k값을 저장하는 코드를 위의 Bisection Method의 구현부에 아래와 같이 정의한다.

```
for i, sqr_sigma in enumerate(short_sqr_sigma):
    a = -0.1 + k_init[i]
    b = 0.1 + k_init[i]

    while (b - a) / 2 > error:
        c = (b + a) / 2
        if f(c, sqr_sigma) == 0:
            break
        elif f(a, sqr_sigma)*f(c, sqr_sigma) > 0:
            a = c
        else:
            b = c
    c = (b + a) / 2

    k.append(c)
```

초기 k값과 Bisection Method로 구한 k값을 위의 코드에 출력문을 아래와 같이 추가구성하여,

```
error = 0.5 * 10**(-6)

for i, sqr_sigma in enumerate(long_sqr_sigma):
    a = -0.1 + k_init[i]
    b = 0.1 + k_init[i]

    while (b - a) / 2 > error:
        c = (b + a) / 2
        if f(c, sqr_sigma) == 0:
            break
        elif f(a, sqr_sigma)*f(c, sqr_sigma) > 0:
```

```
            a = c
        else:
            b = c
    c = (b + a) / 2

    k.append(c)

    # solution.append(f(c, sqr_sigma))
    print("k initial: ", k_init[i])
    print("k:", c)
    print("")
```

출력해보면 다음과 같다.

```
k initial:  3.135870161236885
k: 3.141592588605049

k initial:  2.880321502961319
k: 2.8889263155833893

k initial:  2.6565398217280007
k: 2.668672466381321

k initial:  2.459843722381739
k: 2.4760329161073247

k initial:  2.286367761003905
k: 2.306975900041991

k initial:  2.1328854196396634
k: 2.158080350669937

k initial:  1.996679080573182
k: 2.026437915412049

k initial:  1.8754432647772303
k: 1.9095752685614102

k initial:  1.7672113456359493
k: 1.8053899802795041

k initial:  1.670298876681068
```

```
k: 1.712103762545326

k initial:  1.5832587953844968
k: 1.628210043553442

k initial:  1.5048452653040891
k: 1.5524378098597533

k initial:  1.4339839524861973
k: 1.4837127275106115

k initial:  1.3697472286504564
k: 1.4211254787725265

k initial:  1.3113332551910493
k: 1.3639070314361668

k initial:  1.2580482044709902
k: 1.3114039936555604

k initial:  1.209291072547099
k: 1.2630588490363568

k initial:  1.1645406676354029
k: 1.218395419222317

k initial:  1.1233444460913928
k: 1.177005411057213

k initial:  1.0853089284301563
k: 1.1385373067260547

k initial:  1.050091472001765
k: 1.102687373491023

k initial:  1.0173932105390635
k: 1.0691910773603523

k initial:  0.986952997617885
k: 1.0378185524274555

```

```
k initial:  0.958542213306537
k: 1.0083686445809512

k initial:  0.9319603122171266
k: 0.9806652224954469

k initial:  0.9070310075635305
k: 0.9545525987500537

k initial:  0.8835990001407303
k: 0.9298922588077226

k initial:  0.8615271736676672
k: 0.9065638710553625

k initial:  0.8406941888959165
k: 0.8844575403363462

k initial:  0.8209924184506829
k: 0.863476320428222

k initial:  0.802326172687414
k: 0.843533676959875

k initial:  0.7846101740517668
k: 0.8245511988320404

k initial:  0.7677682436480623
k: 0.8064595736041169

k initial:  0.7517321690682655
k: 0.7891936406258828

k initial:  0.7364407271175379
k: 0.7726975172786708

k initial:  0.7218388389918436
k: 0.7569184135768044

k initial:  0.707876838806769
k: 0.7418081895147767
```

```
k initial:  0.6945098392240203
k: 0.7273242465726532

k initial:  0.6816971803431773
k: 0.7134266877894663

k initial:  0.6694019500828079
k: 0.7000801269626908

k initial:  0.6575905660206002
k: 0.687250218730561

k initial:  0.6462324101430793
k: 0.6749063450796028

k initial:  0.6352995092116726
k: 0.6630212957106961

k initial:  0.6247662545175011
k: 0.6515686989755088

k initial:  0.6146091557014666
k: 0.6405243015754901

k initial:  0.6048066240837825
k: 0.6298657518913997

k initial:  0.595338781599068
k: 0.6195731718578571

k initial:  0.5861872919873506
k: 0.6096274622754366

k initial:  0.5773352113628502
k: 0.6000093904399985

k initial:  0.5687668556839839
k: 0.5907040352494135

k initial:  0.5604676829903933
```

```
k: 0.5816953288644168

k initial:  0.5524241885650082
k: 0.5729682406890316

k initial:  0.5446238114288845
k: 0.5645102097443142

k initial:  0.5370548507902456
k: 0.5563080093595815

k initial:  0.5297063912522475
k: 0.5483507241379897

k initial:  0.5225682357410866
k: 0.5406266311268288

k initial:  0.5156308452510578
k: 0.5331246654414874

k initial:  0.5088852846193221
k: 0.5258358919191269

k initial:  0.5023231736432393
k: 0.5187511674176534

k initial:  0.4959366429395075
k: 0.5118618596143122

k initial:  0.4897182940190196
k: 0.50515904414109

k initial:  0.48366116311598323
k: 0.49863499429274105

k initial:  0.47775868836588586
k: 0.4922839111442062

k initial:  0.4720046799755534
k: 0.4860973160839518

```

```
k initial:  0.46639329307085203
k: 0.4800693642378443

k initial:  0.46091900294445354
k: 0.4741930050196489

k initial:  0.4555765824582212
k: 0.4684637742306822

k initial:  0.4503610813828668
k: 0.46287595870220266

k initial:  0.4452678074820973
k: 0.45742334031901144

k initial:  0.440292309170013
k: 0.4521014674952083

k initial:  0.4354303595894052
k: 0.4469053504341317

k initial:  0.4306779419752245
k: 0.4418309723707323

k initial:  0.42603123618209826
k: 0.4368729872807311

k initial:  0.4214866062676737
k: 0.43202852216122833

k initial:  0.4170405890349343
k: 0.4272933508757547

k initial:  0.4126898834466975
k: 0.4226634094476741

k initial:  0.4084313408344043
k: 0.4181347862689747

k initial:  0.4042619558312086
k: 0.4137052389122633
```

```
234
235 k initial:  0.40017885796638075
236 k: 0.4093703710279042
237
238 k initial:  0.3961793038642717
239 k: 0.4051274392402483
240
241 k initial:  0.39226066999663034
242 k: 0.4009738200210443
243
244 k initial:  0.3884204459420129
245 k: 0.39690700294884884
246
247 k initial:  0.3846562281104369
248 k: 0.39292305855477294
249
250 k initial:  0.3809657138953759
251 k: 0.3890204471717431
252
253 k initial:  0.37734669621872613
254 k: 0.3851961987822028
255
256 k initial:  0.3737970584375337
257 k: 0.38144819674319785
258
259 k initial:  0.37031476958411835
260 k: 0.377773647147595
261
262 k initial:  0.3668978799137715
263 k: 0.3741706002506856
264
265 k initial:  0.36354451673650723
266 k: 0.37063718336248375
267
268 k initial:  0.3602528805114065
269 k: 0.3671715969420707
270
271 k initial:  0.35702124118395967
272 k: 0.36377058505603
273
274 k initial:  0.3538479347484916
```

```
k: 0.3604340095775932

k initial:  0.35073136001927646
k: 0.3571595063815811

k initial:  0.34766997559531965
k: 0.353944008188093

k initial:  0.3446622970050335
k: 0.35078831934390065

k initial:  0.3417068940181608
k: 0.3476887208003874

k initial:  0.3388023881133244
k: 0.34464383403617604

k initial:  0.33594745009051347
k: 0.34165385573016194

k initial:  0.3331407978186657
k: 0.33871521493292345

k initial:  0.33038119410927225
k: 0.33582820033485816

k initial:  0.3276674447076417
k: 0.33299085474182144

k initial:  0.32499839639409356
k: 0.3302012619946795

k initial:  0.32237293518794474
k: 0.3274590710522025

k initial:  0.31978998464768676
k: 0.32476320547288207

k initial:  0.31724850426123585
k: 0.32211186180518114

```

```
k initial:  0.3147474879205983
k: 0.31950403394110616

k initial:  0.31228596247569634
k: 0.31693874873057915

k initial:  0.3098629863624799
k: 0.314414301670097
```

위의 출력문에서 k initial은 Eckart 식에 의해 도출된 초기 k 값이고, k는 Bisection Method에 의해 도출된 k 값이다.

108개의 경우에 대하여 두 k값의 차이에 대한 절댓값의 평균에 대한 백분율은 0.004214%로 도출되었다.

Bisection Method로 도출해낸 108개의 k값을 바탕으로, 파속 C를 저장할 List를 아래와 같이 정의한 다음,

```
C = []
```

아래의 관계식을 이용하여,

$$L = \frac{2\pi}{k}$$

위에서 저장한 108개의 파수 k를 담고 있는 List에 대해 반복문을 돌면서, 아래와 같이 정의되는 파속 C를,

$$C = \frac{L}{T}$$

아래의 코드를 통해 산정하여 이를 위에서 정의한 List C에 저장한다.

```
for i in range(0, len(k)):
    C.append((2 * math.pi / k[i])/T[i])
```

2.8.4 The Comparison of Wave speeds.

마지막으로, 위에서 각 방법들에 의해 최종적으로 산정하여 각각의 명시된 List내에 저장된 파속 C의 값들을 하나의 Figure내에 Plotting하는 아래의 코드로부터,

```
import matplotlib.pyplot as plt
import numpy as np
import math

# Using Numpy to create an array X
```

```
X = np.arange(0, len(C), 1.0)

# Plotting both the curves simultaneously
plt.plot(X, C, color='r', label='Bisection Method')
plt.plot(X, Eckart_C, color='g', label='Eckart')
plt.plot(X, Hunt_C, color='b', label='Hunt')

# Naming the x-axis, y-axis and the whole graph
plt.xlabel("Index")
plt.ylabel("$C$")
plt.title("Wave Velocities")

# Adding legend, which helps us recognize the curve according to it's color
plt.legend()

# To load the display window
plt.show()
```

그 결과를 보이면, 아래와 같다.

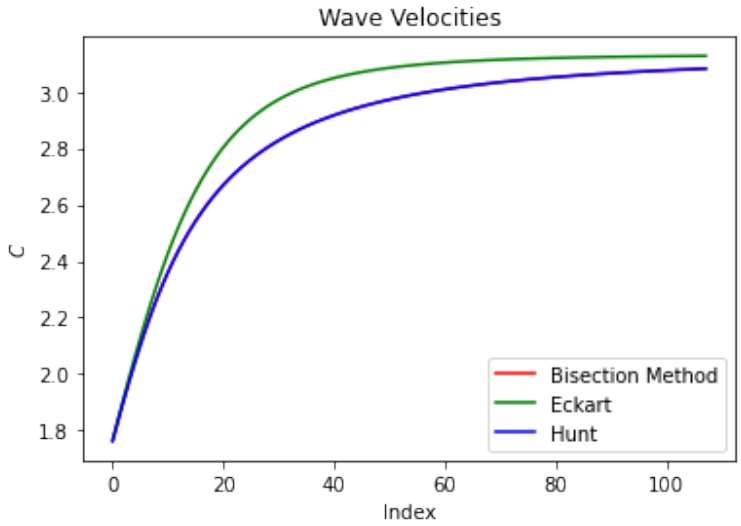

Figure 1: The Comparison of Wave speeds. (Bisection Method vs. Eckart Method vs. Hunt Method)

위에서 Bisection Method와 Hunt Method의 경우 거의 일치하는 모습을 보임을 확인할 수 있는데, You(2003)에 따르면, 여러 개의 파랑분산식의 양해(Hunt, 1979; Nielsen, 1982; Fenton과 McKee, 1990; Guo, 2002; Nielsen, 2002; You, 2003)를 비교하였을 때, Hunt의 해가 $\nu \leq \pi$의 수심범위에서 가장 정확하다는 것을 밝혀 내었다.

여기서 ν는 위의 Hunt의 식에서 정의된 $y = \frac{\sigma^2 h}{g}$와 동일한 것으로서, 심해에서의 상대수심을 나타낸다.

두 방법의 결과가 거의 일치하는 결과가 도출되었다는 사실로부터, 아래와 같이 Hunt 식에서 정의되는 $y = \frac{\sigma^2 h}{g}$ 값들을 List에 저장하는 다음의 코드를 작성한 다음,

```
added_y = []
```

이전에 y를 산정하는 코드에 위의 List added_y에 그 값을 저장하는 코드를 아래와 같이 추가하여,

```
for i in added_sqr_sigma:
    y = i*h/g
    added_y.append(y)
    dny = 0

    for j in range(0, len(d)):
        dny += d[j] * (y ** (j+1))

    sqr_kh.append((y) ** 2 + (y) / (1 + dny))
```

저장된 y값들을 아래의 코드로 Plotting 및 y의 최댓값까지 함께 표시해보면,

```
import matplotlib.pyplot as plt
import numpy as np
import math

# Using Numpy to create an array X
X = np.arange(0, len(C), 1.0)

# Plotting both the curves simultaneously
plt.plot(X, added_y, color='r', label='The y value of the Hunt method.')

# Naming the x-axis, y-axis and the whole graph
plt.xlabel("Index")
plt.ylabel("$y$")
plt.title("y value")

ymax = max(added_y)

s = 'The maximum value of y= ' + str(ymax) + ' '

plt.annotate(s, (20, 2))

# Adding legend, which helps us recognize the curve according to it's color
plt.legend()

```

```
# To load the display window
plt.show()
```

다음과 같다.

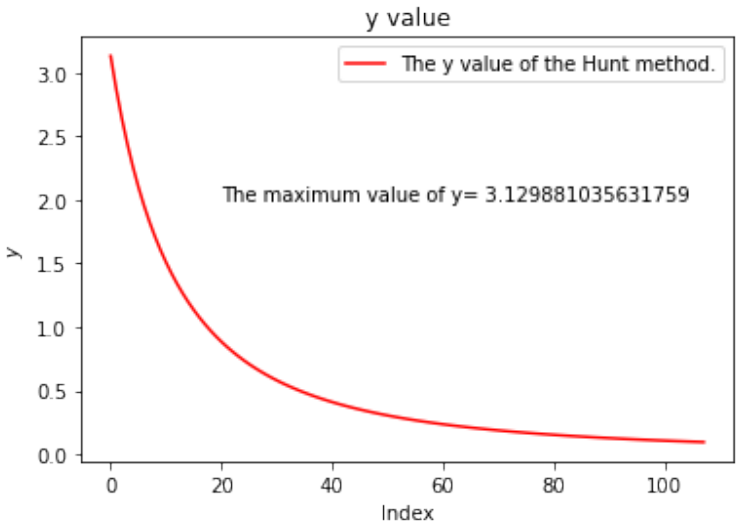

Figure 2: Graph for y of the Hunt Method

위 결과에서도 확인할 수 있듯이, $\nu = y = \frac{\sigma^2 h}{g}$의 최댓값이 대략 3.13 정도로 그 값이 π보다 작은 범위에 있기 때문에, 이렇듯 두 방법에 의한 파속의 결과에서 큰 차이를 보이지 않고 양해범임에도 거의 정확한 파속 C가 도출되고 있는 모습을 확인해볼 수 있다.

또한, Eckart에 의해 산정한 파속 C가 Hunt의 방법과 Bisection 방법보다 상대적으로 더 큰 값으로 도출되는 것 역시 확인해볼 수 있다.

3 Conclusion

3.1 Conclusion

본 보고서에서는 서론에서 언급하였듯이, 기본적으로 아래와 같이 정의되는 분산관계식(Dispersion Relationship)을 이용하여 모든 문제를 해결하였다.

$$\sigma^2 = gktanh(kh)$$

첫 과제로서, kh가 주어지는 경우에는 단순히 위의 분산관계식에 그 값을 대입 또는 주어진 수심 h로부터 k값을

도출하여 이를 대입함으로써, 각진동수 σ를 구하였고, $T = \frac{2\pi}{\sigma}$의 관계식으로부터 주기 T를 산정할 수 있었다.

또한, 두 번째 이후의 과제에서는 첫 번째 과제에서와는 다르게, k값이 주어지지 않은 상황에서 k 값을 산정하고 이로부터 파속 C를 구하는 과정이 수반되었다.

특히, k가 미지인 경우에는 Eckart나 Hunt 식과 같은 양해법(Explicit Method)을 이용하여 k값 산정 및 음해법(Implicit)에 대한 초깃값으로 이를 이용하여, 더 정확하고 정밀한 해를 찾아보는 과정을 살펴보았다. 또한, 파수 k를 산정한 이후에는 $L = \frac{2\pi}{k}$ 관계식으로부터 파장 L을 산정하고, 파속 $C = \frac{L}{T}$를 구하는 절차를 밟았다.

마지막 과제에서는 양해법인 Eckart와 음해법 중 하나인 이분법(Bisection Method) 외 또 다른 양해법인 Hunt 식을 이용하여 각각 k값을 산정하고, 최종적으로 파속 C를 도출하여 그래프를 그려 이들을 서로 비교해보았다.

그 결과, Eckart에 의해 산정된 k값으로부터 구한 파속 C는 다른 두 방법의 결과에 비해 과대평가되는 경향을 확인할 수 있었고, Hunt의 방법은 연구결과에 따라 $\nu \leq \pi$ 범위에서 양해법 중 가장 정확한 결과가 도출되는 것도 음해법으로부터 산정된 파속 C와의 그래프 상의 비교를 통해 확인해볼 수 있었다.

특히, 본 Question 4가 이러한 연구결과에서 시사하고 있는 범위내에 존재 즉 지배받는지의 여부를 확인하기 위해 별도로 $\nu = y = \frac{\sigma^2 h}{g}$의 그래프를 도출해보았고, 그때의 최댓값 또한 출력해봄으로써 명확하게 그 값이 π보다 작음을 확인할 수 있었다. 이로부터 해당 연구에서 시사하고 있는 내용의 성립여부에 대해서도 엄밀하게 분석해볼 수 있었다.

결론적으로, 본 보고서에서는 다양한 방법을 이용하여 분산관계식을 통해 k값을 산정하는 과정이 수록되었고, 기본적인 해안공학의 개념들을 컴퓨터(Computers)내에서 코딩(Coding)을 통해 직접 구현해보는 과정을 통해 이론적인 내용들의 재정립 및 실무적 활용 능력 또한 배양할 수 있겠다.

4 Acknowledgement

본 과제를 부여해주신 Prof. Jung, T.H. 교수님께 감사의 말씀을 드립니다.

5 References

[1] Jonghyeok Kim. *Code Files Written in Python by Kim, J.H. For Report I in Design of Coastal Engineering.* 2022. URL: `https://github.com/enfycius/Coastal-Engineering/blob/main/Assign-1.ipynb` (visited on 11/10/2022).

[2] 장호철 이창훈. "순환 관계에 의한 파랑분산식의 양해". In: *대한토목학회 논문집* (2008), pp. 111–114.