MinWook Kang

January 13, 2023

1 Implement Your Own fixed quad in [-1, 1]

우리는 앞 부분에서, 해를 추정하는 Root Finding Method를 공부했다. 그리고 이번 챕터에서는 Rimann Sum, Trapezoid Rule, Simpsion's Rule 그리고 Gaussian Quadrature을 통하여 어떤 함수 f 가 주어졌을 때, 우리가 원하는 범위 안에서의 적분 값을 실제의 적분 값과 매우 비슷하게 추정할 수 있는 방법을 배웠다. 이것으로 실제로 적분이 필요한 문제들에 대해 접근해 보겠다.

1.1 Problem Recognition

SciPy 모듈에서 fixed quad를 사용하면, Gaussian Quadrature 의 방식으로 원하는 적분 범위 안에서의 적분 값을 유추할 수 있다. 하지만, 이 모듈을 사용하지 않고, 모듈을 만들어서 문제에 적용하는 것을 보일 것이다.

$$\int_{-1}^{1} e^x dx = e - e^{-1} \approx 2.3504023872876029138 \tag{1}$$

$F(x) = e^x$ 를 [-1, 1]로 적분하게 되면 나오는 값을 구하는 모듈을 만들어야 한다. Gauss-Legendre 방식에 의하면, [-1, 1]에서 N에 대한 x_i 값과 w_i 값을 알고 이것을 곱하여 더하게 되면,

$$\int_{-1}^{1} f(x)dx \approx \sum_{i=1}^{n} w_i f(x_i) \equiv G_n(f) \tag{2}$$

위의 공식에 따라, [-1, 1] 사이의 적분 값을 추산할 수 있다. 따라서 우리는 먼저 x_i 값과 w_i 값을 구해줄 것이다. 그러기 위해서는 우선, 임의의 숫자 N에 대한 르장드르 다항식의 해가 필요한데, 이는 SciPy 모듈에서 roots_legendre 을 통해 쉽게 구할 수 있다. 먼저 roots_legendre를 보면, 르장드르 다항식에 들어가는 임의의 숫자 N에 대하여 해 roots 와 무게 weights 값을 알려주는 모듈이다. 만약 N 의 값에 3을 대입하면 아래처럼 구할 수 있다.

```
from scipy.special import roots_legendre, eval_legendre
roots, weights = roots_legendre(3)
roots, weights

(array([-0.77459667,  0.        ,  0.77459667]),
 array([0.55555556, 0.88888889, 0.55555556]))
```

1.2 Development of a solution

def 함수를 이용하여 min_fixed_only_quad(f, N) 함수를 정의하고, roots 와 weights인수를 xi 와 wi 로 저장한 후, zip 함수를 통하여 각 열마다 묶어준 다음, 각각의 (wi, xi) 의 묶음에 따라 식 $w * (f(x))$ 계산한다. 그후 묶음에 대한 이 식들을 더해줌으로서 위의 (2)를 파이썬으로 구현할 것이다. 먼저 $F(x) = e^x$ 이고, 적분 범위는 [-1, 1]이라고 하면 다음과 같다.

```
def f(x):
    return np.exp(x)

def min_fixed_only_quad(f, N):
    roots, weights = roots_legendre(N)
    xi = roots
    wi = weights
    return sum(w*(f(x)) for w, x in zip(wi, xi))
print(f"""
n = 1: I  = {min_fixed_only_quad(f, 1)}, Abs error = {(f(1) - f(-1)) - min_fixed_only_quad(f, 1)}
n = 2: I = {min_fixed_only_quad(f, 2)}, Abs error = {(f(1) - f(-1)) - min_fixed_only_quad(f, 2)}
n = 3: I = {min_fixed_only_quad(f, 3)}, Abs error = {(f(1) - f(-1)) - min_fixed_only_quad(f, 3)}
n = 4: I = {min_fixed_only_quad(f, 4)}, Abs error = {(f(1) - f(-1)) - min_fixed_only_quad(f, 4)}
n = 5: I = {min_fixed_only_quad(f, 5)}, Abs error = {(f(1) - f(-1)) - min_fixed_only_quad(f, 5)}
""")

n = 1: I = 2.0, Abs error = 0.35040238728760276
n = 2: I = 2.3426960879097307, Abs error = 0.007706299377872039
n = 3: I = 2.3503369286800115, Abs error = 6.54586075912178e-05
n = 4: I = 2.3504020921563766, Abs error = 2.9513122612456755e-07
n = 5: I = 2.350402386462826, Abs error = 8.247766913882515e-10
```

1.3 Execution and Assessment

이때, 우리는 실제 값과의 차이를 비교하기 위하여 $f(1) - f(-1)$ 값에다가 추산한 값을 빼서, N 에 따라서 얼만큼 오차가 날 수 있는지 확인해 볼 것이다. 먼저 SciPy package의 fixed_quad로 계산한

결과는 다음과 같다.

```
n = 1: approx = (2.0, None)
n = 2: approx = (2.3426960879097307, None)
n = 3: approx = (2.3503369286800115, None)
n = 4: approx = (2.350402092156377, None)
n = 5: approx = (2.350402386462826, None)
```

이 두 값을 빼보면, N = 4 인 경우 $-4.440892098500626e-16$ 를 제외하고는 0으로 맞아 떨어지는 것을 볼 수 있다. N=4 인 경우에도, 수가 매우 작으므로, 0이라 봐도 무방하다. 하지만 위의 함수는 범위가 [-1, 1] 로 정해져 있기 때문에, 이를 변수변환을 통해 어느 구간에서도 적분 값을 추산할 수 있는 모듈을 만들 것이다.

2 Own fixed quad in any range

2.1 Problem Recognition

위의 상황에서는 범위가 [-1, 1]을 만족하는 상황에서만 적분 값을 구할 수 있었다. 그럼 만약, 적분 범위가 아래와 같은 식일 경우는 어떻게 구할 수 있을까? 이는 적분의 범위를 변수변환을 통하여 바꾸어 주면 된다.

$$\int_0^1 \frac{4}{1+x^2}dx \tag{3}$$

먼저 Gauss' Method에 따르면, x 와 dx는 아래처럼 바꾸어도 무방하다.

$$t = \frac{b-a}{2}x + \frac{b+a}{2} \quad , \quad dt = \frac{b-a}{2}dx \tag{4}$$

2.2 Development of a solution

변수 변환을 통해 적분 범위는 dt에 대해서 [-1, 1]을 만족하게 된다. 그러면 위의 함수를 조금 변형하면 똑같이 적분을 시킬 수 있다. 위의 def 함수에서, x를 t에 대한 식으로 바꾸어지고, 함수에 $(b-a)/2$를 곱하게 해주면 되므로, 위의 식을 $F(x) = \frac{4}{1+x^2}$, 적분 범위는 [0, 1]로 잡고 계산하면 다음과 같다.

```python
def f(t):
    return np.exp(t)
```

```python

def min_fixed_any_range(f, a, b, n):
    g = lambda x: (b - a)/2 * f((b-a)/2*x + (a+b)/2)
    roots, weights = roots_legendre(n)
    xi = roots
    wi = weights
    return sum(w*(g(x)) for w, x in zip(wi, xi))
```

먼저 def로 선언한 함수를 보면 인수를 4개로 두었는데, 이는 구하고자 하는 함수의 범위를 나타내며, 변수 변환을 통해 원하는 [a, b] 의 적분 값을 구할 수 있다. 또한 $g(x)$ 라는 함수에 대해 lambda 를 통하여 아래의 식을 새롭게 정의 하였다.

$$g(x) = \frac{b-a}{2} f(\frac{b-a}{2}x + \frac{b+a}{2}) \tag{5}$$

이는 $f(t)$에서 $t = \frac{b-a}{2}x + \frac{b+a}{2}$ 을 대입한 것이다. 또한, $\frac{b+a}{2}$ 는 적분 범위를 벗어날 수 있기 때문에, 적분을 나중에 해주고, $\frac{b+a}{2}$ 를 곱해줌으로써, $g(x)$에 대하여 위의 코드가 성립됨을 볼 수 있다. 그리고 section 1에서 본 것과 같이, zip(wi, xi) 에 대하여 각각의 w, x값을 할당하여 $g(x)$와 w를 곱한 것을 더해주게 되면, 우리가 구하고자 한 적분 값을 구할 수 있게 된다. 위의 코드로 인한 결과를 보면 다음과 같다.

2.3 Execution and Assessment

```python
print(f"""
n = 1: I = {min_fixed_any_range(f, -1, 1, 1)}, Abs error = {(f(1) - f(-1)) - min_fixed_any_range(f, -1, 1, 1)}
n = 2: I = {min_fixed_any_range(f, -1, 1, 2)}, Abs error = {(f(1) - f(-1)) - min_fixed_any_range(f, -1, 1, 2)}
n = 3: I = {min_fixed_any_range(f, -1, 1, 3)}, Abs error = {(f(1) - f(-1)) - min_fixed_any_range(f, -1, 1, 3)}
n = 4: I = {min_fixed_any_range(f, -1, 1, 4)}, Abs error = {(f(1) - f(-1)) - min_fixed_any_range(f, -1, 1, 4)}
n = 5: I = {min_fixed_any_range(f, -1, 1, 5)}, Abs error = {(f(1) - f(-1)) - min_fixed_any_range(f, -1, 1, 5)}
""")

n = 1: I = 2.0, Abs error = 0.35040238728760276
```

```
n = 2: I = 2.3426960879097307, Abs error = 0.007706299377872039
n = 3: I = 2.3503369286800115, Abs error = 6.54586075912178e-05
n = 4: I = 2.3504020921563766, Abs error = 2.9513122612456755e-07
n = 5: I = 2.350402386462826, Abs error = 8.247766913882515e-10
```

실제의 값 Abs erorr에 우리가 만든 모듈의 차를 구했을때, N 의 값, 즉 legendre 다항식의 차수가 충분하지 않았을 때, error의 폭이 커짐을 볼 수 있고, N이 커질 수록 0에 수렵하는 것을 볼 수 있다. 이와 관련하여 오차를 줄이는 방법에서도 논하여 보겠다.

3 reduce the error 1

3.1 Problem Recognition

$$\int_0^1 \frac{4}{1+x^2}dx \tag{6}$$

위의 함수의 실제 근은 π 이다. 위의 함수에서 N의 값을 임의로 지정하지 않고, 원하는 오차 범위 내에서 적분을 근사적으로 구하려면 어떻게 해야 할까? 즉, 오차 범위는 10^{-5} 보다 크면 안되고, $F(x) = \frac{4}{1+x^2}$ 에서 [0, 1] 까지의 적분값을 유추해야 한다.

3.2 Development of a solution

우리는 앞서서 Root Finding에서도 똑같이 이용한 방식을 이용할 수 있다. 바로 위에서 구한, 실제값에 결과값을 뺀 오차가 우리가 설정한 범위보다 작아질 때 까지 N을 1씩 증가시키면서 반복시키면 된다. while문을 통하여, current_acc 가 우리가 정한 target_acc보다 작아질 때 까지 반복시킨다. 또한, N의 범위도 지정해 주어, 작아졌음에도 계속해서 반복되는 것을 막는다. 이를 코드로 구현하면 다음과 같다.

```
exact_value = np.pi
target_acc = 10**(-5)

current_acc = 1
N = 2
result = 0
while current_acc > target_acc and N < 20:
    result = min_fixed_any_range(lambda x: 4/(1 + x*x), 0, 1, n=N)
    current_acc = abs(exact_value - result)
```

```python
    N += 1

print(f"approximate value = {result}, number of sample points = {N}")
print(f"Abs error = {exact_value - result}")
```

3.3 Execution and Assessment

```
approximate value = 3.141592639884753, number of sample points = 6
Abs error = 1.3705040213807251e-08
```

우리가 원하던 결과 값이 구해졌음을 알 수 있다. 우리가 정한 오차 범위인 10^{-5} 보다 Abs error 값이 더 작은 것을 볼 수 있고, N의 횟수 또한 20보다 작은 6번만에 구해졌음을 볼 수 있다. 마찬가지로 다른 문제에도 이를 적용해보자.

4 reduce the error 2

4.1 Problem Recognition

$$\int_1^2 \frac{1}{x} dx \tag{7}$$

이 식에서 오차는 10^{-7} 보다 작으며, $N < 100$ 을 만족하는 적분 값을 구해보자.$F(x) = \frac{1}{x}$ 를 [1, 2] 에서 적분하게 되면 정확히 1 이 나온다.

4.2 Development of a solution

위와 동일한 방법으로, 적분 범위와 오차만 바꾸어 준다. 또한, $N < 100$이므로 반복 횟수도 100 이하가 되게끔 조정해준다.

```python
exact_value = np.log(2)
target_acc = 10**(-7)

current_acc = 1
N = 2
result = 0
```

```
while current_acc > target_acc and N < 100:
    result = min_fixed_any_range(lambda x: 1/x, 1, 2, n=N)
    current_acc = abs(exact_value - result)
    N += 1

print(f"approximate value = {result}, number of sample points = {N}")
print(f"Abs error = {exact_value - result}")
```

4.3 Execution and Assessment

```
approximate value = 0.6931471578530402, number of sample points = 6
Abs error = 2.270690513395124e-08
```

우리가 원하는 방향대로, 적분값을 구했다. 이번에는 Error Function에서 [0.3] 의 범위내에서 추산한 적분의 값이 실제로 Error Function에서 구한 적분의 값과 어느정도 차이가 있는지 확인해 볼 것이다.

5 Error Function

5.1 Problem Recognition

$$\mathrm{erf}(x) = \frac{2}{\sqrt{\pi}} \int_0^x e^{-t^2} dt \tag{8}$$

Error Function에서 실제 [0, 3]까지를 값을 구하고, 이것을 선으로 구현하고 빨간색 점으로는 우리가 추산한 Error Function의 값을 찍어서, [0, 3] 까지 오차의 차이나 정도의 변화가 있는지 눈으로 확인할 것이다. 그러기 위해선 우선, Error Function의 실제 값을 알아야 한다. 또한 우리가 이제껏 구했던 방식과 달리, [0, 3]의 범위를 [x, x + i] 처럼 적잘하게 잘라, 잘라낸 작은 사이의 적분값을 구한후, 이를 원하는 범위 까지 계속해서 더하는 방식으로 적분을 표현할 것이다. 만약, 잘라낸 작은 사이의 적분값만 구한후 (즉, [x, x + i] 사이에서의) 그래프로 나열하게 되면, 원하는 범위 전까지의 [0, x] 까지의 적분 값은 사라지고, $\frac{2}{\sqrt{\pi}} \int_x^{x+i} e^{-t^2} dt$ 값만 구해질 것이기 때문에 반드시 구했던 모든 적분 값을 중복해서 더해주는 것을 유의해야 한다.

5.2 Development of a solution

먼저 partition 변수를 지정하여 [0, 3] 까지의 적절한 갯수 31개로 나누어 준다. 그리고 zip 함수를 통해 각각의 (a, b)쌍으로 잘게 범위를 쪼개어, 이때의 적분 값을 구한 것을 partial_sum에 저장하여, 이 리스트를 np.cumsum 해줌으로써, 위에서 말한 적분 값을 중복해서 더해줄 것이다. 이를 코드로 구현하면,

```
from scipy import special

partition = np.linspace(0, 3, 31)
partial_sums = [
    min_fixed_any_range(lambda t: np.exp(-t*t), a, b, n=5)
    for (a, b) in zip(partition[:-1], partition[1:])
]

xs = partition
approx_erf = np.cumsum(partial_sums) * 2/np.sqrt(np.pi)
approx_erf = np.hstack(([0], approx_erf)) # erf(0) = 0

x_exact = np.linspace(0, 3, 200)
y_exact = special.erf(x_exact)

plt.figure()

plt.plot(x_exact, y_exact, "-k", label="exact")
plt.plot(xs, approx_erf, ".:r", lw=1, label="approx")
plt.legend()

plt.show()
```

5.3 Execution and Assessment

$Figure 1$을 보면 우리가 원하는 대로 그래프가 그려졌음을 볼 수 있다. 이때 그래프의 오차를 계산하면 다음과 같다.

```
0.00000000e+00, 0.00000000e+00, 0.00000000e+00, 6.93889390e-18,
        0.00000000e+00, 0.00000000e+00, 2.77555756e-17, 4.16333634e-17,
        2.77555756e-17, 2.77555756e-17, 0.00000000e+00, 2.77555756e-17,
```

```
0.00000000e+00,  2.77555756e-17,  2.77555756e-17,  5.55111512e-17,
1.11022302e-16,  5.55111512e-17,  1.11022302e-16,  5.55111512e-17,
0.00000000e+00,  5.55111512e-17,  0.00000000e+00,  0.00000000e+00,
5.55111512e-17,  0.00000000e+00,  5.55111512e-17,  0.00000000e+00,
0.00000000e+00,  0.00000000e+00,  0.00000000e+00,  0.00000000e+00,
0.00000000e+00,  0.00000000e+00,  0.00000000e+00,  0.00000000e+00,
0.00000000e+00,  1.11022302e-16,  0.00000000e+00,  1.11022302e-16,
0.00000000e+00,  1.11022302e-16,  1.11022302e-16,  1.11022302e-16,
1.11022302e-16,  1.11022302e-16,  0.00000000e+00, -1.11022302e-16,
```

N= 5로 두고 그림을 그렸지만, 만약 N의 값을 바꾸어 주게 된다면, 예상할 수 있듯이, 오차가 좀 더 벌어짐을 볼 수 있다. 예를 들어, 오차를 더 정확하게 보기 위해, 적분하고자 하는 값을 [0, 3] 에서 [0, 20] 으로 바꾸어 주고, N의 값을 1, 2, 3, 4 로 두고 그림을 각각 그려보면 다음과 같다.

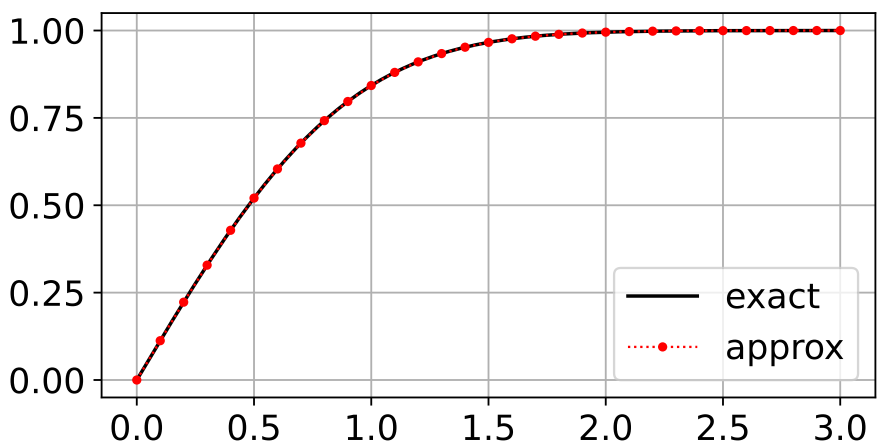

Figure 1: $Error Function$

이처럼, 우리의 예상대로 N의 값이 커질 수록 더 abs error 값이 줄어들면서 실제 값과 유사하게 적분이 될 수 있음을 볼 수 있다.

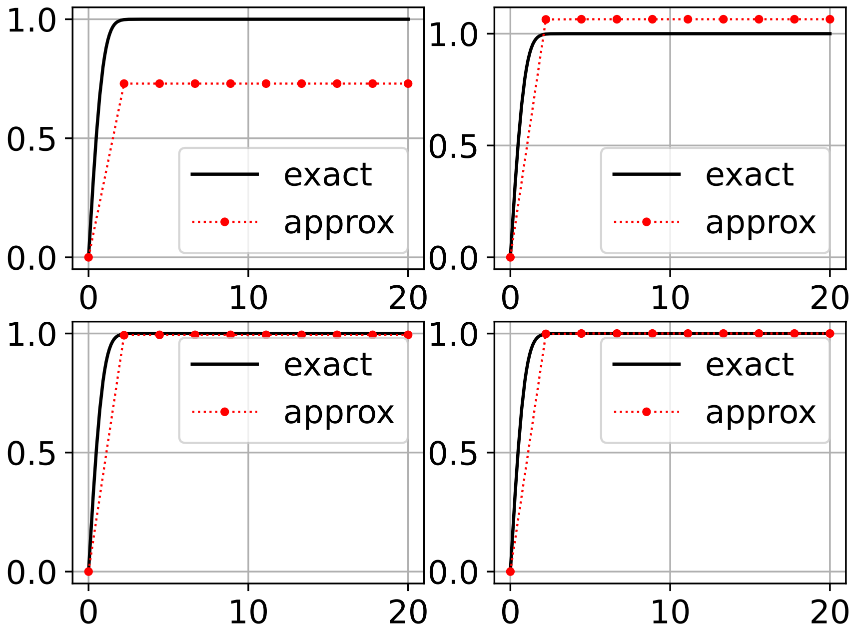

Figure 2: $ErrorFunction\_for\_N = 1, 2, 3, 4$

6 Double Integral

6.1 Problem Recognition

$$I = \int_1^2 \int_0^1 y^x dx dy \approx 1.2292741342880657562 \tag{9}$$

이번에는 함수 I를 구해야 하는데, dx와 dy에 대하여 각각의 범위에 대해 적분을 시켜주려고 한다. $Y(x)$ 를 적분하려면, 먼저 dx 에 대하여 적분을 해준 뒤, y 에 대하여 적분을 해주어야 한다.

6.2 Development of a solution

먼저 우리는 x에 대한 식을 y의 변수와 함께 저장하는 함수를 지정해 주어야 한다. 즉, x값이 변하더라도 y값은 변하지 않는 함수를 만들고, 그 함수에서 x에 대하여 [0, 1] 의 범위를 적분해 준다. 그리고 다시 y에 대해 [1, 2] 범위를 적분해주면서 이차 적분을 풀어줄 수 있다.

즉, 초기의 $F(x)$는 x에 대한 함수로 잡는다. 그리고 x에 대하여 적분을 하면 $F(y)$와 같으므로, 이를 다시 y에 대하여 적분을 한다. 이를 코드로 구현하면 다음과 같다.

6.3 Execution and Assessment

```
def inner_sum(y):
    return min_fixed_any_range(lambda x: y**x, 0, 1, 11)

outer_sum = min_fixed_any_range(inner_sum, 1, 2, 11)

print("approximatio = {}, absolute error = {}"
      .format(outer_sum, abs(outer_sum - 1.2292741342880657562)))

approximatio = 1.229274134361613, absolute error = 7.354716835550335e-11
```

실제 값과의 차를 구하면 0에 수렴할 정도로 작은 것을 볼 수 있다. 이것으로 SciPy 모듈의 fixed quad 모듈을 사용하지 않고, 직접 모듈을 만들어서 문제에 적용해 보았다.

7 Orthogonality of Legendre polynomials

7.1 Problem Recognition

다음 Legendre 다항식의 직교성을 증명하여라.

$$\int_{-1}^{1} P_m(x)P_n(x)dx = \frac{2}{2n+1}\delta_{mn} \tag{10}$$

이 문제는 m 과 n이 임의의 정수일 때, [-1, 1]에서 Legendre 다항식 $P_m(x)$ 과 $P_n(x)$의 곱을 적분하였을 때, $m = n$이면 $\frac{2}{2n+1}$, 다르면 0 이 되는 것을 통하여, 다항식의 직교성을 증명하라는 문제이다. 이 문제를 풀기 위해서는 먼저, $P_m(x)$ 와 $P_n(x)$ 의 곱을 구하고, 그 값을 [-1, 1]에 대하여 적분을 시켜주면 된다. 이때, m, n은 임의의 정수이기 때문에, 같을 때와 다를 때를 나누어 확인해야 한다. 이는 δ_{mn} 의 특성을 유의해야 한다.

즉, 두가지 조건을 정리하면 다음과 같다.

- $n = m$ 인 경우, $\int_{-1}^{1}(P_n(x))^2dx - \frac{2}{2n+1} = 0$ 이 만족되어야 한다.
- $n \neq m$인 경우, $\int_{-1}^{1} P_m(x)P_n(x)dx = 0$이 만족되어야 한다.

7.2 First Condition

먼저 첫번째 조건이 성립하는지를 풀기 위해서, legendre(m) 을 이해하면, m에 어떠한 임의의 정수를 넣었을 때, 르장드르 다항식을 나타내는 함수이다. 유의해야 할 것은, legendre() 모듈은 함수를 나타내는 것이다. 따라서, 우리는 $F_m(x) = Legendre(m, x)$로 잡고, 원하는 문제를 풀 수 있다. 르장드르 모듈에서는 따로 x를 인자로 두지 않기 때문에, $F_m(x) = Legendre(m, x)$ 로 나타내지 않고, Legendre(m)으로 나타내면 된다. 이와 관련하여 몇가지 코드를 확인하면,

```
import random
n, m = random.randint(0,10), random.randint(0,10)

def P_m(m):
    return min_fixed_any_range(legendre(m), -1, 1, 10)
print(legendre(m) )

       4             3        2
4.375 x + 4.857e-16 x - 3.75 x + 2.429e-16 x + 0.375
```

이렇게 위에서 언급했던 것 처럼, m에 임의의 정수를 넣으면 르장드르 다항식을 보여주는 함수이다. 이때, 르장드르 다항식은, m에 따라서 짝함수와 홀함수가 결정되는데, 위의 식에서는 x^3처럼 홀수 차수 항을 볼 수 있지만 계수는 0이므로 고려하지 않아도 된다. 이제 무작위의 n, m에 대해서 $n = m$ 이 성립할 때, 위의 조건에 만족하는지 확인할 것이다.

```
import random
def P_m_P_M(m,n):
    return legendre(m) * legendre(n)
for i in range(10):
    n = random.randint(0,11)
    m = n
    print(f"n = {n}, m = {m}")
    print((2 / (2 * n + 1)) - min_fixed_any_range(P_m_P_M(m,n), -1, 1, 30))

n = 4, m = 4
-8.520961713998076e-15
n = 9, m = 9
-4.3921810632951974e-13
n = 10, m = 10
-4.626687921671646e-12
n = 6, m = 6
-1.6375789613221059e-15
n = 1, m = 1
-5.329070518200751e-15
n = 0, m = 0
0.0
n = 6, m = 6
-1.6375789613221059e-15
n = 9, m = 9
-4.3921810632951974e-13
n = 2, m = 2
-6.827871601444713e-15
n = 10, m = 10
-4.626687921671646e-12
```

범위를 [0, 10] 으로 두고 계산하였지만, 만약, 범위가 10이상 커지게 되면 르장드르 다항식의 차수가 많아지게 되면서, 이를 적분하는데 많은 오차가 발생하게 된다. 따라서 이를 막으려면, 적분을 추산하는

함수의 N의 크기가 더욱 커져야 한다. 만약, 범위를 [10, 21] 로 두고, N = 5 인 경우,

```
n = 14, m = 14
-6.884520453720455e-10
n = 21, m = 21
-0.00011664384684553081
n = 21, m = 21
-0.00011664384684553081
n = 18, m = 18
-4.849815616859987e-06
n = 21, m = 21
-0.00011664384684553081
n = 17, m = 17
7.57868260294059e-07
n = 10, m = 10
-4.626687921671646e-12
n = 15, m = 15
-4.713122087052213e-09
n = 15, m = 15
-4.713122087052213e-09
n = 21, m = 21
-0.00011664384684553081
```

이처럼, Legendre 다항식의 차수가 21에 가까워 질 수록, 오차 값이 커져버리는 경우가 발생할 수 있다. 따라서 적분 값을 추산할 때, N의 값을 조율하면서 맞추어 줄 수 있다. 정리하면, 우리가 적절하게 정한 오차 범위내에서 $n = m$ 을 가정하여 계산했을 경우, 우리가 초기에 설정한 첫번째 조건에 성립하는 것을 볼 수 있다. 그러면 두번째 조건이 타당한지를 확인해 볼 것이다.

7.3 Second Condition

두번째 조건은, $n \neq m$인 경우, $\int_{-1}^{1} P_m(x)P_n(x)dx = 0$이 만족되어야 한다. 이를 확인하기 위해선, 위에서 사용한 방식에서, n 과 m을 다르게 배치하므로써, 문제를 해결할 수 있다.

```
import random

def P_m_P_M(m,n):
    return legendre(m) * legendre(n)

for i in range(10):
```

```
    n = random.randint(0,11)
    m = random.randint(1,12) - 1
    print(f"n = {n}, m = {m}")
    print(min_fixed_any_range(P_m_P_M(m,n), -1, 1, 30))

n = 2, m = 5
-2.0990154059319366e-16
n = 5, m = 8
1.215954420485943e-14
n = 3, m = 11
1.8747243341055153e-13
n = 6, m = 9
-2.9786936805997755e-14
n = 8, m = 6
2.3614790678472275e-14
n = 6, m = 0
8.934693263018545e-15
n = 6, m = 11
2.1583342751929635e-13
n = 4, m = 5
-8.795048023202412e-16
n = 8, m = 4
8.455042221910958e-15
n = 11, m = 5
1.5622919624647125e-13
```

이때 우리는 n, m을 서로 다른 정수로 표현하기 위해서 -1 을 취했는데, n과 m은 무작위 정수이므로 $n < m$ 이나 $n > m$ 에 대하여 고려할 필요가 없다. ($n = m, m = n$을 대입하면 똑같다.) 따라서 이 결과값을 확인하면 모두 0에 수렴하기 때문에, 우리가 초기에 설정한 두번째 조건에도 만족하는 것을 볼 수 있다.

7.4 Execution and Assessment

초기에 결정한 두가지 조건

- $n = m$ 인 경우, $\int_{-1}^{1} (P_n(x))^2 dx - \frac{2}{2n+1} = 0$ 이 만족되어야 한다.
- $n \neq m$인 경우, $\int_{-1}^{1} P_m(x)P_n(x)dx = 0$이 만족되어야 한다.

가 성립하므로 따라서 우리는

$$\int_{-1}^{1} P_m(x)P_n(x)dx = \frac{2}{2n+1}\delta_{mn} \tag{11}$$

이 성립함을 증명할 수 있고 따라서 두 르장드르 다항식은 직교성을 가진다는 것을 확인할 수 있다.

8 Orthogonality of Bessel funcions

8.1 Problem Recognition

이번에는 베셀 함수의 직교성을 증명하는 문제이다. 이제 우리는 Bessel 함수에서 α 와 β 에 따른 직교성을 증명해야 한다. 베셀함수의 경우, Scipy의 special.jn 모듈을 통하여 베셀 함수를 나타낼 수 있다. 하지만 유의할 것은, 위의 르장드르 방정식과 달리, x 값을 따로 설정해 주어야 한다.

$$\int_0^1 J_p(\alpha x)J_p(\beta x)xdx = \frac{1}{2}J_{p+1}^2(\alpha) = \frac{1}{2}J_{p-1}^2(\alpha) = \frac{1}{2}(J_p')^2(\alpha), \qquad \alpha = \beta \tag{12}$$

$$\int_0^1 J_p(\alpha x)J_p(\beta x)xdx = 0, \qquad \alpha \neq \beta \tag{13}$$

유의해야 할 것은 바로, (12) 식에서 볼 수 있는데, $J_p'^2(\alpha)$ 값을 유추해야 한다. 이는 Sympy 의 Derivative() 모듈을 통해 도함수를 구할 수 있으나, 베셀함수의 특징을 생각하면, 어렵지 않게 유추할 수 있다. 바로, 베셀 함수의 반복 관계를 이용하면 된다. 베셀 함수의 반복 관계를 잠시 인용하면,

$$\begin{aligned}
\frac{d}{dx}(x^p J_p(x)) &= x^p J_{p-1}(x)\\
\frac{d}{dx}(x^{-p} J_p(x)) &= -x^{-p} J_{p+1}(x)\\
J_{p-1}(x) + J_{p+1}(x) &= \frac{2p}{x}J_p(x)\\
J_{p-1}(x) - J_{p+1}(x) &= 2J_p'(x)\\
J_p'(x) = -\frac{p}{x}J_p(x) + J_{p-1}(x) &\frac{p}{x}J_p(x) - J_{p+1}(x)
\end{aligned} \tag{14}$$

이 관계중에서 우리는 $J_{p-1}(x) - J_{p+1}(x) = 2J_p'(x)$ 을 이용하여 베셀 함수의 도함수를 구할 것이다.

8.2 Development of a solution

이제, 앞서서 legendre 다항식의 직교성을 구한 것 처럼 두가지의 조건을 성립하는지 확인할 것이다. 조건은 다음과 같다.

- $\alpha = \beta$ 인 경우, 이 만족되어야 한다.
- $\alpha \neq \beta$인 경우, 이 만족되어야 한다.