수학에서 컴퓨터 함수의 적용

10428 최명국

목차

1. 컴퓨터에서의 함수의 개념과 특징, 수학에서의 함수와의 공통점 및 차이점 조사
2. Python 언어를 통해 함수 구현
3. 실제 수학적 함수를 그리는 프로그램 구현
4. 다양한 수학적 함수들 구현
5. 실제 문제에서의 적용

서론

이번 탐구에서는 우선 수학의 함수와 컴퓨터의 함수의 개념에 대해 알아보고 컴퓨터의 함수의 형태, 만드는 방법에 대해 간단히 알아볼 것이다. 그다음 수학의 함수와 컴퓨터의 함수의 차이점에 알아볼 것이다.

컴퓨터의 함수를 이용해 고1 수학 함수 단원에서 배운 개념 대부분을 접목해 그래프로 구현해 볼 것이다. 그다음 구현한 함수들을 이용해 실제 문제들을 풀어보며 실생활에 접목해볼 것이다.

이 탐구를 통해서 수학의 함수와 컴퓨터의 함수 정의에 대해 알아보고 공통점과 차이점에 대해 알아볼 수 있다. 또한 컴퓨터의 함수를 수학의 함수와 연관 지어서 컴퓨터를 통해 수학의 함수를 직접 구현해 볼 수도 있고, 컴퓨터를 이용해 수학 문제를 풀어볼 수도 있다.

수학의 함수 f(x)에서 x에 값을 넣으면 f(x)가 일정한 값으로 결정되는 것처럼 컴퓨터에서의 함수 또한 함수의 인자에 값을 입력하면 다양한 결과물들을 도출할 수 있다. 또한 컴퓨터의 함수를 만드는 과정에서도 수학의 '집합'과 비슷한 개념인 '배열'이 이용되는 등 컴퓨터와 수학은 매우 큰 연관성이 있다.

본론

탐구를 시작하기에 앞서, 이 탐구보고서에서 사용된 프로그래밍 언어는 모두 Python으로 작성되었다. Python은 코드의 길이가 C언어 같은 다른 언어에 비해 짧아 그 의미를 더욱 직관적으로 파악할 수 있으며, 오픈소스(공개적으로 액세스할 수 있게 설계되어 누구나 자유롭게 확인, 수정, 배포할 수 있는 코드)를 이용해 더욱 다양한 활동을 해볼 수 있기 때문이다.

1. 함수의 개념 :
-수학 : "어떤 집합의 각 원소를 다른 집합의 유일한 원소에 대응시키는 이항관계"
-컴퓨터 : "소프트웨어에서 특정 동작을 수행하는 일정 코드 부분"
이처럼 수학과 컴퓨터에서의 함수의 정의는 조금 다르지만 어떠한 값을 통해 다른 값을 도출한다는 점은 같다.

2. 함수의 형태 :

```
def "함수이름"("매개변수") :
    "코드내용"
    return "결괏값"
```

Python에서의 함수의 형태는 대부분 위와 같은 형태로 작성된다. 첫 번째 줄의 def는 함수를 만들 때 사용하는 예약어이다. 함수의 이름은 수학에서의 f(x), g(x)의 f, g처럼 함수의 이름을 정해준다. 이는 함수를 만드는 사람이 임의로 지정할 수 있으며 보통 함수의 기능을 직관적으로 알 수 있도록 지정한다(팀 프로젝트에서는 다른 사람이 나의 함수의 기능을 쉽게 알 수 있도록 이름을 잘 짓는 것이 매우 중요하다). 매개변수는 간단히 말해 '이 함수가 입력받는 값은 이 매개변수로 지정한다'를 의미한다. 함수에 값을 입력하면, 입력받은 값을 매개변수에 저장하여 함수 내에서는 입력받은 값 대신 매개변수를 통해 계산을 수행한다. 코드의 내용은 함수의 기능을 만드는 본론 역할을 하고 return은 그 결괏값을 도출한다(return이 없는 경우도 있다).

```
#함수
def plusOne(x):
    x = x+1
    return x
#본문
a =plusOne(1)
print(a)
```

간단한 코드와 함께 살펴보면 위할 수에서 함수의 이름은 plusOne이고 매개변수는 x이다. 코드의 내용은 'x에 1을 더해라'이다. 본문에서는 a라는 변수에 plusOne 함수에 1을 넣은 값을 저장한다. 여기서 1은 '인자'로, 함수의 매개변수에 넣는 값을 의미한다. plusOne()은 1을 더해주는 함수이므로 a에는 2가 저장되고, 마지막 줄에서 계산한 결과인 a의 값을 출력한다. 그러면 결과적으로 화면에는 2가 출력된다.

3. 수학의 함수와의 차이점

① 수학의 함수는 참조투명성을 가진다.

참조투명성은 함수에 인자가 같다면 함수는 항상 같은 값을 반환한다는 의미이다. 수학에서의 함수는 정의역의 한 원소가 공역의 두 개 이상의 원소로 대응되지 않기 때문에 일정한 값을 넣으면 같은 값을 반환한다. 이를 컴퓨터 함수의 용어로 '순수 함수'라고 한다. 컴퓨터 함수에도 순수 함수가 있지만 그렇지 않은 경우도 많다.

```
globalvariable
variable =1

def plus(x):
    x =+variable
    returnx

a =plus(1)
print(a)
```

예를 들어 위의 코드를 보면 함수의 반환 값은 글로벌 변수인 'variable'의 값을 무엇으로 설정하는지에 따라 결과가 달라진다. 만약 내가 variable을 1이라 정하면 plus() 함수는 인자에 1만큼을 더한 값을 출력하고 100이라 정하면 100을 더한 값을 출력한다. 또한 랜덤한 값을 출력할 때도 함수의 값은 계속 달라질 수 있다. 따라서 컴퓨터의 함수는 외부의 영향에 따라 항상 일정한 값을 가지지는 않는다. 따라서 컴퓨터의 함수는 수학적으로 보면 함수라고 정의되기는 어렵기 때문에 수학과 컴퓨터의 함수에는 차이가 있다.

② 수학의 함수의 인자는 무한하다.

컴퓨터는 저장공간의 한계가 있기 때문에 인자의 개수에 한계가 있다.

f(x) = x+1

예를 들어 이와 같은 함수는 정의역이 실수 전체의 집합이기 때문에 그래프상으로 완전하게 직선을 그리게 된다. 하지만 컴퓨터에서는 이 함수에 모든 실수를 대입할 수 없다.

f(x) = x+1 (x = {1, 2, 3, 4, 5})

함수의 개수를 제한해주기 위해서는 이렇게 정의역을 정해주어야 한다.

③ 컴퓨터 함수은 다양한 형태의 인자를 가질 수 있다.

수학 함수의 경우에는 f(x)에는 x를 '수'로 받지만 컴퓨터 함수는 문자, 참/거짓 관계 등 다양한 인자를 받을 수 있다. 심지어 어떤 함수는 인자를 받지 않기도 한다.

```
def positive_number(x):
    if(x==True) :
        print("양수 또는 0입니다")
    else :
        print("음수입니다")
a=5
if(a>=0) :
    b =True
else :
    b =False
positive_number(b)
```

이 코드는 해당 변수의 값이 양수인지 음수인지 판단하는 코드이다. 만약 변수의 값이 양수라면 b 변수는 True가 되고, 아니면 False가 된다. 함수에서는 인자에 숫자 대신 True/False라는 참/거짓 형태의 자료형을 대입하여 판단하도록 되어있다.

④ 컴퓨터 함수는 다양한 결과를 반환할 수 있다.

수학 함수에서는 함수에 정의역의 원소를 대입하면 그에 대응하는 공역의 원소가 반환된다. 하지만 컴퓨터 함수는 정의에서 보았듯이 하나의 특정 동작을 수행하는 코드이기 때문에 인자로 수 이외의 다양한 형태의 인자를 가질 수 있는 것처럼 결과도 다양한 형태로 도출될 수 있다. print() 함수처럼 문자를 출력하거나, input() 함수처럼 사용자에게서 값을 입력받는 것도 함수이다. 심지어 결괏값이 공집합에 대응되어 결괏값을 반환하지 않는 함수도 있다.

4. 함수 그려보기

앞서 말했듯이, Python은 오픈소스를 활용해 다양한 활동을 할 수 있다. 컴퓨터에서 구현한 수학적 함수를 컴퓨터에서 구현하고 시각적으로 표현할 수 있게 해주는 오픈소스를 이용하여 배운 내용을 활용하여 직접 그래프를 그려볼 것이다.

① 원리

수학과 컴퓨터 함수의 차이점에서 말했듯이, 컴퓨터 함수는 인자의 개수에 한계가 있다. 따라서 컴퓨터 함수에서는 한정된 인자들을 이용해 수학 함수를 구현해야 한다. 이를 위해 나는 일종의 집합과 (엄연히 다르지만) 비슷한 개념인 '배열'이란 개념을 활용할 것이다. 배열은 변수를 여러 개 이어붙인 것으로, 한 번에 여러 개의 값을 저장할 수 있다. 예를 들어,

x = [1, 2, 3, 4, 5]

이와 같은 배열을 만들어 주면 변수를 5개 만들 필요 없이 효율적으로 코드를 작성할 수 있다. 여기서 안의 값마다 번호가 있어 x[0]는 1, x[1]는 2, x[2]는 3. 과 같은 식으로 표현하게 된다. 컴퓨터로 수학의 함수를 구현할 때는 정의역의 범위가 크고, 촘촘할수록 좋으므로, 약 20,000개의 값을 갖는 배열을 만들어 줄 것이다.

x = [0.1, 0.2, 0.3, 0.4, 0.5,..., 1]

이와 같은 식으로 정의역을 선언해주면 치역은

y = [f(0.1), f(0.2), f(0.3),...,f(1)]

이와 같은 식으로 정해진다. 이렇게 하면 첫 번째 점은 (x[0], y[0]), 두 번째 점은 (x[1], y[1])이 된다.

②뼈대 코드

```
1.  from matplotlib import pyplot as plt  #그래프 라이브러리
2.  import numpy  #수학 관련 라이브러리
3.
4.
5.  def f(x):
6.      return 2*x+1
7.
8.
9.  x =[]  #정의역 집합 선언
10. y =[]  #치역 집합 선언
11. for i in numpy.arange(-10,10,0.001):  #정의역 : -10~10의 범위
12.     x.append(i)  #정의역 집합에 정의역 입력
13.     y.append(between(i))  #치역 집합에 함숫값 입력
```

```
14.   #그래프 관련 설정
15. plt.figure(figsize=(7,7))  #그래프 창 크기
16. plt.axis([-2,5,-2,10])  #그래프 숫자 범위
17. plt.grid(True)  #그래프 격자 그리기
18. plt.title('(x-2)^2-1')  #그래프 위에 제목 쓰기
19. plt.plot([-50,50],[0,0],color='black')  #x축 그리기
20. plt.plot([0,0],[-50,50],color='black')  #y축 그리기
21. plt.plot(x,y,color='blue')  #입력한 값들 그래프에 표현
22. plt.show()  #그래프 실행
```

기본적으로 함수의 그래프를 그리는 코드는 다음과 같다. 이 코드에 여러 함수를 추가하여 다양한 함수를 그릴 수 있다. (코드 옆의 숫자는 줄 번호를 나타낸다.)

1~2 : 이미 만들어진 함수들의 모음인 '라이브러리'를 불러온다.

5~6 : 직접 만든 함수 코드이다. 코드의 길이, 개수에 따라 이 부분의 크기가 더 커질 수도 있다.

9~10 : x 와 y 좌표가 들어갈 배열을 선언해준다.

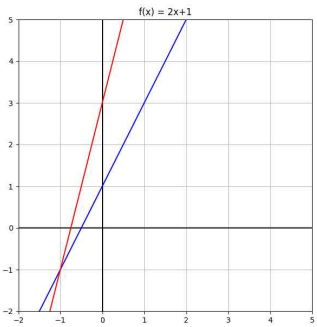
11~13 : x 와 y 좌표 배열에 -10.000~10.000의 값을 넣어주어 총 20,000개의 점을 생성한다.

15~ 22 : 그래프의 크기, 범위, 제목 등을 설정해주고, 그래프를 사용자가 볼 수 있도록 화면에 띄워준다.

③다양한 그래프

1. 다차 함수

1-1. 일차함수



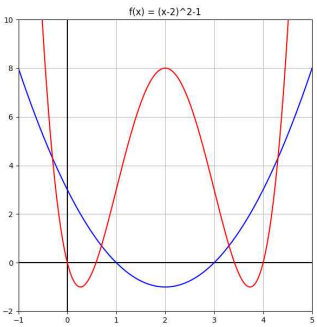
실현 방식 :

```
def f_1(x):
    return 2*x+1
```

파랑(일반) : x = x, y = f_1(x)

빨강(합성) : x = x, y = f_1(f_1(x))

1-2. 이차함수



실현 방식 :

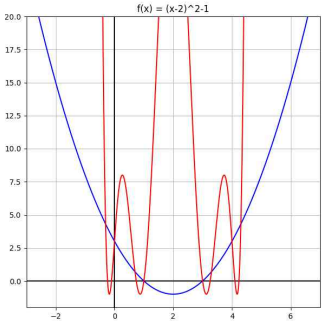
```
def f_2(x):
    return(x-2)**2-1
```

파랑(일반) : x = x, y = f_2(x)

빨강(합성) : x = x, y = f_2(f_2(x))

※ ** 기호는 제곱을 의미한다.

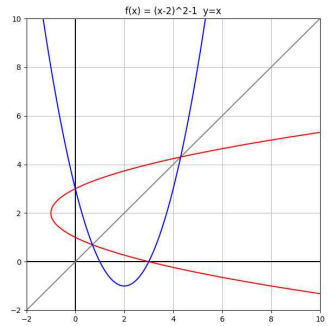
2**3=8, 4**0.5=2



이 그래프는 (x-2)^2-1의 그래프를 두 번 합성한 그래프이다.

실현 방식 :

x = x, y = f_2(f_2(f_2(x)))

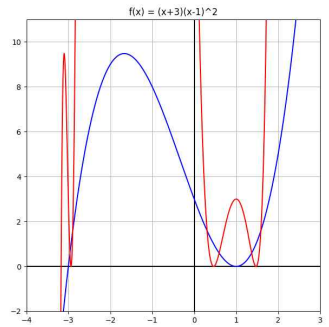


이 그래프는 이차함수를 y=x에 대해 대칭 이동한 그래프이다. 빨간 선의 그래프는 x값 하나에 y 값 두 개가 생기므로 함수식으로 표현할 수 없다. 하지만 아주 간단한 방법으로 위 그래프를 구현할 수 있는데 그것은 바로

x = f_2(x), y = x

이처럼 x와 y 좌표 배열에 반대로 값을 넣는 것이다.

1-3. 삼차함수



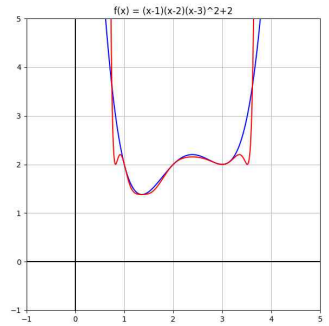
실현 방식 :

```
def f_3(x):
    return(x+3)*(x-1)**2
```

파랑(일반) : x = x, y = f_3(x)

빨강(합성) : x = x, y = f_3(f_3(x))

1-3. 사차함수

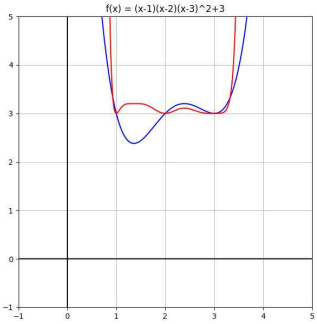


실현 방식 :

```
def f_4(x):
    return(x-1)*(x-2)*(x-3)**2+2
```

파랑(일반) : x = x, y = f_4(x)

빨강(합성) : x = x, y = f_4(f_4(x))

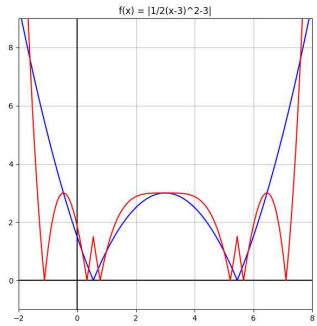



```
def f_4(x):
    return(x-1)*(x-2)*(x-3)**2+3
```

이 그래프는 사차함수 식의 상수항에 +1을 한 그래프의 함수 코드와 그래프 개형이다. y축으로 +1만큼 이동했기 때문에 파랑 그래프는 위쪽으로 이동한 모습이지만 한 번 합성을 한 빨강 그래프는 매우 다른 모습을 하고 있다.

2. 절댓값 함수

2-1. 이차함수 : y = |f(x)|



실현 방식 :

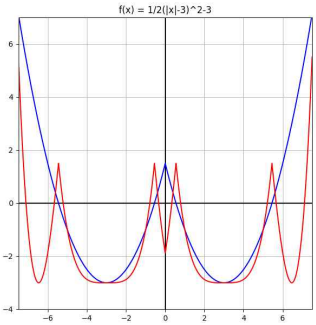
```
def f_abs(x):
    if x<0:
        x =-1*x
    returnx
```

절댓값을 구해주는 함수는 위와 같다. 위 코드를 말로 풀어보면, '만약 x값이 0보다 작다면, x에 -1을 곱해라' 로 나타낼 수 있다. 따라서 이차함수의 y=|f(x)|꼴의 개형을 그리기 위해서는 이차함수에 절댓값 함수를 씌워주면 된다.

파랑(일반) : x = x, y = f_abs(f_2(x))

빨강(합성) : x = x, y = f_abs(f_2(f_2(x)))

2-2. 이차함수 : y = f(|x|)



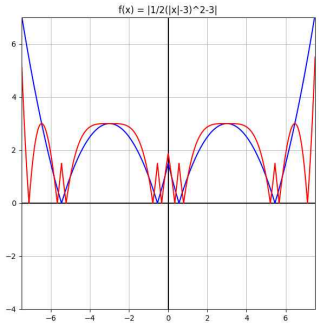
실현 방식 :

파랑(일반) : x = x, y = f_2(f_abs(x))

빨강(합성) : x = x, y = f_2(f_2(f_abs(x)))

y=f(|x|) 꼴에서는 y=|f(x)|와는 반대로 f_abs 함수를 f_2 함수 안에 넣어주면 된다.

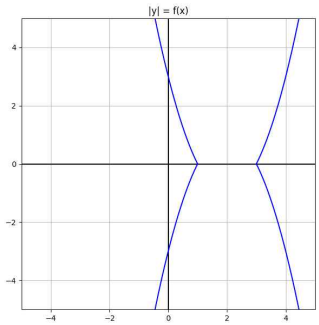
2-3. 이차함수 : y = |f(|x|)|



실현 방식 :

파랑(일반) : x = x, y = f_abs(f_2(f_abs(x)))
빨강(합성) : x = x, y = f_abs(f_2(f_2(f_abs(x))))

2-4. 이차함수 - |y| = f(x)

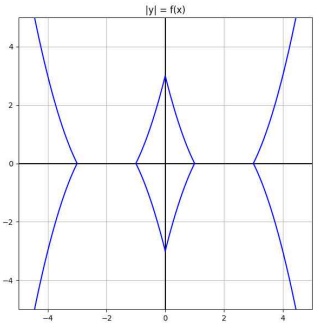


실현 방식 :

```
for i in numpy.arange(-10,10,0.001):
    x1.append(i)
    if i<=1 or i>=3:
        y1.append(f_2(i))
    else:
        y1.append(None)
for i in numpy.arange(-10,10,0.001):
    x2.append(i)
    if i<=1 or i>=3:
        y2.append(-1*f_2(i))
    else:
        y2.append(None)
```

|y| = f(x) 꼴의 그래프는 함수가 아니므로 함수식으로 나타낼 수 없다. 따라서 이러한 그래프를 그리기 위해서는 이차함수 그래프에서 y 값이 0보다 작은 부분은 자르고, 남은 부분을 x축에 대해 대칭시켜 주어야 한다. 이는 y 값이 음수일 때와 양수일 때의 두 상황을 각각 나누어 총 2번의 그래프를 그려야 한다.

2-5. 이차함수 - |y| = f(|x|)

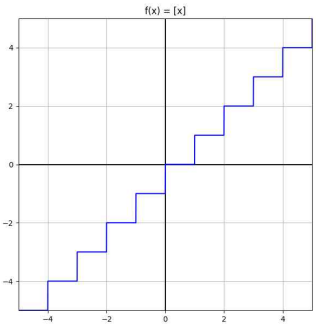



```
for i in numpy.arange(-10,10,0.001):
    if 0<=i<=1 ori>=3:
        x.append(i)
    else:
        x.append(None)
        y.append(f_quadratic(i))
for i in numpy.arange(-10,10,0.001):
    x2.append(i)
    if 0<=i <=1 or>=3:
        y2.append(-1*f_quadratic(i))
    else:
        y2.append(None)
for i in numpy.arange(-10,10,0.001):
    if 0 <=<=1 or>=3:
        x3.append(-1*i)
        y3.append(f_quadratic(i))
    else:
        x3.append(None)
        y3.append(None)
for i in numpy.arange(-10,10,0.001):
    if 0 <=<=1 or>=3:
        x4.append(-1*i)
        y4.append(-1*f_quadratic(i))
    else:
        x4.append(None)
        y4.append(None)
```

|y|=f(|x|) 꼴의 그래프는 |y|=f(x) 꼴의 그래프와 마찬가지로 함수가 아니다. 따라서 이차함수의 제1 사분면의 부분만 남기고 나머지는 자른 다음, x, y, 원점에 대해 대칭한 그래프를 그려주어야 한다. 이는 각 사분면에서의 상황을 나누어 총 4번 그래프를 그리는 것이다.

3. 가우스 기호 함수

3-1. 일차함수 - [f(x)]

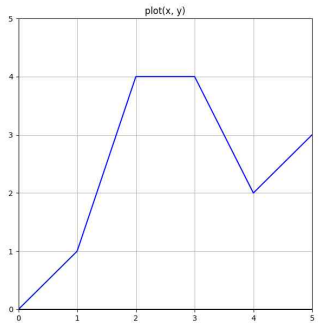


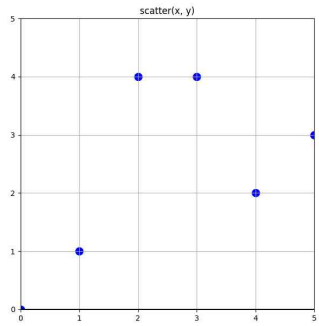
실현 방식 :

```
def f_g(x):
    return math.floor(x)
```

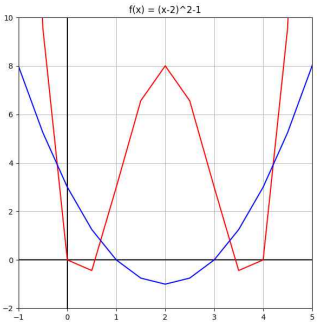
가우스 기호를 구현하기 위해 math 라이브러리에 있는 math.floor 함수를 이용하였다. 이 함수를 이용하면 해당 값보다 작거나 같은 정수를 구해준다.

문제점. 위의 그래프를 보면 일반적인 가우스 기호를 씌운 그래프와는 다르게 그래프가 모두 이어져 있는 것을 볼 수 있다. 이는 내가 그래프를 그리기 위해 사용한 함수와 관련이 있다. 내가 그래프를 그리기 위해 사용한 함수는 plt.plot(x, y)인데, 이 함수를 쓰면 찍은 모든 점을 잇게 된다. 이는 점과 점 사이의 공간을 메꿀 수 있다는 장점이 있지만, 가우스 기호처럼 끊어진 부분이 있어야 하는 그래프에는 맞지 않는다. 따라서 나는 plt.plot(x, y) 대신, plt.scatter(x, y)함수를 사용할 것이다. 이 함수는 기존의 함수와는 달리, 점을 찍어도 그것을 이어주지 않는다. 대신, 찍은 점의 개수가 부족할 경우 plt.plot(x, y) 함수에 비해 그래프가 끊어져 보인다는 단점이 있다. 따라서 plt.scatter(x, y) 함수를 쓸 때는 기존보다 더욱 많은 점을 찍어줄 필요가 있다.

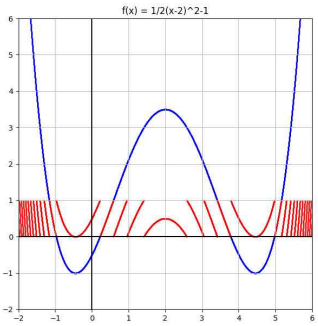


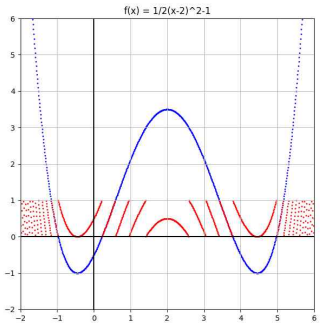


plt.plot(x, y) 함수와 plt.scatter(x, y) 함수를 이용해
(0, 0), (1, 1), (2, 4), (3, 4), (4, 2), (5, 3)을 찍어준 모습

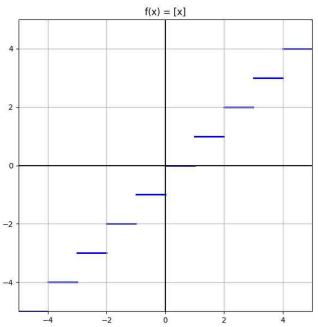


plt.plot(x, y)함수를 이용해 그래프를 그릴 때 충분한 점을 찍어주지 않으면, 이 그래프와 같이 각진 모습의 그래프가 그려진다.





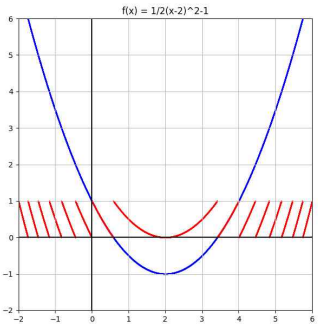
plt.scatter(x, y)함수를 이용해 각각 200,000개와 2,000개의 점을 찍었을 때의 모습
plt.scatter(x, y)함수를 이용해 그래프를 그릴 때 충분한 점을 찍어주지 않으면, 오른쪽의 그래프처럼 그래프가 점으로 끊어진 것처럼 보인다.



plt.scatter(x, y)함수를 이용해 다시 그려준 모습

x = x, y = f_g(f_1(x))

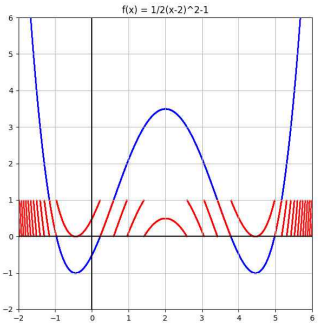
3-2-1. 이차함수 - f(x)-[f(x)]



x = x, y = f_2-f_g(f_2(x))

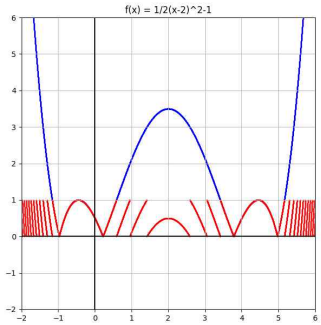
지금까지 만든 다차 함수, 절댓값 함수, 가우스 기호 함수들을 이용하면 다양한 함수들을 만들어볼 수 있다.

3-2-1. f(f(x))-[f(f(x))]



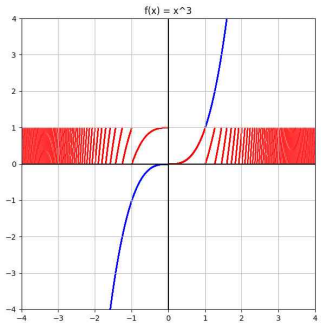
x = x, y = f_2(f_2(x))-f_g(f_2(f_2(x)))

3-2-3. |f(f(x))|-[|f(f(x))|]



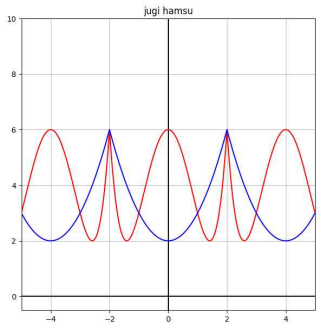
x = x,
y = f_abs(f_2(f_2(x)))-f_g(f_abs(f_2(f_2(x))))

3-3. 3차 함수 - f(x)-[f(x)]



f_3(x)-f_g(f_3(x))

4. 주기함수



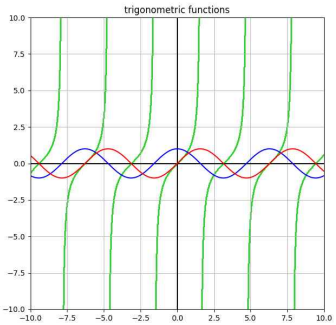
실현 방식 :

```
def f_cycle(x):
    while x<-2:
        x +=4
    while x>2:
        x -=4
    return x**2+2
```

파랑(일반) : x = x, y = f_cycle(x)
빨강(합성) : x = x, y = f_cycle(f_cycle(x))

함수 조건 :
(가) -2≤x≤2에서 f(x)=x^2+2이다.
(나) 모든 실수 x에 대하여 f(x)=f(x+4)이다.
이 함수의 y 값은 x±4n(n≥0인 자연수)이므로 함수 넣은 값을 -2~2의 값으로 변환한 후, 식에 대입하여 계산하면 된다. 위 코드를 한글로 풀이하면 'x<-2인 동안(x<-2가 아닐 때까지) x에 4를 더한다, x>2인 동안(x>2가 아닐 때까지) x에 4를 뺀다.'라고 말할 수 있다.

5. 삼각함수



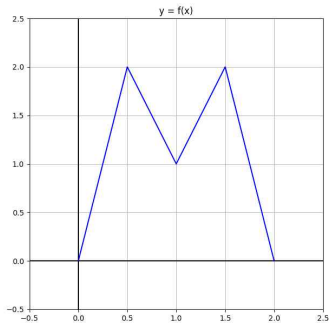
실현 방식 :
math 라이브러리의
math.sin(), math.cos(), math.tan()
함수를 이용하면 손쉽게 삼각비를 구할 수 있다.

함수를 이용하면 아직 배우지 않은 삼각함수 그래프를 그릴 수도 있다. sin, cos 함수는 계속 이어지므로 plt.plot(x, y)함수를 쓰고, tan 함수는 가우스 기호 함수처럼 끊어진 부분이 있으므로 plt.scatter(x, y)함수를 사용한다.

④문제 풀어보기

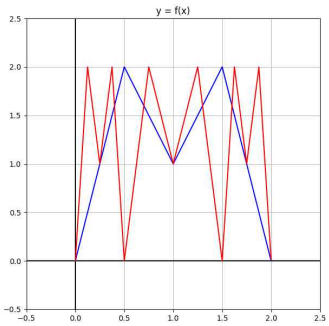
이로써 수(하)의 함수와 관련한 내용 대부분을 컴퓨터 함수로 구현해 보았다. 이를 이용하면 수학 문제들을 컴퓨터를 이용해 시각적으로 나타낼 수 있다.

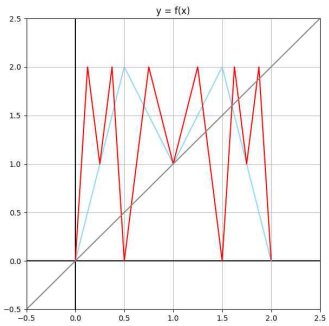
1. 함수 y=f(x)의 그래프가 그림과 같을 때, 방정식 f(f(x))=x의 서로 다른 실근의 개수는? (단 $0 \le x \le 2$)



① 5 ② 6 ③ 7
④ 8 ⑤ 9

```
def M(x):
    if 0<=x<0.5:
        return 4*x
    elif 0.5<=x<1:
        return -2*x+3
    elif 1<=x<1.5:
        return 2*x-1
    else:
        return -4*x+8
```





위 코드는 x의 구간별로 다른 함숫값을 반환하여 문제의 M자 그래프를 구현하는 그래프이다. 이 그래프를 M(M(x)) 형태로 합성하면 왼쪽 그림과 같이 3개의 M이 이어진 모양으로 그려진다. 그 위에 오른쪽 사진처럼 y=x 그래프를 그려주면 M(M(x)) 그래프와 y=x 그래프가 만나는 점의 개수가 9개라는 것을 알 수 있다.

∴ ⑤

2. 실수 전체에서 정의된 함수 f(x)가 다음 조건을 만족시킨다.

(가) 0≤x≤2에서 f(x)=-x^2+2x

(나) 모든 실수 x에 대하여 f(-x)=-f(x)

(다) 모든 실수 x에 대하여 f(x+4)=f(x)

양수 a와 실수 전체에서 정의된 함수

$$g(x)\begin{cases}(x-a)^2-a^2\ (x\geq 0)\\ a^2-(x+a)^2\ (x<0)\end{cases}$$

에 대하여 -a≤x≤2a에서 방정식 f(g(x))=1의 서로 다른 실근의 개수가 3이 되기 위한 양수 a의 범위를 구하시오.

```
from matplotlib import pyplot as plt
import numpy
import math

global a
a_square =5.5
a = math.sqrt(a_square)


def f(x):
    while x<-2:
        x += 4
    while x>2:
        x -= 4

    if 0<=x<=2:
        return -1*x**2+2*x
    if -2<=x<0:
        return x**2+2*x

```

```
22. def g(x):
23.     global a
24.     if x>=0:
25.         return (x-a)**2-a**2
26.     else:
27.         return a**2-(x+a)**2
28.
29.
30. x_f =[]; y_f =[]
31. x_g =[]; y_g =[]
32. x_fus =[]; y_fus =[]
33.
34. for i in numpy.arange(-10,10,0.0001):
35.     x_f.append(i)
36.     y_f.append(f(i))
37.
38. for i in numpy.arange(-10,10,0.0001):
39.     x_g.append(i)
40.     y_g.append(g(i))
41.
42. for i in numpy.arange(-10,10,0.0001):
43.     x_fus.append(i)
44.     if -a <=<=2*a:
45.         y_fus.append(f(g(i)))
46.     else:
47.         y_fus.append(None)
48.
49. plt.figure(figsize=(14,7))
50. plt.axis([-10,10,-5,5])
51. plt.grid(True)
52. plt.title('a^2=%.2lf'%a_square)
53. plt.plot([-10,10],[0,0],color='black')
54. plt.plot([0,0],[-10,10],color='black')
55. plt.plot([-10,10],[1,1],color='gray')
56. plt.plot([-a,-a],[-10,10],color='gray')
57. plt.plot([2*a,2*a],[-10,10],color='gray')
58. plt.plot(x_f,y_f,color='blue')
59. plt.plot(x_g,y_g,color="green")
60. plt.plot(x_fus,y_fus,color='red',linewidth=5)
61. plt.show()
```

1~3 : 그래프, 수학 관련 라이브러리를 불러온다.
5~7 : 이 코드에서 전역적으로 쓰일 변수인 a를 선언해주고 값을 저장한다.
10~19 : f(x) 함수를 구현한다. x의 값을 -2~2의 값으로 변환해준 뒤, 각 식에 대입한다.
22~27 : g(x) 함수를 구현한다. x값의 범위에 따라 각 식에 대입한다.
30~32 : f, g, 합성 함수의 함수값을 저장할 배열을 선언한다.
34~47 : 각 배열에 x값과 함숫값을 저장한다.
49~61 : 그래프관련 설정을 해주고 그래프를 화면에 띄운다.

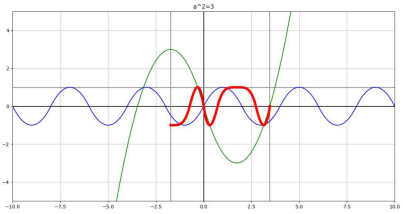


그림 29 a^2 = 3

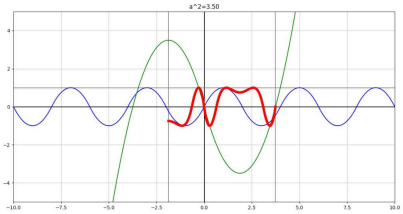


그림 30 a^2 = 3.5

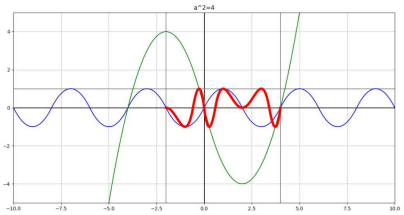


그림 31 a^2 = 4

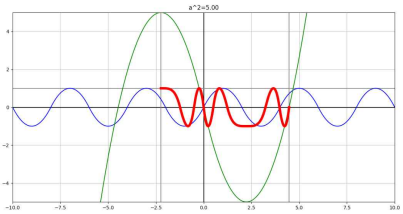


그림 32 a^2 = 5

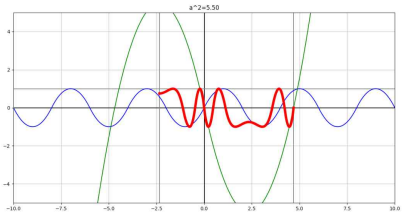


그림 33 a^2 = 5.5

이처럼, f(g(x))=1의 실근의 개수가 3이 되게 하는 a는 3<a^2<5일 때라는 것을 알 수 있다.
∴$\sqrt{3} < a < \sqrt{5}$

결론

컴퓨터 함수는 수학 함수와 달리 가질 수 있는 인자의 개수에 한계가 있어 모든 실수에 대해서 계산할 수는 없지만, 아주 많은 개수의 유리수들을 입력함으로써 실제 함수의 그래프와 거의 동일한 그래프를 그릴 수 있다. 컴퓨터의 함수를 이용하면 사람이 직접 계산하기 어려운 계산을 빠르게 하여 다양한 함수를 변형한 함수의 개형을 그려보고, 이를 통해 문제를 손쉽게 풀 수도 있다.