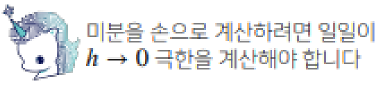
1. 미분

미분(differentiation)은 **변수의 움직임에 따른 함수값의 변화를 측정하기 위한 도구**이다. 최적화에서 가장 많이 사용하는 기법이다.

$$f'(x) = \lim_{h \to 0} \frac{f(x+h) - f(x)}{h}$$



$$f(x) = x^2 + 2x + 3$$
$$f'(x) = 2x + 2 \qquad \frac{f(x+h) - f(x)}{h} = 2x + 2 + h$$

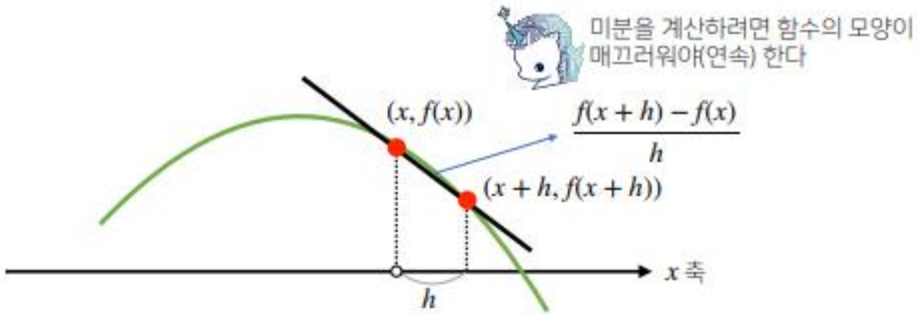
sympy라이브러리의 diff() 함수를 사용하면 미분을 계산할 수 있다.

```
import sympy as sym
from sympy.abc import x

sym.diff(sym.poly(x**2 + 2*x + 3), x)
```

```
Poly(2*x + 2, x, domain='ZZ')
```

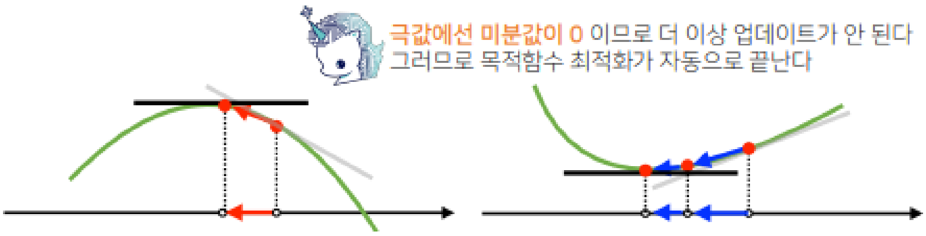
함수f가 있고 두 개의 점이 주어졌다고 하자. 미분은 이 주어진 점 (x, f(x))에서의 접선의 기울기를 구한다. 아래 그래프에서 h를 0으로 보내면 (x, f(x))에서의 접선의 기울기로 수렴하기 때문이다.



이때 한 점에서의 접선의 기울기를 알면 **어느 방향으로 점을 움직여야 함수 값이 증가하는지/감소하는지 알 수 있다**. 2차원에서는 어느 방향으로 움직여야 함수 값이 증가하는지 감소하는지 알기 쉽지만, 5차원, 100차원 등의 **고차원에서는 어느 방향으로 움직여야 하는지 알기 힘들다**. 이때 미분을 사용하면 함수의 최적화가 쉬워진다.. 그럼 어떻게 미분을 이용할 수 있을까?

만약 함수값을 증가시키고 싶다면 미분값을 더하고, 감소시키고 싶다면 미분값을 빼면 된다. 이때 미분값을 더하면 경사상승법(gradient axcent)이라 하며 함수의 극대값의 위치를 구할 때 사용한다. 미분값을 빼면 경사하강법(gradient descent)이라 하며 함수의 극소값의 위치를 구할 때 사용한다. 경사상승/경사하강 방법은 극값에 도달하면 움직임을 멈춘다.

* 최적화: 특정의 집합 위에서 정의된 실수값, 함수, 정수에 대해 그 값이 최대나 최소가 되는 상태를 해석하는 문제



* 미분 참고

미분은 한 점에서의 기울기를 의미한다.

미분은 $f(x)$라는 함수위의 $(a, f(a))$에서의 한 점에서의 기울기

보통 **f'(a)**라고 쓰고 '에이 프라임 에이'라고 읽는다. 아래는 모두 동일한 의미이다.

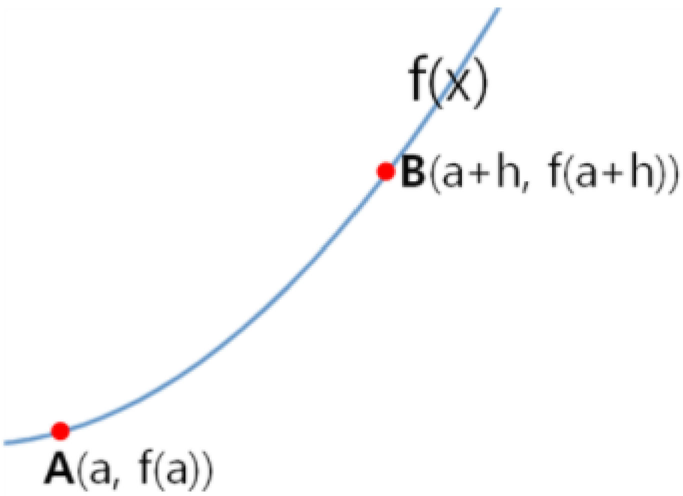
$f'(a)$ = a 라는 점에서의 기울기

= a 라는 점에서의 접선의 기울기

= a 라는 점에서의 미분값

= a 라는 점에서의 미분계수

근데 기울기라는 것은 함수위에 두 점이 있을 때, 두 점을 잇는 직선의 기울기를 구하는 것이다. 그래서 사실 미분도 두 점사이의 기울기를 의미한다. 단, **두 점 사이의 거리가 너무 가까워서 한 점에서의 기울기로 보는 것이다.**

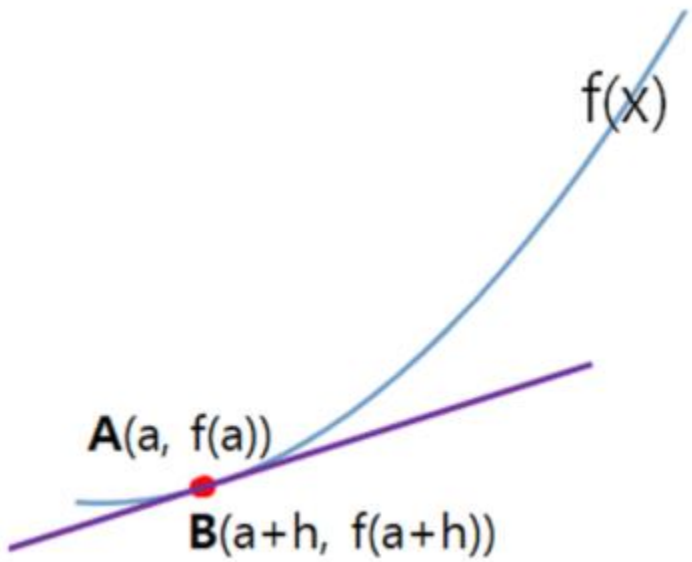


$$\text{기울기} = \frac{y\text{증가량}}{x\text{증가량}}$$

위 함수에서 두 개의 점이 있다. 두 점 사이의 기울기 공식은 아래와 같다.

$$\text{기울기} = \frac{y\text{증가량}}{x\text{증가량}} = \frac{f(a+h)-f(a)}{(a+h)-a}$$

이때, 미분은 x 증가량이 거의 0으로 갈 때의 기울기를 말한다. 즉, B점이 A점에 매우 가깝다는 의미이다. 따라서 미분의 정의는 x=a라는 한 점에서의 접선의 기울기가 된다.



2. 경사하강법: 알고리즘

```
Input: gradient, init, lr, eps, Output: var
-------------------------------------------
# gradient: 미분을 계산하는 함수
# init: 시작점, lr: 학습률, eps: 알고리즘 종료조건

var = init
grad = gradient(var)
while(abs(grad) > eps):
    var = var - lr * grad
    grad = gradient(var)
```

경사하강법이나 경사상승법은 미분값이 0이 되면 update가 더 이상 일어나지 않게 되지만 컴퓨터로 계산할 때 미분이 정확히 0이 되는 것은 거의 불가능하다. 따라서 eps보다 작을 때 종료하는 조건이 필요하다. lr은 학습률로 미분을 통해서 update하는 속도를 조절할 수 있다.

```python
def func(val):
    fun = sym.poly(x**2 + 2*x + 3)
    return fun.subs(x, val), fun

def func_gradient(fun, val):
    _, function = fun(val)
    diff = sym.diff(function, x)
    return diff.subs(x, val), diff

def gradient_descent(fun, init_point, lr_rate=1e-2, epsilon=1e-5):
    cnt=0
    val = init_point
    diff, _ = func_gradient(fun, init_point)
    while np.abs(diff) > epsilon:
        val = val - lr_rate*diff
        diff, _ = func_gradient(fun, val)
        cnt+=1

    print("함수: {}, 연산횟수: {}, 최소점: ({}, {})".format(fun(val)[1], cnt, val, fun(val)[0]))

gradient_descent(fun=func, init_point=np.random.uniform(-2,2))
```

```
함수: Poly(x**2 + 2*x + 3, x, domain='ZZ'), 연산횟수: 636, 최소점: (-0.999995047967832, 2.00000000002452)
```

3. 변수가 벡터인 경우(다변수 함수인 경우)_편미분

벡터가 입력인 다변수 함수의 경우 **'편미분(partial differentiation)'**을 사용한다.

$$\partial_{x_i} f(\mathbf{x}) = \lim_{h \to 0} \frac{f(\mathbf{x} + h\mathbf{e}_i) - f(\mathbf{x})}{h}$$

$$f(x, y) = x^2 + 2xy + 3 + \cos(x + 2y)$$

$$\partial_x f(x, y) = 2x + 2y - \sin(x + 2y)$$

미분과 동일하게 sympy.diff() 함수로 구해볼 수 있다.

```python
import sympy as sym
from sympy.abc import x, y

sym.diff(sym.poly(x**2 + 2*x*y + 3) + sym.cos(x + 2*y), x)
```

```
2*x + 2*y - sin(x + 2*y)
```

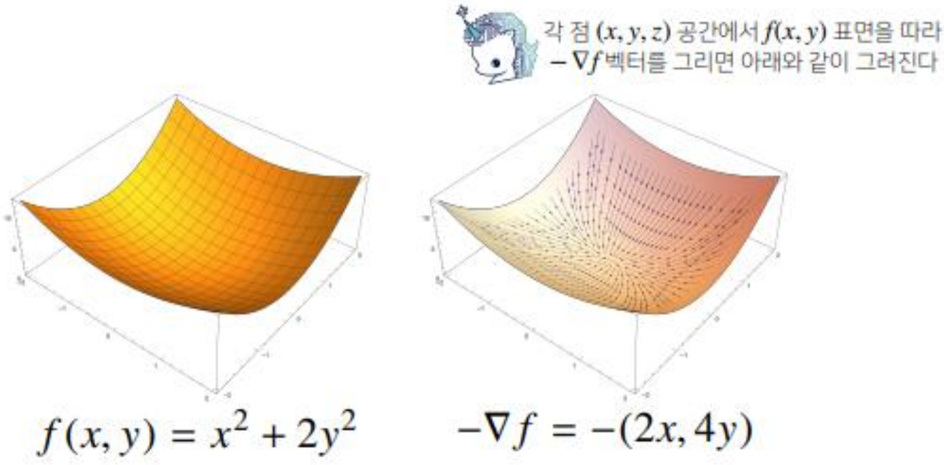
이때 각 변수 별로 편미분을 계산한 **그레디언트(gradient) 벡터**를 이용해 n차원 공간에서 경사하강/경사상승법에 사용할 수 있다.

$$\nabla f = (\partial_{x_1} f, \partial_{x_2} f, \cdots, \partial_{x_d} f)$$

요 기호를 nabla 라 부릅니다

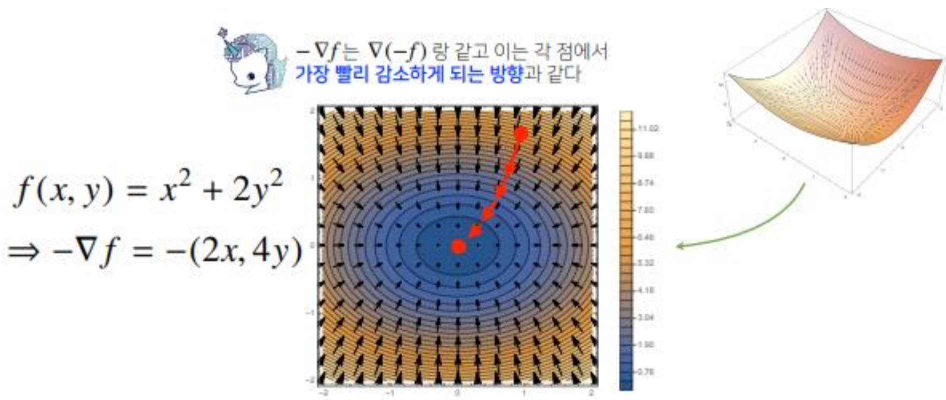
앞서 사용한 미분값인 $f'(x)$ 대신 벡터 ∇f를 사용하여 변수 $\mathbf{x} = (x_1, \ldots, x_d)$ 를 동시에 업데이트 가능합니다

왼쪽의 그림은 f(x,y)의 그림이다. 이때 f(x,y) 표면에 -그레디언트 벡터를 그리면 오른쪽과 같이 극소점으로 향하는 화살표들의 움직임으로 볼 수 있다.



등고선을 그려보자.

그냥 **그레디언트 벡터를 그리면 원점에서 가장 빨리 증가하는 방향으로 벡터가 표시**된다. 단, **-그레디언트 벡터를 그리면 임의의 점에서 출발해 최소점에 가장 빨리 감소하는 방향으로 움직이게 된다**.



경사하강법 알고리즘은 앞의 알고리즘과 비슷하다. 이전에는 미분값의 절대값을 계산했지만 벡터이므로 norm을 이용해야 한다.

```
Input: gradient, init, lr, eps, Output: var
```

```
# gradient: 그레디언트 벡터를 계산하는 함수
# init: 시작점, lr: 학습률, eps: 알고리즘 종료조건

var = init
grad = gradient(var)
while(norm(grad) > eps):
    var = var - lr * grad
    grad = gradient(var)
```

경사하강법 알고리즘은 그대로 적용된다. 그러나 벡터는
절대값 대신 노름(norm)을 계산해서 종료조건을 설정한다

```python
# Multivariate Gradient Descent
def eval_(fun, val):
    val_x, val_y = val
    fun_eval = fun.subs(x, val_x).subs(y, val_y)
    return fun_eval

def func_multi(val):
    x_, y_ = val
    func = sym.poly(x**2 + 2*y**2)
    return eval_(func, [x_, y_]), func

def func_gradient(fun, val):
    x_, y_ = val
    _, function = fun(val)
    diff_x = sym.diff(function, x)
    diff_y = sym.diff(function, y)
    grad_vec = np.array([eval_(diff_x, [x_, y_]), eval_(diff_y, [x_, y_])], dtype=float)
    return grad_vec, [diff_x, diff_y]

def gradient_descent(fun, init_point, lr_rate=1e-2, epsilon=1e-5):
    cnt=0
    val = init_point
    diff, _ = func_gradient(fun, val)
    while np.linalg.norm(diff) > epsilon:
        val = val - lr_rate*diff
        diff, _ = func_gradient(fun, val)
        cnt+=1

    print("함수: {}, 연산횟수: {}, 최소점: ({}, {})".format(fun(val)[1], cnt, val, fun(val)[0]))

pt=[np.random.uniform(-2, 2), np.random.uniform(-2, 2)]
gradient_descent(fun=func_multi, init_point=pt)
```

```
함수: Poly(x**2 + 2*y**2, x, y, domain='ZZ'), 연산횟수: 606, 최소점: ([4.95901570e-06 2.88641061e-11], 2.45918366929856E-11)
```