최적화(optimization)란 함수의 최대값, 최소값을 구하는 문제로 보통 제약식이 없는 경우와 제약식이 있는 경우로 표현된다.

먼저 제약식이 없는 문제들의 해를 찾는 방법을 간단히 살펴보자.

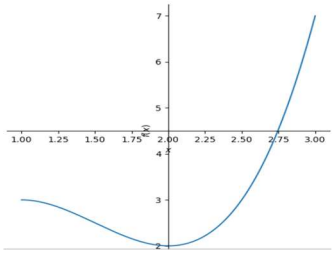
제약식이 없는 문제의 최적해 구하기

문제1. 아래 함수의 최대값과 최소값을 구하여라.

$$f(x) = 2x^3 - 9x^2 + 12x - 2 \ (1 \le x \le 3)$$

'sympy'를 이용해서 먼저 그래프를 그려보자.

```
from sympy import plot,symbols

x=symbols('x',real=True)
f=2*x**3-9*x**2+12*x-2
plot(f,(x,1,3),size=(5,5))
```



위 그래프에서 보면 극값에서 최소값을 갖게 되며, f(3)에서 최대값을 갖는다는 것을 쉽게 알 수 있다. 먼저 1차미분한 함수를 구하면 아래와 같다.

```
# 도함수 구하기
from sympy import diff,pprint,solve
fprime=diff(f,x)
fprime
```

$6x^2 - 18x + 12$

```
# 2차 도함수
f2prime=diff(f,x,2)
f2prime
```

$6 \cdot (2x - 3)$

먼저 1차도함수가 0이 되는 값들을 구하여, 극값을 구하면 아래와 같다.

```
# 극값 구하기
cps=solve(fprime,x) # fprime=0일때의 해를 cps에 저장

# cps에 있는 값들을 f에 대입하여 값을 저장
[f.subs(x,point) for point in cps]
```

```
[3, 2]
```

위 극값을 바탕으로 범위의 양 끝값과 비교해 이 중에서 최대값, 최소값을 선정하면 된다.

```
# 최대값과 최소값 구하기
import numpy as np
fvalues=[f.subs(x,1),f.subs(x,3),3,2]
print(f'최대값: {np.max(fvalues)}')
print(f'최소값: {np.min(fvalues)}')
```

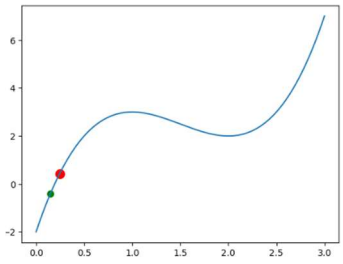
```
최대값: 7
최소값: 2
```

위에서 극값을 구할 때, solve()라는 방정식의 해찾기를 이용했는데, 조금 다른 각도에서 극값을 찾을 수도 있다.

곡선 위 임의의 점 x 에서의 기울기를 Δx라고 할 때, $x+\alpha\Delta x$ 와 같이 업데이트 한다고 하자. 여기서 α 는 양수의 값으로 기울기의 반영비율을 말한다. 기울기 $\Delta x>0$ 이라면, 곡선은 일시적으로 상승하는 구간에 있는 상태이며, $x+\alpha\Delta x$는 x의 오른쪽 점이 되기 때문에 그래프의 더 높은 점으로 이동한다는 의미이다. 이것을 간단한 코딩에 의해 구현해 보면 아래와 같다.

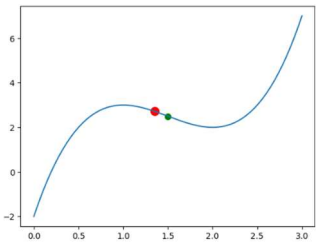
```
from sympy import symbols
import numpy as np
import matplotlib.pyplot as plt

x=symbols('x')
yprime=6*x**2-18*x+12
y=2*x**3-9*x**2+12*x-2

xps=np.linspace(0,3,1000)
yps=2*xps**3-9*xps**2+12*xps-2
```

```
alpha=0.01 # 기울기 가중치
Starting_Point=(xps[50],yps[50])
x0=Starting_Point[0]
x_new=x0+alpha*yprime.subs(x,x0)
y_new=y.subs(x,x_new)
Final_Point=(x_new,y_new)

plt.plot(xps,yps)
plt.scatter(*Starting_Point,color='g',s=50,label='Start')
plt.scatter(*Final_Point,color='r',s=100,label='Final')
```



반대로, $\Delta x<0$ 이라면, 하락하는 구간에 있는 상태이며, $x+\alpha\Delta x$는 x의 왼쪽 점이 되기 때문에 그래프의 더 높은 점으로 이동한다는 의미가 된다.

```
alpha=0.1 # 기울기 가중치
Starting_Point=(xps[500],yps[500])
x0=Starting_Point[0]
x_new=x0+alpha*yprime.subs(x,x0)
y_new=y.subs(x,x_new)
Final_Point=(x_new,y_new)

plt.plot(xps,yps)
plt.scatter(*Starting_Point,color='g',s=50,label='Start')
plt.scatter(*Final_Point,color='r',s=100,label='Final')
```

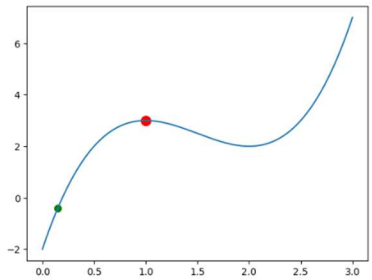


위에서 설명한 부분을 N번 반복 수행하면 우리가 원하는 극대값을 찾을 수 있다. 아래 코드를 참고하여라.

```
# 극대값으로 이동시키기
from sympy import symbols
import numpy as np
import matplotlib.pyplot as plt

x=symbols('x')
yprime=6*x**2-18*x+12
y=2*x**3-9*x**2+12*x-2

xps=np.linspace(0,3,1000)
yps=2*xps**3-9*xps**2+12*xps-2

alpha=0.01 # 기울기 가중치
Starting_Point=(xps[50],yps[50])
x0=Starting_Point[0]

N=10000 # 반복회수
for _ in range(N):
  dx=yprime.subs(x,x0)
  x0=x0+alpha*dx # 좌표 업데이트
```

```
y_new=y.subs(x,x0)
Final_Point=(x0,y_new)

print(f'{N} 반복 후 최종 기울기: {dx}')
print(f'{x0}일 때 극대값 {y_new}을 가진다')

plt.plot(xps,yps)
plt.scatter(*Starting_Point,color='g',s=50,label='Start')
plt.scatter(*Final_Point,color='r',s=100,label='Final')
```

```
10000 반복 후 최종 기울기: 5.32907051820075E-15
0.9999999999999999일 때 극대값 3.0000000000000을 가진다
```

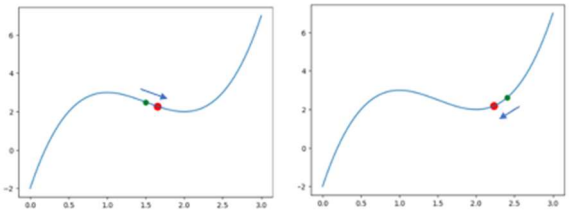


이와 유사하게 극소값을 구하는 것도 아래와 같이 수행하면 된다.

곡선 위 임의의 점 x 에서의 기울기: Δx

⇩

새로운 점 $x - \alpha\Delta x$: 경사를 내려가는 방향



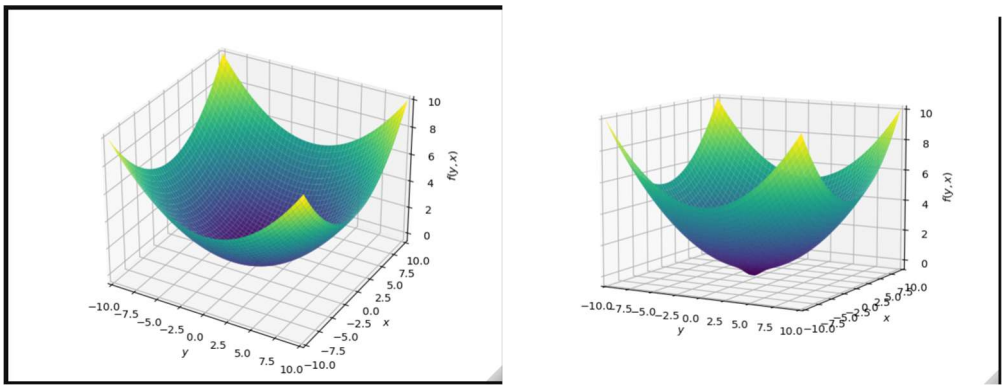
위를 요약해서 다시 설명하자면, 함수 기울기의 (+) 방향으로 계속 이동하면 그래프의 정상 부분인 극대값으로 이동하고, 함수 기울기의 (-) 방향으로 계속 이동하면 그래프의 골짜기 부분인 극소값으로 이동한다는 것이다. 함수 기울기의 (+) 방향으로 계속 이동하면서 극대값을 찾는 것을 경사상승법(Gradient Ascent)이라고 하고, (-) 방향으로 계속 점을 이동하면서 극소값을 찾아가는 방법을 경사하강법(Gradient Descent)이라고 한다.

문제2. 아래 다변량 함수의 그래프를 그리고, 최소값을 구하여 보아라.

$$f(x,y) = xe^{-(x^2+y^2)} + (x^2+y^2)/20$$

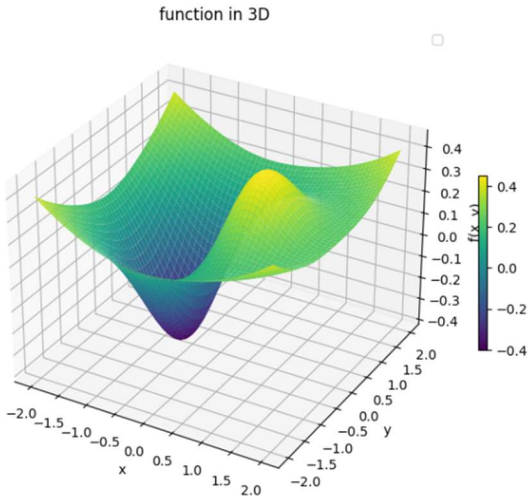
먼저 그래프를 그려보도록 하자. 3차원 상에서 그래프를 그린 후, 그래프를 회전시키다 보면 아래 그림과 같이 원점 근처에 최소값이 존재하는 것을 확인할 수 있다.

```
%matplotlib widget

from sympy import *
x,y=symbols('x y')
plotting.plot3d(x*exp(-x**2-y**2)+(x**2+y**2)/20)
```



'sympy' 이외에 'numpy', 'matplotlib', 'mpl_toolkits' 등을 사용해서 그리면 아래와 같이 사용할 수 있다.

```python
# 그래프 상의 점의 움직임 표현을 위해 numpy, matplotlib 라이브러리 사용
import numpy as np
import matplotlib.pyplot as plt
from mpl_toolkits.mplot3d import Axes3D

# 함수 재정의
def fn(x, y):
    return x+np.exp(-x**2-y**2)+(x**2+y**2)/20

# 3차원 그래프 생성
fig = plt.figure(figsize=(8, 8)) # 특정 크기의 그림판 생성
ax = fig.add_subplot(111, projection='3d') # 3차원 그래프 설정

# Rosenbrock 함수의 등고선 플로팅
x = np.linspace(-2, 2, 400) # -2,2까지의 점을 400개로 분해해 줌
y = np.linspace(-2, 2, 400) # -2,2까지의 점을 400개로 분해해 줌
X, Y = np.meshgrid(x, y) # (x,y) 좌표 순서쌍 생성
Z = fn(X, Y) # 함수값을 출력해서 Z에 저장
# cmap은 colormap을 지정해줌,Z 값에 따른 색깔 변화
# cmap의 값은  'viridis', 'plasma', 'inferno', 'magma' 등이 있음
surf=ax.plot_surface(X, Y, Z,cmap='viridis')

# colorbar 추가 (색과 숫자간의 관계 설명)
# shrink는 0과 1사이의 값으로 표시되는 bar의 크기
# aspect는 세로와 가로의 비율로 값이 클수록 가로방향으로 작아진다.
fig.colorbar(surf, ax=ax, shrink=0.3, aspect=20)

# 그래프 설정
ax.set_xlabel('x') # x축 라벨
ax.set_ylabel('y') # y축 라벨
ax.set_zlabel('f(x, y)') # z축 라벨
ax.set_title('Rosenbrock in 3D') # 그림 제목
ax.legend() # 보여주기
```



변수가 1개가 아닌 함수의 미분은 어떻게 할까? 이 경우는 아래와 같이 다른 문자를 모두 상수라고 생각하고 변수가 1개인 경우와 같이 미분을 하면 된다. 이를 편미분(partial differentiation)이라고 한다. 변수 x에 대한 편미분 fx는 x에 대한 함수 f의 변화율을 의미한다고 생각하면 된다.

예를 들어, $f(x,y) = (1-x)^2 + 100(y-x^2)^2$ 이라는 함수가 주어졌을 때, 각 변수에 대한 편미분 값을 구하면 아래와 같다.

```python
from sympy import symbols

# x,y라는 문자를 실수로 정의
x,y=symbols('x y',real=True)

f=(1-x)**2+100*(y-x**2)**2 # 함수 정의
f
```

$$(1-x)^2 + 100(-x^2+y)^2$$

```python
# 미분값 구하기
from sympy import diff
fx=diff(f,x) # y를 상수로 생각하고 x에 대해 미분
fx
```

$$-400x\left(-x^2+y\right) + 2x - 2$$

```python
fy=diff(f,y) # x를 상수로 생각하고 y에 대해 미분
fy
```

$$-200x^2 + 200y$$

각 좌표에서의 편미분 값을 하나로 모아서 벡터로 표기한 것을 기울기, 경도 혹은 그래디언트(gradient)라고 부른다.


```
# 그래디언트 벡터의 계산
from sympy import Matrix
Gf=Matrix(1,2,[fx,fy]) # 1,2 행렬, 즉 벡터로 표현
Gf
```

$$[-400x\left(-x^2+y\right)+2x-2 \quad -200x^2+200y]$$

그래디언트는 함수의 각 점에서 함수 증가 방향으로의 기울기 벡터라고 해석될 수 있고, 이는 경사의 의미를 갖는다고 한다. 즉, 그래디언트는 다변량 함수의 특정 점에서 함수가 커지는 방향으로 기울기가 가장 큰 방향을 의미한다. 앞 문제의 1변량 함수의 극값을 구하는 문제에서와 유사하게, 다변량 함수의 최대값을 구할 때는 그래디언트 방향으로 이동하는게 좋은데 이를 경사상승법이라고 하며, 함수의 최소값을 구할 때는 그래디언트의 역방향으로 이동해야 해서, 경사하강법(Gradient Descent)이라고 한다.

먼저 위 문제에서의 그래디언트를 계산해 보자.

```
from sympy import symbols,diff,exp
x,y=symbols('x y')
f=x*exp(-x**2-y**2)+(x**2+y**2)/20
gradient_x=diff(f,x)
gradient_y=diff(f,y)
gradient_x
```

$$-2x^2e^{-x^2-y^2}+\frac{x}{10}+e^{-x^2-y^2}$$

```
gradient_y
```

$$-2xye^{-x^2-y^2}+\frac{y}{10}$$

아래는 위 결과를 바탕으로 starting_point 에서 그래디언트를 구하고, 그래디언트의 방향의 일부 즉, 학습율을 반영하여 어떻게 점을 이동시키는지를 코딩한 것이다. 그래디언트는 벡터방향이기 때문에 적절한 크기를 지정해 주어야 하는데, 아래에서는 alpha라는 변수로 이를 정하였다. 딥러닝에서는 보통 이를 학습율 패리미터라고 부른다. 그래디언트 방향으로 한번 이동했더니 함수의 값이 -0.0099에서 0.191로 상승하였다. 이를 경사상승법이라고 한다.

```python
# 경사 상승법 이해하기
import numpy as np
starting_point=(-1.3,0.8) # 초기 (x,y)값 지정

# 해당 점에서 gradient 계산하기
x,y=starting_point[0],starting_point[1] # (x,y)값에 초기값 대입
gradient_x = -2*x**2*np.exp(-x**2-y**2)+x/10*np.exp(-x**2-y**2)
gradient_y =-2*x*y*np.exp(-x**2-y**2)+y/10
Gf=(gradient_x,gradient_y)
print(f'그래디언트 벡터: {Gf}')

# 원래 점에서 그래디언트 벡터 만큼 더해서 새로운 점 만들기
# alpha는 그래디언트의 반영 정도를 나타내며 학습율로 정의
alpha=1.2
x=x+alpha*gradient_x # x값 업데이트
y=y+alpha*gradient_y # x값 업데이트
next_point=(x,y)
print(f'이동된 점의 위치: {next_point}')
```

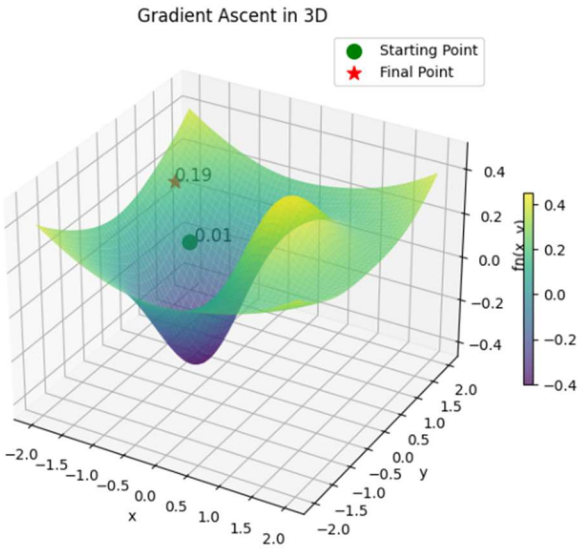
```
그래디언트 벡터: (-0.361563878073088, 0.28237515394622814)
이동된 점의 위치: (-1.7338766536877057, 1.1388501847354737)
```

```python
# 함수값 출력하기

print('처음 위치의 함수값:',fn(starting_point[0],starting_point[1]))
print('처음 위치의 함수값:',fn(next_point[0],next_point[1]))
```

```
처음 위치의 함수값: -0.009984471216392 58
처음 위치의 함수값: 0.1917168879401624
```

위에서 변경되기 전후의 점을 원래 그래프상에서 찍어보면 좀더 경사상승법에 대한 이해가 쉽다. 아래 코드를 확인해 보도록 하자.

```python
# 경사상승법에 의해 이동한 점 표시하기

import numpy as np
import matplotlib.pyplot as plt
from mpl_toolkits.mplot3d import Axes3D


# 3차원 그래프 생성
fig = plt.figure(figsize=(8, 8)) # 특정 크기의 그림판 생성
ax = fig.add_subplot(111, projection='3d') # 3차원 그래프 설정

# 함수의 등고선 플로팅
x = np.linspace(-2, 2, 400) # -2,2까지의 점을 400개로 분해해 줌
y = np.linspace(-2, 2, 400) # -2,2까지의 점을 400개로 분해해 줌
X, Y = np.meshgrid(x, y) # (x,y) 좌표 순서쌍 생성
Z = fn(X, Y) # 함수값을 출력해서 Z에 저장
# alpha를 사용하여 반투명으로 표현
surf=ax.plot_surface(X, Y, Z,cmap='viridis',alpha=0.7)
fig.colorbar(surf, ax=ax, shrink=0.3, aspect=20) # 컬러바 표현

# 출발점과 최종 도달 지점 표시
ax.scatter(starting_point[0], starting_point[1], fn(starting_point[0],starting_point[1]),
           color='g', s=100, marker='o', label='Starting Point')
ax.scatter(next_point[0], next_point[1], fn(next_point[0], next_point[1]),
           color='r', s=100, marker='*', label='Final Point')

# 각 점에 대한 z 값 텍스트 추가
xi,yi,zi=starting_point[0], starting_point[1], fn(*starting_point)
ax.text(xi,yi,zi, f'{zi:.2f}', fontsize=12, color='black')
xi,yi,zi=next_point[0], next_point[1], fn(next_point[0], next_point[1])
ax.text(xi,yi,zi, f'{zi:.2f}', fontsize=12, color='black')

# 그래프 설정
ax.set_xlabel('x')
ax.set_ylabel('y')
ax.set_zlabel('fn(x, y)')
ax.set_title('Gradient Ascent in 3D')
ax.legend()

# 그래프 표시
plt.show()
```



다음은 경사하강법에 의해 다음 점으로 업데이트 하는 방식을 이해해 보도록 하자. 경사하강법은 경사상승법에서 다음 점을 찾을 때, 그래디언트의 부호만 반대로 하면 된다. 아래 그래프를 참조하면, 초기값에서 다음 값으로 업데이트 되었을 때, 함수값은 0.3096에서 -0.1976로 작아졌음을 확인할 수 있다.

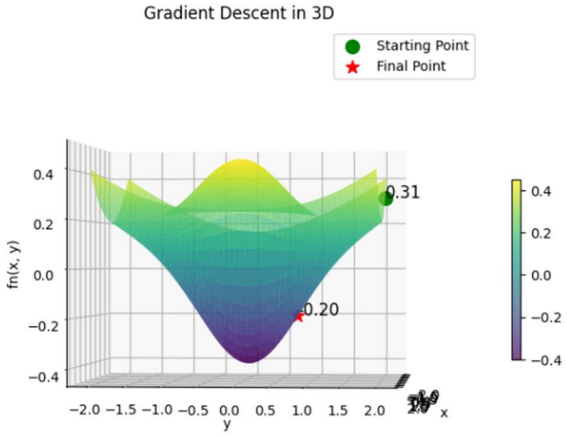
```python
# 경사 하강법 이해하기
import numpy as np
starting_point=(-1.5,2.0) # 초기 (x,y)값 지정

# 해당 점에서 gradient 계산하기
x,y=starting_point[0],starting_point[1] # (x,y)값에 초기값 대입
gradient_x = -2*x**2*np.exp(-x**2-y**2)+x/10+np.exp(-x**2-y**2)
gradient_y =-2*x*y*np.exp(-x**2-y**2)+y/10
Gf=(gradient_x,gradient_y)
print(f'그래디언트 벡터: {Gf}')

# 원래 점에서 그래디언트 벡터 만큼 더해서 새로운 점 만들기
# alpha는 그래디언트의 반영 정도를 나타내면 학습율로 정의
alpha=6
x=x-alpha*gradient_x # x값 업데이트
y=y-alpha*gradient_y # x값 업데이트
next_point=(x,y)
print(f'이동된 점의 위치: {next_point}')
```

```
그래디언트 벡터: (-0.15675658947679696, 0.21158272481736626)
이동된 점의 위치: (-0.5594604631392182, 0.7305036510958025)
```

```python
# 함수값 출력하기

print('처음 위치의 함수값:',fn(starting_point[0],starting_point[1]))
print('처음 위치의 함수값:',fn(next_point[0],next_point[1]))
```

```
처음 위치의 함수값: 0.3096043187956584
처음 위치의 함수값: -0.19759699329544492
```

위의 결과도 그래프상에서 어떻게 이동하는지 살펴보면 아래와 같다.



위에서 설명한 경사하강법에 의해, 주어진 점을 여러 번 업데이트 해 나가는 과정을 위해 함수를 작성하면 아래와 같다. 아래에서 정의한 gradient_descent()함수는 초기값(starting_point), 학습율(learning_rate)과 반복회수(num_iterations)을 입력값으로 하는 경사하강법 진행 함수이다. 결국, 이 함수는 초기값이 주어지면 학습율에 따라 반복회수 만큼 새로운 점을 만들어서 이 모든 점을 리턴해 주는 함수라고 보면 된다. 예를 들어, gradient_descent([1,1],0.2,10)과 같이 주어지면, (1,1)을 시작점으로 하여, 해당점에서의 그래디언트를 계산하고, 0.2만큼을 반영해 업데이트한 점들을 10개 추가로 만들게 되고, 최종적으로 11개의 점을 리턴해 주게 된다.

```python
# 함수 정의
def f(x, y):
    return x*np.exp(-x**2-y**2) + (x**2+y**2)/20

# 경사 하강법 함수 정의
# learning_rate는 그래디언트의 크기의 일부를 반영하는 역할
# num_iterations은 초기값에서 몇 개의 점을 만들지 정하는 역할
def gradient_descent(starting_point, learning_rate, num_iterations):
    points = [starting_point] # 초기값
    x, y = starting_point

    for _ in range(num_iterations): #
        gradient_x = -2*x**2*np.exp(-x**2-y**2)+x/10+np.exp(-x**2-y**2)
        gradient_y =-2*x*y*np.exp(-x**2-y**2)+y/10

        x = x - learning_rate * gradient_x # 좌표 업데이트
        y = y - learning_rate * gradient_y # 좌표 업데이트

        points.append((x, y)) # 점을 리스트에 추가

    return np.array(points) # 초기값부터 갱신된 값 전체 리턴
```

위 함수에 의해 초기 점은 (0,5,2.0)으로 주고, 10000개의 점을 갱신해 보면 아래 trajectory라는 변수에 저장된다. 따라서, 'trajectory'는 10001개의 좌표가 저장되고, 초기 점은 trajectory[0]에 (0.5,2.0)으로 저장된다.

```python
# 경사하강법 반복 수행해 보기
import numpy as np
import matplotlib.pyplot as plt
from mpl_toolkits.mplot3d import Axes3D # 3차원 그래프 그리는 다른 라이브러리

# 초기 위치 설정
starting_point = (0.5, 2.0)

# 학습률 및 반복 횟수 설정
learning_rate = 0.01  # 학습률을 더 작게 조정
num_iterations = 10000 # 점 생성 갯수 설정

# 경사 하강법 수행
trajectory = gradient_descent(starting_point, learning_rate, num_iterations)
```

```python
len(trajectory)
```

```
10001
```

```python
trajectory[0]
```

```
array([0.5, 2. ])
```

```python
# x좌표선택
trajectory[0,0]
```

```
0.5
```

```python
# x좌표선택 다른 방법
trajectory[0][0]
```

```
0.5
```

위에서의 결과에 따라 최소값, 그때의 좌표, 그래디언트를 구해보면 아래와 같다. 최종값을 살펴보면 (-0.6690, 0)에서 최소값 -0.4052를 출력해 주고 있으며, 최소값을 만드는 위치에서의 그래디언트는 (0,0) 임을 알 수 있다.

```python
# 최소값 계산
# trajectory[-1]에 최종 좌표가 들어가 있음
print('x 좌표:',trajectory[-1, 0])
print('y 좌표:',trajectory[-1, 1])
print('최소값:',f(trajectory[-1, 0], trajectory[-1, 1]))
```

```
x 좌표: -0.6690718221499515
y 좌표: 4.305038622038563e-39
최소값: -0.4052368702666903
```

```
# 최소값 위치에서의 그래디언트 살펴보기
x,y=trajectory[-1, 0],trajectory[-1, 1]
gradient_x = -2*x**2*np.exp(-x**2-y**2)+x/10+np.exp(-x**2-y**2)
gradient_y =-2*x*y*np.exp(-x**2-y**2)+y/10
(gradient_x,gradient_y)
```

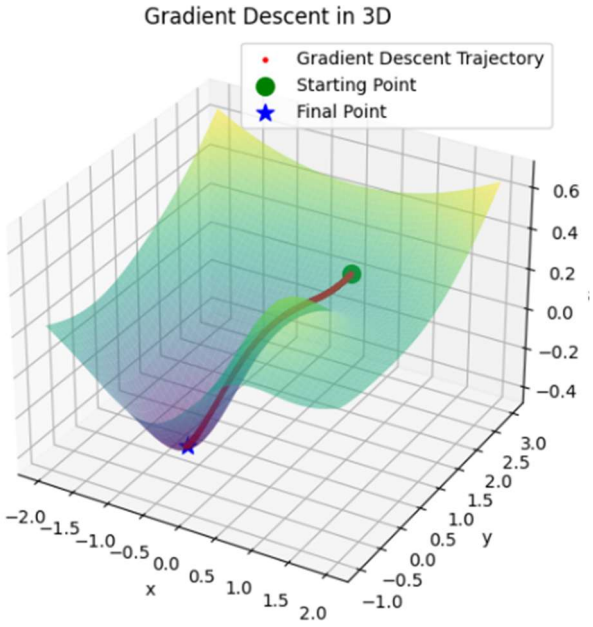
```
(5.440092820663267e-15, 4.112342729216583e-39)
```

경사(상승) 하강법에서 수렴하는 점이 존재한다면 그 좌표에서의 그래디언트는 영벡터가 되는데, 이는 변수가 1개인 상황에서 극값과 동일한 의미라고 볼 수 있다. 극값이 최대, 최소를 보장하지 않는 것처럼 경사(상승) 하강법도 최대, 최소를 보장해 주지 않는다는 사실을 기억하자.

이제 trajectory에 저장된 점들이 최소값을 찾아 가는 지를 시각적으로 보기 위해 아래와 같은 코드를 활용하면 된다.

```
# 3차원 그래프 생성
fig = plt.figure(figsize=(8, 6))
ax = fig.add_subplot(111, projection='3d')

# 함수의 등고선 플로팅
x = np.linspace(-2, 2, 400) # -2에서 2를 400개로 분할
y = np.linspace(-1, 3, 400)
X, Y = np.meshgrid(x, y) # 순서쌍 생성
Z = f(X, Y) # 현재 함수 값을 Z에 저장
ax.plot_surface(X, Y, Z, cmap='viridis', alpha=0.5)

# 경사 하강법의 점들을 plotting
ax.scatter(trajectory[:, 0], trajectory[:, 1], f(trajectory[:, 0], trajectory[:, 1]),
           color='r', s=50, label='Gradient Descent Trajectory')

# 출발점과 최종 도달 지점 표시
ax.scatter(starting_point[0], starting_point[1], f(*starting_point),
           color='g', s=100, marker='o', label='Starting Point')
ax.scatter(trajectory[-1, 0], trajectory[-1, 1], f(trajectory[-1, 0], trajectory[-1, 1]),
           color='b', s=100, marker='o', label='Final Point')

# 그래프 설정
ax.set_xlabel('x')
ax.set_ylabel('y')
ax.set_zlabel('f(x, y)')
ax.set_title('Gradient Descent in 3D')
ax.legend()
```

결과를 살펴보자. 초록색 점이 출발점이고 파란색 점이 최종 도달한 점이다. 아래 오른쪽은 3차원 그래프를 회전시켜 Z축을 옆에서 바라 본 모습으로 최소값에 거의 도달한 것을 알 수 있다.



최적화문제의 대부분은 파이썬 라이브러리 중 scipy.optimize[1] 아래에 있는 함수들을 활용하면 매우 유용하다. 아래는 minimize 함수를 이용하여 위 문제를 해를 구해 본 것으로 위에서 구한 경사하강법과 값이 매우 근사함을 확인할 수 있다. 아래 결과창에서 res.fun에는 함수의 최소값, res.x에는 최소값을 만들어 내는 최적의 좌표값을 생성해 준다. 참고로 아래 함수내에서 사용된 method는 최적화 방법을 선택하는 데 사용되며, nelder-mead이외에도 Powell, CG,BFGS,SLSQP 등이 있다.

```
from scipy.optimize import minimize
import numpy as np
def f(x):
    return x[0]*np.exp(-(x[0]**2+x[1]**2))+(x[0]**2+x[1]**2)/20
res=minimize(f,x0=[-5,5],method='nelder-mead')
res
```

```
      message: Optimization terminated successfully.
      success: True
       status: 0
          fun: -0.4052368684443885
            x: [-6.691e-01 -1.398e-05]
          nit: 43
         nfev: 79
final_simplex: (array([[-6.691e-01, -1.398e-05],
                       [-6.690e-01, -3.536e-05],
                       [-6.691e-01, -9.638e-05]]), array([-4.052e-01, -4.052e-01, -4.052e-01]))
```

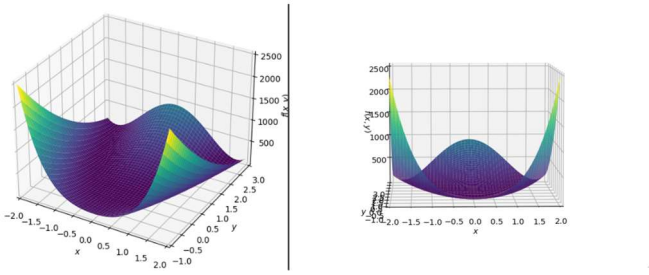
[1] conda install -c conda-forge scipy

문제3. 아래 다변량 함수의 그래프를 그리고, 최대값과 최소값을 구하여 보아라.

$$f(x,y) = (1-x)^2 + 100(y-x^2)^2 \ (-2 \le x \le 2, -1 \le y \le 3)$$

이 함수는 최적화 문제 테스트에 많이 사용되는 Rosenbrock 함수로 알려져 있다. 먼저 3차원 그래프를 그려 살펴보면 노란색 부분에서 최대값이 형성되고, 파란색이 짙을수록 최소값에 가깝다는 것을 시각적으로 확인할 수 있다.

```
# 심파이에서 3차원 그래프를 그리기 위한 함수 호출 필요
from sympy.plotting import plot3d
from sympy import symbols

# x,y라는 문자를 실수로 정의
x,y=symbols('x y',real=True)

f=(1-x)**2+100*(y-x**2)**2 # 함수 정의
plot3d(f, (x, -2, 2), (y, -1, 3)) # x,y범위에서 3차원 그래프 그리기
```



이 문제는 경계값에서의 해가 최대값이 될 수 있기 때문에 경사하강법으로는 해를 찾기 힘들다[2]. 'scipy'를 이용하면 쉽게 해를 찾을 수 있다. 최대값은 (-2,-1)일 때 2509임을 알 수 있다.

```
from scipy.optimize import minimize
# Rosenbrock 함수 정의
def rosenbrock(x):
    return (1 - x[0])**2 + 100 * (x[1]- x[0]**2)**2

def f(x): # 최소값 문제로 바꾸기 위해 (-)를 붙여서
    return -rosenbrock(x)

bounds=[(-2,2),(-1,3)] # 변수들의 상하한 지정
#x0에는 초기값, bounds에는 변수들의 범위 지정
res=minimize(f,x0=[0,0],bounds=bounds)
res
```

```
  message: CONVERGENCE: NORM_OF_PROJECTED_GRADIENT_<=_PGTOL
  success: True
   status: 0
      fun: -2509.0
        x: [-2.000e+00 -1.000e+00]
      nit: 2
      jac: [ 4.006e+03  1.000e+03]
     nfev: 9
     njev: 3
 hess_inv: <2x2 LbfgsInvHessProduct with dtype=float64>
```

[2] 경사하강법은 극값을 찾는 방법의 하나이며, 지역 최적해(local optimum)은 구할 수 있지만 전역 최적해(global optimum)는 보장해 주지 않는다.

반대로 최소값은 아래와 같이 구할 수 있다. (1,1)일 때, 함수값은 0임을 알 수 있다.

```
res=minimize(rosenbrock,x0=[0,0],bounds=bounds)
res
```

```
  message: CONVERGENCE: NORM_OF_PROJECTED_GRADIENT_<=_PGTOL
  success: True
   status: 0
      fun: 7.420087657482253e-12
        x: [ 1.000e+00  1.000e+00]
      nit: 21
      jac: [ 5.751e-06 -2.588e-06]
     nfev: 84
     njev: 28
 hess_inv: <2x2 LbfgsInvHessProduct with dtype=float64>
```

제약식을 갖는 문제의 최적해 구하기

제약식을 갖는 함수의 최적해 문제는 훨씬 더 복잡한 알고리즘을 필요로 한다. 알고리즘 자체를 설명하기에는 이 책의 목적에서 너무 벗어남으로 간단히 예제를 중심으로 scipy의 함수들을 이용하여 선형계획법(LP), 비선형계획법(NLP) 문제들을 해결해 보도록 하자.

선형계획법(Linear Programming)이란 목적함수와 제약식이 모두 선형결합으로 이루어진 경우를 말한다.

문제4. 다음 선형계획법 문제를 풀어보시오.

$$\begin{aligned} \min \quad & -x_0 + 4x_1 \\ \text{s.t.} \quad & -3x_0 + x_1 \leq 6 \\ & -x_0 - 2x_1 \geq -4 \\ & x_1 \geq -3 \end{aligned}$$

선형계획법(linear programming)이란 목적함수와 제약식이 모두 1차식의 선형조합(linear combination)으로 표시된 식을 말한다. Python에서는 SimplexMethod 알고리즘에 바탕을 둔 'linprog' 함수를 이용하여 수치해석적인 방법으로 해를 구하게 된다.

'linprog'에서 목적함수는 항상 최소값을 푸는 문제로 해야 한다. A_ub라는 인자는 제약식 Ax<=b로 표현했을 때의 A라는 행렬을 선택하면 되고, bounds라는 인자는 각 변수들의 범위를 대입해 주면 된다.

아래 결과를 보면 x=[1.000e+01 -3.000e+00]일 때, 목적함수의 최소값(fun)은 -22.0임을 알 수 있다.

```
from scipy.optimize import linprog
f=[-1,4] # 목적함수의 계수
A=[[-3,1],[1,2]] # Ax<=b라고 할 때, A 행렬의 계수만 지정
b=[6,4] # Ax<=b 일 때 b 항
bounds=[(None,None),(-3,None)] # 각 변수의 범위지정
res=linprog(f,A_ub=A,b_ub=b,bounds=bounds)
res
```

```
        message: Optimization terminated successfully. (HiGHS Status 7: Optimal)
        success: True
         status: 0
            fun: -22.0
              x: [ 1.000e+01 -3.000e+00]
            nit: 0
          lower:  residual: [       inf  0.000e+00]
                 marginals: [ 0.000e+00  6.000e+00]
          upper:  residual: [       inf        inf]
                 marginals: [ 0.000e+00  0.000e+00]
          eqlin:  residual: []
                 marginals: []
        ineqlin:  residual: [ 3.900e+01  0.000e+00]
                 marginals: [-0.000e+00 -1.000e+00]
 mip_node_count: 0
 mip_dual_bound: 0.0
        mip_gap: 0.0
```

문제5. 다음 선형계획법 문제를 풀어보시오.

$$\begin{aligned} \min \quad & -2x_1 - x_2 - 4x_3 - 3x_4 - x_5 \\ \text{s.t.} \quad & 2x_2 + x_3 + 4x_4 + 2x_5 \leqslant 54 \\ & 3x_1 + 4x_2 + 5x_3 - x_4 - x_5 \leqslant 62 \\ & x_1, x_2 \geqslant 0, x_3 \geqslant 3.32, x_4 \geqslant 0.678, x_5 \geqslant 2.57 \end{aligned}$$

최적해는 x= [1.979e+01 0.000e+00 3.320e+00 1.138e+01 2.570e+00] 일 때, -89.575가 된다.

```
from scipy.optimize import linprog
f=[-2,-1,-4,-3,-1] # 목적함수의 계수
A=[[0,2,1,4,2],[3,4,5,-1,-1]] # Ax<=b로 표현할 때, A
b=[54,62] # # Ax<=b로 표현할 때, b
bnds=[(0,None),(0,None),(3.32,None),(0.678,None),(2.57,None)] # 각 변수의 범위지정
res=linprog(f,A_ub=A,b_ub=b,bounds=bnds)
res
```

```
        message: Optimization terminated successfully. (HiGHS Status 7: Optimal)
        success: True
         status: 0
            fun: -89.57499999999999
              x: [ 1.979e+01  0.000e+00  3.320e+00  1.138e+01  2.570e+00]
            nit: 0
          lower:  residual: [ 1.979e+01  0.000e+00  0.000e+00  1.071e+01
                              0.000e+00]
                 marginals: [ 0.000e+00  3.500e+00  2.500e-01  0.000e+00
                              1.667e-01]
          upper:  residual: [       inf        inf        inf        inf
                                    inf]
                 marginals: [ 0.000e+00  0.000e+00  0.000e+00  0.000e+00
                              0.000e+00]
          eqlin:  residual: []
                 marginals: []
        ineqlin:  residual: [ 0.000e+00  0.000e+00]
                 marginals: [-9.167e-01 -6.667e-01]
 mip_node_count: 0
 mip_dual_bound: 0.0
        mip_gap: 0.0
```

문제6. 다음 선형계획법 문제를 풀어보시오.

$$\begin{aligned} \min \quad & x_1 + 3x_2 + 4x_3 \\ \text{s.t.} \quad & x_1 + 2x_2 + x_3 = 5 \\ & 2x_1 + 3x_2 + x_3 = 6 \\ & x_2 \geqslant 0,\ x_3 \geqslant 0 \end{aligned}$$

이 문제는 제약식으로 부등식이 아닌 등식을 갖게 되는 문제이다. 함수 linprog의 인자에 A_eq를 사용하여 지정해 주면 된다. 최적해는 x= [-3.000e+00 4.000e+00 0.000e+00]일 때, 9의 값을 갖는다.

```
from scipy.optimize import linprog
f=[1,3,4] # 목적함수의 계수
Aeq=[[1,2,1],[2,3,1]] # Ax=b일 때, A
beq=[5,6] # Ax=b일때, b
bnds=[(None,None),(0,None),(-0,None)] # 변수의 범위 지정
res=linprog(f,A_eq=Aeq,b_eq=beq,bounds=bnds)
res
```

```
       message: Optimization terminated successfully. (HiGHS Status 7: Optimal)
       success: True
        status: 0
           fun: 9.0
             x: [-3.000e+00  4.000e+00  0.000e+00]
           nit: 2
         lower:  residual: [       inf  4.000e+00  0.000e+00]
                marginals: [ 0.000e+00  0.000e+00  2.000e+00]
         upper:  residual: [       inf         inf         inf]
                marginals: [ 0.000e+00  0.000e+00  0.000e+00]
         eqlin:  residual: [ 0.000e+00  0.000e+00]
                marginals: [ 3.000e+00 -1.000e+00]
       ineqlin:  residual: []
                marginals: []
 mip_node_count: 0
 mip_dual_bound: 0.0
        mip_gap: 0.0
```

```
from scipy.optimize import linprog
f=[-2,-1,-4,-3,-1] # 목적함수의 계수
A=[[0,2,1,4,2],[3,4,5,-1,-1]] # Ax<=b로 표현할 때, A
b=[54,62] # # Ax<=b로 표현할 때, b
bnds=[(0,None),(0,None),(3.32,None),(0.678,None),(2.57,None)] # 각 변수의 범위지정
res=linprog(f,A_ub=A,b_ub=b,bounds=bnds)
res
```

```
       message: Optimization terminated successfully. (HiGHS Status 7: Optimal)
       success: True
        status: 0
           fun: -89.57499999999999
             x: [ 1.979e+01  0.000e+00  3.320e+00  1.138e+01  2.570e+00]
           nit: 0
         lower:  residual: [ 1.979e+01  0.000e+00  0.000e+00  1.071e+01
                             0.000e+00]
                marginals: [ 0.000e+00  3.500e+00  2.500e-01  0.000e+00
                             1.667e-01]
         upper:  residual: [       inf        inf        inf        inf
                                   inf]
                marginals: [ 0.000e+00  0.000e+00  0.000e+00  0.000e+00
                             0.000e+00]
         eqlin:  residual: []
                marginals: []
       ineqlin:  residual: [ 0.000e+00  0.000e+00]
                marginals: [-9.167e-01 -6.667e-01]
 mip_node_count: 0
 mip_dual_bound: 0.0
        mip_gap: 0.0
```

문제7. 다음 선형계획법 문제를 풀어보시오.

$$\begin{aligned} \min \quad & 80x_1 + 60x_2 \\ \text{s.t.} \quad & 0.20x_1 + 0.32x_2 \leq 0.25 \\ & x_1 + x_2 = 1 \\ & x_1 \geqslant 0,\ x_2 \geqslant 0 \end{aligned}$$

이 문제는 제약식으로 부등식과 등식을 모두 갖는 문제이다. 최적해는 x=[5.833e-01 4.167e-01] 일 때, 71.6667 임을 알 수 있다.

```
c=[80,60] # 목적함수의 계수
A_ub=[[0.2,0.32]] # Ax<=b로 표현할 때, A
b_ub=[0.25] # Ax<=b로 표현할 때,b
A_eq=[[1,1]]  # Ax=b일 때, A
b_eq=[1] # Ax=b일 때, b
x1b=(0,None) # 첫번째 변수의 범위
x2b=(0,None) # 두번째 변수의 범위
res=linprog(c,A_ub=A_ub,b_ub=b_ub,A_eq=A_eq,b_eq=b_eq,bounds=[x1b,x2b])
res
```

```
        message: Optimization terminated successfully. (HiGHS Status 7: Optimal)
        success: True
         status: 0
            fun: 71.66666666666666
              x: [ 5.833e-01  4.167e-01]
            nit: 0
          lower:  residual: [ 5.833e-01  4.167e-01]
                 marginals: [ 0.000e+00  0.000e+00]
          upper:  residual: [       inf        inf]
                 marginals: [ 0.000e+00  0.000e+00]
          eqlin:  residual: [ 0.000e+00]
                 marginals: [ 1.133e+02]
        ineqlin:  residual: [ 0.000e+00]
                 marginals: [-1.667e+02]
 mip_node_count: 0
 mip_dual_bound: 0.0
        mip_gap: 0.0
```

문제8. 목적함수나 제약식에 하나라도 비선형 결합이 있다면 선형계획법 문제가 아닌 비선형계획법(NLP: Non-linear programming) 문제라고 부른다. 이러한 비선형 최적화 문제에 대한 해법은 현재까지 완전하지 못한 상태로 대부분의 해는 local optimum을 여러 개 찾은 다음 이들 중 최적해를 고르는 상황이다. 즉, 주어진 초기해 근처의 최적해 정도를 찾을 수는 있지만 global optimum을 보증하기는 매우 까다롭다고 알려져 있다. 다음의 문제는 목적함수는 선형이지만 제약식이 비선형인 것을 포함하고 있다. 이 문제의 최적해를 구하여라.

$$\begin{aligned} \min \quad & x_1 + x_2 + x_3 \\ \text{s.t.} \quad & \\ & x_1^2 + x_2 = 3 \\ & x_1 + 3x_2 + 2x_3 = 7 \end{aligned}$$

목적함수나 제약식에 비선형방정식이나 비선형부등식이 하나라도 포함되어 있다면 기존의 SimplexMethod로는 해를 찾기 불가능하다.

파이썬에서는 'scipy.optimize' 아래에 있는 minimize() 함수를 이용하여 초기해 근처에서의 최적해를 찾게 된다[3]. 방정식의 제약식을 정의할 때는 항상 equal to zero가 되도록 정리하도록 한다.

```
from scipy.optimize import minimize
# 목적함수를 따로 정의한다.
def f(x):
    return x[0]+x[1]+x[2]

# 비선형방정식들의 모임을 정의합니다.
# 우변의 항이 모두 0이 되도록 정리해야 합니다.
def eqs(x):
    return [x[0]**2+x[1]-3,x[0]+3*x[1]+2*x[2]-7]

# 제약식의 type과 해당함수를 사전형으로 지정
cnts={'type':'eq','fun':eqs}

# 비선형계획법에서는 반드시 초기해(매개변수 x0=)를 주어야 한다.
# 제약식은 매개변수 constraints로 설정
# method 매개변수의 비선형알고리즘을 지정할 수 있음.
res=minimize(f,x0=[100,1,0],constraints=cnts, method='SLSQP')
res
```

초기값 (100,1,0)에서 시작된 비선형계획법의 최적해는 x=array([-0.5000016 , 2.74999867, -0.3749972]) 이며, 목적함수의 최소값은 fun= 1.8749998666956125의 결과가 도출된다. 하지만, global optimum은 완전히 보장하지 않는다.

```
     fun: 1.8749998666956125
     jac: array([1., 1., 1.])
 message: 'Optimization terminated successfully'
    nfev: 75
     nit: 17
    njev: 17
  status: 0
 success: True
       x: array([-0.5000016 ,  2.74999867, -0.3749972 ])
```

문제9. 다음 비선형문제의 해를 구하여라.

$$\begin{aligned} \min \quad & e^{x_1}(4x_1^2 + 2x_2^2 + 4x_1x_2 + 2x_2 + 1) \\ \text{s.t.} \quad & \\ & x_1^2 + x_2 - 1 = 0 \\ & -x_1x_2 - 10 \le 0 \end{aligned}$$

이 문제는 방정식 형태의 제약식뿐만 아니라 부등식 형태의 제약식도 포함되어 있다. 여기서 주

[3] 경사벡터를 사용한 BFGS(Broyden-Fletcher-Goldfarb-Shanno) 알고리즘, SLSQP(Sequential Least Squares Quadratic Programming) 등 여러가지가 사용된다. 비선형계획법도 목적함수는 모두 최소화문제로 변형하여 풀게 된다.

의할 것은 부등식의 형태가 항상 0보다 큰 상태의 식으로 정리해야 한다는 것이다.

```
from scipy.optimize import minimize
import numpy as np

# 목적함수 설정
def f(x):
    return np.exp(x[0])*(4*x[0]**2+2*x[1]**2+4*x[0]*x[1]+2*x[1]+1)

# 방정식형태의 제약식 정의
def eqs(x):
    return x[0]**2+x[1]-1

# 부등식형태의 제약식 정의
def ieqs(x):
    return x[0]*x[1]+10 # 반드시 greater than zero로 할 것

# 제약식에서 방정식과 부등식 제약식 연결
cons=[{'type':'eq','fun':eqs},{'type':'ineq','fun':ieqs}]

res=minimize(f,x0=[-1,0],method='SLSQP',constraints=cons)
res
```

위 문제에서 방정식과 부등식 제약식 연결부분에서, 동시에 기술하지 않고 아래와 같이 방정식 별도, 부등식 별도로 연결하는 방법도 가능하다.

```
con1={'type':'eq','fun':eqs}
con2={'type':'ineq','fun':ieqs}

res=minimize(f,x0=[0,0],method='SLSQP',constraints=[con1,con2])
res
```

초기해 [0,0] 근처에서의 최적해는 [0.49986966, -0.9998307]이며, 이때 함수의 최적값은 6.101457839897429e-08으로 도출되었다.

```
     fun: 6.101457839897429e-08
     jac: array([-0.00060246,  0.0002569 ])
 message: 'Optimization terminated successfully'
    nfev: 24
     nit: 7
    njev: 7
  status: 0
 success: True
       x: array([ 0.49986966, -0.9998307 ])
```