Optimization - Practical 4

December 18, 2024

Theodoros lambrou

Constrained Optimization: Equality Constraints

1. Experiment 1

Firstly, I define the function f and the equality constraint h, and then compute the gradient and hessian. I also compute the langrangian as $L(x, \lambda) = f(x) - \lambda h(x)$ along its gradient and hessian.

[2]:
```
from math import exp as e
import numpy as np

def f(x_1,x_2):
  return e(3*x_1) + e(-4*x_2)

def h(x_1,x_2):
  return x_1**2 + x_2**2 - 1


def grad_f(x_1, x_2):
    return np.array([3*e(3*x_1), -4*e(-4*x_2)])

def grad_h(x_1, x_2):
    return np.array([2*x_1, 2*x_2])

def hess_f(x_1, x_2):
    return np.array([[9*e(3*x_1), 0], [0, 16*e(-4*x_2)]])

def hess_h(x_1, x_2):
    return np.array([[2, 0], [0, 2]])


def lag(x_1, x_2, lamda):
    return f(x_1, x_2) - lamda*h(x_1, x_2)

def grad_lag(x_1, x_2, lamda):
    return grad_f(x_1, x_2) - lamda*grad_h(x_1, x_2)

def hess_lag(x_1, x_2, lamda):
```

```
    return hess_f(x_1, x_2) - lamda*hess_h(x_1, x_2)
```

Then I define the Newtons based iterative method to find the solution.

[6]:
```
def solveNewtonBased(x_1, x_2, lamda, alpha=1, eps=1e-5, MAX_ITER=100,␣
 ↪verbose=False):

    print(f'Initial point x0 = [{x_1}, {x_2}]')

    for i in range(MAX_ITER):

        gh = grad_h(x_1, x_2)
        grad_lag_value = grad_lag(x_1, x_2, lamda)
        hess_lag_value = hess_lag(x_1, x_2, lamda)

        A = np.block([[hess_lag_value, -gh.reshape(-1, 1)], [-gh, np.
 ↪array([[0]])]])
        b = np.concatenate([-grad_lag_value, [h(x_1, x_2)]])

        delta = np.linalg.solve(A, b) #solving the linear system

        x_1 += alpha * delta[0]
        x_2 += alpha * delta[1]
        lamda += alpha * delta[2]

        if np.linalg.norm(grad_lag(x_1, x_2, lamda)) < eps:
          if verbose: print('Break by Lagrangian Gradient')
          break

        if verbose:
          print('Iterations: {}'.format(i))
          print('x = (x_1, x_2) = ({0:.5f}, {1:.5f}), lamda = {2:.6f}'.
 ↪format(x_1, x_2, lamda))

    return x_1,x_2, lamda


x_1, x_2, lamda = -1, 1, -1

result = solveNewtonBased(x_1, x_2, lamda, alpha=1, eps=1e-5, MAX_ITER=100,␣
 ↪verbose=True)

print("Final result:", result)
```

```
Initial point x0 = [-1, 1]
Iterations: 0
x = (x_1, x_2) = (-0.77423, 0.72577), lamda = -0.351038
Iterations: 1
```

```
x = (x_1, x_2) = (-0.74865, 0.66614), lamda = -0.216059
Break by Lagrangian Gradient
Final result: (-0.7483381762503777, 0.663323446868971, -0.21232390186241443)
```

It can be seen that the solution is reached in 2 iterations (and it matches the given solution).

2. Experiment 2

I repeat the previous method after I define some points which are further away of the optimal solution.

[21]:
```
ranges = [
    (-8, -4),
    (3, 7),
]

points = np.array([
    np.random.uniform(low, high, 3) for low, high in ranges
])

print(points)
```

```
[[-6.23081254 -6.92163864 -7.10536778]
 [ 6.77687368  5.56285214  6.00589472]]
```

[20]:
```
for point in points:
    x_1, x_2, lamda = point
    print('Starting point:\nx = (x_1, x_2) = ({0:.5f}, {1:.5f}), lamda = {2:.
 ↪5f}'.format(x_1, x_2, lamda))
    solveNewtonBased(x_1, x_2, lamda, verbose=True)
```

```
Starting point:
x = (x_1, x_2) = (-7.59140, -5.68915), lamda = -8.62714
Initial point x0 = [-7.591397202007906, -5.689151444425505]
Iterations: 0
x = (x_1, x_2) = (-1.91713, -5.43915), lamda = -6.448436
Iterations: 1
x = (x_1, x_2) = (5.78713, -5.18915), lamda = -25.973879
Iterations: 2
x = (x_1, x_2) = (0.94601, -4.86291), lamda = -121524887.606792
Iterations: 3
x = (x_1, x_2) = (-3.81388, -3.36823), lamda = -611457156.528497
Iterations: 4
x = (x_1, x_2) = (-1.97432, -1.75627), lamda = -294926428.737318
Iterations: 5
x = (x_1, x_2) = (-1.12853, -1.00391), lamda = -126345116.255457
Iterations: 6
x = (x_1, x_2) = (-0.81160, -0.72197), lamda = -35482395.040886
Iterations: 7
```

```
x = (x_1, x_2) = (-0.74971, -0.66692), lamda = -2705417.899128
Iterations: 8
x = (x_1, x_2) = (-0.74717, -0.66465), lamda = -9197.180354
Iterations: 9
x = (x_1, x_2) = (-0.74870, -0.66291), lamda = 18.672930
Iterations: 10
x = (x_1, x_2) = (-1.06219, -0.30885), lamda = -7.831214
Iterations: 11
x = (x_1, x_2) = (-1.01457, -0.11058), lamda = -0.417786
Iterations: 12
x = (x_1, x_2) = (-1.02050, 0.13187), lamda = -0.066761
Iterations: 13
x = (x_1, x_2) = (-0.96013, 0.37604), lamda = -0.085227
Iterations: 14
x = (x_1, x_2) = (-0.84392, 0.58865), lamda = -0.128543
Iterations: 15
x = (x_1, x_2) = (-0.75030, 0.67300), lamda = -0.195296
Iterations: 16
x = (x_1, x_2) = (-0.74854, 0.66317), lamda = -0.212096
Break by Lagrangian Gradient
Starting point:
x = (x_1, x_2) = (-2.09947, -2.74047), lamda = -1.49257
Initial point x0 = [-2.0994736548888566, -2.7404662591639064]
Iterations: 0
x = (x_1, x_2) = (0.17435, -2.49046), lamda = -1.626793
Iterations: 1
x = (x_1, x_2) = (-11.13877, -2.23189), lamda = -583.685251
Iterations: 2
x = (x_1, x_2) = (-5.44247, -1.97334), lamda = -298.492855
Iterations: 3
x = (x_1, x_2) = (-2.54998, -1.71237), lamda = -158.638992
Iterations: 4
x = (x_1, x_2) = (-1.07425, -1.44712), lamda = -91.809718
Iterations: 5
x = (x_1, x_2) = (-0.39516, -1.17446), lamda = -58.206363
Iterations: 6
x = (x_1, x_2) = (-0.41025, -0.94140), lamda = 1.114355
Iterations: 7
x = (x_1, x_2) = (-0.92444, -0.68836), lamda = -0.817271
Iterations: 8
x = (x_1, x_2) = (-0.93213, -0.43947), lamda = -0.092199
Iterations: 9
x = (x_1, x_2) = (-1.01679, -0.18936), lamda = -0.064891
Iterations: 10
x = (x_1, x_2) = (-1.02902, 0.06043), lamda = -0.066495
Iterations: 11
x = (x_1, x_2) = (-0.98415, 0.30707), lamda = -0.078377
Iterations: 12
```

```
x = (x_1, x_2) = (-0.88119, 0.53472), lamda = -0.112358
Iterations: 13
x = (x_1, x_2) = (-0.76501, 0.66782), lamda = -0.178044
Iterations: 14
x = (x_1, x_2) = (-0.74810, 0.66381), lamda = -0.211521
Break by Lagrangian Gradient
```

It can be seen that when the starting points are farther away from the optimal solution, the method diverges.

3. Experiment 3

I define the merit function and its gradient, and then use the gradient descent method with gradient normalization.

[29]:
```
def merit(x_1, x_2, ro=10):
    return f(x_1, x_2) + ro*h(x_1, x_2)**2

def grad_merit(x_1, x_2, ro=10):
    return grad_f(x_1, x_2) + 2 * ro * h(x_1, x_2) * grad_h(x_1, x_2)

def gradient_descent(f, grad_f, w0, f_tol=1e-3, grad_tol=1e-5):
    x = [w0]

    while True:
        gradient_of_f = grad_f(w0[0], w0[1])
        grad_normalized = gradient_of_f / np.linalg.norm(gradient_of_f)
        alpha = 1

        while f(*(w0 - alpha * grad_normalized)) >= f(*w0): alpha /= 2

        w0 = w0 - alpha * grad_normalized
        x.append(w0)

        gradient_of_f = grad_f(w0[0], w0[1])
        grad_normalized = gradient_of_f / np.linalg.norm(gradient_of_f)

        if np.abs(f(*x[-1]) - f(*x[-2])) < f_tol or np.linalg.
 ↪norm(grad_normalized) < grad_tol: return np.array(x)


#testing the method by selecting a far away point
point=points[0]
solution_approx= gradient_descent(merit, grad_merit, point[:2])
print("Approximation:", solution_approx[-1])
```

```
Approximation: [-0.85499654  0.4660693 ]
```

Using gradient descent method (and gradient normalization) on merit function, we get the approximation above.

So it can be seen that we get closer to the optimal solution.

4. Experiment 4

I now use the point that is closer to the optimal solution and minimize the merit function with gradient descent. I then apply the Newton-based method to find the optimal solution by using the minimizer point for the merit function as the starting point for the Newton-based method. Hence, I take advanrage of the Newton-based alogrithm without having the problem of not performing well when the starting point is away from the minimum.

[31]:
```
solution_x =solveNewtonBased(solution_approx[-1,0], solution_approx[-1,1],␣
↪lamda=-1, alpha=1, verbose=True)
```

```
Initial point x0 = [-0.8549965437637832, 0.46606930041339356]
Iterations: 0
x = (x_1, x_2) = (-0.82965, 0.56809), lamda = -0.174847
Iterations: 1
x = (x_1, x_2) = (-0.75848, 0.66231), lamda = -0.197098
Iterations: 2
x = (x_1, x_2) = (-0.74823, 0.66352), lamda = -0.212118
Break by Lagrangian Gradient
```