Optimization Assignment Report

Dajin Han

May 2021

1 Method

1.1 Backtracking linear search

Backtracking linear search is a optimizing algorithm for optimizing the problem. It can attach to the point that maximize or minimize the function(x). It initialize α as the value α_{init} which is 1.

$$\alpha = \alpha_{init} \tag{1}$$

$$f(x_k + \alpha p_k) < f(x_k) \tag{2}$$

α decreases until equation (2) is not satisfied. But this algorithms could not prevent the big α. So in this assignment, Armijo's rule is adjusted.

1.2 Armijo's rule

In Armijo's rule, several details are added. Equation (2) is changed to Equation (3). Armijo's rule make decreasing α simpler.

$$f(x_k + \alpha_k p_k) \leq f(x_k) + \alpha_k \beta p_k^T \nabla f(x_k) \tag{3}$$

2 Implementation

There are two objects, Problem and Optimizer. Problem object contains function, differential function, and constriants. Optimizer is a object calculate the step size for target problem. With step size for each iteration, Optimizer object can optimize the point which minimize or maximize the function output.

2.1 Problem

Problem object consists of function "f", "df" and "check constraints". "f" return the output of function, "df" return the output of differential function and "check constraints" return whether input x satisfying constraints. In this assignment, I made two Problem object specified on statement of assignment. Problem1 is consists of function (4) and constraints (5). Problem2 is consists of function (6) and constraints (7).

Problems

In [3]:
```
class Problem1:
    def __init__(self, weights):
        w1, w2, w3 = weights
        self.w1 = w1
        self.w2 = w2
        self.w3 = w3

    # function f
    def f(self, x):
        x1, x2, x3 = x
        return self.w1*np.log(x1) + self.w2*np.log(x2) + self.w3*np.log(x3) - 1/((1-x1-x2-x3)**2)

    # differential function f about x
    def df(self, x):
        x1, x2, x3 = x
        return np.array([
            self.w1/x1 - 2/((1-x1-x2-x3)**3),
            self.w2/x2 - 2/((1-x1-x2-x3)**3),
            self.w3/x3 - 2/((1-x1-x2-x3)**3)
        ])

    # check if x satisfy constraints
    def check_constraints(self, x):
        x1, x2, x3 = x
        if x1<=0 or x2<=0 or x3<=0 or x1+x2+x3>=1:
            return False
        return True
```

In [4]:
```
class Problem2:
    def __init__(self, weights):
        w1, w2, w3 = weights
        self.w1 = w1
        self.w2 = w2
        self.w3 = w3

    # function f
    def f(self, x):
        x1, x2, x3 = x
        return self.w1*np.log(x1) + self.w2*np.log(x2) + self.w3*np.log(x3) - 1/(1-x1-x2)**2 - 1/(1-x1-x3)**2

    # differential function f about x
    def df(self, x):
        x1, x2, x3 = x
        return np.array([
            self.w1/x1 - 2/((1-x1-x2)**3) - 2/((1-x1-x3)**3),
            self.w2/x2 - 2/((1-x1-x2)**3),
            self.w3/x3 - 2/((1-x1-x3)**3)
        ])

    # check if x satisfy constraints
    def check_constraints(self, x):
        x1, x2, x3 = x
        if x1<=0 or x2<=0 or x3<=0 or x1+x2>=1 or x1+x3>=1:
            return False
        return True
```

Figure 1: Problem implementation

2.2 Optimizer

Optimizer object consists of target problem, parameters for optimizing and optimize function. Target problem is one of the problems created earlier. There are several parameters like α_{init}, β, ρ and ϵ. Each role is explained in comments of Figure 2. When Optimize the target problem, first calculate the differential function value about input x. Let df a differential value of current point. next point is moved to df direction not to $-df$ direction because our optimizer is trying to get the point maximizing the function value. For the case trying to get the point minimizing the function value, $-df$ could be the direction. So get the direction to nx which means the next x. Then calculate the α which satisfying Armijo's rule. Dividing α by 2, optimizer can get the αdf which has

the same meaning of step size.

Optimizer - MAX

In [2]:
```python
class Optimizer:

    def __init__(self, problem, init_alpha=1.0, beta=1e-4, rho=0.5, epsilon=1e-6):
        self.problem = problem          # problem to optimize
        self.init_alpha = init_alpha    # init_alpha to control step_size
        self.beta = beta
        self.rho = rho                  # parameter to be multiplied to alpha
        self.epsilon = epsilon          # very small number to check if there is a significant change

    def optimize(self, x):
        while True:
            df = self.problem.df(x)
            alpha = self.init_alpha

            # Stopping Criterion 1.
            if abs(np.dot(df, alpha*df)) < self.epsilon:
                # No significant gradient on current point
                break

            # Find next_x (= current_x + vector)
            # Direction of vector is df, and size of vector is calculated below
            # Decrease alpha until next_x satisfying all conditions
            while True:
                p = df
                nx = x + alpha*p

                # Check if next_x is satisfying x's constraints
                if self.problem.check_constraints(nx):
                    # Check if next_x is satisfying Armijo's rule
                    if not self.problem.f(nx) <= self.problem.f(x) + alpha*self.beta*np.dot(p, df):
                        break
                alpha *= self.rho

            # Stopping Criterion 2.
            if np.sqrt(np.dot(nx-x, nx-x)) < self.epsilon:
                # Edge case that maximum point is on the edge of domain
                # Judge that the point is converged
                x = nx
                break

            x = nx


        return x
```

Figure 2: Optimizer implementation

2.3 Stopping criterion of optimizer

In Figure 2, there are two stopping criterion. First criterion is to filter the point has the gradient which is not significant. I calculate the 3-dimensional gradient value *df* to 1-dimensional value by multiplying *df* itself. When the squared sum of *df* is smaller than epsilon, optimizer judge that current point has no significant gradient so it is the optimized point enough. Second criterion is to filter the case when the optimized point is on the boundary of domain. when the change of x is too small, optimizer judge that the point is converged enough so stop optimizing. Stopping criterion 2 is added to check the case that the convergence is appeared in the boundary of domain.

3 Experiments

With those objects, I run the optimizer to get the optimized, converged point about various condition.

3.1 Example 1

$$max[w_1 log x_1 + w_2 log x_2 + w_3 log x_3 - \frac{1}{(1 - x_1 - x_2 - x_3)^2}] \tag{4}$$

$$D = \{x \in \mathbb{R}^3 : x_1 > 0, x_2 > 0, x_3 > 0, x_1 + x_2 + x_3 < 1\} \tag{5}$$

3.1.1 Case 1

Case #1 - w1=1, w2=1, w3=1

```
In [5]: weights = np.array([1.0, 1.0, 1.0])
        problem = Problem1(weights)
        optimizer = Optimizer(problem)

        x0 = np.array([0.01, 0.01, 0.01])
        x_opt = optimizer.optimize(x0)
        print("x1 : %.6f\nx2 : %.6f\nx3 : %.6f\n" % (x_opt[0], x_opt[1], x_opt[2]))

        x1 : 0.123943
        x2 : 0.123943
        x3 : 0.123943
```

Figure 3: Result of problem 1 with case 1

In Figure 3, When $w_1 = 1.0, w_2 = 1.0, w_3 = 1.0$ and x_0 is (0.01, 0.01, 0.01), x is converged to the point (0.123943, 0.123943, 0.123943) maximizing the $f(x)$.

3.1.2 Case 2

Case #2 - w1=1, w2=2, w3=3

```
In [6]: weights = np.array([1.0, 2.0, 3.0])
        problem = Problem1(weights)
        optimizer = Optimizer(problem)

        x0 = np.array([0.01, 0.01, 0.01])
        x_opt = optimizer.optimize(x0)
        print("x1 : %.6f\nx2 : %.6f\nx3 : %.6f\n" % (x_opt[0], x_opt[1], x_opt[2]))

        x1 : 0.077237
        x2 : 0.154473
        x3 : 0.231721
```

Figure 4: Result of problem 1 with case 2

In Figure 4, When $w_1 = 1.0, w_2 = 2.0, w_3 = 3.0$ and x_0 is (0.01, 0.01, 0.01), x is converged to the point (0.077237, 0.154473, 0.231721) maximizing the $f(x)$.

3.1.3 Case 3

In Figure 5, When $w_1 = 1.0, w_2 = 4.0, w_3 = 7.0$ and x_0 is (0.01, 0.01, 0.01), x is converged to the point (0.045776, 0.183098, 0.320429) maximizing the $f(x)$.

Case #3 - custom

```
In [7]: weights = np.array([1.0, 4.0, 7.0])
        problem = Problem1(weights)
        optimizer = Optimizer(problem)

        x0 = np.array([0.01, 0.01, 0.01])
        x_opt = optimizer.optimize(x0)
        print("x1 : %.6f\nx2 : %.6f\nx3 : %.6f\n" % (x_opt[0], x_opt[1], x_opt[2]))

        x1 : 0.045776
        x2 : 0.183098
        x3 : 0.320429
```

Figure 5: Result of problem 1 with case 3

3.1.4 Case 4

Case #4 - custom2

```
In [8]: weights = np.array([1.0, -2.0, 3.0])
        problem = Problem1(weights)
        optimizer = Optimizer(problem)

        x0 = np.array([0.01, 0.01, 0.01])
        x_opt = optimizer.optimize(x0)
        print("x1 : %.6f\nx2 : %.6f\nx3 : %.6f\n" % (x_opt[0], x_opt[1], x_opt[2]))

        x1 : 0.013358
        x2 : 0.000000
        x3 : 0.019860
```

Figure 6: Result of problem 1 with case 4

In Figure 6, When $w_1 = 1.0, w_2 = -2.0, w_3 = 3.0$ and x_0 is (0.01, 0.01, 0.01), optimizer return the final x that judged to be converged on the point (0.013358, 0.000000, 0.019860) maximizing the $f(x)$. Optimizer could stop optimizing with the help of stopping criterion 2. $x2$ continuously moved to zero, which is the boundary of domain.

3.2 Example 2

$$max[w_1 log x_1 + w_2 log x_2 + w_3 log x_3 - \frac{1}{(1-x_1-x_2)^2} - \frac{1}{(1-x_1-x_3)^2}] \quad (6)$$

$$D = \{x \in \mathbb{R}^3 : x_1 > 0, x_2 > 0, x_3 > 0, x_1 + x_2 < 1, x_1 + x_3 < 1\} \quad (7)$$

3.2.1 Case 1

In Figure 7, When $w_1 = 1.0, w_2 = 1.0, w_3 = 1.0$ and x_0 is (0.01, 0.01, 0.01), x is converged to the point (0.093324, 0.186638, 0.186638) maximizing the $f(x)$.

3.2.2 Case 2

In Figure 8, When $w_1 = 1.0, w_2 = 2.0, w_3 = 3.0$ and x_0 is (0.01, 0.01, 0.01), x is converged to the point (0.062077, 0.282098, 0.332642) maximizing the $f(x)$.

Case #1 - w1=1, w2=1, w3=1

In [9]:
```
weights = np.array([1.0, 1.0, 1.0])
problem = Problem2(weights)
optimizer = Optimizer(problem)

x0 = np.array([0.01, 0.01, 0.01])
x_opt = optimizer.optimize(x0)
print("x1 : %.6f\nx2 : %.6f\nx3 : %.6f\n" % (x_opt[0], x_opt[1], x_opt[2]))
```

```
x1 : 0.093324
x2 : 0.186638
x3 : 0.186638
```

Figure 7: Result of problem 2 with case 1

Case #2 - w1=1, w2=2, w3=3

In [10]:
```
weights = np.array([1.0, 2.0, 3.0])
problem = Problem2(weights)
optimizer = Optimizer(problem)

x0 = np.array([0.01, 0.01, 0.01])
x_opt = optimizer.optimize(x0)
print("x1 : %.6f\nx2 : %.6f\nx3 : %.6f\n" % (x_opt[0], x_opt[1], x_opt[2]))
```

```
x1 : 0.062077
x2 : 0.282098
x3 : 0.332642
```

Figure 8: Result of problem 2 with case 2

3.2.3 Case 3

Case #3 - custom

In [11]:
```
weights = np.array([1.0, 4.0, 7.0])
problem = Problem2(weights)
optimizer = Optimizer(problem)

x0 = np.array([0.01, 0.01, 0.01])
x_opt = optimizer.optimize(x0)
print("x1 : %.6f\nx2 : %.6f\nx3 : %.6f\n" % (x_opt[0], x_opt[1], x_opt[2]))
```

```
x1 : 0.038755
x2 : 0.384225
x3 : 0.454755
```

Figure 9: Result of problem 2 with case 3

In Figure 9, When $w_1 = 1.0, w_2 = 4.0, w_3 = 7.0$ and x_0 is (0.01, 0.01, 0.01), x is converged to the point (0.038755, 0.384225, 0.454755) maximizing the $f(x)$.

3.2.4 Case 4

In Figure 10, When $w_1 = 1.0, w_2 = -2.0, w_3 = 3.0$ and x_0 is (0.01, 0.01, 0.01), optimizer return the final x that judged to be converged on the point (0.013287, 0.000000, 0.019863) maximizing the $f(x)$. Optimizer could stop optimizing with the help of stopping criterion 2. $x2$ continuously moved to zero, which is the boundary of domain.

Case #4 - custom2

In [12]:
```
weights = np.array([1.0, -2.0, 3.0])
problem = Problem2(weights)
optimizer = Optimizer(problem)

x0 = np.array([0.01, 0.01, 0.01])
x_opt = optimizer.optimize(x0)
print("x1 : %.6f\nx2 : %.6f\nx3 : %.6f\n" % (x_opt[0], x_opt[1], x_opt[2]))
```

```
x1 : 0.013287
x2 : 0.000000
x3 : 0.019863
```

Figure 10: Result of problem 2 with case 4