Homework 1

Vladimir Frants, Yunhua Zhao, Mohamed Ben Zid

September 2020

1 Exercise 1.3

For a continuously differential function $J(\theta)$ the Taylor Series expansion with remainder yields the following expression of the value of J at a point $\theta + td$, for $d \in R^d,\ t \in R^+$:

$$J(\theta + td) = J(\theta) + t\nabla J(\theta')d \quad \forall t > 0$$

Where $\theta' = \theta + \gamma d$ for some intermediate value $\gamma \in [0, t]$.
Or for any descent direction $d(\theta)$ we have $\nabla J(\theta)d(\theta) < 0$.

By continuity of ∇J we get:

$$\exists\ t_1 > 0\ s.t.\ \ \nabla J(\theta + td)d(\theta) < 0\ \ \forall t \in [0, t_1]$$

Then $\exists\ t_1, \gamma$ where $\gamma < t < t_1$ such that:

$$J(\theta + td) = J(\theta) + t\nabla J(\theta')d(\theta') \quad \forall t \in [0, t_1]$$

Where $\theta' = \theta + \gamma d$, and $\nabla J(\theta')d(\theta') < 0$ for some intermediate value $\gamma \in [0, t]$ from which it follows that:

$$J(\theta + td(\theta) \leq J(\theta)$$

.

2 Exercise 1.8

Using Taylor series expansion developed at θ_n to J, Taylor series is:

$$J(\theta_(n+1)) = J(\theta_n) + \nabla J(\theta_n)(\theta_(n+1) - \theta_n) + \frac{1}{2}(\theta_(n+1) - \theta_n)^{\mathrm{T}} \nabla J^2(\xi)(\theta_(n+1) - \theta_n)$$

So:

$$J(\theta_(n+1)) = J(\theta_n) -$$

$\epsilon_n \nabla J(\theta_n)(\nabla J(\theta_n)^{\mathrm{T}} + \beta_n(\theta_n)) + \frac{\epsilon_n^2}{2}(\nabla J(\theta_n)^{\mathrm{T}} + \beta_n(\theta_n))^{\mathrm{T}} \nabla J^2(\xi)(\nabla J(\theta_n)^{\mathrm{T}} + \beta_n(\theta_n))$

name:

$$g_n =$$

$\epsilon_n \nabla J(\theta_n)(\nabla J(\theta_n)^{\mathrm{T}} + \beta_n(\theta_n))$

$$h_n = \frac{\epsilon_n^2}{2}(\nabla J(\theta_n)^{\mathrm{T}} + \beta_n(\theta_n))^{\mathrm{T}} \nabla J^2(\xi)(\nabla J(\theta_n)^{\mathrm{T}} + \beta_n(\theta_n))$$

So

$$J(\theta_(n+1)) = J(\theta_n) - g_n + h_n$$

Because the gradient of J is Lipschitz continuous function, so:

$$h_n = \frac{\epsilon}{2} L \mid \nabla J(\theta_n)^{\mathrm{T}} + \beta_n(\theta_n) \mid^2$$

L is a finite constant.

$$h_n = \frac{\epsilon}{2} L \sqrt{(\nabla J^2(\theta_n) + \beta_n^2(\theta_n) + 2 \mid \nabla J(\theta_n) \beta_n(\theta_n))} \mid$$

Because: the bias is uniformly bounded, so $\beta_n^2(\theta_n)$ is bounded.

And because $\sum \epsilon_n^2 < \infty$ and $\sum \epsilon_n \theta_n < \infty$

So h_n is summable. According to Lemma 1.1 that $J(\theta_n)$ either diverges to $-\infty$ or it converges.

Set $\hat{\theta}$ as an accumulation point of the algorithm, so continuity $J(\theta_{nm})$ converges to $J(\hat{\theta})$ and $\nabla J(\theta_{nm})$ converges to $\nabla J(\hat{\theta})$.

Follows Lemma 1.1, J and $\nabla J(\theta)$ convege and

$$g_n$$

is summable, so:

$$\lim_{n \to \infty} \sum_{i=1}^{n} \epsilon_m i(\mid \nabla J(\theta_m i) \mid^2 + \nabla J(\theta_n) \beta_n(\theta_n)) < \infty$$

Because

$$\sum_{n=1}^{\infty} = +\infty$$

So any accumulation point is a stationary point.

3 Exercise 1.10

3.1 (a)

Minimize the cost function:

$$C(\theta) = \frac{1}{\theta^2}$$

s.t.

$$P(\theta) \leq \alpha$$

where $P(\theta)$ denotes the stationary probability that the queue length is larger than or equal to a threshold b.

Based on the fact that the probability of the stationary queue being n is given by $(1-\theta)\theta^n$, for $n \in \mathbb{N}$ we can compute the probability $P(\theta)$ as:

$$P(\theta) = 1 - \sum_{n=1}^{b-1}(1-\theta)\cdot\theta^n = 1-(1-\theta)\sum_{n=1}^{b-1}\theta^n = 1-(1-\theta)\cdot\frac{1-\theta^b}{1-\theta} = 1-[1-\theta^b] = \theta^b$$

,

So we have the problem on hands: $\min_\theta C(\theta)$, s.t. $P(\theta) \leq \alpha$ which leads to:

$$\min_\theta \frac{1}{\theta^2}$$

, s.t.

$$\theta^b - \alpha \leq 0$$

Constraint:

$$g(\theta) = \theta^b - \alpha \leq 0$$

Lagrangian:

$$\mathcal{L}(\theta, \lambda) = \frac{1}{\theta^2} + \lambda(\theta^b - \alpha)$$

$$\nabla_\theta(\theta, \lambda) = (-\frac{2}{\theta^3} + \lambda \cdot b \cdot \theta^{b-1}, \theta^b - \alpha)$$

$$\frac{\delta\mathcal{L}(\theta, \lambda)}{\delta\theta} = \frac{-2}{\theta^3} + \lambda \cdot b\theta^{b-1} = 0$$

$$\frac{\delta\mathcal{L}(\theta, \lambda)}{\delta\lambda} = \theta^b - \alpha = 0\lambda \cdot b\theta^{b-1} = 0 \Rightarrow a^{\frac{1}{b}} = \theta$$

$$\frac{-2}{(\alpha^{\frac{1}{b}})^3} + \lambda \cdot b \cdot (\alpha^{\frac{1}{3}})^{b-1} = 0$$

$$\lambda^* = \frac{2}{b \cdot \alpha^{\frac{b+2}{b}}}$$

Result:

$$\theta^* = \alpha^{\frac{1}{b}}$$

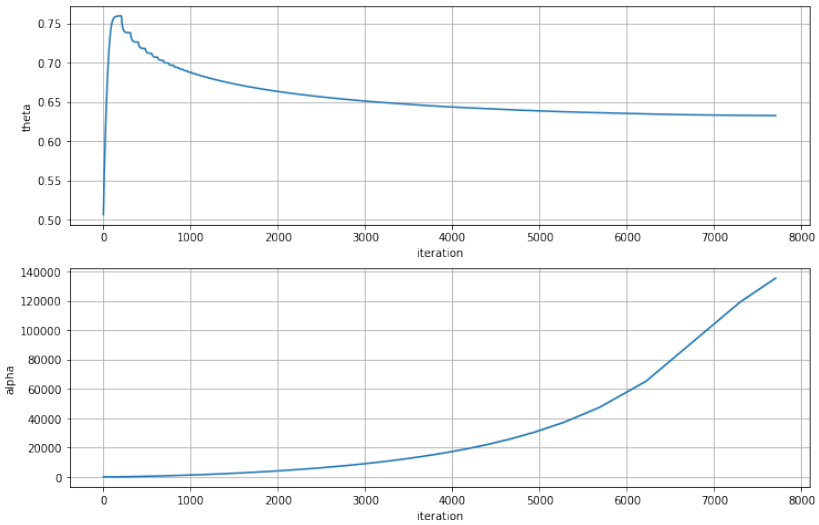
3.2 (b)

For $\alpha = 0.01$ and $b = 10$ we get $\theta^* = 0.01^{\frac{1}{10}} = 0.63$.

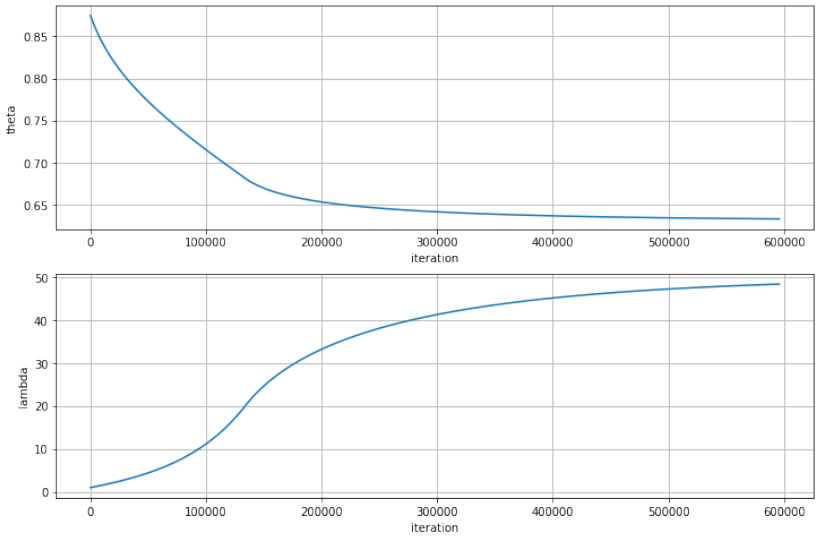
We programmed penalty method and Lagrange duality method to solve this problem numerically (the source code is in separate pdf file).

For both methods we plot the change of θ and of method parameters during optimization. We use finite differences to approximate the derivatives. As you could see in both cases final θ is close to the analytical solution. The penalty method converges faster, but the actual speed of convergence depends of the parameters of the method and could be improved.

3.2.1 Penalty method



3.2.2 Lagrange Duality Method



problem10

September 13, 2020

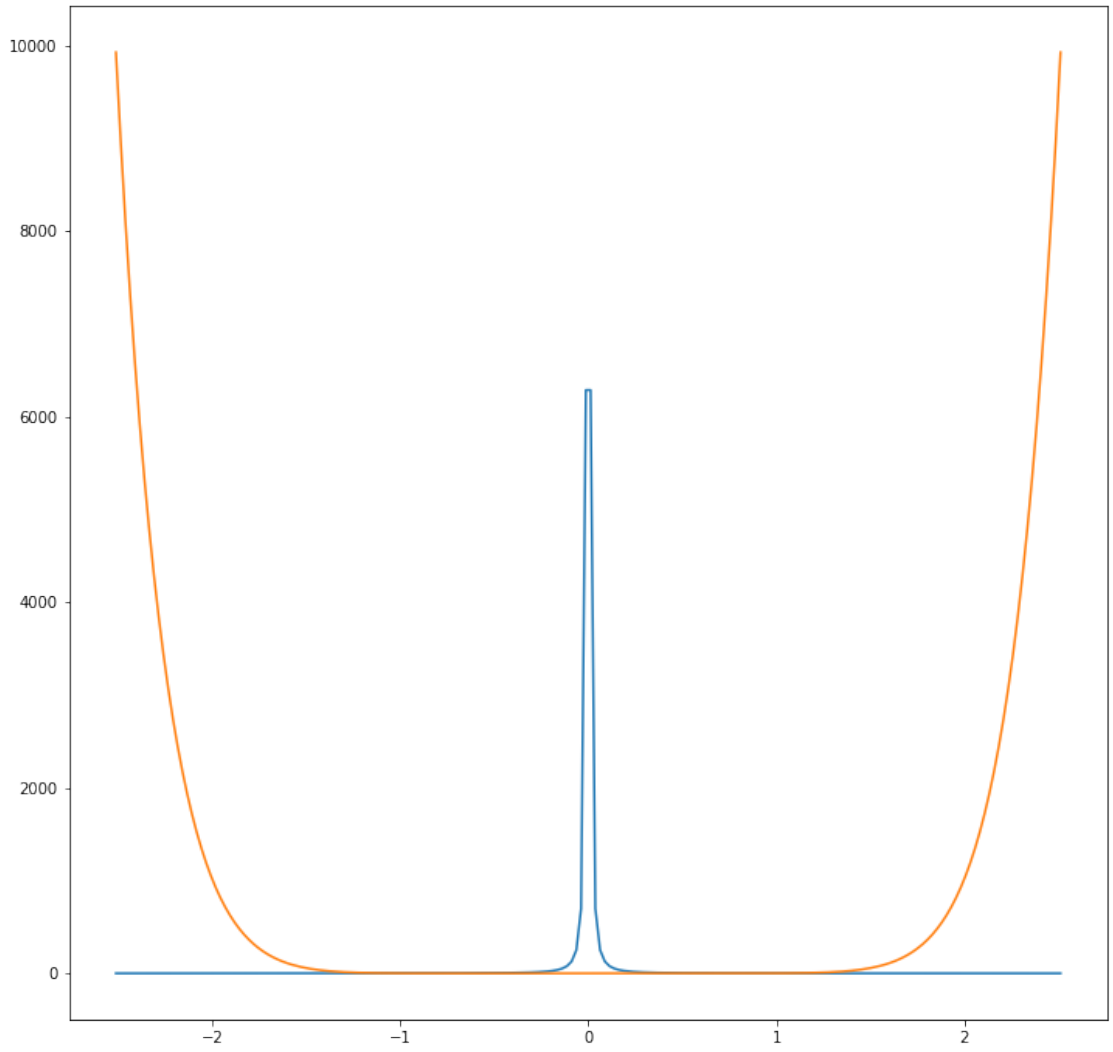
In [1]:
```
import numpy as np
import matplotlib.pyplot as plt
from scipy import optimize
%matplotlib inline
```

In [2]:
```
ALPHA = 0.01
B = 10
```

In [3]:
```
def f(theta):
  return 1/pow(theta, 2)
```

In [4]:
```
def c(theta):
  return pow(theta,B) - ALPHA
```

In [5]:
```
theta = np.linspace(-2.51, 2.51, 200)
f_val = f(theta)
c_val = c(theta)
plt.figure(figsize=(12, 12))
plt.plot(theta, f_val, theta, c_val)
plt.show()
```



1 Let's use scipy function to get an idea what we could get

```
In [6]: fun = f
        cons = ({'type': 'ineq', 'fun': lambda x : -c(x)})
        bnds = ((0), (10))

In [7]: res = optimize.minimize(fun, (0.05), method='SLSQP', constraints=cons)

In [8]: res

Out[8]:      fun: 2.5118864012987694
             jac: array([-7.96214297])
         message: 'Optimization terminated successfully.'
            nfev: 118
```

```
     nit: 36
    njev: 36
  status: 0
 success: True
       x: array([0.63095735])
```

2 Penalty method

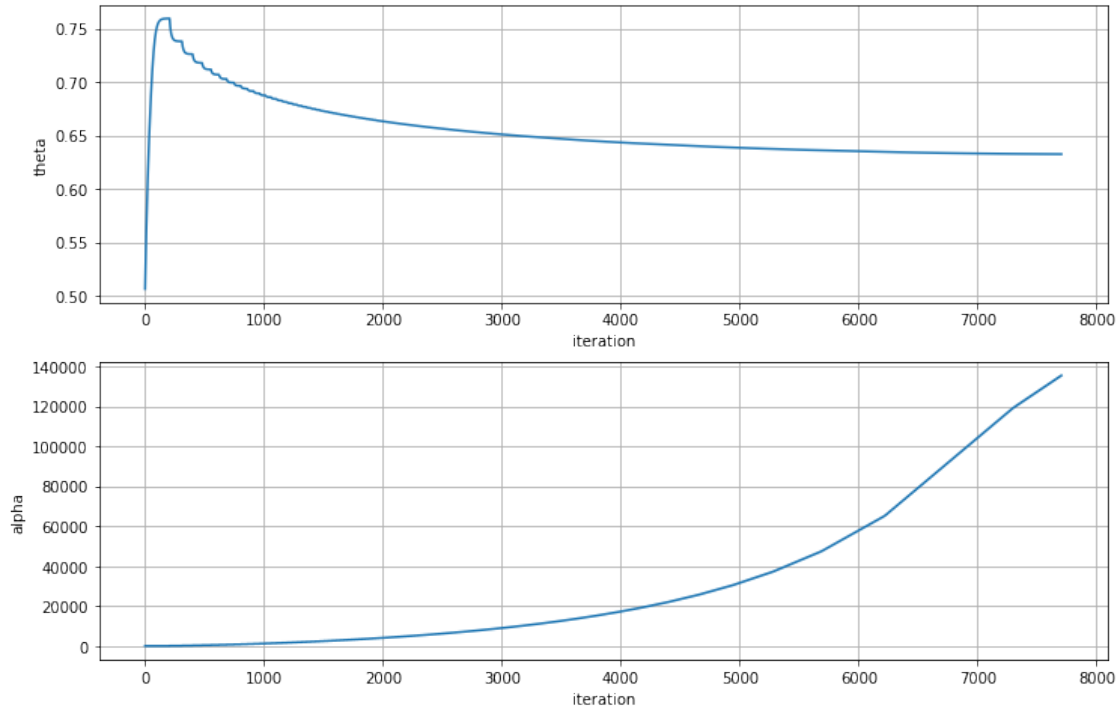
```
In [9]: def obj_func(theta, alpha):
          '''Objective function coupled with the penalty'''
          return 1/(theta**2.0) + (alpha/2)*(np.abs(theta**10 - ALPHA) ** 2)
```

```
In [10]: def obj_drv(theta, alpha):
           '''Derivation of the objective function using finite differences'''
           h = 0.01
           return (obj_func(theta + h, alpha) - obj_func(theta - h, alpha)) / 2*h
```

```
In [11]: theta0 = 0.5 # initial value
         lr = 4.0  # learning rate
         err1 = 0.000001 # condition for the gradient descent
         err2 = 0.000001 # condition for the penalty method termination
         alpha = 100 # initial alpha


         theta = theta0
         thetas = []
         iterations = []
         iteration = 0
         alphas_log = []
         while True:
           cur_theta_outer = theta
           while True:
             prev_theta = theta
             # GD
             theta = theta - lr*obj_drv(theta, alpha=alpha)
             thetas.append(theta)
             alphas_log.append(alpha)
             iterations.append(iteration)
             iteration += 1
             if abs(prev_theta - theta) < err1:
               break

           # Update alpha
           alpha = alpha + 100
           if abs(cur_theta_outer - theta) < err2:
             break
```

```
In [12]: plt.figure(figsize=(12, 8))
         plt.subplot(2, 1, 1)
```

```
plt.plot(iterations, thetas)
plt.ylabel('theta')
plt.xlabel('iteration')
plt.grid()

plt.subplot(2, 1, 2)
plt.plot(iterations, alphas_log)
plt.ylabel('alpha')
plt.xlabel('iteration')
plt.grid()
```



2.0.1 Results for the penalty method

```
In [13]: print(theta, f(theta))
```

```
0.6324816374127435 2.4997936316346596
```

3 Lagrange Duality Method

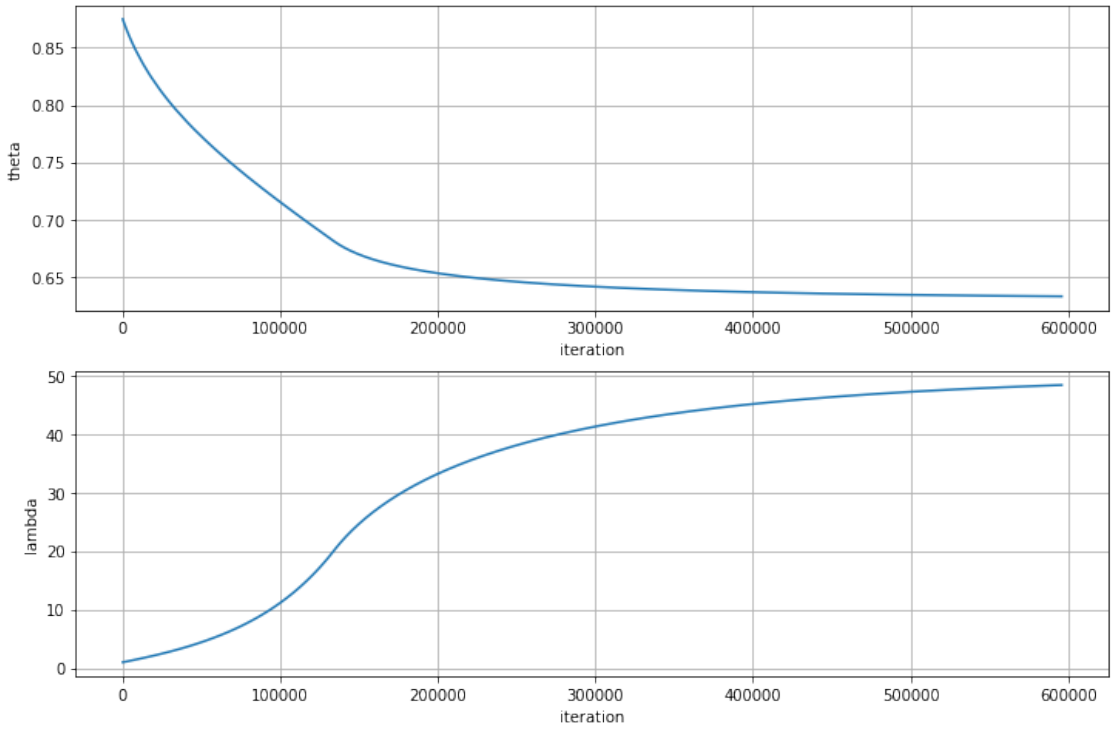
```
In [14]: def obj_func(theta, lmd):
             # Lagrangian
             return 1/(theta**2.0) + lmd*(theta**10-ALPHA)
```

```
        def obj_drv_theta(theta, lmd):
          # Derivative of the lagrangian w.r.t. theta
          h = 0.01
          return (obj_func(theta + h, lmd) - obj_func(theta - h, lmd)) / 2*h

        def obj_drv_lambda(theta, lmd):
          # Derivative of the lagrangian w.r.t. lambda
          h = 0.01
          return (obj_func(theta, lmd + h) - obj_func(theta, lmd - h)) / 2*h
```

In [15]:
```
        theta0 = 0.875 # initial theta
        lmd0 = 1.0 # initial lambda

        lr1 = 100.0 # learning rate for the theta
        lr2 = 300.0 # learning rate for the lambda
        err1 = 0.000001 # termination condition for the theta
        err2 = 0.00000001 # termination condition for the lambda

        thetas = []
        iterations = []
        iteration = 0
        lmd_log = []

        theta = theta0
        lmd = lmd0
        while True:
          cur_theta_outer = theta
          while True:
            prev_theta = theta
            # GD
            theta = theta - lr*obj_drv_theta(theta, lmd=lmd)

            thetas.append(theta)
            lmd_log.append(lmd)
            iterations.append(iteration)
            iteration += 1
            if abs(prev_theta - theta) < err1:
              break

          # Update lambda
          lmd = max(0, lmd + lr2 * obj_drv_lambda(theta, lmd=lmd))
          if abs(cur_theta_outer - theta) < err2:
            break
```

In [16]:
```
        plt.figure(figsize=(12, 8))
        plt.subplot(2, 1, 1)
        plt.plot(iterations, thetas)
        plt.ylabel('theta')
```

```
plt.xlabel('iteration')
plt.grid()

plt.subplot(2, 1, 2)
plt.plot(iterations, lmd_log)
plt.ylabel('lambda')
plt.xlabel('iteration')
plt.grid()
```



3.0.1 Results for the duality method

In [17]: `print(theta, f(theta))`

```
0.6328580249362176 2.496821049149483
```