BOĞAZIÇI UNIVERSITY

NONLINEAR MODELS IN OPERATIONS RESEARCH

IE 440

Homework 5

Authors:

M. Akın Elden

Yunus Emre Karataş

Y. Harun Kıvrıl

Sefa Kayraklık

9 December 2019

Department of Industrial Engineering

Boğaziçi University

1 Introduction

The project is implemented using Python as the programming language. First the given functions and their gradient functions are converted to lambda functions using ”sympy” package.

The source code used to import required dependencies, converting functions to lambda expressions:

```python
import pandas as pd
import numpy as np
from sympy import Symbol, lambdify
import random

x1 = Symbol("x1")
x2 = Symbol("x2")
x3 = Symbol("x3")
x4 = Symbol("x4")

func = 3*x1**2 + 2*x2**2 - 2*x1*x2 - 4*x1 + 2*x2 + 3
f = lambdify([[x1,x2,x3,x4]], func, "numpy")
gf = lambdify([[x1,x2,x3,x4]], func.diff([[x1, x2,x3,x4]]), "numpy")
grad_f = lambda x_arr : np.array(gf(x_arr), 'float64').reshape(1,len(x_arr))

A = np.array([[1, 1, 1, 0],
             [1, 1, 0, 1]])
b = np.array([2,5]).reshape(2,1)
x1_bounds = [2, 5]
x2_bounds = [-1, 6]
x3_bounds = [0, 4]
x4_bounds = [0, 10]
bounds = [x1_bounds, x2_bounds, x3_bounds, x4_bounds]
```

Some useful functions for output table construction:

```python
np_str = lambda x_k : np.array2string(x_k.reshape(len(x_k)), precision=3, separator=',')

f_str = lambda x : "{0:.4f}".format(x)

class OutputTable:
    def __init__(self):
        self.table = pd.DataFrame([],columns=['k', 'x^k', 'f(x^k)', 'd^k', 'a^k', 'x^k+1'])
    def add_row(self, k, xk, fxk, dk, ak, xkp):
        self.table.loc[len(self.table)] = [k, np_str(xk), f_str(np.asscalar(fxk)), np_str(dk), ak, np_str(xkp)]
    def print_latex(self):
        print(self.table.to_latex(index=False))
```

Exact line search algorithm is implemented using Golden Section Method. The source used to implement it:

```
def GoldenSection(f,epsilon=0.005, a=-1000,b=1000):
    golden_ratio = (1+np.sqrt(5))/2
    gama = 1/golden_ratio
    iteration = 0
    x_1 = b - gama*(b-a)
    x_2 = a + gama*(b-a)
    fx_1 = f(x_1)
    fx_2 = f(x_2)
    while (b-a) >= epsilon:
        iteration+=1
        if(fx_1 >= fx_2):
            a = x_1
            x_1 = x_2
            x_2 = a + gama*(b-a)
            fx_1 = fx_2
            fx_2 = f(x_2)
        else:
            b = x_2
            x_2 = x_1
            x_1 = b - gama*(b-a)
            fx_2 = fx_1
            fx_1 = f(x_1)
    x_star = (a+b)/2
    fx_star = f(x_star)
    return x_star

def ExactLineSearch(f, x0, d, eps=0.0000000001):
    alpha = Symbol('alpha')
    function_alpha = f(np.array(x0)+alpha*np.array(d))
    f_alp = lambdify(alpha, function_alpha, 'numpy')
    alp_star = BisectionMethod(f_alp, epsilon=eps)
    return alp_star
```

2 Reduced Gradient Method

The given problem has 2 linear equality constraints and bounds for each variable. Since the variables can be separated as basics and nonbasics, the gradient which is used to compute the descent direction can be reduced. So, reduced gradient method makes use of the advantage of having basic and nonbasic variables in the optimization problem.

First, basic and nonbasic variables are determined; and the directions for the nonbasics are computed using the formula which is derived in the class. After finding the nonbasic directions, the direction of the basic variables are determined by using the nonbasic directions. So, all the directions for the variables are found. Using the exact line search, the step length in the found descent direction is determined. This process is repeated until the norm of the direction vector is smaller than the given ε.

```python
def determineBasicAndNonbasics(xk, bounds, A):
    basics = []
    nonbasics = []
    m = len(A)
    for i in range(len(bounds)):
        if(bounds[i][0] < xk[i,0] < bounds[i][1]):
            basics.append(i)
        else:
            nonbasics.append(i)
    while len(basics) > m:
        random.shuffle(basics)
        new_basics = basics[0:m]
        if np.linalg.det(A[:,new_basics]) == 0: # found set is linearly dependent
            continue
        nonbasics = nonbasics + basics[m:]
        basics = new_basics
    return basics, nonbasics

def ReducedGradient(x0, f=f, gradf=grad_f, eps=0.001, A=A, b=b, bounds=bounds,
    float_prec=6):
    k = 0
    xk = np.array(x0).reshape(len(x0),1)
    output = OutputTable()
    repeat = True
    while(repeat):
        basics, nonbasics = determineBasicAndNonbasics(xk, bounds, A)
        B = A[:,basics]
        N = A[:,nonbasics]
        Binv = np.linalg.inv(B)
        gradfk = gradf(xk)
        gradB = gradfk[:,basics]
        gradN = gradfk[:,nonbasics]
        rNk = gradN - gradB @ Binv @ N
        rBk = 0
        rk = np.zeros((1,len(xk)))
        np.put(rk, nonbasics, rNk) # since rBk is 0, we only put rNk into rk
        dk = np.zeros_like(xk)
        for i in nonbasics:
            if xk[i] == bounds[i][0] and rk[0,i] < 0:
                dk[i,0] = -rk[0,i]
            elif xk[i] == bounds[i][1] and rk[0,i] > 0:
                dk[i,0] = -rk[0,i]
            elif bounds[i][0] < xk[i] < bounds[i][1]:
                dk[i,0] = -rk[0,i]
            else:
                dk[i,0] = 0
        dkB = - Binv @ N @ dk[nonbasics]
        np.put(dk, basics, dkB)
        a_max = 1000 # given upper limit
        for i in range(len(xk)):
            if(dk[i,0] == 0): # max value is infinity if dkj is 0
                continue
            a_max = min(a_max, max((bounds[i]-xk[i])/dk[i,0]))
        ak = ExactLineSearch(f,xk,dk,a_max)
```

```
        output.add_row(k, xk, f(xk), dk, ak)
        xkp = xk + ak * dk
        k += 1
        xk = np.round(xkp, float_prec) # rounding is required since the float calculations can cause precision problem
        if np.linalg.norm(dk) < eps:
            repeat = False
    output.add_row(k, xk, f(xk), np.array([]), None)
    return xk, f(xk).item(), output
```

Solution set 1:

- $x^{(0)}$: [3, -1, 0, 3]
- ε_1 : 0.001

Output of the solution set 1:

k	$x^{(k)}$	$f(x^{(k)})$	$d^{(k)}$	$\alpha^{(k)}$
0	[3,-1, 0, 3]	24.0000	[-40, 24, 16, 16]	0.025000
1	[2. ,-0.4, 0.4, 3.4]	8.1200	[0. , 3.6,-3.6,-3.6]	0.111111
2	[2.,-0., 0., 3.]	7.0000	[0.,0.,0.,0.]	1000.000000
3	[2.,0.,0.,3.]	7.0000	[]	NaN

$$x^* = [2, 0, 0, 3]$$
$$f(x^*) = 7$$

Solution set 2:

- $x^{(0)}$: [2, -1, 1, 4]
- ε_1 : 0.0001

Output of the solution set 2:

k	$x^{(k)}$	$f(x^{(k)})$	$d^{(k)}$	$\alpha^{(k)}$
0	[2,-1, 1, 4]	11.0000	[0, 6,-6,-6]	0.166667
1	[2.,-0., 0., 3.]	7.0000	[0.,0.,0.,0.]	1000.000000
2	[2.,0.,0.,3.]	7.0000	[]	NaN

$$x^* = [2, 0, 0, 3]$$
$$f(x^*) = 7$$

Conclusion:

Two different set of initial points are used to determine the strength of this algorithm. The Reduced Gradient Method quickly converges to the same point in each case.We can concluded that Reduced Gradient Method can be considered well to the problems that the variables of this problem can be separated as basics and non basics.

3 Appendix

The complete source code:

```python
import pandas as pd
import numpy as np
from sympy import Symbol, lambdify
import random


# In[ ]:


x1 = Symbol("x1")
x2 = Symbol("x2")
x3 = Symbol("x3")
x4 = Symbol("x4")

func = 3*x1**2 + 2*x2**2 - 2*x1*x2 - 4*x1 + 2*x2 + 3
f = lambdify([[x1,x2,x3,x4]], func, "numpy")
gf = lambdify([[x1,x2,x3,x4]], func.diff([[x1, x2,x3,x4]]), "numpy")
grad_f = lambda x_arr : np.array(gf(x_arr), 'float64').reshape(1,len(x_arr))

A = np.array([[1, 1, 1, 0],
             [1, 1, 0, 1]])
b = np.array([2,5]).reshape(2,1)
x1_bounds = [2, 5]
x2_bounds = [-1, 6]
x3_bounds = [0, 4]
x4_bounds = [0, 10]
bounds = [x1_bounds, x2_bounds, x3_bounds, x4_bounds]


# ### Useful Functions

# In[ ]:


np_str = lambda x_k : np.array2string(x_k.reshape(len(x_k)), precision=3, separator=','
    )

f_str = lambda x : "{0:.4f}".format(x)


# In[ ]:


class OutputTable:
    def __init__(self):
        self.table = pd.DataFrame([],columns=['k', 'x^k', 'f(x^k)', 'd^k', 'a^k'])
    def add_row(self, k, xk, fxk, dk, ak):
        self.table.loc[len(self.table)] = [k, np_str(xk), f_str(fxk.item()), np_str(dk),
             ak]
    def print_latex(self):
```

```
        print(self.table.to_latex(index=False))


# ### Exact Line Search

# In[ ]:


def GoldenSection(f,epsilon=0.005, a=-1000,b=1000):
    golden_ratio = (1+np.sqrt(5))/2
    gama = 1/golden_ratio
    iteration = 0
    x_1 = b - gama*(b-a)
    x_2 = a + gama*(b-a)
    fx_1 = f(x_1)
    fx_2 = f(x_2)
    while (b-a) >= epsilon:
        iteration+=1
        if(fx_1 >= fx_2):
            a = x_1
            x_1 = x_2
            x_2 = a + gama*(b-a)
            fx_1 = fx_2
            fx_2 = f(x_2)
        else:
            b = x_2
            x_2 = x_1
            x_1 = b - gama*(b-a)
            fx_2 = fx_1
            fx_1 = f(x_1)
    x_star = (a+b)/2
    fx_star = f(x_star)
    return x_star


# In[ ]:

def ExactLineSearch(f, x0, d, a_max, eps=0.0000000001):
    alpha = Symbol('alpha')
    function_alpha = f(np.array(x0)+alpha*np.array(d))
    f_alp = lambdify(alpha, function_alpha, 'numpy')
    alp_star = GoldenSection(f_alp, epsilon=eps, a=0, b=a_max)
    return alp_star


# ## Reduced Gradient Method

# In[ ]:


def determineBasicAndNonbasics(xk, bounds, A):
    basics = []
    nonbasics = []
    m = len(A)
```

```python
    for i in range(len(bounds)):
        if(bounds[i][0] < xk[i,0] < bounds[i][1]):
            basics.append(i)
        else:
            nonbasics.append(i)
    while len(basics) > m:
        random.shuffle(basics)
        new_basics = basics[0:m]
        if np.linalg.det(A[:,new_basics]) == 0: # found set is linearly dependent
            continue
        nonbasics = nonbasics + basics[m:]
        basics = new_basics
    return basics, nonbasics


# In[ ]:


def ReducedGradient(x0, f=f, gradf=grad_f, eps=0.001, A=A, b=b, bounds=bounds,
    float_prec=6):
    k = 0
    xk = np.array(x0).reshape(len(x0),1)
    output = OutputTable()
    repeat = True
    while(repeat):
        basics, nonbasics = determineBasicAndNonbasics(xk, bounds, A)
        B = A[:,basics]
        N = A[:,nonbasics]
        Binv = np.linalg.inv(B)
        gradfk = gradf(xk)
        gradB = gradfk[:,basics]
        gradN = gradfk[:,nonbasics]
        rNk = gradN - gradB @ Binv @ N
        rBk = 0
        rk = np.zeros((1,len(xk)))
        np.put(rk, nonbasics, rNk) # since rBk is 0, we only put rNk into rk
        dk = np.zeros_like(xk)
        for i in nonbasics:
            if xk[i] == bounds[i][0] and rk[0,i] < 0:
                dk[i,0] = -rk[0,i]
            elif xk[i] == bounds[i][1] and rk[0,i] > 0:
                dk[i,0] = -rk[0,i]
            elif bounds[i][0] < xk[i] < bounds[i][1]:
                dk[i,0] = -rk[0,i]
            else:
                dk[i,0] = 0
        dkB = - Binv @ N @ dk[nonbasics]
        np.put(dk, basics, dkB)
        a_max = 1000 # given upper limit
        for i in range(len(xk)):
            if(dk[i,0] == 0): # max value is infinity if dkj is 0
                continue
            a_max = min(a_max, max((bounds[i]-xk[i])/dk[i,0]))
        ak = ExactLineSearch(f,xk,dk,a_max)
```

```python
        output.add_row(k, xk, f(xk), dk, ak)
        xkp = xk + ak * dk
        k += 1
        xk = np.round(xkp, float_prec) # rounding is required since the float calculations can cause precision problem
        if np.linalg.norm(dk) < eps:
            repeat = False
    output.add_row(k, xk, f(xk), np.array([]), None)
    return xk, f(xk).item(), output


# In[ ]:


xk = np.array([3, -1, 0, 3]).reshape(4,1)
xs1, fxs1, out1 = ReducedGradient(xk)
out1.table


# In[ ]:


xk = np.array([2, -1, 1, 4]).reshape(4,1)
xs2, fxs2, out2 = ReducedGradient(xk, eps = 0.0001)
out2.table
```