

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:

```
1 import pandas as pd
2 import numpy as np
3 from sympy import Symbol, lambdify
4 import random
5
6 x1 = Symbol("x1")
7 x2 = Symbol("x2")
8 x3 = Symbol("x3")
9 x4 = Symbol("x4")
10
11 func = 3*x1**2 + 2*x2**2 - 2*x1*x2 - 4*x1 + 2*x2 + 3
12 f = lambdify([[x1,x2,x3,x4]], func, "numpy")
13 gf = lambdify([[x1,x2,x3,x4]], func.diff([[x1, x2,x3,x4]]), "numpy")
14 grad_f = lambda x_arr : np.array(gf(x_arr), 'float64').reshape(1,len(x_arr))
15
16 A = np.array([[1, 1, 1, 0],
17               [1, 1, 0, 1]])
18 b = np.array([2,5]).reshape(2,1)
19 x1_bounds = [2, 5]
20 x2_bounds = [-1, 6]
21 x3_bounds = [0, 4]
22 x4_bounds = [0, 10]
23 bounds = [x1_bounds, x2_bounds, x3_bounds, x4_bounds]
```

Some useful functions for output table construction:

```
1 np_str = lambda x_k : np.array2string(x_k.reshape(len(x_k)), precision=3, separator=',',
2   )
3 f_str = lambda x : "{0:.4f}".format(x)
4
5 class OutputTable:
6     def __init__(self):
7         self.table = pd.DataFrame([],columns=['k', 'x^k', 'f(x^k)', 'd^k', 'a^k', 'x^k+1',
8   ])
9     def add_row(self, k, xk, fxk, dk, ak, xkp):
10         self.table.loc[len(self.table)] = [k, np_str(xk), f_str(np.asscalar(fxk)),
11   np_str(dk), ak, np_str(xkp)]
12     def print_latex(self):
13         print(self.table.to_latex(index=False))
```

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

```

1 def GoldenSection(f,epsilon=0.005, a=-1000,b=1000):
2     golden_ratio = (1+np.sqrt(5))/2
3     gama = 1/golden_ratio
4     iteration = 0
5     x_1 = b - gama*(b-a)
6     x_2 = a + gama*(b-a)
7     fx_1 = f(x_1)
8     fx_2 = f(x_2)
9     while (b-a) >= epsilon:
10         iteration+=1
11         if(fx_1 >= fx_2):
12             a = x_1
13             x_1 = x_2
14             x_2 = a + gama*(b-a)
15             fx_1 = fx_2
16             fx_2 = f(x_2)
17         else:
18             b = x_2
19             x_2 = x_1
20             x_1 = b - gama*(b-a)
21             fx_2 = fx_1
22             fx_1 = f(x_1)
23     x_star = (a+b)/2
24     fx_star = f(x_star)
25     return x_star
26
27 def ExactLineSearch(f, x0, d, eps=0.0000000001):
28     alpha = Symbol('alpha')
29     function_alpha = f(np.array(x0)+alpha*np.array(d))
30     f_alp = lambdify(alpha, function_alpha, 'numpy')
31     alp_star = BisectionMethod(f_alp, epsilon=eps)
32     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 .

```

1 def determineBasicAndNonbasics(xk, bounds, A):
2     basics = []
3     nonbasics = []
4     m = len(A)
5     for i in range(len(bounds)):
6         if(bounds[i][0] < xk[i,0] < bounds[i][1]):
7             basics.append(i)
8         else:
9             nonbasics.append(i)
10    while len(basics) > m:
11        random.shuffle(basics)
12        new_basics = basics[0:m]
13        if np.linalg.det(A[:,new_basics]) == 0: # found set is linearly dependent
14            continue
15        nonbasics = nonbasics + basics[m:]
16        basics = new_basics
17    return basics, nonbasics
18
19 def ReducedGradient(x0, f=f, gradf=grad_f, eps=0.001, A=A, b=b, bounds=bounds,
20 float_prec=6):
21     k = 0
22     xk = np.array(x0).reshape(len(x0),1)
23     output = OutputTable()
24     repeat = True
25     while(repeat):
26         basics, nonbasics = determineBasicAndNonbasics(xk, bounds, A)
27         B = A[:,basics]
28         N = A[:,nonbasics]
29         Binv = np.linalg.inv(B)
30         gradfk = gradf(xk)
31         gradB = gradfk[:,basics]
32         gradN = gradfk[:,nonbasics]
33         rNk = gradN - gradB @ Binv @ N
34         rBk = 0
35         rk = np.zeros((1,len(xk)))
36         np.put(rk, nonbasics, rNk) # since rBk is 0, we only put rNk into rk
37         dk = np.zeros_like(xk)
38         for i in nonbasics:
39             if xk[i] == bounds[i][0] and rk[0,i] < 0:
40                 dk[i,0] = -rk[0,i]
41             elif xk[i] == bounds[i][1] and rk[0,i] > 0:
42                 dk[i,0] = -rk[0,i]
43             elif bounds[i][0] < xk[i] < bounds[i][1]:
44                 dk[i,0] = -rk[0,i]
45             else:
46                 dk[i,0] = 0
47         dkB = - Binv @ N @ dk[nonbasics]
48         np.put(dk, basics, dkB)
49         a_max = 1000 # given upper limit
50         for i in range(len(xk)):
51             if(dk[i,0] == 0): # max value is infinity if dkj is 0
52                 continue
53             a_max = min(a_max, max((bounds[i]-xk[i])/dk[i,0]))
54         ak = ExactLineSearch(f,xk,dk,a_max)

```

```

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

```

Solution set 1:

- $x^{(0)} : [3, -1, 0, 3]$
- $\varepsilon_1 : 0.001$

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:

- $x^{(0)} : [2, -1, 1, 4]$
- $\varepsilon_1 : 0.0001$

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 conclude 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:

```
1 import pandas as pd
2 import numpy as np
3 from sympy import Symbol, lambdify
4 import random
5
6
7 # In[ ]:
8
9
10 x1 = Symbol("x1")
11 x2 = Symbol("x2")
12 x3 = Symbol("x3")
13 x4 = Symbol("x4")
14
15 func = 3*x1**2 + 2*x2**2 - 2*x1*x2 - 4*x1 + 2*x2 + 3
16 f = lambdify([[x1,x2,x3,x4]], func, "numpy")
17 gf = lambdify([[x1,x2,x3,x4]], func.diff([[x1, x2,x3,x4]]), "numpy")
18 grad_f = lambda x_arr : np.array(gf(x_arr), 'float64').reshape(1,len(x_arr))
19
20 A = np.array([[1, 1, 1, 0],
21               [1, 1, 0, 1]])
22 b = np.array([2,5]).reshape(2,1)
23 x1_bounds = [2, 5]
24 x2_bounds = [-1, 6]
25 x3_bounds = [0, 4]
26 x4_bounds = [0, 10]
27 bounds = [x1_bounds, x2_bounds, x3_bounds, x4_bounds]
28
29
30 # ### Useful Functions
31
32 # In[ ]:
33
34
35 np_str = lambda x_k : np.array2string(x_k.reshape(len(x_k)), precision=3, separator=',',
36 )
37
38 f_str = lambda x : "{0:.4f}".format(x)
39
40 # In[ ]:
41
42
43 class OutputTable:
44     def __init__(self):
45         self.table = pd.DataFrame([],columns=['k', 'x^k', 'f(x^k)', 'd^k', 'a^k'])
46     def add_row(self, k, xk, fxk, dk, ak):
47         self.table.loc[len(self.table)] = [k, np_str(xk), f_str(fxk.item()), np_str(dk),
48         ak]
49     def print_latex(self):
```

```

49     print(self.table.to_latex(index=False))
50
51
52 # ### Exact Line Search
53
54 # In[ ]:
55
56
57 def GoldenSection(f,epsilon=0.005, a=-1000,b=1000):
58     golden_ratio = (1+np.sqrt(5))/2
59     gama = 1/golden_ratio
60     iteration = 0
61     x_1 = b - gama*(b-a)
62     x_2 = a + gama*(b-a)
63     fx_1 = f(x_1)
64     fx_2 = f(x_2)
65     while (b-a) >= epsilon:
66         iteration+=1
67         if(fx_1 >= fx_2):
68             a = x_1
69             x_1 = x_2
70             x_2 = a + gama*(b-a)
71             fx_1 = fx_2
72             fx_2 = f(x_2)
73         else:
74             b = x_2
75             x_2 = x_1
76             x_1 = b - gama*(b-a)
77             fx_2 = fx_1
78             fx_1 = f(x_1)
79     x_star = (a+b)/2
80     fx_star = f(x_star)
81     return x_star
82
83
84 # In[ ]:
85
86 def ExactLineSearch(f, x0, d, a_max, eps=0.0000000001):
87     alpha = Symbol('alpha')
88     function_alpha = f(np.array(x0)+alpha*np.array(d))
89     f_alp = lambdify(alpha, function_alpha, 'numpy')
90     alp_star = GoldenSection(f_alp, epsilon=eps, a=0, b=a_max)
91     return alp_star
92
93
94 # ### Reduced Gradient Method
95
96 # In[ ]:
97
98
99 def determineBasicAndNonbasics(xk, bounds, A):
100     basics = []
101     nonbasics = []
102     m = len(A)

```

```

103     for i in range(len(bounds)):
104         if(bounds[i][0] < xk[i,0] < bounds[i][1]):
105             basics.append(i)
106         else:
107             nonbasics.append(i)
108     while len(basics) > m:
109         random.shuffle(basics)
110         new_basics = basics[0:m]
111         if np.linalg.det(A[:,new_basics]) == 0: # found set is linearly dependent
112             continue
113         nonbasics = nonbasics + basics[m:]
114         basics = new_basics
115     return basics, nonbasics
116
117
118 # In[ ]:
119
120
121 def ReducedGradient(x0, f=f, gradf=grad_f, eps=0.001, A=A, b=b, bounds=bounds,
122     float_prec=6):
123     k = 0
124     xk = np.array(x0).reshape(len(x0),1)
125     output = OutputTable()
126     repeat = True
127     while(repeat):
128         basics, nonbasics = determineBasicAndNonbasics(xk, bounds, A)
129         B = A[:,basics]
130         N = A[:,nonbasics]
131         Binv = np.linalg.inv(B)
132         gradfk = gradf(xk)
133         gradB = gradfk[:,basics]
134         gradN = gradfk[:,nonbasics]
135         rNk = gradN - gradB @ Binv @ N
136         rBk = 0
137         rk = np.zeros((1,len(xk)))
138         np.put(rk, nonbasics, rNk) # since rBk is 0, we only put rNk into rk
139         dk = np.zeros_like(xk)
140         for i in nonbasics:
141             if xk[i] == bounds[i][0] and rk[0,i] < 0:
142                 dk[i,0] = -rk[0,i]
143             elif xk[i] == bounds[i][1] and rk[0,i] > 0:
144                 dk[i,0] = -rk[0,i]
145             elif bounds[i][0] < xk[i] < bounds[i][1]:
146                 dk[i,0] = -rk[0,i]
147             else:
148                 dk[i,0] = 0
149         dkB = - Binv @ N @ dk[nonbasics]
150         np.put(dk, basics, dkB)
151         a_max = 1000 # given upper limit
152         for i in range(len(xk)):
153             if(dk[i,0] == 0): # max value is infinity if dkj is 0
154                 continue
155             a_max = min(a_max, max((bounds[i]-xk[i])/dk[i,0]))
156         ak = ExactLineSearch(f,xk,dk,a_max)

```



```

156     output.add_row(k, xk, f(xk), dk, ak)
157     xkp = xk + ak * dk
158     k += 1
159     xk = np.round(xkp, float_prec) # rounding is required since the float
        calculations can cause precision problem
160     if np.linalg.norm(dk) < eps:
161         repeat = False
162     output.add_row(k, xk, f(xk), np.array([]), None)
163     return xk, f(xk).item(), output
164
165
166 # In[ ]:
167
168
169 xk = np.array([3, -1, 0, 3]).reshape(4,1)
170 xs1, fxs1, out1 = ReducedGradient(xk)
171 out1.table
172
173
174 # In[ ]:
175
176
177 xk = np.array([2, -1, 1, 4]).reshape(4,1)
178 xs2, fxs2, out2 = ReducedGradient(xk, eps = 0.0001)
179 out2.table

```