HW4_Question_7

March 11, 2018

Group: Lawrence Moore, Austin Wang, Joe Higgins

0.1 Introduction

For this question, we implement the ADMM algorithm and explore the behavior with various betas and objective functions.

0.2 Part a)

```
In [1]: import numpy as np
```

```
In [2]: def M_beta(beta):
            LHM = np.matrix([
                [3*beta, 0,      0,      0, 0, 0],
                [4*beta, 6*beta, 0,      0, 0, 0],
                [5*beta, 7*beta, 9*beta, 0, 0, 0],
                [1*beta, 1*beta, 1*beta, 1, 0, 0],
                [1*beta, 1*beta, 2*beta, 0, 1, 0],
                [1*beta, 2*beta, 2*beta, 0, 0, 1],
            ])

            RHM = np.matrix([
                [0, -4*beta, -5*beta, 1, 1, 1],
                [0,       0, -7*beta, 1, 1, 2],
                [0,       0,       0, 1, 2, 2],
                [0,       0,       0, 1, 0, 0],
                [0,       0,       0, 0, 1, 0],
                [0,       0,       0, 0, 0, 1],
            ])

            return (np.linalg.inv(LHM) * RHM)[1:,1:]
```

```
In [3]: def find_optimal(M):
            xy = np.matrix([[1], [2], [3], [4], [5]])
            xy_k = xy
            check = 10**6
            max_iter = 1000
            k = 0
```

```
            np.linalg.eig(M)

            while(check > 10**-8 and k < max_iter):
                k += 1
                xy_k1 = M * xy_k
                check = np.linalg.norm(xy_k1 - xy_k)
                xy_k = xy_k1
            return xy_k, k
```

In [4]:
```
optimal_beta_p1, num_iter1 = find_optimal(M_beta(.1))
optimal_beta_1, num_iter2 = find_optimal(M_beta(1))
optimal_beta_10, num_iter3 = find_optimal(M_beta(10))
print("Optimal value")
print([0, 0, 0])
print("Number of Iterations for beta = 0.1")
print(num_iter1)
print("Optimal for beta = 0.1")
print(optimal_beta_p1[0:3,:])
print("Number of Iterations for beta = 1")
print(num_iter2)
print("Optimal for beta = 1")
print(optimal_beta_1[0:3,:])
print("Number of Iterations for beta = 10")
print(num_iter3)
print("Optimal for beta = 10")
print(optimal_beta_10[0:3,:])
```

```
Optimal value
[0, 0, 0]
Number of Iterations for beta = 0.1
1000
Optimal for beta = 0.1
[[ 2.37133293e+12]
 [-3.28160297e+12]
 [-3.67229831e+11]]
Number of Iterations for beta = 1
1000
Optimal for beta = 1
[[ 1.71634730e+12]
 [-8.37461739e+11]
 [-3.16129952e+12]]
Number of Iterations for beta = 10
1000
Optimal for beta = 10
[[ 1.65084874e+12]
 [-5.93047616e+11]
 [-3.11019964e+13]]
```

As we can see above, the algorithm never converges for any beta. The lack of influence of beta can be easily explained by the fact that M does not change (within rounding error), as shown below. This means that the spectral radius always remains above 1.

```
In [5]: print("M for beta .1")
        print(M_beta(.1))
        print("M for beta 1")
        print(M_beta(1))
        print("M for beta 10")
        print(M_beta(10))
        print("Spectral Radius: ")
        print(abs(max(np.linalg.eig(M_beta(1))[0])))
```

```
M for beta .1
[[ 0.88888889 -0.05555556 -0.55555556 -0.55555556  1.11111111]
 [ 0.04938272  0.9691358  -0.30864198  0.80246914 -0.49382716]
 [ 0.03950617  0.07530864  0.75308642 -0.35802469 -0.39506173]
 [ 0.0345679  -0.02160494 -0.21604938  0.5617284  -0.34567901]
 [-0.05432099 -0.01604938 -0.16049383 -0.38271605  0.54320988]]
M for beta 1
[[ 0.88888889 -0.05555556 -0.05555556 -0.05555556  0.11111111]
 [ 0.04938272  0.9691358  -0.0308642   0.08024691 -0.04938272]
 [ 0.39506173  0.75308642  0.75308642 -0.35802469 -0.39506173]
 [ 0.34567901 -0.21604938 -0.21604938  0.5617284  -0.34567901]
 [-0.54320988 -0.16049383 -0.16049383 -0.38271605  0.54320988]]
M for beta 10
[[ 8.88888889e-01 -5.55555556e-02 -5.55555556e-03 -5.55555556e-03
   1.11111111e-02]
 [ 4.93827160e-02  9.69135802e-01 -3.08641975e-03  8.02469136e-03
  -4.93827160e-03]
 [ 3.95061728e+00  7.53086420e+00  7.53086420e-01 -3.58024691e-01
  -3.95061728e-01]
 [ 3.45679012e+00 -2.16049383e+00 -2.16049383e-01  5.61728395e-01
  -3.45679012e-01]
 [-5.43209877e+00 -1.60493827e+00 -1.60493827e-01 -3.82716049e-01
   5.43209877e-01]]
Spectral Radius:
1.027839303276622
```

0.3 Part b)

```
In [6]: def M_beta(beta):

            LHM = np.matrix([
                [3*beta+1, 0,        0,        0, 0, 0],
                [4*beta,   6*beta+1, 0,        0, 0, 0],
                [5*beta,   7*beta,   9*beta+1, 0, 0, 0],
```

```
            [1*beta,   1*beta,   1*beta,   1, 0, 0],
            [1*beta,   1*beta,   2*beta,   0, 1, 0],
            [1*beta,   2*beta,   2*beta,   0, 0, 1],
        ])

        RHM = np.matrix([
            [0, -4*beta, -5*beta, 1, 1, 1],
            [0,       0, -7*beta, 1, 1, 2],
            [0,       0,       0, 1, 2, 2],
            [0,       0,       0, 1, 0, 0],
            [0,       0,       0, 0, 1, 0],
            [0,       0,       0, 0, 0, 1],
        ])

        return (np.linalg.inv(LHM) * RHM)[1:,1:]
```

In [7]:
```
optimal_beta_p1, num_iter1 = find_optimal(M_beta(.1))
optimal_beta_1, num_iter2 = find_optimal(M_beta(1))
optimal_beta_10, num_iter3 = find_optimal(M_beta(10))
print("Optimal value")
print([0, 0, 0])
print("Number of Iterations for beta = 0.1")
print(num_iter1)
print("Optimal for beta = 0.1")
print(optimal_beta_p1[0:3,:])
print("Number of Iterations for beta = 1")
print(num_iter2)
print("Optimal for beta = 1")
print(optimal_beta_1[0:3,:])
print("Number of Iterations for beta = 10")
print(num_iter3)
print("Optimal for beta = 10")
print(optimal_beta_10[0:3,:])
```

```
Optimal value
[0, 0, 0]
Number of Iterations for beta = 0.1
775
Optimal for beta = 0.1
[[-2.93033503e-08]
 [-9.89961652e-08]
 [ 5.28840640e-07]]
Number of Iterations for beta = 1
127
Optimal for beta = 1
[[ 1.05245707e-08]
 [-4.67365615e-09]
 [-9.68404504e-09]]
```

```
Number of Iterations for beta = 10
1000
Optimal for beta = 10
[[  8559.14743958]
 [-12400.66457325]
 [-96080.2342266 ]]
```

From above, we can see that we now converge for beta equal to 0.1 and 1, though not for 10. This is because the beta actually changes the matrix of transformation, with the spectral radius less than 1 as shown below.

```
In [8]: print("M for beta .1")
        print(M_beta(.1))
        print("M for beta 1")
        print(M_beta(1))
        print("M for beta 10")
        print(M_beta(10))
        print("Spectral Radius: ")
        print(abs(max(np.linalg.eig(M_beta(1))[0])))
```

```
M for beta .1
[[ 0.07692308 -0.34134615  0.43269231  0.43269231  1.05769231]
 [ 0.05263158  0.22697368  0.16447368  0.69078947  0.46052632]
 [ 0.01781377  0.04989879  0.86336032 -0.18927126 -0.22874494]
 [ 0.01255061  0.02720142 -0.15308704  0.7416498  -0.27479757]
 [ 0.0048583   0.06133603 -0.19635628 -0.30161943  0.6194332 ]]
M for beta 1
[[ 0.57142857 -0.28571429  0.          0.          0.14285714]
 [ 0.1         0.825      -0.025       0.075      -0.025     ]
 [ 0.32857143  0.71071429  0.775      -0.325      -0.36785714]
 [ 0.22857143 -0.11428571 -0.2         0.6        -0.34285714]
 [-0.34285714  0.17142857 -0.2        -0.4         0.51428571]]
M for beta 10
[[ 8.46113168e-01 -8.98995241e-02 -4.75938657e-03 -4.75938657e-03
   1.16340561e-02]
 [ 5.81121681e-02  9.55364044e-01 -3.07413369e-03  7.91487730e-03
  -4.69546318e-03]
 [ 3.86097245e+00  7.47438706e+00  7.55754557e-01 -3.54135552e-01
  -3.91966574e-01]
 [ 3.27985077e+00 -2.07925337e+00 -2.13504106e-01  5.66715675e-01
  -3.45011942e-01]
 [-5.18128091e+00 -1.18025813e+00 -1.65910240e-01 -3.85690460e-01
   5.38647497e-01]]
Spectral Radius:
0.857648347684974
```

0.4 Part c)

In [9]:
```
# For function in part a
def M_beta(beta, permutation):

    LHM = np.matrix([
        [3*beta,   4*beta,   5*beta,   0, 0, 0],
        [4*beta,   6*beta,   7*beta,   0, 0, 0],
        [5*beta,   7*beta,   9*beta,   0, 0, 0],
        [1*beta,   1*beta,   1*beta,   1, 0, 0],
        [1*beta,   1*beta,   2*beta,   0, 1, 0],
        [1*beta,   2*beta,   2*beta,   0, 0, 1],
    ])

    RHM = np.matrix([
        [0, 0, 0, 1, 1, 1],
        [0, 0, 0, 1, 1, 2],
        [0, 0, 0, 1, 2, 2],
        [0, 0, 0, 1, 0, 0],
        [0, 0, 0, 0, 1, 0],
        [0, 0, 0, 0, 0, 1],
    ])

    not_visited = set(permutation)
    for i in permutation:
        not_visited.remove(i)

        for j in not_visited:
            RHM[i,j] = -1*LHM[i,j]
            LHM[i,j] = 0

    return (np.linalg.inv(LHM) * RHM)

xy = np.matrix([[1], [2], [3], [4], [5], [6]])
xy_k = xy
check = 10**6
max_iter = 2000
k = 0

while(check > 10**-8 and k < max_iter):
    perm = np.random.permutation([0, 1, 2])
    M = M_beta(1, perm)
    k += 1
    xy_k1 = M * xy_k
    check = np.linalg.norm(xy_k1 - xy_k)
    xy_k = xy_k1
print("For Part a) Function")
print("Number of iterations")
```

```
        print(k)
        print("Optimal value:")
        print([0, 0, 0])
        print("Generated solution:")
        print(xy_k[0:3,:])
```

```
For Part a) Function
Number of iterations
984
Optimal value:
[0, 0, 0]
Generated solution:
[[-3.94681930e-09]
 [-1.11649710e-08]
 [ 1.08765438e-08]]
```

By permuting the order, we do indeed converge for the function in part a).

```
In [10]: # For function in part b
         def M_beta(beta, permutation):

             LHM = np.matrix([
                 [3*beta+1, 0,         0,         0, 0, 0],
                 [4*beta,   6*beta+1, 0,         0, 0, 0],
                 [5*beta,   7*beta,   9*beta+1, 0, 0, 0],
                 [1*beta,   1*beta,   1*beta,   1, 0, 0],
                 [1*beta,   1*beta,   2*beta,   0, 1, 0],
                 [1*beta,   2*beta,   2*beta,   0, 0, 1],
             ])

             RHM = np.matrix([
                 [0, 0, 0, 1, 1, 1],
                 [0, 0, 0, 1, 1, 2],
                 [0, 0, 0, 1, 2, 2],
                 [0, 0, 0, 1, 0, 0],
                 [0, 0, 0, 0, 1, 0],
                 [0, 0, 0, 0, 0, 1],
             ])

             not_visited = set(permutation)
             for i in permutation:
                 not_visited.remove(i)

                 for j in not_visited:
                     RHM[i,j] = -1*LHM[i,j]
                     LHM[i,j] = 0
```

```
            return (np.linalg.inv(LHM) * RHM)

        xy = np.matrix([[1], [2], [3], [4], [5], [6]])
        xy_k = xy
        check = 10**6
        max_iter = 2000
        k = 0

        while(check > 10**-8 and k < max_iter):
            perm = np.random.permutation([0, 1, 2])
            M = M_beta(1, perm)
            k += 1
            xy_k1 = M * xy_k
            check = np.linalg.norm(xy_k1 - xy_k)
            xy_k = xy_k1
        print("For Part b) Function")
        print("Number of iterations")
        print(k)
        print("Optimal value:")
        print([0, 0, 0])
        print("Generated solution:")
        print(xy_k[0:3,:])
```

```
For Part b) Function
Number of iterations
1553
Optimal value:
[0, 0, 0]
Generated solution:
[[-8.65723377e-09]
 [-6.32547709e-09]
 [ 9.90424942e-09]]
```

Again, by permuting the order, we do converge for the function in part b).

0.5 Part d)

```
In [11]: top_identity = np.eye(10, 6)
         bottom_identity = np.eye(10, 6, k=-4)
         A_1 = A_2 = A_3 = A_4 = top_identity
         A_5 = bottom_identity
         A = np.hstack([A_1, A_2, A_3, A_4, A_5])

         # Q = identity of size 30 by 30
         # objective = 1/2 * xT * Q * x
         x = np.random.rand(30,1)
         b = np.ones((10, 1))
```

```
        y = np.random.rand(10,1)
        beta = 1
```

```
In [12]: def update(cols_to_pick):
             A_chunk = A[0:10, cols_to_pick]
             LHS = beta * np.dot(np.transpose(A_chunk), A_chunk) + np.identity(6)
             dat_sum = np.dot(A, x) - np.dot(A_chunk, x[cols_to_pick])
             RHS = np.dot(np.transpose(A_chunk), y) - beta * \
                   np.dot(np.transpose(A_chunk), dat_sum)
             x[cols_to_pick] = np.linalg.solve(LHS, RHS)
```

```
In [13]: check = 10**6
         max_iter = 10000
         k = 0

         while(check > 10**-6 and k < max_iter):
             k += 1
             perm = range(0, 5)
             old_x = x.copy()
             for index in perm:
                 cols_to_pick = range(index*6, index*6+6)
                 update(cols_to_pick)
             y = y - beta * (np.dot(A, x) - b)
             check = np.linalg.norm(x - old_x)
         print("Number of iterations:")
         print(k)
         print("Real solution for generic ADMM:")
         print(b)
         print("Generated Result:")
         print(np.dot(A, x))
```

```
Number of iterations:
22
Real solution for generic ADMM:
[[1.]
 [1.]
 [1.]
 [1.]
 [1.]
 [1.]
 [1.]
 [1.]
 [1.]
 [1.]]
Generated Result:
[[0.99999993]
 [1.00000001]
 [0.99999992]
```

```
 [0.99999995]
 [1.00000019]
 [1.00000006]
 [0.99999955]
 [0.99999974]
 [0.99999966]
 [0.99999962]]
```

For the simple ADMM implementation, our solution converges.

```
In [14]: top_identity = np.eye(10, 6)
         bottom_identity = np.eye(10, 6, k=-4)
         A_1 = A_2 = A_3 = A_4 = top_identity
         A_5 = bottom_identity
         A = np.hstack([A_1, A_2, A_3, A_4, A_5])

         # Q = identity of size 30 by 30
         # objective = 1/2 * xT * Q * x
         x = np.random.rand(30,1)
         b = np.ones((10, 1))
         y = np.random.rand(10,1)
         beta = 1

In [15]: check = 10**6
         max_iter = 10000
         k = 0

         while(check > 10**-6 and k < max_iter):
             k += 1
             old_x = x.copy()
             perm = np.random.permutation(range(0, 5))
             for index in perm: # generate index from the random permutation
                 cols_to_pick = range(index*6, index*6+6)
                 update(cols_to_pick)
             y = y - beta * (np.dot(A, x) - b)
             check = np.linalg.norm(x - old_x)
         print("Number of iterations:")
         print(k)
         print("Real solution with random block update:")
         print(b)
         print("Generated Result:")
         print(np.dot(A, x))
```

```
Number of iterations:
28
Real solution with random block update:
[[1.]
 [1.]
```

```
 [1.]
 [1.]
 [1.]
 [1.]
 [1.]
 [1.]
 [1.]
 [1.]]
Generated Result:
[[1.00000005]
 [1.00000002]
 [1.00000005]
 [1.        ]
 [1.00000027]
 [1.00000033]
 [0.99999999]
 [0.99999999]
 [0.99999999]
 [1.        ]]
```

The algorithm still converges when choosing random blocks, with roughly the same speed.

```
In [16]: top_identity = np.eye(10, 6)
         bottom_identity = np.eye(10, 6, k=-4)
         A_1 = A_2 = A_3 = A_4 = top_identity
         A_5 = bottom_identity
         A = np.hstack([A_1, A_2, A_3, A_4, A_5])

         # Q = identity of size 30 by 30
         # objective = 1/2 * xT * Q * x
         x = np.random.rand(30,1)
         b = np.ones((10, 1))
         y = np.random.rand(10,1)
         beta = 1

In [17]: check = 10**6
         max_iter = 10000
         k = 0

         big_perm = np.random.permutation(range(0, 30))
         while(check > 10**-6 and k < max_iter):
             k += 1
             old_x = x.copy()
             for index in perm:
                 cols_to_pick = range(index*6, index*6+6)
                 update(big_perm[cols_to_pick]) # choose these columns from big perm
             y = y - beta * (np.dot(A, x) - b)
```

```
            check = np.linalg.norm(x - old_x)
        print("Number of iterations:")
        print(k)
        print("Real solution with random sampling update:")
        print(b)
        print("Generated Result:")
        print(np.dot(A, x))
```

```
Number of iterations:
22
Real solution with random sampling update:
[[1.]
 [1.]
 [1.]
 [1.]
 [1.]
 [1.]
 [1.]
 [1.]
 [1.]
 [1.]]
Generated Result:
[[1.        ]
 [0.99999996]
 [1.00000004]
 [1.00000005]
 [1.00000007]
 [0.99999993]
 [0.99999958]
 [0.99999961]
 [0.99999971]
 [0.99999972]]
```

The algorithm still converges when choosing a random sample of columns to update, with slightly fewer iterations than in the random block update.