Solution to Assignment 4

I will appreciate it if you could give me some advice on my assignment!

December 26, 2018

phd exercises

1. Suppose that D^0 is *symmetric* and we apply **SR1** method to solve the positive definite quadratic problem $f(x) = \frac{1}{2}x^{\mathrm{T}}Qx - b^{\mathrm{T}}x$, i.e., D^k is updated according to the formula

$$D^{k+1} = D^k + \frac{(y^k)(y^k)^{\mathrm{T}}}{\langle q^k, y^k \rangle}, \tag{1}$$

where $y^k = p^k - D^k q^k$.

(a) Show that we have

$$D^{k+1}q^i = p^i, \quad \text{for all } k \text{ and } i \leq k, \tag{2}$$

(b) Conclude that for a positive definte quadratic problem, after n steps for which n linearly independent increments $q^0, q^1, \dots, q^{n-1}$ are obtained, D^n is equal to the *inverse Hessian* of the cost function.

Proof. Notations:

- D^k: Approximation of the inverse of Hessain matrix $[\nabla^2 f(x^k)]^{-1}$ at kth iteration
- p^k: difference between argument at kth iteration, i.e., $p^k = x^{k+1} - x^k$
- q^k: difference between gradient of function at kth iteration, i.e., $q^k = \nabla f(x^{k+1}) - \nabla f(x^k)$

(a) We apply induction on k to show this formula.

- As $k = 0$, as the *secant equation* (2.43) in the textbook has to be satsified, i.e.,

$$D^{k+1}(\nabla f^{k+1} - \nabla f^k) = x^{k+1} - x^k,$$

  and thus taking $k = 0$ we obtain the desired formula $D^1 q^0 = p^0$.
- Then we assume (2) holds for some value $k > 1$, and show that it also holds for $k+1$. (i.e., given $D^{k+1}q^i = p^i$ with $i \leq k$, we aim to show $D^{k+2}q^i = p^i$ with $i \leq k+1$)
  (i) First consider the term $\langle p^{k+1} - D^{k+1}q^{k+1}, q^i \rangle$ for $i \leq k$:

$$\begin{aligned}
\langle p^{k+1} - D^{k+1}q^{k+1}, q^i \rangle &= \langle p^{k+1}, q^i \rangle - \langle q^{k+1}, (D^{k+1})^{\mathrm{T}} q^i \rangle && (3)\\
&= \langle p^{k+1}, q^i \rangle - \langle q^{k+1}, D^{k+1} q^i \rangle && (4)\\
&= \langle p^{k+1}, q^i \rangle - \langle q^{k+1}, p^i \rangle && (5)\\
&= \langle p^{k+1}, Qp^i \rangle - \langle Qp^{k+1}, p^i \rangle && (6)\\
&= 0, && (7)
\end{aligned}$$

where (5) is obtained by applying the hypothesis $D^{k+1}q^i = p^i$; and (6) is obtained by computing the formula $q^j = \nabla f(x^{j+1}) - \nabla f(x^j) = Qx^{j+1} - Qx^j = Q(x^{j+1} - x^j) = Qp^j$.

(ii) Thus we can derive the formula for $D^{k+2}q^i$:

$$\begin{aligned} D^{k+2}q^i &= D^{k+1}q^i + \frac{(y^{k+1})(y^{k+1})^{\mathrm{T}}}{\langle q^{k+1}, y^{k+1}\rangle} q^i && (8) \\ &= D^{k+1}q^i + \frac{y^{k+1}\langle y^{k+1}, q^i\rangle}{\langle q^{k+1}, y^{k+1}\rangle} && (9) \\ &= D^{k+1}q^i + \frac{y^{k+1}\langle p^{k+1} - D^{k+1}q^{k+1}, q^i\rangle}{\langle q^{k+1}, y^{k+1}\rangle} && (10) \\ &= 0, && (11) \end{aligned}$$

where (8) is due to the update formula (1); (10) is due to the formula $y^k = p^k - D^k q^k$; (11) is due to the formula in (i)

Combining (i) and (ii), the proof in (a) is complete.

(b) If this algorithm is performed n steps and $q^0, \ldots, q^{n-1}$ are linearly independent, taking $k = n-1$ at (2),

$$p^i = D^n q^i = D^n Q p^i, \quad i = 0, 1, \ldots, n-1 \tag{12}$$

Note that here $Q := H$ is the Hessian matrix of the cost function. Thus we re-write (12) as:

$$D^n QP = P \Longleftrightarrow (D^n Q - I)P = 0,$$

where $P := \begin{pmatrix}(p^0)^{\mathrm{T}} & (p^1)^{\mathrm{T}} & \cdots & (p^{n-1})^{\mathrm{T}}\end{pmatrix}^{\mathrm{T}}$ is nonsingular due to the linear independence of q^j's and $p^j = Q^{-1}q^j$ for $j = 0, \ldots, n-1$. It follows that

$$D^n Q - I = 0 \implies D^n = Q^{-1},$$

i.e., D^n is equal to the *inverse Hessian* of the cost function.

$\square$

Project 1: Gauss-Newton Method for Truncated SVD

A copy of my Code

```
function [X, iter] = myGN(A,X0,tol,maxiter)
% Input:
%        A: given matrix
%       X0: initial guess
%      tol: tolerance
%  maxiter: maximum iterations
%Output:
%        X: solution to the opt
%     iter: number of iterations

k = size(X0,2);
I = speye(k);
for iter = 1:maxiter
    M = X0'*X0;
    Y = X0/M;
    Z = A*Y;
    X = Z - X0 * ((Y'*Z-I)/2);
    if norm(X - X0,'fro')^2 <= tol^2 * trace(M),break;end
    X0 = X;
end
end
```

Matlab screen printout

```
Command Window
>> test_panda
Image 1 or 2: 2
image size: m = 2848, n = 4272
m = 2848, n = 4272, k = 128, tol = 5e-03

SVDs ... Elapsed time is 5.399176 seconds.
yzGN ... Elapsed time is 0.898560 seconds.
myGN ... Elapsed time is 0.880033 seconds.

SVDs    : fM = 6.8525e-04, rU = 2.0068e-15
iter  17: fM = 6.9761e-04, rU = 2.5968e-05
iter  17: fM = 6.9708e-04, rU = 2.5986e-05

>>
```

Figure 1: Matlab screen printout from Image #2

Figure generated from run 2

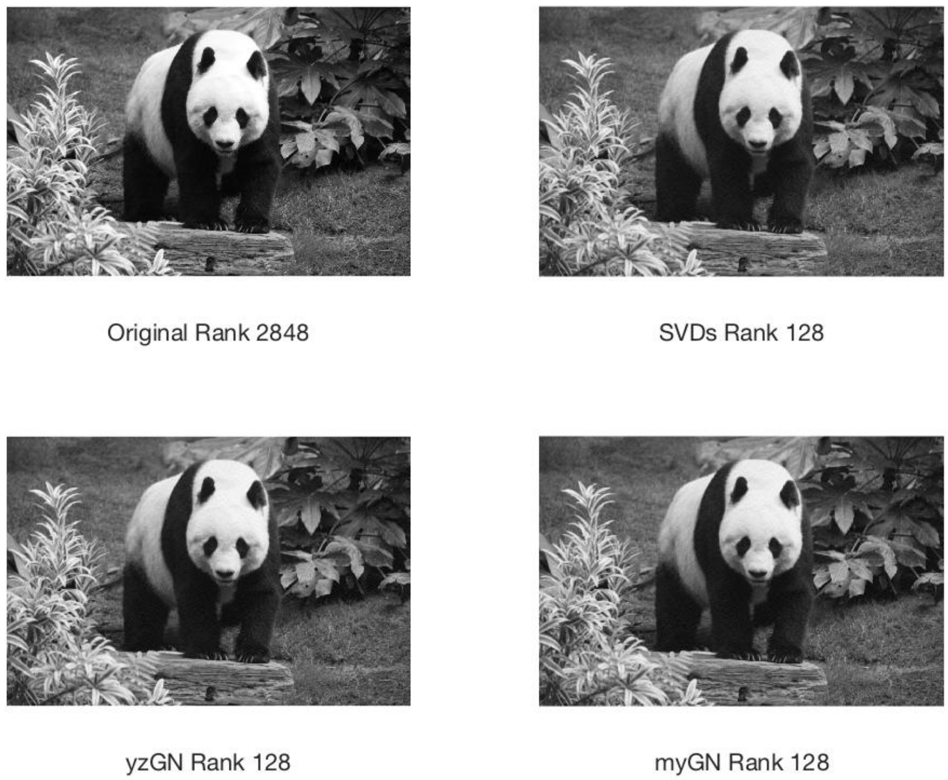

Figure 2: Figure generated from Image #2

A short summary

Introduction This project aims to apply Gauss-Newton method to compute a symmetric rank product XX^{T} that is closest to A in Frobenius norm. The process is just two-line MATLAB code:

$$\begin{aligned} Y &\leftarrow X^k((X^k)^{\mathrm{T}}X^k)^{-1} \\ X^{k+1} &\leftarrow AY - X^k(Y^{\mathrm{T}}AY - I)/2 \end{aligned}$$

Although the process is simple, we still need to care about the arrangement of computation, otherwise our code will have long and unnecessary computing time.

Do Avoid repeated computation

1. For example, the update of X^{k+1} requires twice computation of AY, so it's better to compute this value and save it using a new variable.
2. Another example is that the stopping criteria requires the value of norm(X), i.e., $\sqrt{\mathrm{trace}(X^{\mathrm{T}}X)}$, and the update for Y also requires the computation of $X^{\mathrm{T}}X$, so we should compute this value in advance and save it using a new variable.
3. Moreover, computing $X * [(Y^{\mathrm{T}}AY - I)/2]$ is faster than computing $[X(Y^{\mathrm{T}}AY - I)]/2$, since the former only requires the division operation on a smaller size matrix.

Save necessity before iteration The update of X requires the call of large scale identity for each iteration, which is time-consuming. So we should save the identiy matrix in sparse form *before* iteration.

Project 2: Solving LP Barrier Systems by Newtons Method

Deviations

The Newton's method to the Barrier system gives:

$$\begin{pmatrix} x^{k+1} \\ y^{k+1} \\ z^{k+1} \end{pmatrix} = \begin{pmatrix} x^{k} \\ y^{k} \\ z^{k} \end{pmatrix} - [F'_\mu(x,y,z)]^{-1} F_\mu \implies \begin{pmatrix} dx \\ dy \\ dz \end{pmatrix} = -[F'_\mu(x,y,z)]^{-1} F_\mu$$

Leftmultiplying with $F'_\mu(x,y,z)$, we derive:

$$F'_\mu(x,y,z) \begin{pmatrix} dx \\ dy \\ dz \end{pmatrix} = -F_\mu := \begin{pmatrix} r_d \\ r_p \\ r_c \end{pmatrix}$$

We set

$$\begin{aligned} f_1 &:= A^{\mathrm{T}} y + z - c \\ f_2 &:= Ax - b \\ f_3 &:= (x_1 z_1 - \mu, x_2 z_2 - \mu, \dots, x_n z_n - \mu)^{\mathrm{T}} \end{aligned}$$

- Here we derive a concise expression for the *Jacobian matrix* $F'_\mu(x,y,z)$. Note that

$$\begin{aligned} \nabla_x f_1 = 0, \qquad & \nabla_y f_1 = A, \qquad & \nabla_z f_1 = I \\ \nabla_x f_2 = A^{\mathrm{T}}, \qquad & \nabla_y f_2 = 0, \qquad & \nabla_z f_2 = 0 \\ \frac{\partial f_3(j)}{\partial x_j} = z_j, \qquad & \frac{\partial f_3(j)}{\partial y_j} = 0, \qquad & \frac{\partial f_3(j)}{\partial z_j} = x_j, \end{aligned}$$

  Therefore,

  - the Jacobian matrix for f_3 is

$$J_3 = [\frac{\partial f(i)}{\partial x_j, y_j, z_j}]_{n \times (3n)} = \begin{bmatrix} Z & 0 & X \end{bmatrix}$$

    with $Z := \mathrm{diag}(z_1, \dots, z_n)$ and $X := \mathrm{diag}(x_1, \dots, x_n)$.
  - the Jacobian matrices for f_1, f_2 are:

$$\begin{aligned} J_1 &= \begin{bmatrix} \nabla_x^{\mathrm{T}} f_1 & \nabla_y^{\mathrm{T}} f_1 & \nabla_z^{\mathrm{T}} f_1 & \end{bmatrix} = \begin{pmatrix} 0 & A^{\mathrm{T}} & I \end{pmatrix} \\ J_2 &= \begin{bmatrix} \nabla_x^{\mathrm{T}} f_2 & \nabla_y^{\mathrm{T}} f_2 & \nabla_z^{\mathrm{T}} f_2 & \end{bmatrix} = \begin{pmatrix} A & 0 & 0 \end{pmatrix} \end{aligned}$$

  - The Jacobian matrix for $F_\mu(x,y,z)$ is given by:

$$F'_\mu(x,y,z) = \begin{pmatrix} J_1 \\ J_2 \\ J_3 \end{pmatrix} = \begin{pmatrix} 0 & A^{\mathrm{T}} & I \\ A & 0 & 0 \\ Z & 0 & X \end{pmatrix},$$

    where $Z := \mathrm{diag}(z_1, \dots, z_n)$ and $X := \mathrm{diag}(x_1, \dots, x_n)$.

- After rearranging, it suffices to solve the system

$$\begin{pmatrix} A^{\mathrm{T}} & I & 0 \\ 0 & 0 & A \\ 0 & X & Z \end{pmatrix} \begin{pmatrix} dy \\ dz \\ dx \end{pmatrix} = \begin{pmatrix} r_d \\ r_p \\ r_c \end{pmatrix}$$

Or we write it into *augumented form* and solve it for dy first:

$$\left[\begin{array}{ccc|c} A^{\mathrm{T}} & I & 0 & r_d \\ 0 & 0 & A & r_p \\ 0 & X & Z & r_c \end{array}\right] \xRightarrow{\text{row 1 left-multiply } X} \left[\begin{array}{ccc|c} XA^{\mathrm{T}} & X & 0 & Xr_d \\ 0 & 0 & A & r_p \\ 0 & X & Z & r_c \end{array}\right] \xRightarrow{\text{Add row 3 into row 1}}$$

$$\left[\begin{array}{ccc|c} XA^{\mathrm{T}} & 0 & -Z & Xr_d - r_c \\ 0 & 0 & A & r_p \\ 0 & X & Z & r_c \end{array}\right] \xRightarrow{\text{row 1 divided by } Z} \left[\begin{array}{ccc|c} \frac{X}{Z}A^{\mathrm{T}} & 0 & -I & \frac{Xr_d - r_c}{Z} \\ 0 & 0 & A & r_p \\ 0 & X & Z & r_c \end{array}\right] \xRightarrow{\text{row 1 leftmutliply by } A}$$

$$\left[\begin{array}{ccc|c} A\frac{X}{Z}A^{\mathrm{T}} & 0 & -A & A\frac{Xr_d - r_c}{Z} \\ 0 & 0 & A & r_p \\ 0 & X & Z & r_c \end{array}\right] \xRightarrow{\text{Add row 2 into row 1}} \left[\begin{array}{ccc|c} A\frac{X}{Z}A^{\mathrm{T}} & 0 & 0 & A\frac{Xr_d - r_c}{Z} + r_p \\ 0 & 0 & A & r_p \\ 0 & X & Z & r_c \end{array}\right]$$

Hence, we derive a formula for solving dy:

$$A\frac{X}{Z}A^{\mathrm{T}}dy = A\frac{Xr_d - r_c}{Z} + r_p$$

Then we want to apply back substitution to solve for dx and dz. Recall the formula above that $Xdz + Zdx = r_c$, and to solve dx, we apply Gaussian elimination again:

$$\left[\begin{array}{ccc|c} A^{\mathrm{T}} & I & 0 & r_d \\ 0 & 0 & A & r_p \\ 0 & X & Z & r_c \end{array}\right] \xRightarrow{\text{Add row 1 leftmutliplying } -X \text{ into row 3}} \left[\begin{array}{ccc|c} A^{\mathrm{T}} & I & 0 & r_d \\ 0 & 0 & A & r_p \\ -XA^{\mathrm{T}} & 0 & Z & r_c - Xr_d \end{array}\right]$$

Therefore,

$$-XA^{\mathrm{T}}dy + Zdx = r_c - Xr_d$$

In summary, we derive formulas for solving dx, dy and dx:

$$\begin{cases} A\frac{X}{Z}A^{\mathrm{T}}dy = A\frac{Xr_d - r_c}{Z} + r_p \\ -XA^{\mathrm{T}}dy + Zdx = r_c - Xr_d \\ Xdz + Zdx = r_c \end{cases}$$

We solve the small linear system dy only, and after solving these small systems, dz, dx are recovered by backsubstitutions:

$$\begin{cases} dx = [XA^{\mathrm{T}}dy + (r_c - Xr_d)]/Z \\ dz = \frac{r_c - Zdx}{X} \end{cases}$$

A Copy of my Code

```
function [dx,dy,dz] = mylinsolve(A,rd,rp,rc,x,z)
% Usage: Solving LP Barrier Systems by Newtons Method
% Input:
%       A: m * n matrix
%      rd: n * 1 vector
%      rp: m * 1 vector
%      rc: n * 1 vector
%       x: n * 1 vector
%       z: n * 1 vector
%Output:
%      dx: n * 1 vector
%      dy: m * 1 vector
%      dz: n * 1 vector
n = length(rd);
d = x./z;
B = A * sparse(1:n,1:n,d) * A';
t1 = -x.*rd + rc;
t2 = A * (-t1./z) + rp;
dy = B \ t2;
dx = (t1 + x.*(A'*dy))./z;
dz = (rc - z.*dx)./x;
end
```

MATLAB Screen Printout for $p = 1$

Command Window

```
>> test_barrier
[size m = 500*p, n = 5*m] p = 1
(m,n) = (500,2500)

=============== using mylinsolve ===============
iter  1: errors = [1.25e+05, 2.70e+05, 2.81e+08]
iter  2: errors = [7.18e+04, 1.55e+05, 1.70e+08]
iter  3: errors = [5.05e+04, 1.09e+05, 1.24e+08]
iter  4: errors = [2.97e+04, 6.41e+04, 7.64e+07]
iter  5: errors = [1.20e+04, 2.59e+04, 3.34e+07]
iter  6: errors = [1.95e+03, 4.22e+03, 5.98e+06]
iter  7: errors = [8.04e+01, 1.74e+02, 3.34e+05]
iter  8: errors = [1.29e+01, 2.78e+01, 1.05e+05]
iter  9: errors = [5.90e+00, 1.27e+01, 5.88e+04]
```

Command Window

```
iter 37: errors = [8.12e-12, 1.95e-11, 2.79e-04]
iter 38: errors = [3.09e-12, 1.30e-11, 1.23e-04]
iter 39: errors = [8.46e-13, 2.41e-11, 4.24e-05]
iter 40: errors = [1.61e-13, 4.74e-11, 1.15e-05]
iter 41: errors = [3.59e-14, 3.44e-10, 3.42e-06]
iter 42: errors = [3.90e-15, 8.12e-11, 3.50e-07]
iter 43: errors = [3.59e-15, 2.26e-11, 3.50e-08]

=============== using yzlinsolve ===============
iter  1: errors = [1.25e+05, 2.70e+05, 2.81e+08]
iter  2: errors = [7.18e+04, 1.55e+05, 1.70e+08]
iter  3: errors = [5.05e+04, 1.09e+05, 1.24e+08]
iter  4: errors = [2.97e+04, 6.41e+04, 7.64e+07]
iter  5: errors = [1.20e+04, 2.59e+04, 3.34e+07]
```

Command Window

```
iter 34: errors = [8.17e-11, 1.76e-10, 2.01e-03]
iter 35: errors = [3.45e-11, 7.54e-11, 9.62e-04]
iter 36: errors = [1.26e-11, 3.04e-11, 4.10e-04]
iter 37: errors = [8.12e-12, 1.93e-11, 2.79e-04]
iter 38: errors = [3.09e-12, 1.14e-11, 1.23e-04]
iter 39: errors = [8.46e-13, 2.16e-11, 4.24e-05]
iter 40: errors = [1.61e-13, 5.08e-11, 1.15e-05]
iter 41: errors = [3.59e-14, 3.91e-10, 3.42e-06]
iter 42: errors = [3.18e-15, 7.80e-11, 3.50e-07]
iter 43: errors = [3.28e-15, 2.47e-11, 3.50e-08]

Using mylinsolve, time = 1.551851
Using yzlinsolve, time = 1.465034
```

Figure 3: MATLAB Screen Printout for $p = 1$

MATLAB Screen Printout for $p = 4$

```
Command Window
>> test_barrier
[size m = 500*p, n = 5*m] p = 4
(m,n) = (2000,10000)

=============== using mylinsolve ===============
iter  1: errors = [1.00e+06, 4.47e+06, 9.00e+09]
iter  2: errors = [5.89e+05, 2.63e+06, 5.56e+09]
iter  3: errors = [4.08e+05, 1.82e+06, 3.99e+09]
iter  4: errors = [2.66e+05, 1.19e+06, 2.71e+09]
iter  5: errors = [1.14e+05, 5.08e+05, 1.25e+09]
iter  6: errors = [1.48e+04, 6.61e+04, 1.80e+08]
iter  7: errors = [6.36e+02, 2.85e+03, 8.95e+06]
iter  8: errors = [4.73e+01, 2.12e+02, 1.29e+06]
iter  9: errors = [2.01e+01, 8.98e+01, 6.96e+05]
```

```
Command Window
iter 47: errors = [8.86e-15, 1.16e-08, 1.82e-06]
iter 48: errors = [8.34e-15, 1.57e-08, 9.14e-07]
iter 49: errors = [8.76e-15, 4.24e-09, 2.44e-07]
iter 50: errors = [9.56e-15, 1.05e-08, 2.43e-08]

=============== using yzlinsolve ===============
iter  1: errors = [1.00e+06, 4.47e+06, 9.00e+09]
iter  2: errors = [5.89e+05, 2.63e+06, 5.56e+09]
iter  3: errors = [4.08e+05, 1.82e+06, 3.99e+09]
iter  4: errors = [2.66e+05, 1.19e+06, 2.71e+09]
iter  5: errors = [1.14e+05, 5.08e+05, 1.25e+09]
iter  6: errors = [1.48e+04, 6.61e+04, 1.80e+08]
iter  7: errors = [6.36e+02, 2.85e+03, 8.95e+06]
iter  8: errors = [4.73e+01, 2.12e+02, 1.29e+06]
```

```
Command Window
iter 42: errors = [2.01e-13, 2.59e-10, 2.96e-04]
iter 43: errors = [7.70e-14, 2.89e-10, 1.31e-04]
iter 44: errors = [3.53e-14, 5.28e-10, 6.58e-05]
iter 45: errors = [1.23e-14, 1.04e-09, 2.21e-05]
iter 46: errors = [8.59e-15, 4.15e-09, 6.57e-06]
iter 47: errors = [8.29e-15, 1.09e-08, 1.82e-06]
iter 48: errors = [7.96e-15, 1.41e-08, 9.14e-07]
iter 49: errors = [8.47e-15, 3.35e-09, 2.44e-07]
iter 50: errors = [9.20e-15, 1.03e-08, 2.43e-08]

Using mylinsolve, time = 12.840127
Using yzlinsolve, time = 13.152927

>>
```

Figure 4: MATLAB Screen Printout for $p = 4$

A short summary

Introduction This project aims to solve a LP Barrier Systems by Newtons Method, i.e., applying Newton's method to solve linear systems.

Compute dx and dz Smartly During the computation, we need to solve a small linear system dy first, after which we should not solve for dz using the *echelon echol form* directly, i.e., do not compute $dz = A^{-1}r_p$, which is computationally expansive. Instead, we derive the formula for dx and dz in terms of dy sufficiently. The advantage is that we use the extra information for solving this system, and avoid some large-scale matrix inverse calculation processes.

Appreciate the sparse form When computing the inverse of $A * \text{diag}(d) * A'$, we should use the sparse form since the dimension for this matrix is large and most entries are zero.

Arrange Reasonable Computation For example, we need to use the balue $-x/*rd + rc$ for many times, so we can save it in advance and call them if necessary. Moreover, saving the matrix $\text{diag}(x_1/z_1, \ldots, x_n/z_n)$ is meaningless, since we can arrange computation such that it suffices to save the vector form $x./z$.