Minh-Thi Nguyen
MAT321: Numerical Methods
Professor Nicolas Boumal
Due: 17 October 2018

Homework 2: $Ax = b$ and $A = QR$

1 Linear systems with structure

$$A = \begin{pmatrix} a_1 & b_2 & & & \\ b_2 & a_2 & b_3 & & \\ & b_3 & a_3 & b_4 & \\ & & \ddots & \ddots & b_n \\ & & & b_n & a_n \end{pmatrix}$$

1. By Gaussian Elimination, we see that we can produce an $L_1 A$ such that it is zero everywhere below the top left entry by multiplying the first row by $-\frac{b_2}{a_1}$ and adding it to the second row. This operation gives us an $L_1 = I + M$ where M is a matrix such that:

$$MA = \begin{pmatrix} 0 & 0 & & & \\ -b_2 & \times & 0 & & \\ & 0 & 0 & 0 & \\ & & \ddots & \ddots & 0 \\ & & & 0 & 0 \end{pmatrix}$$

So, we have:

$$M = \begin{pmatrix} 0 & 0 & & & \\ -\frac{b_2}{a_1} & 0 & 0 & & \\ & 0 & 0 & 0 & \\ & & \ddots & \ddots & 0 \\ & & & 0 & 0 \end{pmatrix}$$

So, because $L_1 A = (I + M)A = A + MA$, which gives us our desired matrix, we see that:

$$L_1 = \begin{pmatrix} 1 & 0 & & & \\ -\frac{b_2}{a_1} & 1 & 0 & & \\ & 0 & 1 & 0 & \\ & & \ddots & \ddots & 0 \\ & & & 0 & 1 \end{pmatrix}$$

Which gives us:

$$L_1 A = \begin{pmatrix} a_1 & b_2 & & & & \\ 0 & (-\frac{b_2^2}{a_1} + a_2) & b_3 & & & \\ & b_3 & & a_3 & b_4 & \\ & & & \ddots & \ddots & b_n \\ & & & & b_n & a_n \end{pmatrix}$$

For this operation to be valid, we have to assume that the leading element (a_1) of the diagonal is nonzero. We also assume that A is symmetric and nonsingular.

2. Again, we let $L_1 = I + M$ where I is the $n \times n$ identity matrix. So, $L_1^T = (I+M)^T = I + M^T$. So, $L_1 A L_1^T = (I+M)A(I+M)^T = (A+MA)(I+M^T) = A + MA + AM^T + MAM^T$.

We know:

$$MA = \begin{pmatrix} 0 & 0 & & & \\ -b_2 & (-\frac{b_2^2}{a_1}) & 0 & & \\ & 0 & 0 & 0 & \\ & & \ddots & \ddots & 0 \\ & & & 0 & 0 \end{pmatrix}$$

and can easily see that because A is symmetric ($A = A^T$),

$$AM^T = (MA)^T = \begin{pmatrix} 0 & -b_2 & & & \\ 0 & (-\frac{b_2^2}{a_1}) & 0 & & \\ & 0 & 0 & 0 & \\ & & \ddots & \ddots & 0 \\ & & & 0 & 0 \end{pmatrix}$$

and

$$MAM^T = \begin{pmatrix} 0 & 0 & & & \\ 0 & (\frac{b_2^2}{a_1}) & 0 & & \\ & 0 & 0 & 0 & \\ & & \ddots & \ddots & 0 \\ & & & 0 & 0 \end{pmatrix}$$

Thus, $L_1 A L_1^T = A + MA + AM^T + MAM^T$:

$$L_1 A L_1^T = \begin{pmatrix} a_1 & & 0 & & & \\ 0 & (a_2 - \frac{b_2^2}{a_1} - \frac{b_2^2}{a_1} + \frac{b_2^2}{a_1}) & b_3 & & & \\ & b_3 & & a_3 & b_4 & \\ & & & \ddots & \ddots & b_n \\ & & & & b_n & a_n \end{pmatrix}$$

$=$

$$\begin{pmatrix} a_1 & 0 & & & \\ 0 & (a_2 - \frac{b_2^2}{a_1}) & b_3 & & \\ & b_3 & a_3 & b_4 & \\ & & \ddots & \ddots & b_n \\ & & & b_n & a_n \end{pmatrix}$$

$=$

$$\begin{pmatrix} \times & \\ & A' \end{pmatrix}$$

Thus, we see that

$$A' = \begin{pmatrix} (a_2 - \frac{b_2^2}{a_1}) & b_3 & & \\ b_3 & a_3 & b_4 & \\ & \ddots & \ddots & b_n \\ & & b_n & a_n \end{pmatrix}$$

So, A' is a tridiagonal matrix of size $(n-1)\times(n-1)$. Because A is symmetric, $A^T = A$, we see $L_1AL_1^T$ is symmetric because $(L_1AL_1^T)^T = L_1A^TL_1^T = L_1AL_1^T$. Therefore, because $L_1AL_1^T$ is symmetric and

$$L_1AL_1^T = \begin{pmatrix} \times & \\ & A' \end{pmatrix}$$

we observe that A' must also be symmetric because there is only a leading diagonal term that is not part of A', and the diagonal is symmetric. We also see that A' is tridiagonal because it is basically the left lower $(n-1)\times(n-1)$ matrix of A with a_2 changed to $a_2 - \frac{b_2^2}{a_1}$.

3. We are given that L' is the elimination matrix for A', so

$$L'A'L'^T = \begin{pmatrix} \times & \\ & A'' \end{pmatrix}$$

Thus, we see that

$$L_2L_1AL_1^TL_2^T = \begin{pmatrix} \times & & \\ & \times & \\ & & A'' \end{pmatrix}$$

=

$$\begin{pmatrix} \times & \\ & L'A'L'^T \end{pmatrix}$$

Hence, we see that:

$$L_2 = \begin{pmatrix} 1 & 0 \\ 0 & L' \end{pmatrix}$$

4. We propose an algorithm to create the unit lower matrix L and diagonal matrix D such that $A = LDL^T$. We let:

$$D = \begin{pmatrix} d_1 & 0 & & & \\ 0 & d_2 & 0 & & \\ & 0 & d_3 & 0 & \\ & & \ddots & \ddots & 0 \\ & & & 0 & d_n \end{pmatrix}$$

We observe that $L_{n-1}L_{n-2}...L_2L_1AL_1^TL_2^T...L_{n-2}^TL_{n-1}^T = D$. Thus, if we let $L_{total} = L_{n-1}L_{n-2}...L_2L_1$, so $L_{total}AL_{total}^T = D$, with:

$$L_{total} = \begin{pmatrix} 1 & 0 & & & \\ -\frac{b_2}{a_1} & 1 & 0 & & \\ & -\alpha_3 & 1 & 0 & \\ & & \ddots & \ddots & 0 \\ & & & -\alpha_n & 1 \end{pmatrix}$$

where α_i follows the constant required to eliminate A', A'', etc. We see that $\alpha_i = \frac{b_i}{d2}$. Furthermore, we can see that $d_1 = a_1$, $d_2 = a_2 - \frac{b_2^2}{a_1}$, $d_3 = a_3 - b_3^2/d_2$, ..., $d_n = a_n - b_n^2/d_{n-1}$

We see that our unit lower matrix L becomes L_{total}^{-1} (because we know that if $L_{total} = I + M$, then $L_{total}^{-1} = I - M$:

$$L = L_{total}^{-1} = \begin{pmatrix} 1 & 0 & & & \\ \frac{b_2}{a_1} & 1 & 0 & & \\ & \alpha_3 & 1 & 0 & \\ & & \ddots & \ddots & 0 \\ & & & \alpha_n & 1 \end{pmatrix}$$

Thus, we construct our matrices L and D from the iterations described above, and I attached my code to perform the LU decomposition below:

```
% Create algorithm to construct L and D such that A = LDL^T where
% A is tridiagonal and all leading values of the diagonal of A
% are nonzero.

function [L,D] = symmetric_tridiagonal_LU(A)
%function to compute LU factorization

%determine the size of matrix A (rows and columns)
n = size(A,1);

D = zeros(n,n);
%Set the diagonals of D to be the diagonals of A
for i = 1 : n
    D(i,i) = A(i,i);
end

L = eye(n); %identity matrix

%determine the diagonal entries for each iteration
for k = 2 : n

    %check to see if the leading entry is nonzero
    if D(k-1,k-1) == 0
        break;
    end

    %update L
    L(k,k-1) = A(k-1,k) / D(k-1,k-1);

    %update D
    D(k,k) = D(k,k) - A(k-1,k) * L(k,k-1);

end

end
```

Because we divide, subtract, and multiply once within the loop between 2 and n in our algorithm, we see that the complexity of this algorithm consists of 3 operations, thus $\boxed{\approx 3n}$ flops.

5. As demonstrted in (4) and much of the explanation in previous problems, by Gaussian elimination, because we want to reduce A into a diagonal matrix, we see that the lower unit

triangular matrix takes the form:

$$L = L_{total}^{-1} = \begin{pmatrix} 1 & 0 & & & \\ \frac{b_2}{d_1} & 1 & 0 & & \\ & \frac{b_3}{d_2} & 1 & 0 & \\ & & \ddots & \ddots & 0 \\ & & & \frac{b_n}{d_{n-1}} & 1 \end{pmatrix}$$

where the diagonals of the matrix are 1 and $L_{21} = b_2/d_1$, $L_{32} = b_3/d_2$, ... where d_i is the diagonal value determined by the iterative process described by part (4). Thus, assuming that d_i is always going to be nonzero, the only the values of L which are not zero or one are $\ell_{(k+1)(k)}$ for $0 < k < n$. Thus, the structure of L consists of 0 everywhere above the diagonal and below the second left lower diagonal, with a diagonal of values 1, and a lower left diagonal of values $\ell_{(k+1)(k)} = \frac{b_{k+1}}{d_k}$. This will always be the structure of L because we know that we must choose an L_k such that for x_k, the kth column of matrix A, we have $L_k x_k = \begin{pmatrix} x_{1k} \\ x_{2k} \\ \vdots \\ x_{kk} \\ 0 \\ \vdots \\ 0 \end{pmatrix}$. We know that this matrix L_k would take the form $L_k = I - \ell_k e_k^*$ where e_k is the column matrix with 1 in the kth position and 0 everywhere else:

$$L_k = \begin{pmatrix} 1 & & & & & \\ & 1 & & & & \\ & & \ddots & & & \\ & & & 1 & & \\ & & & -\frac{x_{(k+1)k}}{x_{kk}} & & \\ & & & -\frac{x_{(k+2)k}}{x_{kk}} & & \\ & & & & \ddots & \\ & & & -\frac{x_{nk}}{x_{kk}} & 1 & \end{pmatrix}$$

Thus, in Gaussian elimination, the final L must take the form:

$$L = \begin{pmatrix} 1 & & & & & \\ \frac{x_{21}}{x_{11}} & 1 & & & & \\ \frac{x_{31}}{x_{11}} & \frac{x_{32}}{x_{22}} & \ddots & & & \\ \vdots & \vdots & & 1 & & \\ & & & \frac{x_{(k+1)k}}{x_{kk}} & & \\ & & & \frac{x_{(k+2)k}}{x_{kk}} & & \\ & & & \vdots & \ddots & \\ & & & \frac{x_{nk}}{x_{kk}} & & 1 \end{pmatrix}$$

We see that because our matrix A is tridiagonal, in the column vector $x_k = \begin{pmatrix} x_{1k} \\ x_{2k} \\ \vdots \\ x_{kk} \\ x_{(k+1)k} \\ \vdots \\ x_{nk} \end{pmatrix}$, we know that x_{nk} for $n > (k+1)$, $x_{nk} = 0$. Thus, for values in L_k, $\ell_{nk} = 0$. So, we have:

$$L_k = \begin{pmatrix} 1 & & & & & \\ & 1 & & & & \\ & & \ddots & & & \\ & & & 1 & & \\ & & & -\frac{x_{(k+1)k}}{x_{kk}} & & \\ & & & 0 & & \\ & & & & & \ddots \\ & & & 0 & & 1 \end{pmatrix}$$

Thus,

$$L = \begin{pmatrix} 1 & & & & & & \\ \frac{x_{21}}{x_{11}} & 1 & & & & & \\ 0 & \frac{x_{32}}{x_{22}} & \ddots & & & & \\ \vdots & \vdots & & 1 & & & \\ & & & \frac{x_{(k+1)k}}{x_{kk}} & & & \\ & & & 0 & & & \\ & & & \vdots & & \ddots & \\ & & & 0 & & & 1 \end{pmatrix}$$

We see that

$L_k x_k L_K^T = \begin{pmatrix} 0 \\ 0 \\ \vdots \\ x_{kk} \\ 0 \\ \vdots \\ 0 \end{pmatrix}$. So the diagonal D has diagonals $d_i = x_{ii}$.

Thus,

$$L = \begin{pmatrix} 1 & 0 & & & \\ \frac{b_2}{d_1} & 1 & 0 & & \\ & \frac{b_3}{d_2} & 1 & 0 & \\ & & \ddots & \ddots & 0 \\ & & & \frac{b_n}{d_{n-1}} & 1 \end{pmatrix}$$

Hence, we can develop a simple algorithm to solve the function $Lx = b$. For a vector $x = \begin{pmatrix} x_1 \\ x_2 \\ \vdots \\ x_n \end{pmatrix}$ and for a vector $b = \begin{pmatrix} b_1 \\ b_2 \\ \vdots \\ b_n \end{pmatrix}$, we have Lx gives us: $x_1 = b_1$, $\ell_{21}x_1 + x_2 = b_2$, ... $\ell_{n(n-1)}x_{n-1} + x_n = b_n$, so we have: $x_1 = b_1$, $x_2 = b_2 - \ell_{21}x_1$, ... $x_n = b_n - \ell_{n(n-1)}x_{n-1}$, which we can implement recursively.

```
% Create algorithm to solve the solution to Lx = b from matrix A that is
%tridiagonal

function [X] = solve_linear_LXB(A, B)
% use code to decompose A = LDL^T
n = size(A,1);
% determine size of matrix A and dimensions of X and B

[L,D] = symmetric_tridiagonal_LU(A);

%initialize X
X = zeros(n,1);

% set initial value for x1
X(1) = B(1);

%start recursion
for i = 2 : n
    %recursive function
    X(i) = B(i) - L(i,i-1)*X(i-1);
end

end
```

Because there are two operations inside the loop, the complexity of this algorithm is just also $\approx 2n$ flops.

6. Thus, the complete algorithm to solve for $Ax = b$ can be obtained by noticing that we can solve for a y such that $Ly = b$, and then we observe that because $A = LDL^T$, we have $DL^Tx = y$. We observe that DL^T is an upper triangular matrix:

$$DL^T = \begin{pmatrix} d_{11} & b_2 & & & \\ 0 & d_{22} & b_3 & & \\ & 0 & d_{33} & b_4 & \\ & & \ddots & \ddots & b_n \\ & & & 0 & d_{nn} \end{pmatrix}$$

Thus, we have the system of equations for $DL^Tx = y$ (notice our notation for "b" here is different from $Ax = b$: $d_{11}x_1 + b_2x_2 = y_1$, $d_{22}x_2 + b_3x_2 = y_2$, ... , $d_{nn}x_n = y_n$. So, $d_{11}x_1 + \ell_{21}d_{11}x_2 = y_1$, $d_{22}x_2 + \ell_{32}d_{33}x_3 = y_2$, ... , $d_{nn}x_n = y_n$, so we have: $x_n = y_n/d_{nn}$,

$x_{n-1} = (y_{n-1} - \ell_{n(n-1)} d_{nn} x_n)/d_{(n-1)(n-1)}$, ... Thus, we can first use our algorithm in (5) to find a y such that $Ly = b$, and then after obtaining y, we can find an x such that $DL^T x = y$. We implement the complete algorithm through the code shown below:

```
% Create algorithm to solve the solution to Ax = b from matrix A that is
%tridiagonal

function [X] = solve_tridiagonal(A, B)
% use code to decompose A = LDL^T
n = size(A,1);
% determine size of matrix A and dimensions of X and B


%1. Decompose A such that A = LDL'
D = zeros(n,n);
%Set the diagonals of D to be the diagonals of A
for i = 1 : n
    D(i,i) = A(i,i);
end

L = eye(n); %identity matrix

%determine the diagonal entries for each iteration
for k = 2 : n

    %check to see if the leading entry is nonzero
    if D(k-1,k-1) == 0
        break;
    end

    %update L
    L(k,k-1) = A(k-1,k) / D(k-1,k-1);

    %update D
    D(k,k) = D(k,k) - A(k-1,k) * L(k,k-1);

end


%2. Find y such that Ly = b
%initialize Y
Y = zeros(n,1);

% set initial value for x1
Y(1) = B(1);

%start recursion
for i = 2 : n
    %recursive function
    Y(i) = B(i) - L(i,i-1)*Y(i-1);
end


%3. Solve for X:
%initialize X
X = zeros(n, 1);
```

```
53
54 % set initial value for x1
55 X(n) = Y(n)/D(n,n);
56
57 %start recursion
58 for i = n-1:-1:1
59
60     %recursive function
61     X(i) = ((Y(i) - L(i+1,i)*D(i+1,i+1)*X(i+1)))/D(i,i);
62 end
63
64 end
```

We see that in the first decomposition of A to L and D, we have a loop with 3 scalar multiplication/addition, which gives us $\approx 3n$ flops. The second step, which is solving for y in $Ly = b$, we have a loop with two operations, which gives us $\approx 2n$ flops. The final step, solving for x from $DL^T x = y$, we have a loop with three operations, which gives us $\approx 3n$ flops. Therefore, the total overall complexity of the full algorithm consists of $8n$ flops.

Alternatively, we can also solve $Ax = b$ by finding a y such that $Ly = b$, and then find a z such that $Dz = y$, and finally an x such that $L^T x = z$, so we have $LDL^T x = Ax = b$.

So, first, after finding y such that $Ly = b$, we must find a z such that $Dz = y$. We see that because our matrix D is a diagonal matrix, we can easily see that $d_{11}z_1 = y_1$, $d_{22}z_2 = y_2$, ..., $d_{nn}z_n = y_n$. Thus, we can implement the following code to find the value of vector z:

```
1  %Solve Dz = y
2
3  function [X] = solve_linear_DZY(D, Y)
4
5  % use code to solve Dz = Y, given D and Y
6  n = size(D,1);
7  % determine size of matrix D and dimensions of Y and Z
8
9  X = zeros(n,1);
10
11 for i = 1 : n
12
13     X(i) = Y(i)/D(i,i);
14 end
15
16 end
```

Now, we must find a vector (solution) x such that $L^T x = z$. We see that our matrix L^T takes the form:

$$L^T = \begin{pmatrix} 1 & \ell_1 & & & \\ 0 & 1 & \ell_2 & & \\ & 0 & 1 & \ell_3 & \\ & & \ddots & \ddots & \ell_{n-1} \\ & & & 0 & 1 \end{pmatrix}$$

So, we see that for $L^T x = z$, we have $x_1 + \ell_1 x_2 = z_1$, $x_2 + \ell_2 x_3 = z_3$, ..., $x_{n-1} + \ell_{n-1} x_n = z_{n-1}$, $x_n = z_n$. Thus, we can implement the following code to find the value of vector x:

```
%Solve L^Tx = z

function [X] = solve_linear_LTXZ(L, Z)

% use code to solve L^Tx = Z, given lower triangular matrix L from A
% decomposition
n = size(L,1);
% determine size of matrix L and dimensions of X and Z

L_T = L';
%take transpose

%set up solution
X = zeros(n,1);

%initialize x_n
X(n) = Z(n);

%recursively find X
for i = n-1 : -1: 1

    X(i) = Z(i) - L_T(i,i+1)*X(i+1);

end

end
```

Thus, in this solution, the complete algorithm takes the form:

```
%Solve Ax = b Method 2

% Create algorithm to solve the solution to Ax = b from matrix A that is
%tridiagonal

function [X] = solve_tridiagonal2(A, B)
% use code to decompose A = LDL^T
n = size(A,1);
% determine size of matrix A and dimensions of X and B


%1. Decompose A such that A = LDL'
[L,D] = symmetric_tridiagonal_LU(A);

%2. Find y such that Ly = b
%initialize Y
[Y] = solve_linear_LXB(A,B);

%3. Solve for Z:
[Z] = solve_linear_DZY(D, Y);

%4. Solve for X:
[X] = solve_linear_LTXZ(L,Z);

end
```

We see that in this alternative method, we still have $3n + 2n + n + 2n = 8n$ flops total.

7. We now test our implementation with the given matrices.

```
%Test Ax = b tridiagonal implementation
A = [12 1 0 0 0 0; 1 18 2 0 0 0; 0 2 4 -5 0 0; 0 0 -5 8 -2 0; 0 0 0 -2 6 2; 0 0 0 0 0 2
    16];
B = [1;2;3;4;5;6];

%Find L and U:
[L,D] = symmetric_tridiagonal_LU(A);
%print L and D
disp(L)
disp(D)

%print norm
product = L*D*L';
difference = A - product;
Norm_difference = norm(difference);
fprintf('%.16e\n',Norm_difference)

%Solve Ax = b
[X] = solve_tridiagonal(A, B); %method 1
[X1] = solve_tridiagonal2(A, B); %method 2

%print X
fprintf('%.16e\n',X) %method 1
fprintf('%.16e\n',X1) %method 2

%Compare to A\b
disp(A\B);
```

Which outputs:

```
L =
    1.0000         0         0         0         0         0
    0.0833    1.0000         0         0         0         0
         0    0.1116    1.0000         0         0         0
         0         0   -1.3239    1.0000         0         0
         0         0         0   -1.4487    1.0000         0
         0         0         0         0    0.6446    1.0000

D =
12.0000         0         0         0         0         0
         0   17.9167         0         0         0         0
         0         0    3.7767         0         0         0
         0         0         0    1.3805         0         0
         0         0         0         0    3.1026         0
         0         0         0         0         0   14.7108

norm =
0.0000000000000000e+00
```

```
X =
1.0828711025365960e-01
-2.0055872799220370e-01
1.3069593460546310e+01
2.5482724137649104e+01
6.1163841597223163e+00
-2.9931201626143389e-01

X1 =
2.4611054647799230e-01
-1.9533265577359078e+00
1.8456883746384175e+01
1.3384176374012977e+01
5.3944961300914711e+00
-2.9931201626143389e-01

A\B =
   0.2461
   -1.9533
   18.4569
   13.3842
    5.3945
   -0.2993
```

Thus, we see that our x vector in method 1 differs from the MATLAB backslash result even though our norm for A and LDL^T is very close to 0. We observe that the x_6 value is close to that of the backslash result, but because of our recursion, the error of the operations accumulate, resulting in discrepancies in the other values of the X vector.

Nonetheless, our second method gives an x vector that is very accurate to the MATLAB backslash.

8. If A is positive definite, then for all vectors $y \neq 0$, we have $y^T Ay > 0$. Thus, we see that if we let $y = \hat{e_k}$, we see that $y^T Ay = a_k$ where a_k is the kth element along the diagonal of A. Thus, we see that for $1 \geq k \leq n$, we have $a_k > 0$, so all the diagonals of A are positive.

We know that A is positive definite. Thus, for an invertible matrix M, we can define a new vector $y_1 = M^T y$, so $y_1^T = (M^T y)^T = y^T M$, so we know that since we have $y^T Ay > 0$ for any vector $y \neq 0$, we can let $y = y_1$ because we know that $y_1 \neq 0$ because if $y_1 = M^T y$ and we know that $y \neq 0$, then $y_1 \neq 0$ because $y = (M^{-1})^T y_1$ and if $y_1 = 0$, we would have $y = 0$, which is a contradiction.

Thus, we see that if A is positive definite, then because L_1 is invertible ($\mid L_1 \mid = 1 > 0$), we see that $L_1 AL_1^T$ is also positive definite. Thus, by induction, if we know that if (for the case $n-1$) $L_{n-1}L_{n-2}...L_1 AL_1^T ...L_{n-2}^T L_{n-1}^T$ is positive definite, and we know that L_n is invertible because the diagonals are all 1, we have $L_n L_{n-1}L_{n-2}...L_1 AL_1^T ...L_{n-2}^T L_{n-1}^T L_n^T = D$ must also be positive definite. Therefore, the diagonals of a positive definite matrix must be positive, so the diagonals of D must be positive. Because our algorithm consists of recursively dividing by the diagonal entries of D, and we have to assume that the diagonals of D are nonzero, since if D is positive definite, the diagonals are positive and thus nonzero, we see that if A is positive definite, our algorithm can't fail.

2 Conditioning of least-squares problems

We know that for our least squares problems, perturbations in b will result in perturbations of the least squares solution x. We know that the condition number for a full-rank, rectangular matrix A is:

$$\kappa(A) = \frac{\sigma_{max}(A)}{\sigma_{min}(A)} = || A |||| A^{+} || \tag{1}$$

We also know the relative sensitivity of x with respect to perturbations in b in a least squares problem in terms of the condition number of A and problem-dependent norms as:

$$\frac{|| \delta x ||}{|| x ||} \leq \kappa(A) \frac{|| b ||}{|| A |||| x ||} \frac{|| \delta b ||}{|| b ||} \tag{2}$$

We are given that:

$$\frac{|| \delta b ||}{|| b ||} \approx 10^{-3} \tag{3}$$

We are given the two strategies of choosing 250 uniformly distributed data or take 125 quick pictures during the first 0.1 seconds and 125 during the last 125 seconds.

I would choose the first strategy. This is because in the second strategy, the change in time sampled for the first and last 125 data sets is very small ($\delta t = 0.0008s$) compared to the uniform distribution. Because we know that our b_i has a relative error of about 0.001, which is comparable to our δt. This small δt can be problematic and make the problem ill-conditioned because small perturbations in b can significantly alter the solution x. For instance, we consider a simple case of just a 3×2 matrix A_0 with δt and any term comparable to δt be ϵ and the starting time of the sample be t_0:

$$A_0 = \begin{bmatrix} 1 & t_0 & t_0^2 \\ 1 & t_0 + \epsilon & (t_0 + \epsilon)^2 \end{bmatrix}$$

Let's say for this system, we have:

$$b_0 = \begin{bmatrix} t_0 \\ t_0 + \epsilon \end{bmatrix}$$

Thus, for the linear system $A_0 x = b_0$ where $0 < \epsilon \approx 0.001 \ll 1$, we have the solution:

$$x = \begin{bmatrix} 0 \\ 1 \\ 0 \end{bmatrix}$$

Nevertheless, small perturbations in b can change this solution significantly. For instance, if we measured $\hat{b}$ to be:

$$\widetilde{b_0} = \begin{bmatrix} t_0 \\ t_0 \end{bmatrix}$$

because of the relative error in measuring b_i. We see that if this is the case, our solution to the least squares problem becomes:

$$\widetilde{x} = \begin{bmatrix} 1 \\ 0 \\ 0 \end{bmatrix}$$

This makes $\widetilde{x}$ significantly different from x that was supposed to be obtained. Thus, we see that by measuring our data points with very small time increments can result in potential ill-conditioning and bad accuracy on x. Furthermore, in the second strategy, a lot of information

about the least squares solution is missing from the center interval between 0.1 and 9.9 seconds because all the data is concentrated at the ends of the interval.

We let $A1$ be the matrix corresponding to the first strategy and $A2$ be the matrix corresponding to the second strategy. We can confirm our choice in strategy by calculating explicitly the condition number for matrices A_1 and A_2 on MATLAB, seeing that the matrix with the lower conditioning number will result in a better relative accuracy on x.

Our code is implemented below:

```
%Implement Strategy 1
m = 250;
t1 = linspace(0,1,m);
t2 = t1.^2;
t1 = t1';
ones = zeros(m,1);

for i = 1 : m
    ones(i) = 1;
end

A1 = [ones t1 t2'];
%find pseudoinverse of A_1
A1_pseudo = inv(A1'*A1)*A1';
%display condition number of A_1
disp(norm(A1)*norm(A1_pseudo));

%Implement Strategy 2
delta = 0.1;
ta = linspace(0, delta, m/2);
tb = linspace(1-delta, 1, m/2);
t3 = [ta tb];
t4 = t3.^2;

A2 = [ones t3' t4'];
%find pseudoinverse of A_2
A2_pseudo = inv(A2'*A2)*A2';
%display condition number of A_2
disp(norm(A2)*norm(A2_pseudo));
```

Which outputs:

```
K(A_1) =
   22.7183


K(A_2) =
   68.5427
```

Therefore, we see that the condition number $\kappa(A_1)$ is significantly less than $\kappa(A_2)$, so the first strategy is significantly more well-conditioned than the second.

Using the fact that $Ax = b$ and

$$|| \delta x ||\leq|| A^+ |||| \delta b ||=|| A^+ |||| \delta b || \frac{|| b ||}{|| b ||} =|| A^+ |||| \delta b || \frac{|| Ax ||}{|| b ||} =|| A^+ |||| Ax || \frac{|| \delta b ||}{|| b ||} \quad (4)$$

So,

$$\frac{|| \delta x ||}{|| x ||} \leq \frac{|| A^{+} |||| Ax ||}{|| x ||}(10^{-3}) = || A^{+} |||| A ||_p (10^{-3}) \tag{5}$$

.

Thus, comparing Strategy 1 and Strategy 2 for A_1 and A_2, we see that:

$$\frac{|| \delta x_1 ||}{|| x_1 ||} \leq || A_1^{+} |||| A_1 ||_p (10^{-3}) \tag{6}$$

.

$$\frac{|| \delta x_2 ||}{|| x_2 ||} \leq || A_2^{+} |||| A_2 ||_p (10^{-3}) \tag{7}$$

.

We can confirm our hypothesis by calculating explicitly the upper bound of the relative error for each Strategy through the MATLAB implementation shown below for a random theoretical value of x:

```
%test with a sample theoretical x:
height = rand(1)*1000000;
velocity = 500*randn - 1000;
gravity = -9.81;
x = [height velocity gravity/2];
x = x';
norm_x = norm(x);

%strategy 1:
b1 = A1 * x; %image of A1
norm_b1 = norm(b1);
relerr_x1 = (norm(A1_pseudo))*(norm_b1)*(10.^(-3))/norm_x;
disp(relerr_x1);

%strategy 2:
b2 = A2 * x; %image of A2
norm_b2 = norm(b2);
relerr_x2 = (norm(A2_pseudo))*(norm_b2)*(10.^(-3))/norm_x;
disp(relerr_x2);
```

Which gives us the consistent output:

```
relerr_x1 =
  0.0191

relerr_x2 =
  0.0540
```

Thus, we can expect the upper bound of the relative error of x in Strategy 1 to be:

$$\frac{|| \delta x ||}{|| x ||} \leq \boxed{0.0191} \tag{8}$$

Comparing the two upper bounds of relative error of the two strategies, we also see that the first Strategy would give us a less perturbed value of x for perturbed b, so the first Strategy is better conditioned.

Note that in this problem, we assume that if there is no measurement error, then there would be no perturbations in b and we would have $Ax = b$. Nevertheless, if b is perturbed already on a least squares solution, we would have to consider the angle between b and the image of A.

3 QR factorizations

We are asked to compute:

$$w = || AB^T - CD^T ||_F = \sqrt{Tr((AB^T - CD^T)^T(AB^T - CD^T))} \tag{9}$$

Thus, we must compute: $(AB^T - CD^T)^T(AB^T - CD^T) = (BA^T - DC^T)(AB^T - CD^T) =$

$$\begin{pmatrix} B & D \end{pmatrix} \begin{pmatrix} A^T \\ -C^T \end{pmatrix} \begin{pmatrix} A & -C \end{pmatrix} \begin{pmatrix} B^T \\ D^T \end{pmatrix}$$

Hence, we can calculate w by using two QR factorization using the MGS technique. To improve the orthonormality of Q, we can perform the MGS technique twice, with the second time on Q. We can factor the $m \times 2r$ matrices into QR:

$$\begin{pmatrix} B & D \end{pmatrix} = Q_1 R_1 = Q_{12} R_{12} R_1 = Q_{12} R'_1 = Q'_1 R'_1$$

where Q is a $m \times 2r$ matrix and R is a $2r \times 2r$ matrix. Similarly, we can also produce:

$$\begin{pmatrix} A & -C \end{pmatrix} = Q_2 R_2 = Q_{21} R_{22} R_2 = Q_{21} R'_2 = Q'_2 R'_2$$

The QR factorization of both matrices take a total of $2(2(m)(2r)^2) = 2(8mr^2)$ flops each, so both operations give us $32mr^2$ flops total.

We can then represent:

$$\begin{pmatrix} B & D \end{pmatrix} \begin{pmatrix} A^T \\ -C^T \end{pmatrix} \begin{pmatrix} A & -C \end{pmatrix} \begin{pmatrix} B^T \\ D^T \end{pmatrix} = Q'_1 R'_1 R'^T_2 Q'^T_2 Q'_2 R'_2 R'^T_1 Q'^T_1 = Q'_1 R'_1 R'^T_2 R'_2 R'^T_1 Q'^T_1$$

because Q_2 is orthogonal.

We let $M = R'_1 R'^T_2 R'_2 R'^T_1$. We wish to compute $|| Q'_1 R'_1 R'^T_2 R'_2 R'^T_1 Q'^T_1 ||_F = || Q'_1 M Q'^T_1 ||_F$.

We see that for any two square $n \times n$ matrices U and V, we have:

$$Tr(UV) = \sum_{i=1}^{n} u_{ii} v_{ii} = \sum_{i=1}^{n} v_{ii} u_{ii} = Tr(VU) \tag{10}$$

We see that (because $Q'^T_1 Q'_1 = I$):

$$|| Q'_1 M ||^2_F = Tr((Q'_1 M)^T (Q'_1 M)) = Tr(M^T Q'^T_1 Q'_1 M) = Tr(M^T M) = || M ||^2_F \tag{11}$$

Furthermore, we see that:

$$|| Q'_1 M ||^2_F = Tr((Q'_1 M)^T (Q'_1 M)) = Tr(M^T Q'^T_1 Q'_1 M) = Tr(Q'_1 M M^T Q'^T_1) \tag{12}$$

Since $M = R'_1 R'^T_2 R'_2 R'^T_1$, we have: $M^T = R'_1 R'^T_2 R'_2 R'^T_1 = M$. Thus, we have:

$$|| Q'_1 M ||^2_F = Tr(Q'_1 M M^T Q'^T_1) = Tr(Q'_1 M^T M Q'^T_1) = Tr((M Q'^T_1)^T (M Q'^T_1)) = || M Q'^T_1 ||^2_F = || M ||^2_F \tag{13}$$

Hence, we see that:

$$|| Q'_1 M Q'^T_1 ||^2_F = || M Q'^T_1 ||^2_F = || M ||^2_F \tag{14}$$

Therefore, we can calculate w by observing that:

$$w^2 = Tr((AB^T - CD^T)^T(AB^T - CD^T)) = Tr(R_1' R_2'^T R_2' R_1'^T) \tag{15}$$

We observe that the multiplication of $R_1' R_2'^T R_2' R_1'^T$ is the three times the multiplication of 2 R matrices, and because they are of size $2r \times 2r$ each, we require a total of $3(2(2r)(2r)(2r)) = 48r^3$ flops.

Thus, the QR decomposition and matrix product gives us a total complexity of $32mr^2 + 48r^3$ total flops for these operations.

Once we have the matrix product, we have to compute the trace of the overall matrix to obtain the Forbenius norm. Thus, we have to sum up the diagonals of the final matrix. This requires m addition operations, thus resulting in m total flops. We know that $m = r^3$. Thus, the total number of flops to compute w is $32mr^2 + 48r^3 + m = 32mr^2 + 49r^3$. Since a float requires 4 bytes of memory, we see that based on our calculation of the algorithm, we would need $(32mr^2 + 49r^3)(4) \approx 1.28$ Terabytes.

We implement the QR factorization using the modified Grahm-Schmidt algorithm twice for an economy-size QR decomposition. We chose the modified Gram-schmidt algorithm rather than the classical GS orthogonalization algorithm to reduce numerical instability and to reduce the sensitivity of the effects of rounding errors during our computational calculations. Furthermore, because the MGS algorithm produces a QR factorization where $\hat{Q}$ is still not quite orthonormal, we can perform MGS twice to get a very orthonormal $\hat{Q}_2$ such that $A = \hat{Q}_2 \hat{R}_2 \hat{R}_1 = \hat{Q}' \hat{R}'$, which gives us a more accurate QR factorization because this Q' is very nearly orthonormal.

We define the MGS function shown below from precept:

```
function [Q, R] = modified_gram_schmidt(A)
% Modified Gram-Schmidt orthogonalization algorithm (better stability than CGS)

    [m, n] = size(A);
    Q = zeros(m, n);
    R = zeros(n, n);

    for j = 1 : n

        v = A(:, j);

        R(j, j) = norm(v);
        Q(:, j) = v / R(j, j);

        R(j, (j+1):n) = Q(:, j)' * A(:, (j+1):n);
        A(:, (j+1):n) = A(:, (j+1):n) - Q(:, j) * R(j, (j+1):n);

    end

end
```

Our implementation of the algorithm is shown below:

```
%Problem 3: QR factorization to calculate the Forbenius norm

function[w] = Forbenius_norm(A, B, C, D);

%assume full common rank
```

```
%determine the size of A, B, C, D
[m, r] = size(A);

n = 2*r
%decompose [B D]
matrix_1 = [B D];
%MGS#1
[Q1_1, R1_1] = modified_gram_schmidt(matrix_1);
%MGS#2
%matrix_1 = Q1R1_2R1_1
[Q1, R1_2] = modified_gram_schmidt(Q1_1);

%decompose [A -C]
matrix_2 = [A -C];
%MGS#1
[Q2_1, R2_1] = modified_gram_schmidt(matrix_2);
%MGS#2
%matrix_2 = Q2R2_2R2_1
[Q2, R2_2] = modified_gram_schmidt(Q2_1);

%matrix_1 = Q1R1
R1 = R1_2 * R1_1;

%matrix_2 = Q2R2
R2 = R2_2 * R2_2;

%find the matrix
w_matrix = R1*R2'*R2*R1';

%find the trace
trace_w = sum(diag(w_matrix));

%find the forbenius norm
w = sqrt(trace_w);

disp(w)

end
```

We implement a test to see the accuracy of our algorithm by choosing an A, B, C, and D such that we know the value of w. We let $A = C$ and $B = D + X$, where $X = Q$ where $A = QR$, the orthonormal matrix from the QR decomposition of A after two MGS calls. Thus, $AB^T - CD^T = A(D^T + X^T) - AD^T = AX^T = AQ^T$. Thus, $w =|| AB^T - CD^T ||_F =|| AQ^T ||_F =|| A ||_F$, from the identity we proved above because $Q^TQ = I$.

```
%Test forbenius norm algorithm

%set matrix dimensions:
m = 10.^6;
r = 10.^2;

%generate four matrices A, B, C, D
A = randn(m,r);
D = randn(m,r);

%find the forbenius norm of A
```

```
disp(norm(A,'fro'));

%find the QR decomposition of A
[Q1,R1] = modified_gram_schmidt(A);
[Q,R] = modified_gram_schmidt(Q1);

%Q^TQ = I;
X = Q;

%define B and C
B = D + X;
C = A;

%calculate the forbenius norm using our algorithm
w_mint = Forbenius_norm(A, B, C, D);
```

We obtain the output:

```
norm_A =
      9.9996e+03


w =
   1.4856e+04
```

We see that the Forbenius norm calculated by MATLAB is 10^3 orders of magnitude different from our calculated value of w, but our matrix sizes were also very large, so this degree of error is not too significant. We see that when we decrease the matrix size to $m = 50$ and $r = 5$, we obtain the following result:

```
norm_A =
   15.7327


w =
   13.1417
```

Thus, our algorithm is fairly accurate.