Name: Hongda Li
Class: AMATH 584

Files and MATLAB Scripts

1. BadPolynomial.m:
   Summing up a badly conditioned polynomial terms by terms.

2. GoodPolynomial.m:
   The RHS of the equation as in HW3 sheet.

3. ModifedGS.m
   A Native matlab implementations of the Mofified Gram Shimitz process.

4. PerformanceSubroutine.m
   This is a file that takes in different functions handles for different schemes and them measure their numerical accuracy.

5. notes.mlx
   This is the file where I generated all of the plots that will be in this reports, run the cells in sequence and you will see all the plots.

Here is the link to man Github repo containing all the scripts that produces this report: link, PLEASE read the "notes.mlx" in particular it has all the codes that glue everything together and produces all the plots in this paper.

QR Algorithms

0.1 Algorithms and Performence Metrics

Two metrics are used to measure the performance of QR decomposition for each of the following algorithms:

1. The Householder Triangulation developed in class.

2. The Modified Grams Schimtz Process Implemented in MATLAB.

3. The qr function in MATLAB.

Notice that, some of these algorithm produces a full-QR decomposition while other don't, therefore, all matrices used for testing are squared matrices. But for the discussion below we will assume that the matrix $A \in \mathbb{C}^{m\times n}$ for generality.

Metric 1: The ability to reconstruct the original matrix from the original matrix. This is computed by:

$$\frac{\|QR - A\|}{mn}$$

Notice that the error from the matrix is averaged out by the total number of elements in the matrix.

Metric 2: How orthogonal is the matrix Q. This is computed via:

$$\frac{\|Q^H Q - I\|}{n^2}$$

Notice that it's also average out by the total size of the matrix.

0.2 Matrices that Tests for Numerical Accuracy

Four different types of matrices are tested on each of the algorithms, here is the type of matrices.

1. Uniform Random matrix produced by "rand" command in matlab
2. Low Rank Noisy Matrix
3. Hilbert Matrix
4. Pathological Vandermond Matrix

Low Rank Noisy Matrix: Produced by outer product of 2 vectors and then a noise that is close to machine epsilon is applied to the matrix.

$$A = v^H v + \text{rand(n, n)} * (10^{-15})$$

Because the matrix is inherently low rank, it will be singular, but just a tiny bit of noise will prevent it from being detected as a singular matrix or causing "Nan" to propagates.

Hilbert Matrix: The Hilbert Matrix is given by the formula:

$$H_{i,j} = \frac{1}{(i+j)}$$

This is a matrix that has atrocious condition number and it's ill-conditioned.

Pathological Vandermonde Matrix:
This matrix is the matrix that used for polynomial interpretation and it's looking like this:

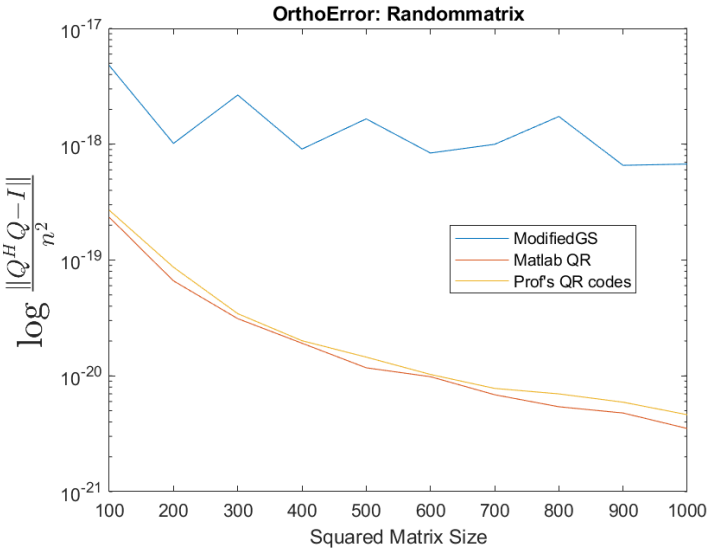
$$V = \begin{bmatrix} 1 & x_1 & x_1^2 & x_1^3 & \cdots & x_1^n \\ 1 & x_2 & x_2^2 & x_2^3 & \cdots & x_2^n \\ \vdots & & & & & \\ 1 & x_n & x_n^2 & x_n^3 & \cdots & x_n^n \end{bmatrix}$$

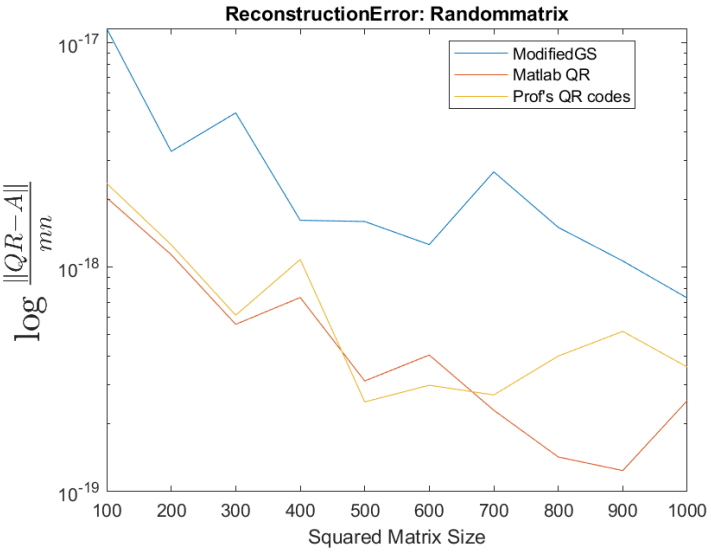
And this matrix can be generated by giving the vector of x_i using the "vander" command in matlab. This matrix is interested to us because the conditions number is completely pathological and the matrix is hardly singular, making it hard to detect numerical instability when it's running in the algorithms.

In our experiments, we take the x_i evenly from the interval: $[-2, 2]$ so that it actually blows up in values.

0.3 Plots and Figures

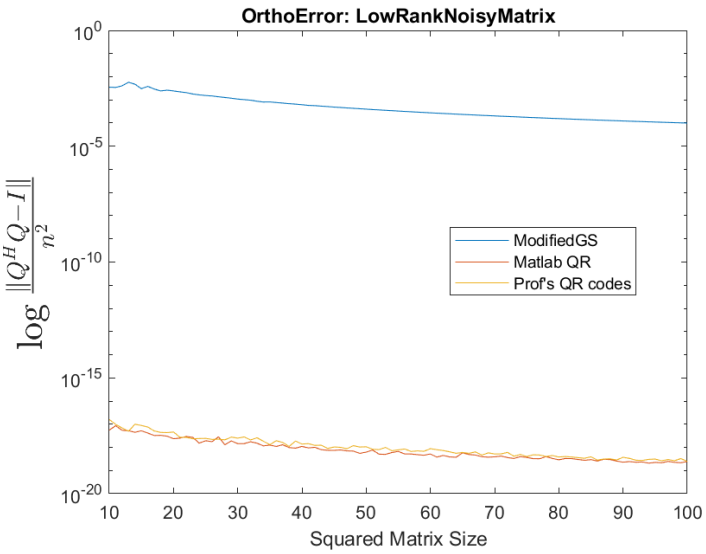
Testing using random matrix with huge huge size.

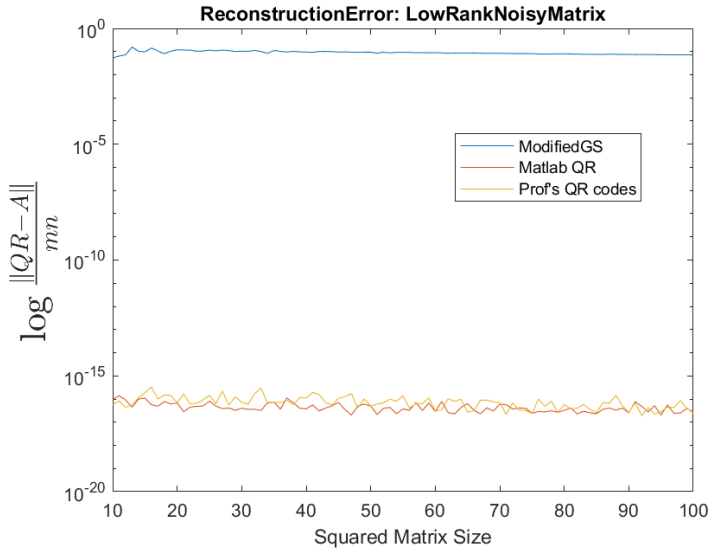




The errors averaged out to each elements are almost smaller than machine epsilon.

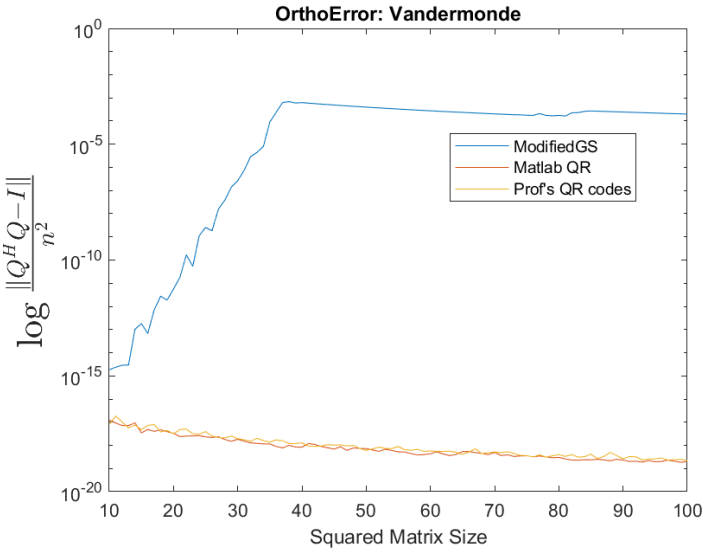
Testing using the Noisy Low Rank Matrix:

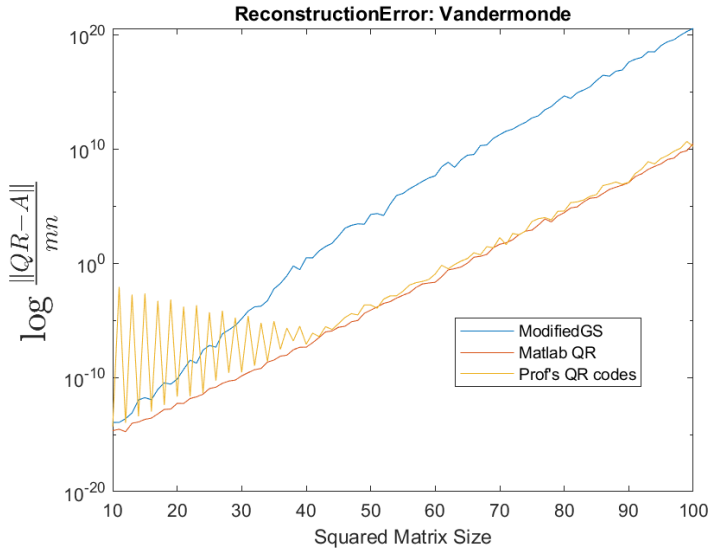




Please observe the relative small size of the matrix is need to destroy the accuracy for the Modified GS algorithm under both metrics. And notice that the errors seem to converge to a constant, meaning that, the total errors will be quadratic with respect to the dimension of the matrix.

Testing using the Vandermonde Matrix:

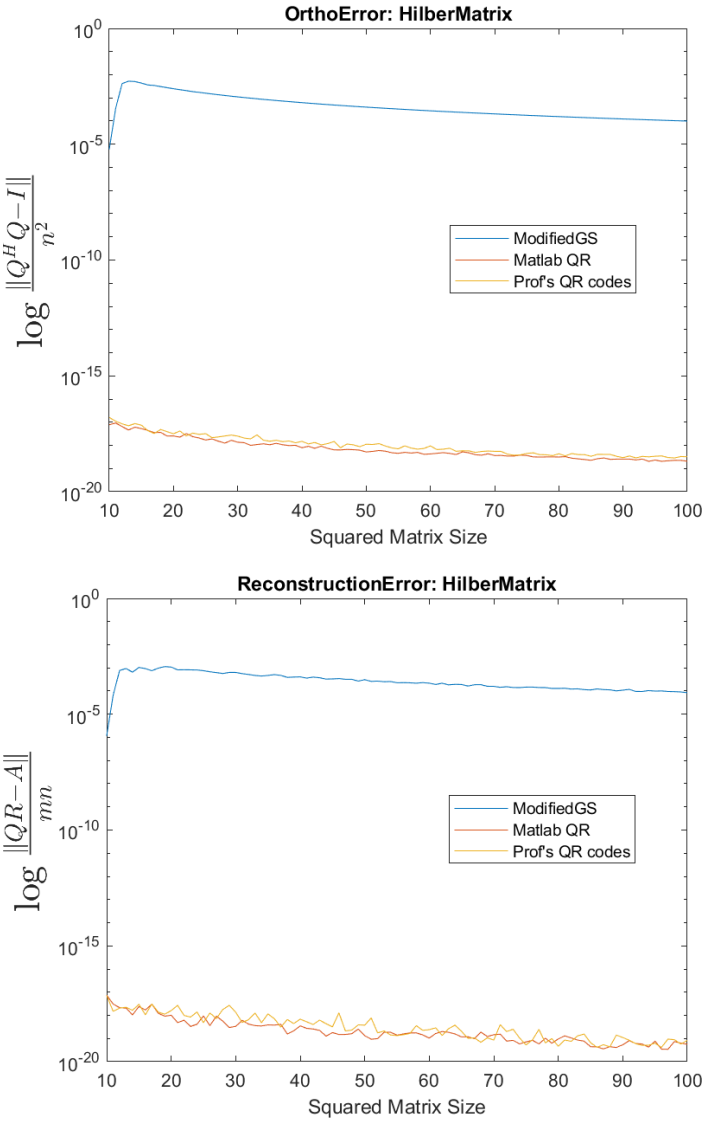




Please observe that, all of the algorithm's errors blows up astronomically with a very small size for the matrix, also observe the oscillations in Prof's algorithm indicates some unwanted artifacts, this implies that there are some difference between Matlab's implementations of the Householder Transformation and the Prof's implementation of the same algorithm.

Please observe that, despite of the bad reconstruction error, the householder transformation still produces good quality unitary matrices from the Vandermonde Matrix.

Testing using Hilbert Matrix:



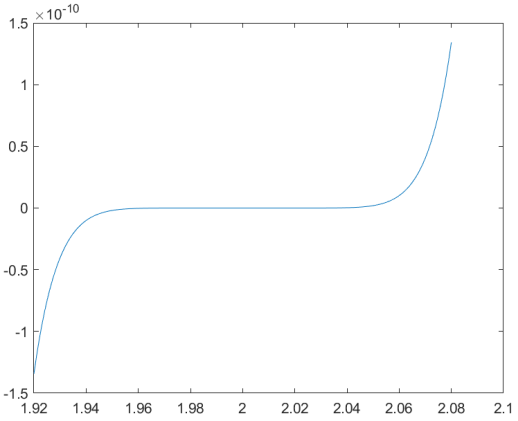
Observe that for the Hilbert Matrix, Modified GS performs poorly and the error spikes up and then averaged out as the size of the matrix increased. The other 2 algorithms maintain impressive accuracy for small HilbertMatrix.

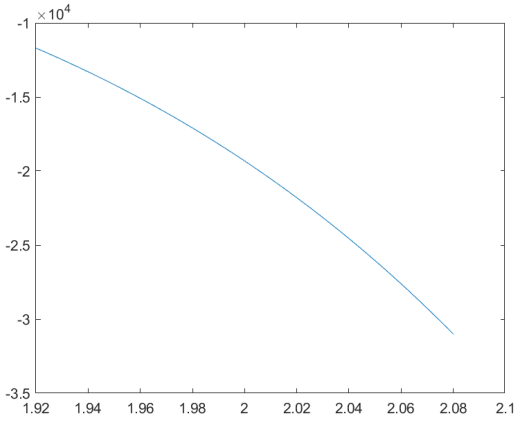
1 Polynomial Conditioning

For repeated roots, polynomial will be come very hard to evaluation using the Nested Multiplication alorithm and it's also ill-conditioned. This is discussed as an example in *Trefethen's* Book, Lecture 12.

Here are 2 plots of the same polynomial: $(x-2)^9$, close to the region to the repeating roots, and

$$x^9 - 18x^8 + 144x^7 - 672x^6 + 2016x^5 - 4023x^4 + 5376x^3 - 4608x^2 + 2304x - 512$$





And these 2 plots are very different and people should not trust the results when evaluating polynomial to a very high degree with the possibility that the polynomial having roots with high multiplicity.

2 Matrix Conditioning

2.1 Repeated Columns, Det, and Cond

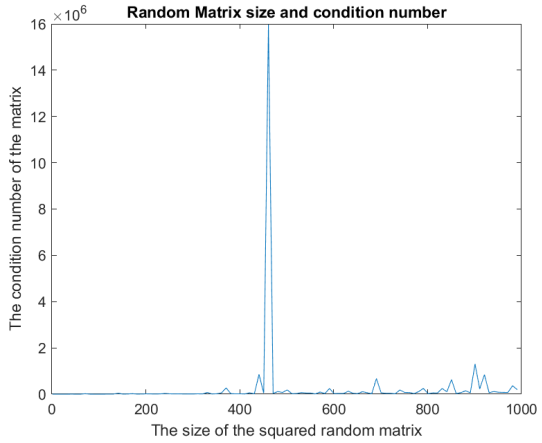
A repeated column is added to a random matrix and it's been setup in a way so that the resulting matrix with the repeating columns will be a square matrix in the end.

When the matrix is small (50×50), the repeated column causes the matrix to have a very large condition number (Ill-conditioned) and a very small determinant (close to singular) .

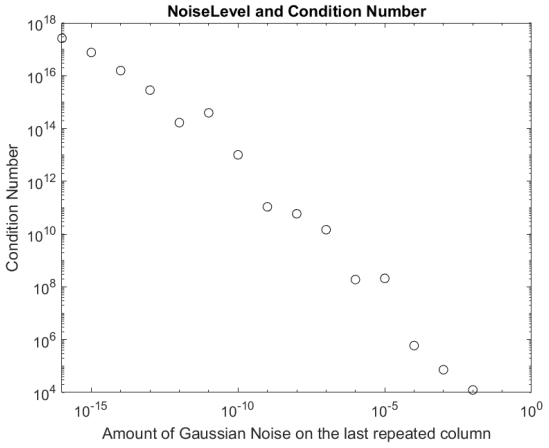
However, for matrix double the size (100×100) the repeated column causes the matrix to be ill-conditioned, but the difference this time is that the matrix determinant is huge (13+e18), meaning that the matrix is not singular but in fact it is.

Therefore we can't trust a sizable matrix with a large determinant and it might be singular. The condition

number of a matrix as a function of the size of the matrix doesn't seem to have any kind of observable patterns, most of them are contained within an reasonable range and here is the plot:



The following plot is the amount of noise involved in the repeated column of the matrix and the condition number.



The slope under the log log base 10 is around 2.5.

file: BadPolynomial.m

```
function output = BadPolynomial(x)

    Coefficients = ...
        [1, -18, 144, -627, 2016, -4032, 5376, -4608, 2304, -512];

    PowerAccumulate = ones(1, length(x));
    output = zeros(1, length(x));
    for C = Coefficients
        output = output + C*PowerAccumulate;
        PowerAccumulate = PowerAccumulate.*x;
    end
end
```

file: GoodPolynomial.m

```
function outputArg1 = GoodPolynomial(x)
    outputArg1 = (x - 2).^9;
end
```

file: ModifiedGS.m

```
function [Q, R] = ModifiedGS(A)
    % Given a matrix A, this function produces a reduced Gram Schitmz for
    % the matrix, using the Modified Gram Schimtz process.
    [~, n] = size(A);
    Q = A;  % Copy
    Q(:, 1) = Q(:, 1)/norm(Q(:, 1));
    for I = 2: n
        q = Q(:, I - 1);
        if norm(Q) < 1e-15
           error("Rank Deficit matrix can't use Modified GS. ") ;
        end
        P = q*q';
        V =  Q(:, I: end);
        Q(:, I: end) = V - P*V;
        q = Q(:, I);
        Q(:, I) = q/norm(q);
    end
    R = triu(Q'*A);
end
```

file: PerformenceSubtroutine.m

```
function [orthoErrors, restructErrors] = ...
    PerformenceSubroutine(matrices, schemes)
    % Given the generator of matrices, and 2 schemes that you want to
    % compare with, this function performs analysis on them and returns all
    % the errors.
    % Schemes:
    %   Cell arrays containing all the function handles.
    % Matrices:
    %   The matrices should be given as a cells java.until.arraylist.
    %
    % Return:
    %   The Reconstruction Errors and the Orthogonality Errors.
    N = matrices.size();
    S = length(schemes);
    orthoErrors = zeros(S , N);
    restructErrors = zeros(S, N);

    for I = 1: N
        for J = 1:S
            scheme = schemes{J};
            [E1, E2] = ErrorGet(matrices.get(I - 1), scheme);
            restructErrors(J, I) = E1;
            orthoErrors(J, I) = E2;
        end
    end

end

function [ReconstructionError, OrthogonalityError] = ...
    ErrorGet(matrix, scheme)
    %% For a given matrix and a given scheme, get the Reconstruction Error
    % and the Orthogonality Error.
    [m, n] = size(matrix);
    [Q, R] = scheme(matrix);
    ReconstructionError = norm(Q*R - matrix)/(m*n);
    OrthogonalityError  = norm(Q'*Q - eye(n))/(size(Q, 2)^2);
end
```

file: QRFactorFromClass.m

```
function [Q,R] = qrfactor(A)

    [m,n] = size(A);
    Q=eye(m);
    for k = 1:n
        % Find the HH reflector
        z = A(k:m,k);
        v = [ -sign(z(1))*norm(z) - z(1); -z(2:end) ];
        v = v / sqrt(v'*v);   % remoce v'*v in den

        % Apply the HH reflection to each column of A and Q
        for j = 1:n
            A(k:m,j) = A(k:m,j) - v*( 2*(v'*A(k:m,j)) );
        end
        for j = 1:m
            Q(k:m,j) = Q(k:m,j) - v*( 2*(v'*Q(k:m,j)) );
        end
    end
    Q = Q';
    R = triu(A);  % exact triangularity

end
```