Algorithms for Computing Strong RRQR Factorization

Shashi Satyam, Shiuli Subhra Ghosh, Shreyan Patra, Shubham Parashar

July 11, 2021

Abstract

Finding the numerical rank of a matrix has applications in subset selection, least squares, regularization, matrix approximation etc. Traditionally, SVD has been the go-to rank-revealing tool due to it's high accuracy. However, it can be computationally expensive and is sensitive to change in the initial matrix. In this project we have reviewed an alternative to the same which is the strong rank revealing QR (strong RRQR) factorization of a matrix M. In addition to the numerical rank, a strong RRQR factorization provides a basis for the approximate right null space of M. We have a look at the existence of such a factorization and explore the algorithms for finding it.

Contents

Work contribution:

Shashi Satyam	Contributed in writing the abstract and introduction of the project. Explained Algo 1 and Algo 2 during the presentation
Shiuli Subhra Ghosh	Contributed in writing section 2, 3 and section 4 of the report Presented Algo 3 and efficiency & stability of the Algo 5
Shreyan Patra	Contributed in writing section 2, 3 and section 4 of the report Explained Algo 4 and Algo 5 during the presentation
Shubham Parashar	Contributed in writing introduction and section 4 of the report Worked on and presented the MATLAB simulations for the presentation

1 Introduction

Given a matrix $M \in \mathbb{R}^{m \times n}$ with $m \geq n$, we consider a partial QR factorization of the form,

$$M\Pi = QR = Q \begin{pmatrix} A_k & B_k \\ & C_k \end{pmatrix}$$

where,

- $Q \in \mathbb{R}^{m \times m}$ is orthogonal and $\Pi \in \mathbb{R}^{n \times n}$ is a permutation matrix.
- $A_k \in \mathbb{R}^{k \times k}$ is upper triangular with non negative diagonal elements
- $B_k \in \mathbb{R}^{k \times n-k}$ is linearly dependent on A_k
- $C_k \in \mathbb{R}^{m-k \times n-k}$ has a sufficiently small norm

Here, $1 \leq k \leq n$ is the smallest integer for which $||C_k||_2$ is sufficiently small. This k is called the numerical rank of M. Such a QR factorization is called rank-revealing if it satisfies two conditions,

$$\sigma_{min}(A_k) \geq \frac{\sigma_k(M)}{p(k,n)} \quad \text{and} \quad \sigma_{max}(C_k) \leq \sigma_{k+1}(M)p(k,n)$$

where $p(k,n)$ is a function bounded by a low degree polynomial in k and n.

If $\sigma_k(M) >> \sigma_{k+1}(M) \approx 0$, then the numerical rank of M is k. Thus, the essential aim of RRQR algorithms is to find a Π for which $\sigma_{min}(A_k)$ is sufficiently large and $\sigma_{max}(C_k)$ is sufficiently small.

While traditional RRQR algorithms work in most cases, there are certain classes of matrices (more on them in section 3) on which the algorithmic computations can not be performed. Moreover, some numerical applications such as rank-deficient least-squares and subspace tracking require a basis for the approximate right null space of M.

All these drawbacks lead us to a modification of the traditional RRQR methods, which both works on a broader class of matrices and produces extra information in the form of a basis for the approximate right null space. Moreover, the computational complexity of this new factorization remains about the same as the RRQR counterparts. This modification is known as the strong RRQR factorization.

Formally stating, a QR factorization strong rank revealing if, for all $1 \leq i \leq k$ and $1 \leq j \leq n-k$

- $\sigma_i(A_k) \geq \frac{\sigma_i(M)}{q_1(k,n)}$ and $\sigma_j(C_k) \leq \sigma_{k+j}(M)q_1(k,n)$
- $|(A_k^{-1}B_k)_{i,j}| \leq q_2(k,n)$

Where, q_1 and q_2 are functions bounded by a low degree polynomial in k and n. In such a case, columns of $N = \Pi \begin{pmatrix} -A_k^{-1}B_k \\ I_{n-k} \end{pmatrix}$ form a basis for the approximate right null space of M.

In this report we talk about various RRQR and strong RRQR algorithms along with their implementations, flop count and stability. We also discuss about termination condition and the ideas in improving the efficiency of algorithms.

2 Algorithms and Main Results

2.1 Overview

In this section-

1. We will review QR with column pivoting
2. We will review the Chandrasekaran and Isepn algorithm for computing RRQR factorization.
3. We will review the strong RRQR factorization.
4. We discuss about the new algorithm which computes strong RRQR factorization and bound the total number of operations required when $f > 1$

As we move from algorithm to algorithm, we point out key differences and possible improvements that lead us into making the modifications in question.

2.2 Notation

The upcoming section uses the following convention,

- $A_k, \bar{A}_k \in \mathbb{R}^{k\times k}$ denote upper triangular matrices with non negative diagonal elements
- $B_k, \bar{B}_k \in \mathbb{R}^{k\times n-k}$ and $C_k, \bar{C}_k \in \mathbb{R}^{m-k\times n-k}$ denote general matrices
- $1/\omega_i$ denotes the 2-norm of the i^{th} row of A^{-1} (where $A \in \mathbb{R}^{l\times l}$ is non singular)
- γ_i denotes the 2-norm of the i^{th} column of C (where C has l columns)
- $\omega_*(A) = (\omega_1(A), \omega_2(A), \cdots, \omega_l(A))^T$ and $\gamma_*(A) = (\gamma_1(A), \gamma_2(A), \cdots, \gamma_l(A))$

In the partial QR factorization $X = Q \begin{pmatrix} A_k & B_k \\ & C_k \end{pmatrix}$ of a matrix $X \in \mathbb{R}^{m\times n}$, where the diagonal elements of A_k are non negative, we write

$$\mathcal{A}_k(X) = A_k, \quad \mathcal{C}_k(X) = C_k, \quad \mathcal{R}_k(X) = \begin{pmatrix} A_k & B_k \\ & C_k \end{pmatrix}$$

2.3 Algorithm - 1 (QR with column pivoting)

This QR with column pivoting is a modified version of the ordinary QR algorithm. Our aim is to find a factorization of the form $M\Pi = QR$.

The algorithm :

ALGORITHM 1. QR with column pivoting.

$k := 0$; $R := M$; $\Pi := I$;
while $\max_{1\le j\le n-k} \gamma_j(\mathcal{C}_k(R)) \ge \delta$ **do**
  $j_{\max} := \text{argmax}_{1\le j\le n-k}\ \gamma_j(\mathcal{C}_k(R))$;
  $k := k + 1$;
  Compute $R := \mathcal{R}_k(R\,\Pi_{k,k+j_{\max}-1})$ and $\Pi := \Pi\,\Pi_{k,k+j_{\max}-1}$;
endfor;

Steps of the algorithm : We initialize the loop parameters k at 0 and Π at I. Then for every iteration,

1. The while condition finds the maximum 2-norm of all $n-k$ columns of C_k.
2. It checks if the 2-norm of the columns is greater then δ
3. If the check is successful, then k is updated to $k+1$
4. Next, it swaps that column with greater 2-norm with the first column of C_k. The permutation matrix is also modified to reflect upon the swap just made.

The algorithm halts when the maximum 2-norm of the $n-k$ columns is less than δ, i.e. Columns of C_k reach a small enough norm.

This algorithm follows the greedy approach. So, suppose we already have $l < k$ well conditioned columns of M, then this greedy algorithm picks a column from the remaining $n-l$ columns of M such that the 2 norm of the selected column is as large as possible. This is done k times to pick the k well-conditioned columns of M. This algorithm, doesn't discard any column once it is chosen.

Termination condition of the algorithm : When this algorithm halts, we have

$$\sigma_{max}(C_k(M\Pi)) \leq \sqrt{n-k}$$
$$\max_{1\leq j\leq n-k} \gamma_j(C_k(M\Pi)) \leq \sqrt{n-k}\delta$$

Flop Count : If the vector column norms are updated rather than recomputed from scratch each time, then the flop count is about

$$4mnk - 2k^2(m+n) + \frac{4k^3}{3}$$

Termination condition of the algorithm : This algorithm follows a greedy strategy for finding the well conditioned columns. It gradually interchanges the columns so that $\det[A_k(R)]$ increases for each iteration. Thinking recursively, when the first k columns are already determined, it tries to pick the column from the remaining $n-k$ columns that maximizes $\det[A_{k+1}(R)]$. It is possible when there are only a few well conditioned columns and it also can find the left null space. But, it fails to find a strong RRQR factorization for the example which we will state in section 3.

2.4 Algorithm - 2 (Hybrid-III(k))

As Algorithm 1 uses a greedy strategy y for finding well-conditioned columns it fails to find a RRQR factorization in few cases. When $m = n$ and the numerical rank of M is close to n, Stewart suggested applying Algorithm 1 to M^{-1}. Chandrasekaran and Ipsen combined these ideas to construct an algorithm Hybrid-III(k) that is guaranteed to find an RRQR factorization, given k. Algorithm 2 is presented in a different form to motivate a constructive proof of the existence of a strong RRQR factorization.

The Algorithm :

ALGORITHM 2. Hybrid-III(k).

$R := M$; $\Pi := I$;
repeat
  $i_{\min} := \text{argmin}_{1 \le i \le k}\ \omega_i\ (\mathcal{A}_k(R))$;
  if there exists a j such that $\det\left[\mathcal{A}_k(R\,\Pi_{i_{\min}, j+k})\right] / \det\left[\mathcal{A}_k(R)\right] > 1$ **then**
    Find such a j;
    Compute $R := \mathcal{R}_k(R\,\Pi_{i_{\min}, j+k})$ and $\Pi := \Pi\,\Pi_{i_{\min}, j+k}$;
  endif;
  $j_{\max} := \text{argmax}_{1 \le j \le n-k}\ \gamma_j\ (\mathcal{C}_k(R))$;
  if there exists an i such that $\det\left[\mathcal{A}_k(R\,\Pi_{i, j_{\max}+k})\right] / \det\left[\mathcal{A}_k(R)\right] > 1$ **then**
    Find such an i;
    Compute $R := \mathcal{R}_k(R\,\Pi_{i, j_{\max}+k})$ and $\Pi := \Pi\,\Pi_{i, j_{\max}+k}$;
  endif;
until no interchange occurs;

Steps of the algorithm : Similar to Algorithm 1, we initialise $R = M$ and permutation Π as identity matrix I. Moreover k is known here.

1. It first finds the index i_{min} for which $\omega_i(\mathcal{A}_k(R))$ is the least i.e. 2-norm of i^{th} row of A_k^{-1} is greatest for $i = i_{min}$.

2. Then it finds for a j such that after interchange of column i_{min} with column $j + k$ the new $\det(A_k)$ is strictly greater than the old $\det(A_k)$.

3. Then it finds the index j_{max} for which $\gamma_j(\mathcal{C}_k(R))$ is the maximum i.e. 2 norm of j^{th} row of C_k is greatest for $j = j_{max}$.

Basically, Algorithm 2 keeps interchanging the most "dependent" of the first k columns (column i_{min}) with one of the last $n-k$ columns, and interchanging the most "independent" of the last $n-k$ columns (column j_{max}) with one of the first k columns, as long as det $[\mathcal{A}_k(R)]$ strictly increases.

Termination of the algorithm : Since $\det(\mathcal{A}_k(R))$ increases with every interchange no permutation repeats. And there are finite number of permutations, Algorithm 2 eventually halts.

2.5 Algorithm - 3 (Computing a strong RRQR factorization given k)

2.5.1 Explanation of Algorithm

Given k and $f \ge 1$, Algorithm 3 constructs a strong RRQR factorization by using column interchanges to try to maximize $\det(A_k)$.

The algorithm :

ALGORITHM 3. Compute a strong RRQR factorization, given k.

$R := \mathcal{R}_k(M)$; $\Pi := I$;

while there exist i and j such that $\det(\bar{A}_k))/\det(A_k) > f$,

where $R = \begin{pmatrix} A_k & B_k \\ & C_k \end{pmatrix}$ and $\mathcal{R}_k(R\,\Pi_{i,j+k}) = \begin{pmatrix} \bar{A}_k & \bar{B}_k \\ & \bar{C}_k \end{pmatrix}$, **do**

Find such an i and j;

Compute $R := \mathcal{R}_k(R\,\Pi_{i,j+k})$ and $\Pi := \Pi\,\Pi_{i,j+k}$;

endwhile;

Steps of the algorithm :
Here we initialize $R = \mathcal{R}_k(M)$ i.e. R is the upper triangular matrix of QR factorization of M. As always, the permutation matrix is initialised as Identity matrix i.e. $\Pi = I$.

1. The algorithm finds i and j such that after interchange of i^{th} column with $(j+k)^{th}$ column inside R, the determinant of new A_k(i.e. $\overline{A}_k$) is f times the determinant of original A_k where $f \geq 1$.

2. If such an i and j exists, the columns of R are interchanged and the permutation matrix is also updated.

While Algorithm 2 interchanges either the most "dependent" column of A_k or the most "independent" column of C_k, Algorithm 3 interchanges any pair of columns that sufficiently increases $\det(A_k)$.

Termination condition of the algorithm : Similarly to Algorithm 2, there are finite number of permutations and none can repeat, so Algorithm 3 eventually halts.

2.5.2 Proof of the Lemma 3.1 [1]

Let $R = \begin{pmatrix} A_k & B_k \\ & C_k \end{pmatrix}$ and $\mathcal{R}_k(R\,\Pi_{i,j+k}) = \begin{pmatrix} \bar{A}_k & \bar{B}_k \\ & \bar{C}_k \end{pmatrix}$, where A_k has positive diagonal elements. Then

$$\frac{det(\overline{A}_k)}{det(A_k)} = \sqrt{(A_k^{-1}B_k)_{i,j}^2 + (\gamma_j(C_k)/\omega_i(A_k))^2}$$

Essence: What this shows is that our halting condition creates bounds on $(A_k^{-1}B_k)$, which was one of the necessary conditions for computiong a strong RRQR factorization. We will exploit this fact late on the make the algorithm more efficient.

Proof: Assume that $i < k$ or $j > 1$. Let $A_k\,\Pi_{i,k} = \tilde{Q}\tilde{A}_k$ is the QR factorization of $A_k\,\Pi_{i,k}$ and $\tilde{B}_k = \tilde{Q}^T B_k\,\Pi_{i,j}$ and let $\tilde{\Pi} = \text{ diag}(\Pi_{i,k}, \Pi_{1,j})$. Then,

$$R\,\tilde{\Pi} \equiv \begin{pmatrix} A_k\,\tilde{\Pi}_{i,k} & B_k\,\tilde{\Pi}_{1,j} \\ & C_k\,\tilde{\Pi}_{1,j} \end{pmatrix} = \begin{pmatrix} \tilde{Q} & \\ & I_{(m-k)} \end{pmatrix} \begin{pmatrix} \tilde{A}_k & \tilde{B}_k \\ & \tilde{C}_k \end{pmatrix}$$

Since A_k and $\tilde{A}_k$ both have positive diagonal elements and are upper triangular, we have $\det A_k = \det(\tilde{A}_k)$.

Thus,

$$\begin{aligned}
A_k\ \Pi_{i,k} &= \tilde{Q}\tilde{A}_k \\
\tilde{A}_k &= \tilde{Q}^{-1}A_k\ \Pi_{i,k} \\
\tilde{A}_k^{-1} &= (\tilde{Q}^{-1}A_k\ \Pi_{i,k})^{-1} \\
\tilde{A}_k^{-1} &= \Pi_{i,k}{}^{-1}A_k{}^{-1}\ (\tilde{Q}^{-1})^{-1} \\
\tilde{A}_k^{-1} &= \Pi_{i,k}{}^{T}A_k{}^{-1}\ \tilde{Q} \\
\tilde{A}_k^{-1}\tilde{B}_k &= \Pi_{i,k}{}^{T}A_k{}^{-1}\ (\tilde{Q}\tilde{Q}^{T})B_k\ \Pi_{i,j} \\
\tilde{A}_k^{-1}\tilde{B}_k &= \Pi_{i,k}{}^{T}A_k{}^{-1}B_k\ \Pi_{i,j}
\end{aligned}$$

Then we have, $(\tilde{A}_k^{-1}\tilde{B}_k)_{i,j} = (A_k{}^{-1}B_k)_{k,1}$. Since, $\tilde{A}_k^{-1} = \Pi_{i,k}{}^{T}A_k{}^{-1}\ \tilde{Q}$ and post multiplication by an orthogonal matrix leaves the 2-norms of the rows unchanged, we have $\omega_i(A_k) = \omega_k(\tilde{A}_k)$. And finally, we have, $\gamma_j(C_k) = \gamma_1(\tilde{C}_k)$ as the permutation matrix takes the column with maximum 2-norm in the 1st column. We can consider the special case, $i = k$ and $j = 1$.

A simple partition will show,

$$\mathcal{R}_{k+1}(R) = \begin{pmatrix} A_{k-1} & b_1 & b_2 & B \\ & \gamma_1 & \beta & c^T{}_1 \\ & & \gamma_2 & c^T{}_2 \\ & & & C_{k+1} \end{pmatrix}$$

Then $\omega_i(A_k) = \gamma_1$ and $\gamma_j(C_k) = \gamma_2$ and $(A_k^{-1}B_k)_{i,j} = \beta/\gamma_1$. But, $\det A_k = \det(A_{k-1})\gamma_1$ and $\det(\bar{A}_k) = \det(A_{k-1})\sqrt{\beta^2+\gamma_2^2}$. So that,

$$\frac{det(\overline{A}_k)}{det(A_k)} = \sqrt{(\beta/\gamma_1)^2 + (\gamma_2/\gamma_1)^2} = \sqrt{(A_k^{-1}B_k)_{i,j}^2 + (\gamma_j(C_k)/\omega_i(A_k))^2}$$

2.6 Algorithm - 4 (Computing a strong RRQR factorization given k)

The Algorithm :

ALGORITHM 4. Compute a strong RRQR factorization, given k.

Compute $R \equiv \begin{pmatrix} A_k & B_k \\ & C_k \end{pmatrix} := \mathcal{R}_k(M)$ and $\Pi = I$;

while $\rho(R,k) > f$ **do**

Find i and j such that $\sqrt{\left(A_k^{-1}B_k\right)_{i,j}^2 + \left(\gamma_j(C_k)/\omega_i(A_k)\right)^2} > f$;

Compute $R \equiv \begin{pmatrix} A_k & B_k \\ & C_k \end{pmatrix} := \mathcal{R}_k(R\ \Pi_{i,j+k})$ and $\Pi := \Pi\ \Pi_{i,j+k}$;

endwhile;

Explanation of Algorithm : Algorithm 4 is equivalent to Algorithm 3. Here we denote $\det(\overline{A}_k)/\det(A_k)$ in terms of $\omega_i(A_k)$, $\gamma_j(C_k)$ and $(A_k^{-1}B_k)_{i,j}$. We use lemma 3.1 to devise,

$$\rho(R,k) = \max_{1\le i\le k, 1\le j\le n-k} \sqrt{(A_k^{-1}B_k)_{i,j}^2 + (\gamma_j(C_k)/\omega_i(A_k))^2}$$

Similar to Algorithm 3, we initialise $R = \mathcal{R}_k$ and $\Pi = I$. Then for every iteration,

1. It finds i and j such that $\rho(R,k) > f$.

2. It interchanges i^{th} and $(j+k)^{th}$ column of R and updates the permutation matrix accordingly until there does not exist such i, j such that $\rho(R,k) > f$.

Termination condition of the algorithm : Since Algorithm 4 is equivalent to Algorithm 3, it eventually halts when there are no more permutations available and finds a Π for which $\rho(\mathcal{R}_k(M\Pi), k) < f$.

2.7 Algorithm - 5 (Compute k and a strong RRQR factorization)

Given $f > 1$ and a tolerance $\delta > 0$, Algorithm 5 below computes both k and a strong RRQR factorization. It is a combination of the ideas in Algorithms 1 and 4 but uses

$$\hat{\rho}(R,k) = \max_{1\le i\le k, 1\le j\le n-k} max\left\{|(A_k^{-1}B_k)_{i,j}|, \gamma_j(C_k)/\omega_i(A_k)\right\}$$

instead of $\rho(R,k)$ and computes $\omega_*(A_k), \gamma_*(C_k)$ and $A_k^{-1}B_k$ directly for greater efficiency.
The Algorithm :

ALGORITHM 5. Compute k and a strong RRQR factorization.
$k := 0$; $R \equiv C_k := M$; $\Pi := I$;
Initialize $\omega_*(A_k)$, $\gamma_*(C_k)$, and $A_k^{-1}B_k$;
while $\max_{1\le j\le n-k} \gamma_j(C_k) \ge \delta$ **do**
  $j_{\max} := \text{argmax}_{1\le j\le n-k}\ \gamma_j(C_k)$;
  $k := k + 1$;
  Compute $R \equiv \begin{pmatrix} A_k & B_k \\ & C_k \end{pmatrix} := \mathcal{R}_k(R\,\Pi_{k,k+j_{\max}-1})$ and $\Pi := \Pi\,\Pi_{k,k+j_{\max}-1}$;
  Update $\omega_*(A_k)$, $\gamma_*(C_k)$, and $A_k^{-1}B_k$;
  while $\hat{\rho}(R,k) > f$ **do**
    Find i and j such that $\left|\left(A_k^{-1}B_k\right)_{i,j}\right| > f$ or $\gamma_j(C_k)/\omega_i(A_k) > f$;
    Compute $R \equiv \begin{pmatrix} A_k & B_k \\ & C_k \end{pmatrix} := \mathcal{R}_k(R\,\Pi_{i,j+k})$ and $\Pi := \Pi\,\Pi_{i,j+k}$;
    Modify $\omega_*(A_k)$, $\gamma_*(C_k)$, and $A_k^{-1}B_k$;
  endwhile;
endwhile;

Explanation of Algorithm : Similar to Algorithm 1 here also we initialize $R = M, k = 0$ and $\Pi = I$. Before applying the algorithm we also initialize $\omega_*(A_k), \gamma_*(C_k)$ and $A_k^{-1}B_k$ i.e. we calculate these values for the initialised matrix R.

Steps of the algorithm :

1. As in Algorithm 1, the first while loop finds the largest 2-norm among $n - k$ columns of C_k and checks if it is greater than δ.
2. If the condition satisfies it stores the index of the largest 2-norm column of C_k in j_{max}.
3. k gets incremented by 1.
4. The j_{max} column gets interchanged by the first column of C_k and the permutation matrix gets updated.
5. $\omega_*(A_k), \gamma_*(C_k)$ and $A_k^{-1}B_k$ gets updated which is used in the second while loop which gives the strong RRQR factorization.
6. The second while loop checks if $\hat{\rho}(R, k) > f$ i.e. it checks if the determinant of A_k after permutation increases sufficiently.
7. If the condition in second while loop satisfies then it finds i, j such that $|(A_k^{-1}B_k)_{i,j}| > f$ or $\gamma_j(C_k)/\omega_i(A_k) > f$.
8. The i^{th} and $(j + k)^{th}$ column of R gets interchanged and the permutation matrix gets updated. $\omega_*(A_k), \gamma_*(C_k)$ and $A_k^{-1}B_k$ gets modified accordingly.
9. The second while loop runs until for a given k it finds a permutation matrix Π such that $\hat{\rho}(R, k) \leq f$ which implies $\rho(\mathcal{R}_k(M\Pi, k)) \leq \sqrt{2}f$.
10. After exiting the second while loop the next iteration of the first while loop starts and in the same manner it runs for every k until $\underset{1 \leq j \leq n-k}{max} \gamma_j(C_k) < \delta$ occurs for a k. This k gives the rank of M and the permuted R matrix gives the upper triangular matrix of the strong RRQR factorization.

Termination condition of the algorithm : The second while loop terminates when $\hat{\rho}(R, k) \leq f$ after every possible permutation occurs. The first while loop and hence algorithm 5 terminates when for a particular k the largest 2-norm of all $n - k$ columns is smaller than δ.

3 Examples and Explorations

3.1 Failure of Algorithm 1:

Algorithm 1 follows greedy strategy for finding the well conditioned columns from the remaining $n-k$ columns that maximizes the determinant of $A_{k+1}(R)$. This algorithm perfectly works when there are few well conditioned columns this algorithm works fine. But in the example given below, this algorithm fails to find RRQR factorization.

$$S_n = \begin{pmatrix} 1 & 0 & 0 & \dots & 0 \\ 0 & s & 0 & \dots & 0 \\ 0 & 0 & s^2 & \dots & 0 \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & 0 & \dots & s^{n-1} \end{pmatrix} \quad K_n = \begin{pmatrix} 1 & -c & -c & \dots & -c \\ 0 & 1 & -c & \dots & -c \\ 0 & 0 & 1 & \dots & -c \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & 0 & \dots & 1 \end{pmatrix}$$

Where $c, s > 0$ and $c^2 + s^2 = 1$. Here all the columns of the matrix $M = S_n K_n$, are well conditioned.The maximum condition number is 1 and the columns become more well conditioned as we move through the column index. So, This matrix will not permute the columns, yet it can be shown that,

$$\frac{\sigma_k(M)}{\sigma_{\min}(A_k)} \geq \frac{c^3(1+c)^{n-1}}{2s}$$

and the right handed side grows faster than any polynomial in k and n.

3.2 Failure of Algorithm 2:

Considering the previous example where $M = S_n K_n$, let $k = n-2$.

$$M = \begin{pmatrix} S_{k-1}K_{k-1} & 0 & 0 & -cS_{k-1}d_{k-1} \\ & \mu & 0 & 0 \\ & & \mu & 0 \\ & & & \mu \end{pmatrix}$$

where $d_{k-1} = (1, \dots, 1)^T \in \mathbb{R}^{k-1}$ and $\mu = \frac{1}{\sqrt{k}} \underset{1 \leq i \leq k-1}{min} \omega_i(S_{k-1}K_{k-1})$
Then the algorithm does not permute the columns of M, yet it can be shown that,

$$\frac{\sigma_{k-1}(M)}{\sigma_{k-1}(A_k)} \geq \frac{c^3(1+c)^{n-4}}{2s} \quad \text{and} \quad ||A_k^{-1}B_k||_\infty = c(1+c)^{k-2}$$

and the right hand side grows faster than any polynomial in k and n.
Since Algorithm 1 does not permute the columns of M, this example also shows that Algorithm 2 may not compute a strong RRQR factorization even when it is run as a postprocessor to Algorithm 1.

3.3 Iterating Algorithm 4 over various values of k

For this experiment we let k (our approximation for rank) vary and observe how it affects the running of Algorithm 4. Our choice of A here is a numerically row rank matrix with entries in between 0,1 and

dimensions 2000 x 2000. We initialise k as 5 and interatively increase it to 1000. Some observations of the setup are as follows -

- Algorithm 5 tells that the approximate rank is infact around 55, unlike what one would expect- there is so significant difference between iteration 55 and the iteration in its vicinity.
- In fact, iteration 5 through 450 give error that is close to 0 and only start to fail once the matrices become so sparse that they have near 0 determinant.
- Although it seems like a downside but having a reasonable leeway when making a guess for k in when we run the algorithm is important. More specifically, it is fine if we have some near 0 rows/columns as long as we are able to reduce the dimension of our problem without losing a lot of information.

3.4 Iterating Algorithm 5 over various values of f

For this experiment we let f (our bound used for the approximate right null space calculation) vary and observe how it affects the running and outcomes of Algorithm 5. Our choice of A here is a numerically row rank matrix with entries in between 0,1 and dimensions 2000 x 2000.

Moreover, the tolerance is set close to 0, in order to simulate a QR factorisation which is very close to the actual one. We initialise f as 1.01 and interatively increase it to 1.01^{200}. Some observations of the setup are as follows -

- The approximate rank and error remain almost constant though all the values of f. Since A is extremely ill-conditioned, a large part of this can be attributed to the intrinsic stability of Algorithm 5.
- As mentioned in the paper, the run time of Algorithm is dependent on the value of f. In our case, the running time was much much slower when f was very close to 1.
- The largest entry in $A^{-1}B$ increases with f as expected. It stabilises after a threshold in our case though. (Section 5 of paper guarantees this to happen in most cases)

4 Results/observations

4.1 Efficiency of Algorithm 5

Algorithm 1 has a flop count of $4mnk - 2k^2(m+n) + 4k^3/3$. Despite the fact that Algorithm 5 is providing more information while covering a broader class of matrices, the efficiency is almost as same as the Algorithm 1.

Item	Flops
Updating procedure	$2(2m-k)(n-k)$
Reduction procedure	$3k(2n-k)$
Modifying procedure	$4m(n-k)+k^2$
Finding $\hat{\rho}(R,k)$	$2k(n-k)$

Table 1: Flop Count for Algorithm 5

Let, k_f be the final value of k when Algorithm 5 terminates and the total number of interchanges is denoted by t_{k_f}. t_{k_f} is bounded by $k_f log_f \sqrt{n}$. Then the total cost is about $2mk_f(2n-k_f) + 4t_{k_f}n(m+n)$. When f is taken to be small power of n the total cost is $\mathcal{O}(mnk_f)$.

Comparing Algorithms 1 and 5

- When $m >> n$, Algorithm 5 is almost as fast as Algorithm 1.
- When $m \approx n$ Algorithm 5 is about 50% more expensive.

4.2 Numerical Stability of Algorithm 5

- Since we updated and modified $\omega_*(A_k), \gamma_* C_k$ and ${A_k}^{-1}B_k$,rather than recompute them, we might expect some loss of accuracy.
- But Since we use these quantities for deciding which pairs of columns to interchange, Algorithm 5 could be only unstable if they are extremely inaccurate.
- We give an upper bound of $\rho(R,k)$ during interchanges. Since the bound grows slowly with k, A_k can never be extremely ill conditioned provided that $\sigma_k(M)$ is not very much smaller than $||M||_2$. This implies that ${A_k}^{-1}B_k$ cannot be too inaccurate.

References

[1] Minh Gu and Stanley C. Eisenstat (1996) Efficient Algorithms for Computing A Strong Rank-Revealing OR Factorization *SIAM J. SCI. COMPUT* Vol. 17, No. 4, pp. 848-869, July 1996

[2] Shivkumar Chandrasekaran and Ilse C.F. Ipsen (1994) On Rank Revealing Factorisations *SIAM J. MATRIX ANAL. APPL.* Vol. 15, No. 2, pp. 592-622, April 1994

[3] Xin Xing (2019) Strong Rank Revealing QR decomposition, MATLAB Central File Exchange. (https://www.mathworks.com/matlabcentral/fileexchange/69139-strong-rank-revealing-qr-decomposition) Retrieved July 1, 2021.