UNIVERSITY OF PISA

Resolution of the Linear Least Square Problem for augmented matrix adapting QR factorization through Householder

Computational Mathematics for Learning and Data Analysis project report

Amendola Maddalena, Rosasco Andrea
Group 30 - Project 16

CONTENTS

1. Introduction

In this report we solve a problem assigned for the exam of Computational Mathematics for Learning and Data Analysis a.a. 2019/2020 University of Pisa. The project is number 16 in the list of NO-ML project.

2. Problem

Considering a tall-thin data matrix $A \in \mathbb{R}^{m \times n}$ and the target values $b \in \mathbb{R}^m$, it's possible to solve the following Least Square Problem (LSP)

$$\min_x \|Ax - b\|$$

using the thin QR factorization computed with the Householder reflectors with complexity equal to $O(mn^2)$.
Let us assume that $\hat{A} \in \mathbb{R}^{m \times p}$ with $p > n$ is an *'agumented'* data matrix created from A with $p-n$ new columns that are transformations of the columns of A. Our goal is to solve the least square problem

$$\min_x \left\|\hat{A}x - b\right\|$$

through the thin QR factorization computed starting from the thin QR factorization of A. We want to achieve a complexity smaller than $O(mp^2)$ that is the cost of computing the thin QR factorization of $\hat{A}$ from scratch.

3. Theory

In this section we will show the theory behind the problem.

3.1. QR factorization

The Linear Least Square Problem consists in coumputing the set of coefficients $x_1, x_2, ...x_n$ such that the norm $\|A\mathbf{x} - b\|$ is minimized for a given $A \in \mathbb{R}^{m \times n}$ and $b \in \mathbb{R}^m$.
Formally, we want to solve

$$\min_x \|Ax - b\|$$

There is a way to solve the Least Square Problem using the QR factorization of the matrix A.

Definition 3.1. For a given matrix $A \in \mathbb{R}^{m \times n}$ there is an orthogonal matrix $Q \in \mathbb{R}^{m \times m}$ and an upper triangular matrix $R \in \mathbb{R}^{m \times n}$ such that $A = QR$.

3.2. HOUSEHOLDER REFLECTORS

One of the most used method to compute a QR factorization is the Householder triangularization.
An Householder transformation of a vector x is its reflection with respect to a plane through the origin, represented by its normal unit vector v ($vv^T = \|v\|^2 = 1$). Such transformation, applied to a vector, takes the form:

$$x' = x - 2vv^T x \tag{3.1}$$

where $vv^T x$ is the projection of x into v.
We call Householder Matrix the matrix defined as follow:

$$P = I - 2vv^T \tag{3.2}$$

Since any vector u can be normalized to have a unit norm ($v = u/\|u\|$), the Householder matrix defined in (3.2) can be written as:

$$P = I - 2vv^T = I - \frac{2uu^T}{\|u\|^2} \tag{3.3}$$

Given a vector x and the first standard basis vector $e = [1, 0, ..., 0]^T$ with all zeros except the first one equal to 1, we can define the vector

$$u = x - \|x\| e_1 \tag{3.4}$$

When the Householder matrix defined in (3.3) based on the vector defined in (3.4) is applied to the vector x itself, it produces its reflection $\|x\| e_1$. In particular, all elements in x' will be zeros except the first one, which is gonna be the norm of x. The reason for this, is that the Householder transformation is orthogonal and can't change the norm of its input vector.
The algorithm we use to compute the Householder reflector is defined in Algorithm 1. It takes as input a vector x and it outputs s and v.

Algorithm 1 householder_vector

$x \leftarrow$ vector to rotate
$s \leftarrow -sign(x_1) \times \|x\|$
$v = x$
$v_1 = x_1 - s$
$v \leftarrow v/\|v\|$

3.3. QR FACTORIZATION THROUGH HOUSEHOLDER REFLECTORS

The Householder reflectors can be effectively used to compute the QR factorization of a given matrix. Given a matrix $A \in \mathbb{R}^{m \times n}$, the algorithm consists in building a series of Householder matrices that transform A in an upper triangular matrix $R \in \mathbb{R}^{m \times n}$ and in storing their product as the matrix $Q \in \mathbb{R}^{m \times m}$.

The process is incremental, meaning that at each step it zeros out all the element below the diagonal of the current column by multiplying to A the Householder matrix Q_j, computed from the householder vector of the column. The process goes on until step n. The result is:

$$R = Q_n ... Q_1 A \tag{3.5}$$

hence, considering that the Householder matrices are orthogonal, we can compute the QR factorization of A as

$$A = (Q_n ... Q_1)^T R = Q_1 ... Q_n R = QR \tag{3.6}$$

3.4. THIN QR

Looking at how the matrices Q and R are composed we notice that the last $m - n$ lines of R are filled with zeros. If we try to examine the situation trough block matrix arithmetic we notice the following:

$$Q = \begin{bmatrix} Q_1 & Q_2 \end{bmatrix} \qquad R = \begin{bmatrix} R_1 \\ 0 \end{bmatrix} \tag{3.7}$$

When we multiply the matrices Q and R, the second block of the matrix Q gets multiplied by zeros not contributing to the final result, A. This means that we can drop Q_2 and the lower half of R and simply write

$$Q_1 R_1 = A \tag{3.8}$$

with $Q1 \in \mathbb{R}^{m \times n}$ and $R \in \mathbb{R}^{n \times n}$

4. SOLUTION

The goal is to solve the least square problem associated to the augmented matrix $\hat{A}$, through its thin QR factorization, computed starting from the thin QR factorization of A, the original dataset.

4.1. COMPUTING QR OF A

To compute the thin QR factorization of a matrix, we follow Algorithm 2. The algorithm takes in input the matrix $A \in \mathbb{R}^{m \times n}$ and it outputs the upper triangular matrix $R \in \mathbb{R}^{n \times n}$ and the list of householder reflectors v. We don't build the orthogonal matrix Q explicitly since it is not useful for the purpose. However, the list of the reflectors is important to restart the QR factorization of the augmented dataset $\hat{A}$.

In the algorithm we do not compute the Householder matrix as in equation (3.2) since it's expensive, instead we apply the transformation to the input matrix using directly the Householder vector. Another optimization we adopted consists in manually building the j_{th} column by setting s on the diagonal and zeros below it.

Algorithm 2 QR factorization

1: $q \leftarrow list\ of\ n\ elements$
2: **for** $j = 1$ `to` n **do**
3: $v, s \leftarrow householder\_vector(A_{\text{j:m, j}})$
4: $A_{\text{j, j}} \leftarrow s, \quad A_{\text{j+1:m, j}} \leftarrow 0$
5: $A_{\text{j:m, j+1:n}} \leftarrow A_{\text{j:m, j+1:n}} - 2v * (v^T A_{\text{j:m, j+1:n}})$
6: $q[j] \leftarrow v$
7: $R \leftarrow A$

4.2. Computing QR of $\hat{A}$

The Algorithm 3 describes how to start the thin QR factorization of the augmented dataset $\hat{A} \in \mathbb{R}^{m \times p}$ from the thin QR factorization of $A \in \mathbb{R}^{m \times n}$. The algorithm takes as input $R1 \in \mathbb{R}^{n \times n}$ and the list of the householder reflectors q computed by Algorithm 2 on the matrix A and works in two steps.
First, we have to transform the new columns of $\hat{A}$ multiplying them for the first n Householder vectors. This is necessary since the transformations that produced R1 were not applied to these new columns.
This process is described at lines 4-5 of Algorithm 3. After this short process, the thin QR factorization continues as in the Algorithm 2.
Together, the solution is described in Algorithm 3.

Algorithm 3 Resume QR factorization

1: $X \leftarrow$ new columns of $\hat{A}$
2: $R \leftarrow$ upper triangular matrix of A
3: $q \leftarrow$ list of householder reflectors of A
4: **for** $j = 1$ `to` n **do**
5: $X_{\text{j:m, 1:p-n}} \leftarrow X_{\text{j:m, 1:p-n}} - 2q_j * (q_j^T X_{\text{j:m, 1:p-n}})$
6: **for** $j = n+1$ `to` $(p-n)$ **do**
7: $v, s \leftarrow householder\_vector(X_{\text{j:m, j}})$
8: $X_{\text{j, j}} \leftarrow s, \quad X_{\text{j+1:m, j}} \leftarrow 0$
9: $X_{\text{j:m, j+1:n}} \leftarrow X_{\text{j:m, j+1:n}} - 2v * (v^T X_{\text{j:m, j+1:n}})$
10: $q_{j+n+1} \leftarrow v$
11: $R = [R, X]$

4.3. Solve the least square problem

Finally, we solve the least square problem $\min_x \left\| \hat{A}x - b \right\|$. Having R_1 and the p Householder vectors, this is fairly simple and relatively low cost. We have to compute the product $c = Q^t b$, but we don't have Q. We can compute the product using the Householder vectors as shown in Algorithm 4.

Algorithm 4 Compute $Q^T b$

1: **for** $j = 1$ `to` p **do**
2: $\quad b_{j:m} \leftarrow b_{j:m} - 2v_j(v_j^T b_{j:m})$
3: $c \leftarrow b_{1:n}$

Then, since R_1 is square and upper triangular, we proceed to solve by back-substitution the linear system $R_1 x = c$.

Algorithm 5 Back-substitution

for $j = n$ `to` 1 **do**
$\quad x_j \leftarrow \frac{b_j - (R_{1_{j,j+1:n}} * x_{j+1:n})}{R_{1_{j,j}}}$

4.4. COMPLEXITY

The cost of the Algorithm 2 is dominated by the row 5. Each iteration, we have:

1. $(m-j)(n-j)$ operations for the outer product $v_k(...)$
2. $2(m-j)(n-j)$ operations for the products $v_k^T A_{j:m,j:n}$
3. $(m-j)(n-j)$ operations for the subtraction $A_{j:m,j:n}$ - ...

In total it is $4(m-j)(n-j)$.

$$\sum_{j=1}^{n}(m-j)(n-j) = 4\sum_{j=1}^{n}(mn-k(m+n)+k^2) \approx 4mn^2 - \frac{4}{2}(m+n)n^2 + \frac{4}{3}n^3 = 2mn^2 - \frac{2}{3}n^3 \quad (4.1)$$

When we have a tall-thin matrix where $m >> n$, the complexity becomes $O(mn^2)$. Since all of the other operations have smaller costs, this is the complexity of solving the least square problem of A using its thin QR factorization. Similarly, if we solve the least square problem of $\hat{A}$ computing its thin QR factorization from scratch the cost is $O(mp^2)$.

We now analyze the complexity of the Algorithm 3. The lines that dominate the complexity are 5 and 9. Following the procedure described above, we obtain that the line 5 has a complexity equal to:

$$\sum_{j=1}^{n}(m-j)(p-n) = 4\sum_{j=1}^{n}(mp-mn-j(p-n)) \approx 4mnp - 4mn^2 + 2n^2 - 2n^2 p =$$
$$4mn^2 + 4nmz + 2n^2 - 4mn^2 - 2n^3 - 2zn^2 = 4nmz + 2n^2 - 2n^3 - 2zn^2 \approx O(nmz) \quad (4.2)$$

where $z = p - n$ is the number of new columns.
So far, we still have a complexity less than $O(mp^2)$. Let us analyze the complexity at line 9. In this for loop, we continue the factorization of $\hat{A}$ starting from the first new column, at position $n+1$ up to p. The operation is repeated z times and it starts from row $y = m - n$.

$$\sum_{j=1}^{n}(y-j)(z-j) = 4\sum_{j=1}^{n}(zy-j(z+y)+j^2) \approx 4yz^2 - 2(z+y)z^2 + \frac{4}{3}z^3 = 2yz^2 - \frac{2}{3}z^3 \approx O(yz^2) \quad (4.3)$$

Since $y < m$ and $z < p$ we have $O(yz^2) < O(mp^2)$. In total, the Algorithm 3 has a complexity equal to

$$O(mnz) + O(yz^2) = \begin{cases} O(mnz), & \text{if } z < n \\ O(yz^2), & \text{if } z > n \end{cases}$$

The cost of the Algorithm 4 is $O(mn)$ and the cost of the Algorithm 5 is $O(n^2)$. This means that the cost to solve the least square problem is negligible with respect to the costs to resume the thin QR factorization.

4.5. Stability

Since the algorithm is gonna be implemented and run on a computer, we are working with floating point representation.
This introduces an error in the order of the machine precision $u \approx 2e^{-16}$.
As a consequence, for a given input matrix $A \in \mathbb{R}^{m \times n}$ we will have:

$$\widetilde{A} = A + \delta_A \tag{4.4}$$

with $\delta_A = O(u\|A\|)$.
The error on the output is gonna be determined by the stability of the algorithm.
In particular, QR factorization algorithm is known to be backward stable, meaning that a small perturbation on the output can be seen as the results of executing the algorithm on a perturbed input. We can write:

$$\widetilde{Q}\widetilde{R} = \widetilde{A}$$

where $\widetilde{A}$ is defined in Equation 4.4.
We want to show that the same stability result holds for Algorthm 3. To achieve this we'll compare the computations made by Algorthm 3 with the one made by the QR factorization (Algorithm 2). Assuming the matrix $A \in \mathbb{R}^{m \times n}$ and its augmented version $\hat{A} \in \mathbb{R}^{m \times p}$, we define $X \in \mathbb{R}^{m \times z}$ as the matrix composed only by the new columns of $\hat{A}$.
Let's start analyzing the for loop at lines 4-5 of the Algorithm 3:

for $j = 1$ `to` n **do**
  $X_{\text{j:m, 1:p-n}} \leftarrow X_{\text{j:m, 1:p-n}} - 2v_j * (v_j^T X_{\text{j:m, 1:p-n}})$

The goal of these lines is to modify the new columns of $\hat{A}$ as if they were in A at the beginning of the factorization. This operation is important to *resume* the factorization.
Looking at lines 5 of Algorithm 2:

for $j = 1$ `to` n **do**
  $\hat{A}_{\text{j:m, j:p}} \leftarrow \hat{A}_{\text{j:m, j:p}} - 2v_j * (v_j^T \hat{A}_{\text{j:m, j:p}})$

we can observe that the Algorithm 3 applies the same operations of the Algorithm 2. After that, the Algorithm 3 performs the thin-QR factorization (lines 6-10) of the sub-matrix $X_{n:m,n:p} \in \mathbb{R}^{y \times z}$ with $y = m - n$ and $z = p - n$ as in the Algorithm 2.
Since the Algorithm 2 is backward stable and, as we shown, the Algorithm 3 performs exactly the same operations of the Algorithm 2, we can conclude that our implementation of the $resumeQR$ algorithm is backward stable.

5. EXPERIMENTS

The experiments are run on the ML-CUP-2019 dataset which is comprised of 1765 rows and 22 columns. The first 20 columns are the input data, the last 2 columns are the target. Formally, we define: $A \in \mathbb{R}^{1765\times 20}$ and $b \in \mathbb{R}^{1765\times 2}$.
The goal of our experiments is to show that solving the LSP of an augmented dataset resuming the thin-QR factorization of the original dataset with our algorithm, is faster than solving the same LSP computing the thin-QR factorization from scratch.
We also want also compare our solution with MatLab off-the-shelf solver in both accuracy and time. In total we perform three experiments:

1. we check how the time spent by our implementation of Algorithm 3 grows as we augment the dataset with an increasing number of columns.
2. we verify that solving the LSP resuming the thin-QR factorization is faster than solving it computing the thin-QR factorization from scratch.
3. we compare our implementation of the thin-QR factorization with the MatLab off-the-shelf solver in accuracy and time.

We chose 4 different functions to generate the new columns of A. These functions take as input one column and return a non-linear transformation of it. Considering that A has 20 columns and we use 4 functions, we can add to $\hat{A}$ at most 80 columns. Remaining consistent with the previous notation, we call z the number of columns we add to A to form $\hat{A}$. We compute the experiments for all possible value of $z = 1, ..., 80$. Since these values are too much to show in table, we will report for each experiments only some of them. Specifically we will show here only the result obtained every 5 values of z ($z = \{5, 10, 15, .., 70, 75, 80\}$).
To take reliable time measurements without any jittering, we used the MatLab function $timeit$. It measures the target function multiple times and then it takes the average of the time spent for each run.

5.1. LSP Experiment

The first experiment measures the time spent by solving the LSP with the function resumeQR (MatLab implementation of Algorithm 3) varying the number of columns added to its input matrix $\hat{A}$. We can observe that generally the time increases, especially when the number of columns we add it's not too small. The time is equal or slightly less when we add one or two columns. In the Table 5.1 are reported only some of the values computed. The first column is the number of columns added, the second column is the time measured and the third column is the difference between the time spent for a given number of columns added z and its previous value $z-1$ (i.e. in table 5.1, the Delta value in the first row with $z=5$ is the difference in time between the LSP with $z=5$ and $z=4$).

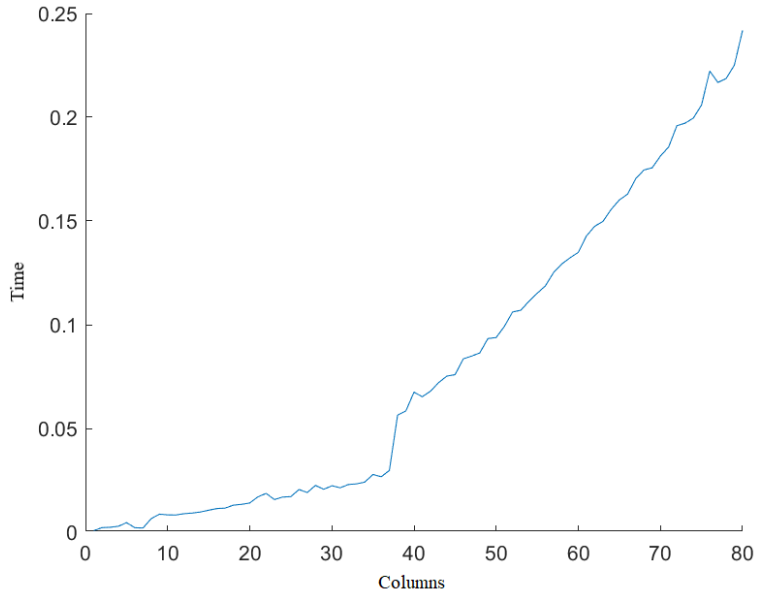

Figure 5.1: Time of LSP with resumeQR method.

Z	Time	Delta
05	0.00460621	0.00174230
10	0.00834111	-0.00035400
15	0.01060271	0.00087220
20	0.01407241	0.00064850
25	0.01714391	0.00021630
30	0.02239671	0.00173530
35	0.02781861	0.00369390
40	0.06751871	0.00911520

Z	Time	Delta
45	0.07592091	0.00067710
50	0.09381631	0.00047590
55	0.11517171	0.00383260
60	0.13486741	0.00261090
65	0.16015951	0.00459290
70	0.18130571	0.00565960
75	0.20585511	0.00629360
80	0.24182731	0.01674500

Table 5.1: Time of LSP with resumeQR method.

5.2. RESUMEQR EXPERIMENT

In the second experiment we first compare the time of the LSP using QR factorization of $\hat{A}$ computed with two different approaches. The first one computes the QR factorization of A and then resumes it for the new columns of $\hat{A}$ using the algorithm $resumeQR$. The second one, instead, applies the thin-QR factorization algorithm to $\hat{A}$.
In Table 5.2, the second column is the time of the LSP with $resumeQR$, the third table is the time of LSP computing the thin-QR factorization from scratch and the Delta column contains the difference between the second and the third column. The values in Delta are always negative, meaning that resuming the factorization is faster than compute it from scratch.

Z	resumeLSP	scratchLSP	Delta	Z	resumeLSP	scratchLSP	Delta
05	0.00288689	0.01059419	-0.007	45	0.07561999	0.09122739	-0.015
10	0.00969249	0.01326759	-0.003	50	0.09816599	0.12338039	-0.025
15	0.01019499	0.01659069	-0.006	55	0.11067259	0.12400909	-0.013
20	0.01258289	0.02061819	-0.008	60	0.12723279	0.14191589	-0.014
25	0.01586409	0.03272999	-0.016	65	0.14542189	0.16195239	-0.016
30	0.01790649	0.04582329	-0.027	70	0.17180739	0.19301469	-0.021
35	0.02350909	0.05682809	-0.033	75	0.20230529	0.21766869	-0.015
40	0.05798269	0.06961859	-0.011	80	0.21889979	0.23784059	-0.018

Table 5.2: Comparison in time between resumeLSP and scratchLSP.

This results are graphically shown in Figure 5.2 where the blue line is the $resumeQR$ algorithm and the orange line is the standard factorization. As we shown in section 4.4, we expect that the complexity of the $resumeQR$ algorithm is:

$$\text{w } O(mnz) + O(yz^2) = \begin{cases} O(mnz), & \text{if } z < n \\ O(yz^2), & \text{if } z > n \end{cases}$$

To check if this behavior holds, we plot also the two functions: $f_1 = mnz$ and $f_2 = yz^2$ where $z = p - n$ and $y = m - n$ assuming we are working with the original dataset $A \in \mathbb{R}^{m \times n}$ and the augmented dataset $\hat{A} \in \mathbb{R}^{m \times p}$ with $p > n$. Obviously, the two functions outputs very high values impossible to plot in the same figure. We reduced these values multiplying them by a factor of $1e^{-8}$. We observe that up to $2n$ columns added, the algorithm follows the function f_1. After that, the cost fits perfectly the function f_2.

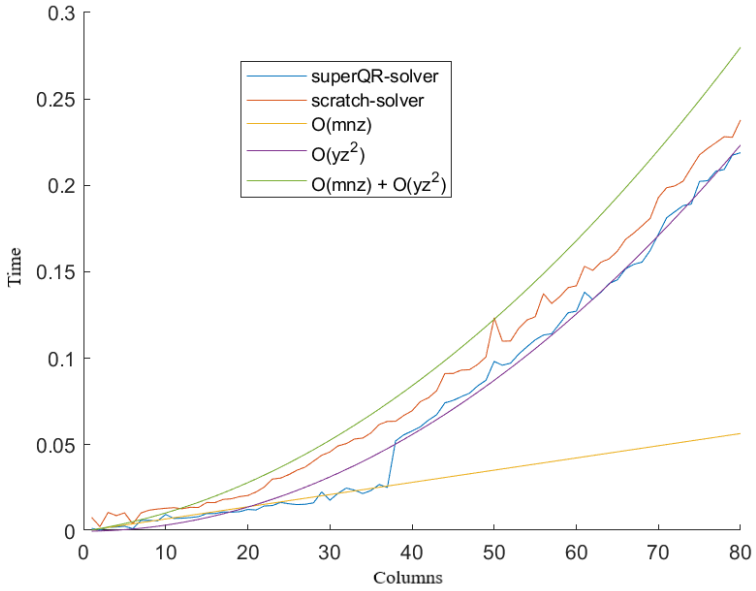

Figure 5.2: Comparison in time between resumeLSP and scratchLSP.

We also performed two more experiments. The first experiment is very similar to the previous one but, this time, we measure the cost of $resumeQR(\hat{A})$ algorithm including the cost of computing standard thin-QR factorization of A. We call this method *two-steps* because we divide the factorization in two step: thin-QR factorization of A and $resumeQR(\hat{A})$. We compare this cost with the standard thin-QR factorization of $\hat{A}$. As the Figure 5.3 shows, the behavior is the same as in Figure 5.2.

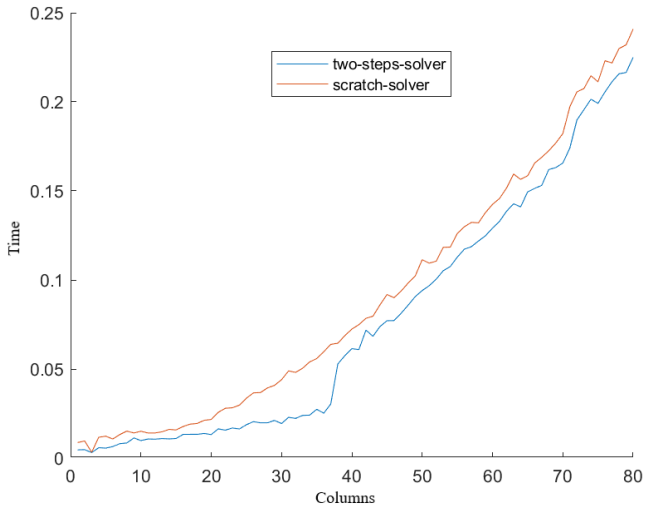

Figure 5.3: Comparison in Time between the *two-steps* method and scratchLSP.

We can try to explain this result as follow: we know that the cost of the thin-QR factorization for a given matrix $A \in \mathbb{R}^{m \times n}$ is $O(mn^2)$. Equivalently, the cost for the augmented matrix

$\hat{A} \in \mathbb{R}^{m \times p}$ with $p > n$ is $O(mp^2)$. Let us analyze the cost of the *two-steps* method. The first step costs $qr(A) \approx O(mn^2)$. We now analyze the two cases for the second step:

1. $O(mnz)$ if $z < n$ => $O(mn^2) + O(mnz) \approx O(mn^2) < O(mp^2)$
2. $O(yz^2)$ if $z > n$ => $O(mn^2) + O(yz^2) \approx O(yz^2) < O(mp^2)$

since $z = p - n < p$ and $y = m - n < m$

In the second experiment, we measured the accuracy of the two methods. The accuracy for a given matrix $A \in \mathbb{R}^{m \times n}$ with factorization $Q \in \mathbb{R}^{m \times n}$, $R \in \mathbb{R}^{n \times n}$ is computed as:

$$\frac{\|A - QR\|}{\|A\|} \tag{5.1}$$

As already explained in Section 4.5, the algorithms are backward stable. For this reason we expect accuracy to be of the order of machine precision. The accuracy of the two methods is almost the same: half of the times $resumeQR$ is more accurate, and viceversa. The difference between the two methods is of the order $1e^{-16}$ or greater. The results are shown in Table 5.3.

Z	ResumeQR	QR	Delta	Z	ResumeQR	QR	Delta
05	1.092e-15	8.827e-16	2.093e-16	45	1.631e-15	1.515e-15	1.154e-16
10	9.614e-16	1.155e-15	-1.94e-16	50	1.591e-15	1.355e-15	2.365e-16
15	1.957e-15	1.326e-15	6.304e-16	55	1.403e-15	1.446e-15	-4.31e-17
20	2.339e-15	1.628e-15	7.110e-16	60	1.320e-15	1.340e-15	-1.99e-17
25	1.136e-15	1.229e-15	-9.31e-17	65	1.156e-15	1.144e-15	1.203e-17
30	1.427e-15	1.696e-15	-2.69e-16	70	1.219e-15	1.728e-15	-5.09e-16
35	1.085e-15	1.242e-15	-1.57e-16	75	1.118e-15	1.149e-15	-3.02e-17
40	1.132e-15	1.264e-15	-1.31e-16	80	1.247e-15	1.190e-15	5.685e-17

Table 5.3: Comparison in terms of Accuracy between the LSP using $resumeQR$ and QR from scratch.

5.3. Off-the-shelf Experiment

Finally, we want to compare our solution with the MatLab off-the-shelf solver. This experiment is composed of two sub-experiments:

1. We compare the time of the our implementation of the LSP with the Algorithm 3 with respect to the MatLab solver.
2. We compare the accuracy of the Algorithm 3 with respect to the MatLab function *qr(Â,0)*.

The MatLab solver is faster than our implementation. Precisely, as shown in Table 5.4, where the fourth column is the difference between the second one and the third one, the Delta between the two methods becomes higher row by row. The time of our implementation grows a lot with the increase of the number of columns, instead the time of the MatLab solver grows slightly.

Z	QR	MatLab-QR	Delta	Z	QR	MatLab-QR	Delta
05	0.00931392	0.00109202	0.008	45	0.09275402	0.00330422	0.089
10	0.01131752	0.00141102	0.009	50	0.10529892	0.00374082	0.101
15	0.01417312	0.00144352	0.012	55	0.12555392	0.00405202	0.121
20	0.02403142	0.00190312	0.022	60	0.14420022	0.00447002	0.139
25	0.03160342	0.00222412	0.029	65	0.16161802	0.00508682	0.156
30	0.04420892	0.00257052	0.041	70	0.18499322	0.00701472	0.177
35	0.05705612	0.00302572	0.054	75	0.21651482	0.00655572	0.209
40	0.07197042	0.00353092	0.068	80	0.24427222	0.00644502	0.237

Table 5.4: Comparison in terms of Time between our QR implementation and the Matlab off-the-shelf solver.

The accuracy is computed as in Equation (5.1). The Table 5.5 shows the difference in Accuracy between the two methods. Again, the MATLAB off-the-shelf it's better, but the Delta values are low (order of $1e^{-16}$).

Z	QR	MatLab-QR	Delta	Z	QR	MatLab-QR	Delta
05	5.867e-16	5.382e-16	4.85e-17	45	1.357e-15	1.102e-15	2.54e-16
10	1.618e-15	1.272e-15	3.46e-16	50	1.100e-15	1.075e-15	2.44e-17
15	2.568e-15	1.126e-15	1.44e-15	55	1.181e-15	1.085e-15	9.63e-17
20	1.301e-15	1.175e-15	1.25e-16	60	1.801e-15	1.020e-15	7.81e-16
25	1.163e-15	1.026e-15	1.37e-16	65	1.133e-15	1.027e-15	1.06e-16
30	1.479e-15	9.511e-16	5.28e-16	70	1.432e-15	1.044e-15	3.88e-16
35	1.372e-15	1.089e-15	2.83e-16	75	1.493e-15	9.917e-16	5.01e-16
40	1.768e-15	1.062e-15	7.06e-16	80	1.220e-15	1.009e-15	2.10e-16

Table 5.5: Comparison in terms of Accuracy between our QR implementation and the Matlab off-the-shelf solver.

A. DOCUMENTATION

In this section, we briefly describe the documentation of the MatLab implementation of this project. The code we are going to describe is hosted at `https://github.com/andrew-r96/resumed-qr.git`.
The most principal functions are implemented in the following files:

- *qr_v.m*: it implements the Algorithm 1 through the Algorithm 2;
- *resumeQR.m*: it implements the Algorithm 3;
- *solve.m*: it implements the Algorithm 4 using Algorithm 5.

There are six live function files which constitutes the experiments. These files are organized as follows:

- *Experiment*_1.*mlx* is the Experiment 5.1;
- *Experiment*_2.*mlx*, *Experiment*_3.*mlx* and *Experiment*_4.*mlx* together constitute the second Experiment 5.2;
- *Experiment*_5.*mlx* and *Experiment*_5.*mlx* together constitute the second Experiment 5.3;

These live function files use the following ausiliar files to run the experiments:

- *importfile.m*: it implements an helper function to easily load the dataset;
- *augment.m*: it implements a function that takes in input a matrix A and an integer k and add k new columns to A. This is done by applying 5 hard-coded functions to the columns of A in a random fashion. The algorithm avoid to apply the same function to the same column more than once, since this would lead to non full column rank;
- *get_Q.m*: it implements an helper function that takes the householder vectors as input and it constructs the corresponding matrix Q.

To execute the code it is sufficient to open one of the experiment file, add the folder to the MatLab path and run the script.
Each experiment generates a file txt in which there will be stored the results and a plot which will shows graphically the behaviour of the tested functions.