Using the QR Factorization to swiftly update least squares problems

Oscar Olsson* and Tommy Ivarsson†

Centre for Mathematical Sciences, Numerical Analysis

The Faculty of Engineering at Lund University, LTH

June 5, 2014

Abstract

In this paper we study how to update the solution of the linear system $Ax = b$ after the matrix A is changed by addition or deletion of rows or columns.

Studying the QR Factorization of the system, more specifically, the factorization created by the Householder reflection algorithm, we find that we can split the algorithm in two parts. The result from the first part is trivial to update and is the only dependency for calculating the second part.

We find that not only can this save a considerable amount of time when solving least squares problems but the algorithm is also very easy to implement.

*osse.olsson@gmail.com

†tommy.ivarsson@gmail.com

Contents

1 Introduction

> If everything seems under control, you're not going fast enough.
>
> Mario Andretti

1.1 Motivation and overview

Schoolbook examples of algorithms that solve systems of equations typically have the time complexity $\mathrm{O}(n^3)$. In fact the fastest known algorithm, Williams algorithm, is $\mathrm{O}(n^{2.3727})$[8, p. 1]. In other words, big systems cause big problems in terms of computation times.

Let us assume that we, in the general case, can not go any faster than Williams. Under this assumption one could still look at specific cases for which a faster solution may be found.

Imagine we have just solved system S_1 and want to add or delete rows to or from it, creating system S_2. Could we use the structures we created solving S_1 to speed up the process of solving S_2?

In mathematical terms, assume the existence of a solution to the linear system $A\bar{x} = b$, with $A \in \mathbb{R}^{n \times m}$ where n is very large. Can we under this assumption minimize the computations required to solve the new system with our modified A?

1.2 Preliminaries and notation

- A: Upper-case letters describe matrices.
- $A[i, j]$: Row i, column j of matrix A.
- $A[i : j, k : l]$: The submatrix created from row i to and including j and column k to and including l of matrix A.
- $a_{i,j}$: Element i, j of a matrix.
- $\bar{a}_{*,j}$: Column j of a matrix.
- $\bar{a}_{i,*}$: Row i of a matrix.

- a: Lower-case letters describe scalars.
- $\bar{a}$: Lower-case letters with a bar describe vectors.
- A^T or $\bar{a}^\mathsf{T}$: Superscript T describes that the entity is transposed.

2 Research

> If we knew what it was we were doing, it would not be called research, would it?
>
> Albert Einstein

2.1 Least squares

The standard approach to approximate a solution to an overdetermined system is the least squares method. The name of the method comes from the fact that it minimizes the sum of the errors squared in the results of every equation.

Overdetermined systems are everywhere. With this in mind, one could quite easily imagine a machine in the real world that depends on the solution to such a system to operate. A dependency on such a system would likely be due to the system describing the real world to the machine in one way or another. Looking at it this way the overdetermined system becomes a log of reality.

Our machine would require updates of what is happening in the real world, however it is likely that the reaction of the machine does not only depend on the update but also on the previous state. Obviously we would not want to solve the entire system again. What we would like to do is somehow use the previous calculations in order to get the solution to the updated system faster than if we solved the new system from scratch.

These things considered, we chose to concentrate our efforts on solving least squares problems that have been slightly modified.

2.2 Solving large systems computationally

Computationally solving systems of equations using traditional methods could introduce huge errors if the matrices used in the calculations are illconditioned. The standard solution to this problem is using the QR Factorization, $A = QR$. Q is an orthogonal matrix and R is upper triangular. This gives us $A\bar{x} = \bar{b} \Leftrightarrow QR\bar{x} = \bar{b} \Leftrightarrow R\bar{x} = Q^{\mathsf{T}}\bar{b}$. As R is upper triangular we can then solve the system using back substitution.

2.2.1 QR Factorization algorithms

There are several algorithms available for QR Factorization. We were interested in a fast but numerically stable algorithm.

- Graham Schmidt
  Not numerically stable [7, p. 224].
- Householder reflection
  Requires fewer operations than Graham Schmidt and is more stable in the sense of rounding errors and error amplification [7, p. 224].
- Givens rotations
  Has numerical properties as favorable as those for Householder reflection [2, p. 217]. However, not as efficient as Householder reflection [4, p. 3].

Under these considerations we chose to study the Householder reflection algorithm.

2.3 The Householder reflection algorithm

We consider the general case where A is any $n \times m$ matrix,

$$A = \begin{bmatrix} a_{1,1} & a_{1,2} & a_{1,3} & \dots & a_{1,m} \\ a_{2,1} & a_{2,2} & a_{2,3} & \dots & a_{2,m} \\ a_{3,1} & a_{3,2} & a_{3,3} & \dots & a_{3,m} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ a_{n,1} & a_{n,2} & a_{n,3} & \dots & a_{n,n} \end{bmatrix}$$

With each iteration i of the algorithm we create the matrix H_i. After m iterations we have $R = H_m \dots H_1 A$ and $Q^\mathsf{T}\bar{b} = H_m \dots H_1 \bar{b}$. This enables us to solve $R\bar{x} = Q^\mathsf{T} b$.

To construct H_i we let $\bar{u}_i$ be the i:th column excluding the first $i-1$ elements of $H_{i-1} \dots H_1 A$, that is $\bar{u}_i = (H_{i-1} \dots H_1 A)[i : n, i]$. Let $w_i = -\frac{\bar{u}_i[1]}{|\bar{u}_i[1]|}|\bar{u}_i|$, the Euclidian distance of $\bar{u}_i$ with opposite sign of $\bar{u}_i[1]$, and $\bar{v}_i = \bar{w}_i - \bar{u}_i$, where,

$$\bar{w}_i = \begin{bmatrix} w_i \\ 0 \\ \vdots \\ 0 \end{bmatrix}$$

The sign of w_i is chosen to be opposite of the first element in $\bar{u}_i$ to prevent possible loss of accuracy in case $w_i \approx \bar{u}_i[1]$. Then $H_i = I - 2\frac{\bar{v}\bar{v}^\mathsf{T}}{\bar{v}^\mathsf{T}\bar{v}}$.

2.3.1 Calculating H_1A and $H_1\bar{b}$

Our first iteration will create H_1A and $H_1\bar{b}$. Here,

$$\bar{v}_1 = \begin{bmatrix} w_1 - a_{1,1} \\ -a_{2,1} \\ \vdots \\ -a_{n,1} \end{bmatrix},\ w_1 = \pm\sqrt{\sum_1^n a_{i,1}^2}$$

The projection matrix P is constructed by the matrix $\bar{v}_1\bar{v}_1^\mathsf{T}$ where each element is divided by the scalar $\bar{v}_1^\mathsf{T}\bar{v}_1$.

$$\bar{v}_1^\mathsf{T}\bar{v}_1 = 2w_1(w_1 - a_{1,1})$$

$$\bar{v}_1\bar{v}_1^\mathsf{T} = \begin{bmatrix} (w_1 - a_{1,1})^2 & -a_{2,1}(w_1 - a_{1,1}) & \dots & -a_{n,1}(w_1 - a_{1,1}) \\ -a_{2,1}(w_1 - a_{1,1}) & a_{2,1}^2 & \dots & a_{2,1}a_{n,1} \\ \vdots & \vdots & \ddots & \vdots \\ -a_{n,1}(w_1 - a_{1,1}) & a_{n,1}a_{2,1} & \dots & a_{n,1}^2 \end{bmatrix}$$

As $H_1 = I - 2P$ we find that each element of the matrix H_1, and thereby also H_1A and $H_1\bar{b}$, can be determined individually using nothing but the elements in A and w_1. This means that we can find expressions for each element in H_1A and $H_1\bar{b}$, consequently eliminating the need to calculate any of the structures leading up to this point in the traditional sense of the algorithm.

The elements of H_1A

$$H_1A[i,j] = \begin{cases} 0 & \text{if } j = 1 \text{ and } i > 1 \\ \dfrac{\bar{a}_{*,1} \cdot \bar{a}_{*,i}}{w_1} & \text{if } i = 1 \\ a_{i,j} + \dfrac{a_{i,1}(w_1 a_{1,j} - \bar{a}_{*,1} \cdot \bar{a}_{*,j})}{w_1(w_1 - a_{1,1})} & \text{otherwise} \end{cases}$$

The elements of $H_1\bar{b}$

$$H_1\bar{b}[i] = \begin{cases} \dfrac{\bar{a}_{*,1} \cdot \bar{y}_*}{w_1} & \text{if } i = 1 \\ y_i + \dfrac{a_{i,1}(w_1 y_1 - \bar{a}_{*,1} \cdot \bar{y}_*)}{w_1(w_1 - a_{1,1})} & \text{otherwise} \end{cases}$$

This first iteration leaves us with H_1A and $H_1\bar{b}$. The first row and column in H_1A have now been made triangular and are thereby in their final correct state. This means that one can view $H_1A[2:n, 2:m]$ as the non-triangular submatrix. The first element in $H_1\bar{b}$ is now also in its final correct state.

2.3.2 Calculating H_2H_1A and $H_2H_1\bar{b}$

Calculating H_2 is almost as easy as reiterating the process to calculate H_1 using H_1A and $H_1\bar{b}$ instead of A and $\bar{b}$ as the basis of our calculations. However we will only be using the non-triangular part of H_1A, $H_1A[2:n, 2:m]$.

If one were to consider A in our first iteration as the initial input, the input for the second iteration would be $H_1A[2:n, 2:m]$. This means that the output from our second iteration is a matrix with one less row and column than H_1. We call this matrix $\hat{H}_2$. $\hat{H}_2$ has a static relationship to H_2 described in (1).

$$H_2 = \begin{bmatrix} 1 & 0 & \dots & 0 \\ 0 & & & \\ \vdots & & \hat{H}_2 & \\ 0 & & & \end{bmatrix} \tag{1}$$

Just as when we calculated H_1, every element in $\hat{H}_2$ can be expressed using only the elements in the input matrix, in this case $H_1A[2:n, 2:m]$, and w_i.

$$w_2 = \sqrt{\sum_{i=2}^{n} H_1A(i,1)^2}$$

The elements of H_2H_1A

Let $t_{i,j} = H_1A[i+1, j+1]$.

$$H_2H_1A[i,j] = \begin{cases} 0 & \text{if } i > j \text{ and } j \leq 2 \\ H_1A[i,j] & \text{if } i = 1 \\ \dfrac{\bar{t}_{*,1} \cdot \bar{t}_{*,j-1}}{w_2} & \text{if } i = 2 \text{ and } j > 1 \\ t_{i-1,j-1} + \dfrac{t_{i-1,1}(w_2 t_{1,j-1} - \bar{t}_{*,1} \cdot \bar{t}_{*,j-1})}{w_2(w_2 - t_{1,1})} & \text{otherwise} \end{cases}$$

$$\bar{t}_{*,q} \cdot \bar{t}_{*,k} = \bar{a}_{*,q+1} \cdot \bar{a}_{*,k+1} - \frac{(\bar{a}_{*,1} \cdot \bar{a}_{*,k+1})(\bar{a}_{*,1} \cdot \bar{a}_{*,q+1})}{(\bar{a}_{*,1} \cdot \bar{a}_{*,1})}$$

$$\bar{a}_{*,1} \cdot \bar{a}_{*,1} = w_1^2$$

The elements of $H_2H_1\bar{b}$

Let $s_i = H_1\bar{b}[i+1]$ and $t_{i,j} = H_1A[i+1, j+1]$.

$$H_2H_1\bar{b} = \begin{cases} H_1\bar{b}[i] & \text{if } i = 1 \\ \dfrac{\bar{t}_{*,1} \cdot \bar{s}_*}{w_2} & \text{if } i = 2 \\ s_{i-1} + \dfrac{t_{i-1,1}(w_2 s_1 - \bar{t}_{*,1} \cdot \bar{s}_*)}{w_2(w_2 - t_{1,1})} & \text{otherwise} \end{cases}$$

$$\bar{t}_{*,k} \cdot \bar{s}_* = \bar{a}_{*,k+1} \cdot \bar{y}_* - \frac{\bar{a}_{*,1} \cdot \bar{y}_*}{\bar{a}_{*,1} \cdot \bar{a}_{*,1}} \bar{a}_{*,1} \cdot \bar{a}_{*,k+1}$$

$$\bar{a}_{*,1} \cdot \bar{a}_{*,1} = w_1^2$$

The second row of H_2H_1A depends on the dot products between the second column and all columns to its right in A, as well as the dot products between the first column and all other columns in A. All of these dot products have to be available at this point.

2.3.3 Calculating $H_k \dots H_1A$ and $H_k \dots H_1\bar{b}$

In order to arrive at the point where we have the structures required to solve the system we will need m iterations, that is, as many iterations as A has columns. $H_k \dots H_1A$ will with each iteration move towards its final triangular state by making one row and one column triangular per iteration. Similarly $H_k \dots H_1\bar{b}$ will gain one correct element per iteration before arriving at its final state, after the last iteration, where every element is correct.

In short, iteration k will not change the first $k-1$ rows in neither $H_k \dots H_1A$ nor $H_k \dots H_1\bar{b}$ and will render the k:th row correct in both.

The elements of $H_k \dots H_1 A$

Let

$$t_{i,j} = H_{k-1} \dots H_1 A[i+k-1, j+k-1]$$
$$s_{i,j} = H_{k-2} \dots H_1 A[i+k-2, j+k-2]$$

then

$$H_k \dots H_1 A[i,j] = \begin{cases} 0 & \text{if } i > j \text{ and } j \leq k \\ H_{k-1} \dots H_1 A[i,j] & \text{if } i < k \\ \dfrac{\bar{t}_{*,1} \cdot \bar{t}_{*,j-k+1}}{w_k} & \text{if } i = k \text{ and } j \geq k \\ t_{i-k+1,j-k+1} + \dfrac{t_{i-k+1,1}(w_k t_{1,j-k+1} - \bar{t}_{*,1} \cdot \bar{t}_{*,j-k+1})}{w_k(w_k - t_{1,1})} & \text{otherwise} \end{cases}$$

$$\bar{t}_{*,q} \cdot \bar{t}_{*,p} = \bar{s}_{*,q+1} \cdot \bar{s}_{*,p+1} - (\bar{s}_{*,1} \cdot \bar{s}_{*,p+1})(\bar{s}_{*,1} \cdot \bar{s}_{*,q+1})$$

Clearly we are very dependent on the dot products between all the columns in A. During the k:th iteration, the k:th row in $H_k \dots H_1 A$ will be calculated from the dot products of the k:th column and all the columns to its right in A.

The elements of $H_k \dots H_1 \bar{b}$

Let

$$
\begin{aligned}
s_i &= H_{k-1} \dots H_1 \bar{b}[i+k-1] \\
\mu_i &= H_{k-2} \dots H_1 \bar{b}[i+k-2] \\
t_{i,j} &= H_{k-1} \dots H_1 A[i+k-1, j+k-1] \\
\gamma_{i,j} &= H_{k-2} \dots H_1 A[i+k-2, j+k-2]
\end{aligned}
$$

then

$$
H_k \dots H_1 \bar{b} = \begin{cases} H_{k-1} \dots H_1 \bar{b}[i] & \text{if } i < k \\ \dfrac{\bar{t}_{*,1} \cdot \bar{s}_*}{w_k} & \text{if } i = k \\ s_{i-1} + \dfrac{t_{i-1,1}(w_k s_1 - \bar{t}_{*,1} \cdot \bar{s}_*)}{w_k(w_k - t_{1,1})} & \text{otherwise} \end{cases}
$$

$$
\bar{t}_{*,p} \cdot \bar{s}_* = \bar{\gamma}_{*,p+1} \cdot \bar{\mu}_* - (\bar{\gamma}_{*,1} \cdot \bar{\mu}_*)(\bar{\gamma}_{*,1} \cdot \bar{\gamma}_{*,p+1})
$$

When $k = m$ we have reached the point where we have calculated $Q^\mathsf{T}\bar{b} = H_m \dots H_1 \bar{b}$ and $R = H_m \dots H_1 A$. The system can now easily be solved using the equation $R\bar{x} = Q^\mathsf{T}\bar{b}$ and back substitution.

2.3.4 The value w_k

It may be apparent to the reader that w_k has no effect on the exact solution to the system. As can be seen in $H_k \dots H_1 A$ and $H_k \dots H_1 \bar{b}$ each row is divided by w_k in both matrices so this does not affect the equality and therefore does not affect the solution. However, w_k affects the numerical properties of back substitution.

2.4 The Recycling Householder Algorithm

The algorithm will be expressed as matrix-vector operations. This not only lets us express it neatly, which improves readability, but also has the beautiful consequence that easier implementation.

Furthermore, in yet another attempt to improve readability, we redefine addition and subtraction between matrices which have differing dimensions.

When such a situation arises it will be implied that the smaller matrix be padded with zeros on the upper and left side until the dimensions of both matrices agree.

The algorithm, expressed as matrix operations

Define $U(M)$ as the upper triangular part of M, including the diagonal, and $I_k(c)$ as the identity matrix with the k:th diagonal element replaced by c and let

$$\begin{aligned} G_1 &= I_1(\frac{1}{\sqrt{\bar{a}_{*,1} \cdot \bar{a}_{*,1}}})U(A^\mathsf{T}A) \\ T_1 &= I_1(\frac{1}{\sqrt{\bar{a}_{*,1} \cdot \bar{a}_{*,1}}})A^\mathsf{T}b \\ \bar{v}_k &= G_k[k, (k+1) : m]. \end{aligned}$$

Then

$$\begin{aligned} G_k &= I_k(\frac{1}{\sqrt{G_{k-1}[k,k] - G_{k-1}[k-1,k]^2}})(G_{k-1} - U(\bar{v}_{k-1}^\mathsf{T}\bar{v}_{k-1})) \\ T_k &= I_k(\frac{1}{\sqrt{G_{k-1}[k,k] - G_{k-1}[k-1,k]^2}})(T_{k-1} - T_{k-1}[k-1]\bar{v}_{k-1}^\mathsf{T}) \\ R &= G_m \\ Q^\mathsf{T}\bar{b} &= T_m \end{aligned}$$

NOTE: $G_{k-1}[k,k] - G_{k-1}[k-1,k]^2 = G_k[k,k]$ and multiplying with $I_k(\frac{1}{c})$ is equivalent to dividing each element in row k with the element c.

Expressing the algorithm this way makes it clear that it has two distinct parts:

1. calculation of the basis for the factorization, G_1, and
2. factorization of the basis.

Calculating the dot products, clearly the most computationally heavy part of the algorithm, takes place during the first part. Every step thereafter relies

upon them. This means that by saving them we can update them easily if we add or delete rows from the system. This lets us bypass having to calculate the dot products from scratch, thereby massively reducing the amount of computations required to solve the new system.

2.4.1 MATLAB implementation

Listing 1: First part of algorithm

```
function [G0,T0] = adaptiveqr_prepare(A, b)
  G0 = triu(A'*A);
  T0 = A'*b;
end
```

Listing 2: Second part of algorithm

```
function [R,d] = adaptiveqr_factor(R, d, varargin)
  [m,~] = size(R);
  if nargin > 2
    m0 = varargin{1};
  else
    [R,d] = finalize_row(R, d, 1);
    m0 = 2;
  end
  % If starting row in greater than 2, perform facotrization
     related to the
  % starting row and below done in the iterations 2 to the
     starting row index
  for k = 2:(m0-1)
    [R,d] = eliminate_row(R, d, k, m0, m);
  end
  % Perform the factorization from the starting row
  for k = m0:m
    [R,d] = eliminate_row(R, d, k, k, m);
    [R,d] = finalize_row(R, d, k);
  end
end

function [R,d] = eliminate_row(R, d, k, col, m)
    Gr = R(k-1,col:m);
    d(col:m,1) = d(col:m,1) - (d(k-1).*Gr');
    R(col:m,col:m) = R(col:m,col:m) - triu(Gr'*Gr);
end
```

```
function [R,d] = finalize_row(R, d, k)
    w = sqrt(R(k,k));
    R(k,:) = R(k,:)./w;
    d(k) = d(k)/w;
end
```

Listing 3: Solve system given first part

```
function [x, R, d] = adaptiveqr_solve(G0, T0)
  [R,d] = adaptiveqr_factor(G0,T0);
  x = R\d;
end
```

The *prepare*-function completes in $nm^2 + 3nm - m^2 - \frac{3m}{2} \Rightarrow \mathrm{O}\left(nm^2\right)$ operations, *factor* requires $\frac{m^3}{3} + \frac{m^2}{2} + \frac{m}{6} \Rightarrow \mathrm{O}\left(m^3\right)$, a complete solution has the complexity $\mathrm{O}\left(nm^2 + m^3\right)$, where n is the number of rows and m is the number of columns.

2.4.2 Add rows

When adding rows, one would simply add the dot products of the update to the existing dot products. This implies that the dot products of a previous solution, G_1 and T_1, must be available.

Adding rows to a least squares system

$$\begin{cases} \hat{A} = \begin{bmatrix} A \\ A_\delta \end{bmatrix} \\ \hat{b} = \begin{bmatrix} b \\ b_\delta \end{bmatrix} \end{cases} \Rightarrow \begin{cases} \hat{A}^\mathsf{T}\hat{A} = A^\mathsf{T}A + A_\delta^\mathsf{T}A_\delta \\ \hat{A}^\mathsf{T}\hat{b} = A^\mathsf{T}b + A_\delta^\mathsf{T}b_\delta \end{cases} \tag{2}$$

Listing 4: Add rows to first part

```
function [G0,T0] = adaptiveqr_add_rows(G0, T0, Ahat, Bhat)
```

```
  G0 = G0 + triu(Ahat'*Ahat);
  T0 = T0 + Ahat'*Bhat;
end
```

Adding another p rows to the system would only require $pm^2 + 3pm - \frac{m^2}{2} \Rightarrow \mathrm{O}(pm^2)$ additional computations.

This function only replaces the *prepare*-function. The rest of the *solve*-function still requires $\mathrm{O}(m^3)$ calculations.

2.4.3 Delete rows

Deleting rows works similarly to adding rows. However, in this case, the dot products are instead subtracted from the original dot products.

Deleting rows from a least squares system

$$\begin{cases} A = \begin{bmatrix} \hat{A} \\ A_\delta \end{bmatrix} \\ b = \begin{bmatrix} \hat{b} \\ b_\delta \end{bmatrix} \end{cases} \Rightarrow \begin{cases} \hat{A}^\mathsf{T}\hat{A} = & A^\mathsf{T}A - A_\delta^\mathsf{T}A_\delta \\ \hat{A}^\mathsf{T}\hat{b} = & A^\mathsf{T}b - A_\delta^\mathsf{T}b_\delta \end{cases} \tag{3}$$

Listing 5: Delete rows from first part

```
function [G0, T0] = adaptiveqr_delete_rows(G0, T0, a, b)
  G0 = G0 - triu(a'*a);
  T0 = T0 - a'*b;
end
```

2.4.4 Add columns

Adding another p columns to the $n \times m$ matrix A requires $pm + p^2$ new dot products, but no previous dot product has to be updated.

Adding columns to a least squares system

$$\hat{A} = \begin{bmatrix} A & A_\delta \end{bmatrix} \Rightarrow \begin{cases} \hat{A}^\mathsf{T}\hat{A} &= \begin{bmatrix} A^\mathsf{T}A & A^\mathsf{T}A_\delta \\ A_\delta^\mathsf{T}A & A_\delta^\mathsf{T}A_\delta \end{bmatrix} \\ \hat{A}^\mathsf{T}b &= \begin{bmatrix} A^\mathsf{T}b \\ A_\delta^\mathsf{T}b \end{bmatrix} \end{cases} \tag{4}$$

Listing 6: Add columns to first part

```
function [G0,T0,A] = adaptiveqr_add_columns(G0,T0,A,b,cols)
  [n,m] = size(A);
  [~,p] = size(cols);
  G0(:,m+1:m+p) = A'*cols;
  G0(m+1:m+p,m+1:m+p) = triu(cols'*cols);
  T0(m+1:m+p) = cols'*b;
  A(:,m+1:m+p) = cols;
end
```

The time complexity of this algorithm is $\mathrm{O}\big(np(m+p)\big)$, compared to the *prepare*-function, which has a time complexity of $\mathrm{O}\big(n(m+p)^2\big)$.

2.4.5 Delete columns

Deleting p columns is simply a matter of removing all dot products related to any of the deleted columns. More specifically, deleting the k:th column is accomplished by removing the k:th column and the k:th row in G_0. This operation depends on the implementation of the matrix but the worst case is $\mathrm{O}\big((m-k)^2\big)$ as every column to the right of the k:th column, and every row below the k:th row must be realigned. Another $\mathrm{O}\big((m-p)^3\big)$ operations are required to perform the factorization.

Deleting columns from a least squares system

$$\begin{cases} A = \begin{bmatrix} \hat{A}_1 & A_\delta & \hat{A}_2 \end{bmatrix} \\ \hat{A} = \begin{bmatrix} \hat{A}_1 & \hat{A}_2 \end{bmatrix} \end{cases} \Rightarrow \begin{cases} \hat{A}^\mathsf{T}\hat{A} &= \begin{bmatrix} \hat{A}_1^\mathsf{T}\hat{A}_1 & \hat{A}_1^\mathsf{T}\hat{A}_2 \\ \hat{A}_2^\mathsf{T}\hat{A}_1 & \hat{A}_2^\mathsf{T}\hat{A}_2 \end{bmatrix} \quad (5) \\ \hat{A}^\mathsf{T}b &= \begin{bmatrix} \hat{A}_1^\mathsf{T}b \\ \hat{A}_2^\mathsf{T}b \end{bmatrix} \quad (6) \end{cases}$$

Removing column i from A is equivalent to removing column i and row i from $A^\mathsf{T}A$. No new elements have to be calculated.

Listing 7: Delete columns from first and second part

```
function [G0, T0] = adaptiveqr_delete_columns(G0, T0, cols)
  G0(cols,:)=[];
  G0(:,cols)=[];
  T0(cols)=[];
end
```

A more efficient but still easily implemented algorithm would be to remove the k:th column and row from R and recalculating the elements below the k:th row.

Listing 8: Delete columns from first and second part

```
function [R, d, G0, T0] = adaptiveqr_delete_columns2(G0, T0, R
    , d, cols)
  [G0, T0] = adaptiveqr_delete_columns(G0, T0, cols);
  [m,~] = size(G0);
  R(cols,:)=[];
  R(:,cols)=[];
  d(cols)=[];
  m0 = min(cols);
  R(m0:m,m0:m) = G0(m0:m,m0:m);
  d(m0:m) = T0(m0:m);
  [R,d] = adaptiveqr_factor(R,d,m0);
end
```

This algorithm requires $\mathrm{O}\big(m^2 + (m-k-p)^3\big)$ with k being the index of the leftmost deleted column.

2.4.6 Change elements in A

If a few elements in A are changed, the dot products would have to be recalculated. Each dot product with a changed column is added to the difference $(a_{i,j} - \hat{a}_{i,j})a_{i,k}$, where i is the column with a changed element.

Listing 9: Changing elements in A

```
function [G0, T0] = change_elem(row, col, new_e, A, B)
  [~,m] = size(old_row)
  diff_e = new_e - A(row,col);
  for c = 1:m
    if c <= col
      G0(c,col) = G0(c,col) + diff_e*A(row,c);
    else
      G0(col,c) = G0(col,c) + diff_e*A(row,c);
    end
  end
  T0(col) = T0(col) + diff_e*B(row);
```

This would require $\mathrm{O}\big(pm\big)$ calculations for p changed elements. The algorithm actually only requires the rows from A and $\bar{b}$ which are affected by the change but for readability we included the full matrices.

2.4.7 Taking advantage of sparsity

Sparsity in A could be considered in the calculation of each dot product. For example, adding a row with only one element would only require recalculation of the dot product with that column and itself.

Adding a diagonal block to A,

$$\hat{A} = \begin{bmatrix} & & & 0 & \dots & 0 \\ & A & & \vdots & \vdots & \vdots \\ & & & 0 & \dots & 0 \\ 0 & \dots & 0 & & & \\ \vdots & \vdots & \vdots & & C & \\ 0 & \dots & 0 & & & \end{bmatrix}$$

would not require the previous dot product to be recalculated, however the dot products with the new columns would have to be calculated. For a $q \times p$ matrix, the number of operations to compute the dot products would be $\mathrm{O}(qp^2)$.

2.4.8 Limitations

As the algorithm depends on the dot products of all the columns in A a very large representation of floating point numbers might be required. The dot product of any two columns, k and l, is limited by $|\bar{a}_{*,k}^{\mathsf{T}}\bar{a}_{*,l}| \leq ||A||^2$. The Cauchy-Schwarz inequality states that $|v^{\mathsf{T}}u| \leq ||v|| \cdot ||u||$ for any vectors u and v. Furthermore, $\bar{a}_{*,k} = A\bar{e}_k$ where $\bar{a}_{*,k}$ is the k:th column of A and e_k is the k:th unit vector. This implies the inequality $||\bar{a}_{*,k}|| \leq ||A|| \cdot ||\bar{e}_k||$ with $||\bar{e}_k|| = 1$ which gives $||\bar{a}_{*,k}|| \cdot ||\bar{a}_{*,l}|| \leq ||A||^2$.

The above stated limitation might only become a problem if $||A||^2$ is larger than the maximum floating point value on the machine.

2.5 Testing the algorithm

The typical approach to testing an algorithm would be to run the algorithm for different inputs and compare our results to the correct results. However this would require specific input which could lead to biased results.

Instead we test specific characteristics of the solution which can be determined by the input. This enables us to generate random input which avoids biased test results.

2.5.1 Testing the relative error

1. Generate random A with condition number $\kappa \leq 10$.
2. Generate random $\bar{x}$ with $||\bar{x}|| = 1$.
3. Compute $\bar{b} = A\bar{x}$.
4. Solve $A\hat{x} = \bar{b}$ for $\hat{x}$.
5. Compute $\delta\bar{b} = A\hat{x} - \bar{b}$.
6. Check that $||\hat{x} - \bar{x}|| \leq \kappa\frac{||\delta\bar{b}||}{||\bar{b}||}$.

We ran this test on matrices where $A \in \mathbb{R}^{n \times n}$ and $n = [10^2 \dots 10^3]$ in steps of 10^2. Each size tested with 1000 different matrices. As can be seen in Figure 1, $||\hat{x} - \bar{x}||$ was strictly smaller than $\kappa \frac{||\delta \bar{b}||}{||b||}$.

Listing 10: Testing the relative error

```
function [result] = test_relative_error()
    result = [];
    nbr_tests = 1000;
    cond = 10;
    rnd_space = 1000;
    for n = 100:100:1000
        fprintf('%d: ', n);
        for i = 1:nbr_tests

            % Generate As with K <= 10
            A = randnCond(n,n,cond,rnd_space);

            % Generate x with norm(x) = 1
            x = rand(n,1)*rnd_space - rnd_space/2;
            x = x/norm(x);

            % Compute b = Ax
            b = A*x;

            % Compute delta_b = A*tilde_x-b
            AQR = AdaptiveQR(A,b);
            [sol_x,~,~] = AQR.solve();
            delta_b = A * sol_x - b;

            % norm(tilde_x - x) <= K*norm(delta_b)/norm(b)
            result = [result; n norm(sol_x - x) cond * norm(
                delta_b) / norm(b)];
            fprintf('.');
        end
        fprintf('\n');
    end
```

2.5.2 Testing the numerical stability

Testing the numerical stability is far from trivial. To capture any numerical instability in the algorithm we must first ensure nothing but the algorithm affects the result, hence, well conditioned matrices are required. If we were

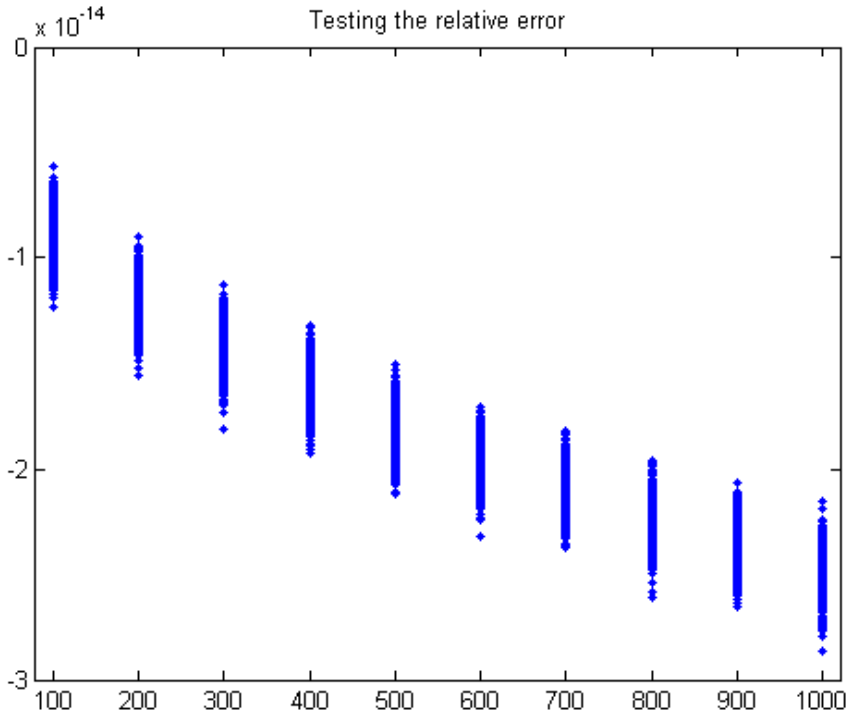

Figure 1: The difference between $||\hat{x} - \bar{x}||$ and $\kappa \frac{||\delta\bar{b}||}{||\bar{b}||}$.

to solve random well conditioned systems, we would expect random solutions satisfying $\bar{x} = A^{-1}\bar{b}$. However computing $A^{-1}\bar{b}$ might introduce errors and reduce the certainty of the test. Instead, if we generate A with $||A^{-1}|| = 1$ and $\bar{b}$ with $||\bar{b}|| = 1$, we could test the algorithm by checking the constraint $||\bar{x}|| \leq 1$. If $||\bar{x}||$ ever becomes larger than 1 the algorithm suffers from error magnification. However, for a given A and $\bar{b}$, the inequality $||\bar{x}|| \leq 1$ might hold even though the error is magnified. In fact, equality is reached when $\bar{b}$ is aligned with the signular vector of A corresponding to the smallest singular value. This is not likely to occur for a single choice of A and $\bar{b}$, but generating several $\bar{b}$'s for each A and testing with several A's would increase the chance of hitting such a $\bar{b}$.

1. Generate random A with $||A^{-1}|| = 1$.
2. Generate random $\bar{b}$ with $||\bar{b}|| = 1$.
3. Solve $A\bar{x} = \bar{b}$ for $\bar{x}$ with our algorithm.
4. Check that $||\bar{x}|| = ||A^{-1}\bar{b}|| \leq ||A^{-1}|| \cdot ||\bar{b}|| \leq 1$.

This test was run with 1000 different $A \in \mathbb{R}^{100\times100}$. Each A was tested with 1000 different $\bar{b}$ vectors. The difference $||\bar{x}|| - 1$ was plotted, see Figure 2.

The largest error encountered in the test was $\max_i \varepsilon_i = 17\varepsilon_{mach} \approx 1.89 \cdot 10^{-15} < 2 \cdot 10^{-15}$, $\varepsilon_i = |\ ||x_i|| - 1\ |$. This is within our accepted error bounds. Only 0.1567% of the errors were larger than 10^{-15}.

Listing 11: Testing the delta space

```
function [result] = test_delta_space(test_sizes,
   tests_per_size)
  rnd_space = 1000;
  result = [];
  for n = test_sizes
    fprintf('n=%d', n);
    A = randnCond(n,n,1,rnd_space);
    for i = 1:tests_per_size
      B = randnCond(n,1,1,rnd_space);
      AQR = AdaptiveQR(A,B);
      [x,~,~] = AQR.solve();
      xn = norm(x);
      bn = norm(B);
      result = [result; n xn bn];
      fprintf('.');
```

```
    end
    fprintf('\n');
end
```

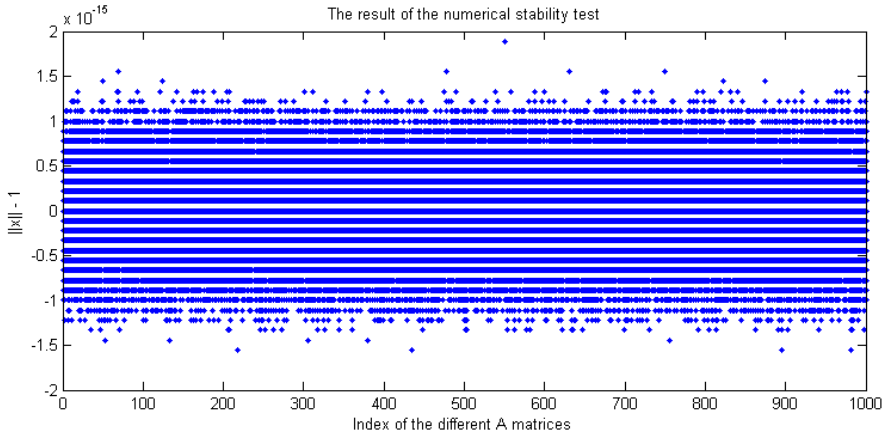

Figure 2: The difference $||\bar{x}|| - 1$ for each test, plotted with dots. The y-axis represents the index of the different A matrices. For each index 1000 dots are plotted, one for each $\bar{b}$ vector. $||\bar{x}|| - 1$ should always be 0, which is not the case. However, the distance from 0 is within accepted error bounds which implies success.

2.6 Previous work

In 2008 Sven Hammarling and Craig Lucas set out to solve the same problem but ended up finding a different solution [4]. They too use the QR Factorization of the system. However, instead of updating intermediate results, they update the final factorization.

2.6.1 Time complexity

The time complexity of both algorithms depend on which operation one wants to execute, see Table 1 where we compare the time complexity of Hammarling/Lucas to that of our algorithm.

Hammarling/Lucas outperform Olsson/Ivarsson in all mentioned updates. Most notable is the growth rate m^3 in Olsson/Ivarsson. However, this is negligable in comparison to the other growth rates.

Update	**Hammarling/Lucas**	**Olsson/Ivarsson**	**Note**
$\pm p$ rows	$\mathrm{O}\big(m^2 p\big)$	$\mathrm{O}\big(m^2 p + m^3\big)$	$p \ll n$
$+p$ cols	$\mathrm{O}\big(nmp\big)$	$\mathrm{O}\big(nmp + np^2 + m^3\big)$	$p \ll m$
$-p$ cols	$\mathrm{O}\big(mp(m-p-k) + p^3\big)$	$\mathrm{O}\big(m^2 + (m-k-p)^3\big)$	$p \ll m$

Table 1: Time complexity of updating with Hammarling/Lucas compared to Olsson/Ivarsson. When deleting columns, k is the left most index of the removed columns. With $A \in \mathbb{R}^{n \times m}$ and $m \ll n$.

2.6.2 Memory consumption

Another interesting aspect is the memory consumption, see table 2 where we compare the memory consumption.

Update	**Hammarling/Lucas**	**Olsson/Ivarsson**	**Note**
$+p$ rows	$\mathrm{O}\big(m^2 + mp\big)$	$\mathrm{O}\big(m^2 + mp\big)$	$p \ll n$
$-p$ rows	$\mathrm{O}\big(m^2\big)$	$\mathrm{O}\big(m^2 + mp\big)$	$p \ll n$
$+p$ cols	$\mathrm{O}\big(m^2 + np\big)$	$\mathrm{O}\big(nm + np + m^2\big)$	$p \ll m$
$-p$ cols	$\mathrm{O}\big(m^2\big)$	$\mathrm{O}\big(m^2\big)$	$p \ll m$

Table 2: Memory consumption of updating with Hammarling/Lucas compared to Olsson/Ivarsson. With $A \in \mathbb{R}^{n \times m}$ and $m \ll n$.

Hammarling/Lucas require less than or equal memory compared to Olsson/Ivarsson. The largest difference is found when adding columns.

2.6.3 Ease of implementation

How easy an algorithm is to implement is of course a subjective matter. Not only does it depend on which problem one is solving but it also depends on who is performing the implementation. Typically one would try to consider how complicated the structures and operations in the algorithm are. One could also look at the cyclomatic complexity and the number of lines of code.

MATLAB lets the programmer express an algorithm in ways very close to those typically used in mathematics. Therefore many mathematical algorithms, regardless of complicated structures and operations, are not difficult to write in MATLAB. This is true as the language MATLAB uses was designed specifically for mathematics. However, if one must use a programming

language which was not designed for mathematics the case might be completely different.

Our algorithm uses far less complicated structures and operations than Hammarling/Lucas and can also be implemented using fewer lines of code. Furthermore our cyclomatic complexity is lower. Under these considerations we would like to argue that our algorithm is easier to implement. Perhaps not much so using MATLAB, but very much so using more traditional programming languages like C/C++.

3 Conclusion

> I think and think for months and years. Ninety-nine times, the conclusion is false. The hundredth time I am right.
>
> Albert Einstein

The benefits of our solution are most apparent when updating rows in least squares problems. If p is the number of rows we want to add or delete, and m is the number of columns in the system, our time complexity for this operation is $\mathrm{O}\big(pm^2 + m^3\big)$. At first glance this may not look too good. However, considering that the least squares problem in general uses very few columns, m becomes insignificant and thereby one could argue a time complexity closer to $\mathrm{O}\big(p\big)$. In other words, we update a solution in time linearly proportional to the number of altered rows.

In fact, whilst not as fast, our algorithm lets us make almost any type of modification to the original system, and does so with increased performance compared to solving the new system from scratch. As our time complexities are all heavily dependent on m this is of course not true for systems with a square or nearly square matrix.

Yet another argument for using our algorithm when updating least squares problems is memory consumption. Adding or deleting rows does not require any knowledge of the individual elements in A. All we need is the dot products from the previous solution and the dot products from the matrix formed by the rows we are adding or deleting. Depending on the size of A this could significantly reduce the amount of memory our algorithm requires.

A somewhat beautiful consequence of the minimal memory consumption is that we could theoretically solve systems where n has any size by simply reading a few rows at a time and adding their dot products to the existing dot products. We would never have to keep a huge A-matrix in memory. However, in practice we might run into the problem of saving huge dot products efficiently.

Unfortunately the operations of adding or deleting columns, or changing a single element in A, all require access to individual elements in A.

Analysing our results, it becomes apparent that what we have found is in essence Gaussian elimination with a stabilisation factor w. Considering all the effort which has been put into different issues surrounding Gaussian

elimination, a few examples being the research of the numerical stability [5], LU-factorization [2, p. 94], and the overall speed [3], it would not be unreasonable to assume that our algorithm could benefit from existing research.

Furthermore, if the conditions are such that one knows that the systems in question are well conditioned one could replace the Gaussian elimination and the stabilisation factor with a faster algorithm, for example Strassen [6], Coppersmith-Winograd [1] or Williams algorithm [8].

Our work has been concentrated around general systems. If we were instead to assume a positive definite system we could perhaps exchange the QR Factorization for something like the Cholesky decomposition which could increase our performance.

Whilst we have not even tried to break the Williams barrier we have found a specific case where our algorithm allows for a faster solution than taking the general approach. The fact that using the specifics to our favor had such an impact will have serious consequences for how we approach problems in the future.

4 Future topics

> It is always wise to look ahead,
> but difficult to look further than
> you can see.
>
> Winston Churchill

As our solution depends on having the factorization of a related system available we assumed the knowledge that the systems in question were in fact very much alike. Clearly this is not always the case. One could very easily end up in a situation where it is not known if the new system is related to a previously solved system or not.

Consider a particular software that solves linear systems of equations. How would that software determine if a system has been solved previously unless the systems are solved succesively? Also, there could be cases in which a user wants to solve a system and is not aware that the system is related to a previously solved system.

Simply comparing systems would take too long but if there was an efficient way to determine whether two matrices are related or not the areas of application for our solution would widen immensely.

We have assumed a general system, that is, we have not made any assumptions about the systems being, for example, positive-definite. However, if we could assume constraints on the input there would be room to investigate other methods to factorize the system. For example, if one made the assumption that the systems are positive-definite, faster factorizations, as the Cholesky decomposition, could be considered.

It would be very interesting to find out if it is possible to use our algorithm to solve least squares problems where n is so large that we can not fit A in memory.

One could also look at using the algorithm and general idea for solving other types of problems. Could perhaps FFT of a continuous signal benefit from our work? At least DFT uses a matrix which grows deterministically and FFT is just a smart way of breaking DFT into subproblems [7, p. 475].

In fact one could quite possibly be able to use an adaptation of our work in many cases where a new solution is dependent on previously calculated dot products.

References

[1] D. Coppersmith and S. Winograd. Matrix multiplication via arithmetic progressions. In *Proceedings of the Nineteenth Annual ACM Symposium on Theory of Computing*, STOC '87, pages 1–6, New York, NY, USA, 1987. ACM.

[2] Gene H. Golub et. al. *Matrix Computations*. The Johns Hopkins University Press, Stanford University, 1996.

[3] I. Gohberg, T. Kailath, and V. Olshevsky. Fast Gaussian elimination with partial pivoting for matrices with displacement structure, 1995.

[4] Sven Hammarling and Craig Lucas. Updating the QR Factorization and the least squares problem. Unpublished manuscript, 2008.

[5] Nicholas J. Higham. Gaussian elimination. *Wiley Interdisciplinary Reviews: Computational Statistics*, 3(3):230–238, 2011.

[6] Steven Huss-Lederman, Elaine M. Jacobson, Jeremy R. Johnson, Anna Tsao, and Thomas Turnbull. Implementation of Strassen's algorithm for matrix multiplication. In *In Proceedings of Supercomputing '96*, pages 9–6, 1996.

[7] Timothy Sauer. *Numerical Analysis*. Pearson Education Inc., George Mason University, 2006.

[8] Virginia Vassilevska Williams. Multiplying matrices faster than coppersmith-winograd. In *Proceedings of the Forty-fourth Annual ACM Symposium on Theory of Computing*, STOC '12, pages 887–898, New York, NY, USA, 2012. ACM.