QR FACTORIZATION WITH SPARSE ADDITIVE UPDATES

JAMES FAIRBANKS AND DOMENIC CARR

Abstract. Abstract goes here.

Key words. QR factorization, Incremental Computation, Streaming Linear Algebra.

AMS subject classifications. 15A15, 15A09, 15A23

1. Introduction and examples. This paper concerns the goal of maintaining a least squares fit with changing data. In the literature, low rank updates, and the addition of new data points have been studied [**?**] [**?**]. In this work we address making changes to dimensions of the observed data. We take the case where A is an $m \times n$ matrix with $m \geq n$. We have an update to A called ΔA which has only a few nonzero columns. This can correspond to the fact that a few dimensions have updates. For example when studying the term document matrix $A_{i,j}$ represents the number of occurances of word i in document j. If the jth document changes and thus acquires new words then ΔA will contain nonzero entries only in the jth column. We can take advantage of this structure and accelerate the QR decomposition of $A + \Delta A$ by leveraging the information contained in the QR factorization of A.

Another example is a dynamic graph where new edges are presented and the algorithm is using a low rank representation of vertices based on QR_k where R_k a rank k truncated version of R. Then updates to A are the new edges and changes in this low rank representation can be used to study the dynamics of the graph [**?**].

1.1. Goals. In this paper we present two specialized forms of the givens rotation algorithm that perform these updates. We give an equation counting the flops used by each method, and compare them to standard Householder QR algorithm. Since these equations depend on which columns are updated, we give an average case analysis under the model of randomly selected columns. We experimentally estimate the cross over point for various values of m, n, k where m is the number of rows, n is the number of columns and k is the number of updated columns.

2. Algorithm. We exploit a key structure in this problem, which is that if $X = xe_i^T$ for some vector $x \in \mathbb{R}^m$ and Q is a matrix in $\mathbb{R}^{m \times n}$ then $Q^T X$ has nonzero entries only in the ith column. These implies that updates to any number of data elements that occur in the same dimension, incur the same number of nonzero entries when an attempt is made to update the QR factorization. We also use the fact that Givens rotations are able to selectively eliminate entries of a matrix while creating the orthogonal transformation Q and upper triangular factor R. This selectivity allows us to use Givens rotations in a way that we could not use Householder transformations. For example if the ith column of A contained nonzero elements, then applying a Householder transformation on the ith dimension would create fill-in entries in potentially every element in columns i through n. Thus our sparsity structure has been completely nullified in the process of a single Householder transformation. And this would necessitate performing $n-i$ transformations to fix the damage caused by changing the ith column.

The Givens rotations allow us to selectively eliminate entries of the matrix and in the next section we will consider different possible orderings of the rotations and count the fill-in introduced by each ordering.

3. Annihilation Ordering. We must decide to go row-wise or column-wise and decide whether to eliminate from the top to bottom or bottom to top. We first eliminate row-wise operation. If we go from top to bottom row-wise, we will introduce zeros in the entire row by eliminating against row 1. Eliminating the original entries of row j will introduce fill in in all of row j. If we go from bottom to top, then we introduce oscillatory fill in. eliminating entry m, i_2 will introduce a zero in entry m, i_1 and vice versa.

This indicates that we should operate column-wise. If we eliminate entries from the left most nonzero column, we introduce fill-in entries on the first off diagonal of the entire matrix. We then eliminate the fill-in entries between the leftmost nonzero column (which has now been zeroed) and then next leftmost nonzero column. This incurs the cost of eliminating $i_{j+1} - i_j$ fill-in entries. We then eliminate column i_{j+1} which introduces another off diagonal of fill-in entries. The total number of fill-in entries eliminated is

$$Lazy = \sum_{j=1}^{k} \sum_{t=1}^{j} (i_{j+1} - i_j - t) = \sum_{j=1}^{k} j(i_{j+1} - i_j - \frac{j+1}{2})$$

This is called the lazy elimination strategy because we defer elimination of fill-in entries until the last possible moment.

The eager strategy eliminates fill-in as soon as it is created. We eliminate column i_j and this fills in the first off diagonal. Then before eliminating column i_{j+1} we annihilate the first off-diagonal entries. Then when eliminating column i_{j+1} we reintroduce fill-in on the first off diagonal between columns i_{j+1} and n. The total number of fill-in entries removed is

$$Eag = \sum_{j=1}^{k} j(n - i_j) = n \sum_{j=1}^{k} j - \sum_{j=1}^{k} j i_j = \frac{nk(k+1)}{2} - \sum_{j=1}^{k} i_j$$

Both methods must eliminate the original nonzeros which are counted by $\sum_{j=1}^{k} n - i_j$.

$$Eager - Lazy = \sum_{j=1}^{k} nj - \sum_{j=1}^{k} j i_j - \sum_{j=1}^{k} j(i_{j+1} - i_j + \frac{j+1}{2}) \tag{3.1}$$

$$= \sum_{j=1}^{k} nj - \sum_{j=1}^{k} j(i_{j+1} + \frac{j+1}{2}) \tag{3.2}$$

Since $n \geq i_{j+1}$ for all j, we conclude that the lazy strategy is more efficient. That is, the lazy elimination strategy uses fewer eliminations than the eager strategy. The core cause of this discrepancy is that the lazy strategy eliminates the k off diagonals and the eager strategy eliminates the first off diagonal k times, and since the ith off diagonal has $n - i$ elements, this is more efficient.

3.1. Another interpretation of the eager method. We will now explain the eager method using an alternative definition. Suppose that $\Delta A = \sum_{j=1}^{k} A_j$ where each A_j has nonzero entries only in the i_jth column. That is $A_j = x e_{i_j}^T$ Then we can

use k updates on a single column each. We make use of the equations

$$\begin{aligned} Q^T A_1 + R &= Q_1 R_1 \\ Q_1^T A_2 + R_1 &= Q_2 R_2 \\ &\vdots \\ Q_{k-1}^T A_k + R_{k-1} &= Q_k R_k \end{aligned}$$

If we iteratively add one column at a time, then Givens rotations allows us to compute Q_{j+1}, R_{j+1} from $Q_j, R_j + Q_j A_{j+1}$. This requires $m - i_j$ rotations to elinate the one column and fills in one off diagonal with $n - i_j$ elements. This gives

$$6(n - i_j)(m - i_j) + 6(n - i_j)(n - i_j)$$

flops to perform the rotations and

$$6n(m - i_j) + 6n(n - i_j)$$

flops to update Q Each iteration also requires a single dense matrix vector product because we must compute $Q_j^T A_{j+1} + R_j$ We must perform this k times.

If we examine this method, we see that when performing the factorization

$$Q_{k-1}^T A_k + R_{k-1} = Q_k R_k$$

we introduce the fill in entries and immediately eliminate them. This recovers the eager strategy discussed earlier.

Should we present this interpretation first?

4. Comparison to Householder QR factorization. From Golub and Van Loan, we have the flop counts of the Householder reflection method where we are accumulating the orthogonal basis Q as $4(m^2n - mn^2 + n^3/3) + 2(n^2m - n/3)$. We know that the static Givens rotations methods takes $6mn(n+1)/2 + 6mn^2$ to accumulate the orthogonal matrix Q. Our method depends on which particular columns are perturbed by the update and so gives much more complicated formula. Let $(\cdot)^+$ be the function $max(\cdot, 0)$. For the sparse update we use at most

$$\sum_{j=1}^{k} 6(n - i_j)(m - i_j) + 6(n - i_j)^+$$

flops for eliminating the fill in. This is an upper bound because when counting the flops for eliminating the fill in entries, we count all of the fill in entries that are in the jth gap, as if they are at the position (i_{j+1}, i_j). These flops are also accompanied by the flops necessary to update Q, which are counted as

$$\sum_{j=1}^{k} 6n(m - i_j) + 6nj\left(i_{j+1} - i_j + \frac{j+1}{2}\right)$$

We can compare the flop counts of full recomputation with our sparse update methods. For the purpose of analytic tractability we will use the eager method and

make use order of growth bounds. Under the assumption that $m \geq n$ and $m \approx m-1, n \approx n-1$ $k=1$ and $i_1 = 1$

$$\begin{aligned} T_{eager}(m,n,k) &= 6(n-1)(m-1) + 6(n-1)(n-1) + 6n(m-1) + 6n(n-1) + m^2 \\ &\approx 12(nm+n^2) + m^2 \end{aligned}$$

We can thus suppose that we are making k updates to an m,n matrix, we bound the cost of this operation by the k times the cost to update the first column, which is the most expensive column to update. Thus we can derive a lower bound on the ratio of T_{full} to T_{eager}.

$$\frac{T_{full}}{T_{eager}}(m,n,k) \geq \frac{4m^2n - 4mn^2 + 4n^3/3 + 2n^2(m-n/3)}{12k(nm+n^2) + km^2} \tag{4.1}$$

$$\geq \frac{4m^2n - 2mn^2 + 2n^3/3}{12(nm+n^2) + m^2} \tag{4.2}$$

For a fixed m large enough, the Householder based methods are cubic in n and the sparse updating methods are quadratic in n because we need to accumulate Q. Thus making this simplification we can model the performance ratio as

$$r_e(m,n,k) = \frac{C_m n^3}{n^2 k} = \frac{Cn}{k} \tag{4.3}$$

The relationship between m and n is accounted for in the dependence of C_m on m. This model is useful for understanding the experimental results below. We see that as n increases the performance ration should increase for a fixed k, and for a fixed n the performance ratio is inversely proportional to k. From Equation 4.1 we can see that as m increases for a fixed n,k pair the proportionality parameter C_m should increase. When $r_e = 1$ the two methods will take the same amount of time, and the k for which this occurs is called the break even point. The model described by Equation4.3 can produce a prediction about the break even point by solving $C_m n/k = 1$ that is $C_m n = k$. So this tells us that the break even point will increase as the size of the matrix increases. We verify this experimentally in the next section.

5. Testing. Our input space has three dimensions: the number of rows of A, m; the number of columns of A, n; and, the number of nonzero columns in ΔA, k. We examined values of m of 100, 500, and 1000. Based on the value of m, we chose the values of n and k from the following sets:

$$\begin{aligned} n \in &\{2^i \mid 1 \leq i \leq \log(m)\} \\ k \in &\{2^i \mid 1 \leq i \leq \log(n/2)\} \end{aligned}$$

Altogether, this resulted in over 100 distinct combinations of (m,n,k) . To compare each algorithm's runtime for a (m,n,k)-combination, we applied the following testing procedure:

For given values of m, n, and k,

1. Generate a random $m \times n$ matrix A
2. Generate k vectors uniformly from $\mathbb{R}^k$
3. Place the k vectors into random columns of ΔA
4. Execute each QR Algorithm on A+ΔA

5. Repeat Steps (1)-(4) 30 times collecting the runtime
6. Determine median runtime
7. Perform Signed-Rank Test to determine statistical significance

The signed rank test tells us if the difference of the two median runtimes are distinct, taking advantage of the fact that the runtime of the methods depend on which positions the perterbations occurred in. We implemented the three algorithms in Matlab and performed all testing on a laptop computer with an Intel i7 processor operating at 2.2 GHz.

6. Results and Discussion. After computing t_{full}, t_{eager} and t_{lazy}, the median runtime for each algorithm, we investigated the ratios of algorithm runtimes for various cross-sections in our data. The most interesting ratios were

$$r_l(m,n,k) = \frac{t_{full}(m,n,k)}{t_{lazy}(m,n,k)} \qquad \text{and} \qquad r_e(m,n,k) = \frac{t_{full}(m,n,k)}{t_{eager}(m,n,k)}$$

Whenever these ratios are greater than 1, the specialized updating algorithms perform better than re-computing the QR factorization using Householder transformations. Once these ratios become less than 1, however, it is more advantageous to re-compute the QR factorization. In Figure 6.1 (left), we plot rl (m,n,k) versus log(k) for fixed values of n FOOTNOTE. This perspective is useful when the data vectors are of a fixed dimension and a practitioner is determining how columns can be altered in one batch before recomputation is more efficient than updating. From these plots, we see that as log(k) increases, rl decreases for fixed n. Specifically, we see a negative linear relationship, which confirms the inverse proportionality we anticipated in Section 4.

FOOTNOTE We only show rl since lazy outperformed eager in nearly every case

On these plots we also include a line called "breakeven", which indicates when rl is equal to 1. All points above this line imply the lazy updating algorithm is faster, while all points below this line imply the Householder algorithm is faster. By comparing the rl curves to the breakeven line, we see that the breakeven k value increases as n increases. We also see that as m increases (successively lower plots), the curves shift up and to the right, indicating improved performance of the lazy algorithm. For example, the n=64 curve crosses the breakeven line at log(k) = 1.25 for m = 100; at log(k) = 2.6 for m = 500; and, at log(k) = 2.8 for m =1000. This tells us that the breakeven k value increases as m increases, for fixed n. Thus, we have experimentally shown what we had predicted in Section 4: as the matrix size increases in either dimension, the breakeven point increases.

Another perspective is determining for a fixed batch size k, how much faster is updating only that batch compared to the entire factorization as a function of the dimensionality of the problem n. We show this type of perspective in the plots of Figure 6.1 (right). Here, we plot rl (m,n,k) versus n for fixed values of k FOOTNOTE. We see that there is nearly linear growth in $r(m_0, n, k_0)$ as n increases, when m is large. This is what we would expect from the analytic bounds on flop counts, as detailed in Section 4. We also see that as m increases, (successively lower plots), the curves shift upward for a fixed combination of n, k. For example, for m=100, the k=1 curve has an rl value of 5.5 at n = 100; for m=500, the k=1 curve has an rl value of 22 at n = 500; and, for m=1000, the k=1 curve has an rl value of 32 for m =1000. Thus, as m increases for a fixed n,k pair, the Cm value increases, which corroborates our theory from Section 4. We provide the tables that contain all of the data used to construct the plots in Figure 6.1 (left and right) in Figure 6.2.

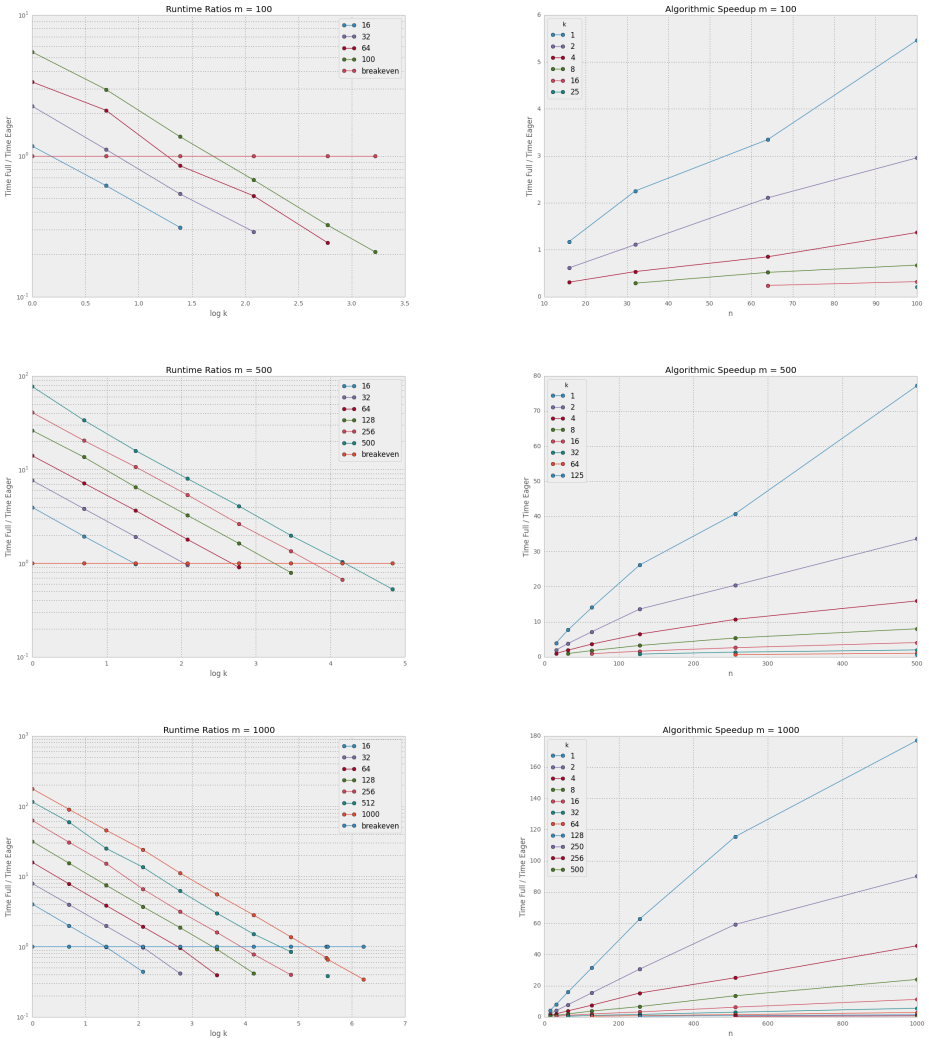

Fig. 6.1: r_e as a function of k (left) and as a function of n (right)

As detailed in Section 4, the model described by section Equation 4.3 can produce a prediction about the breakeven point by solving Cm n = k. As described previously, we can determine Cm experimentally from our data, and then use this value to predict where the breakeven point will occur. Figure 6.3 contains a table of experimentally determined breakeven points by using this method. Comparing these values to their counterparts in Figure 6.4, which contains our analytic model's lower and upper bound predictions for the breakeven points, we see that all of our experimental results fall within the bounds predicted by our analytic model.

7. Conclusion. In this paper, we have shown how specialized algorithms can improve the performance of QR factorization updating under sparse additive updates. We presented an analytic model and from this predicted the anticipated performance

n	16	32	64	100
k				
1	1.174464	2.254186	3.348890	5.464774
2	0.614962	1.110850	2.107137	2.959988
4	0.311029	0.537040	0.852289	1.369830
8	—	0.289581	0.520050	0.674347
16	—	—	0.241873	0.322460
25	—	—	—	0.208223

n	16	32	64	128	256	500
k						
1	3.963033	7.662772	14.071474	26.146007	40.734077	77.218781
2	1.942927	3.820333	7.162948	13.604650	20.400014	33.636736
4	0.979985	1.917975	3.650417	6.493244	10.671382	15.939933
8	—	0.956929	1.798891	3.253542	5.382873	8.006895
16	—	—	0.905242	1.624700	2.619620	4.078864
32	—	—	—	0.796405	1.351518	1.983535
64	—	—	—	—	0.671680	1.034494
125	—	—	—	—	—	0.529203

n	16	32	64	128	256	512	1000
k							
1	4.049259	7.944472	15.926325	31.507983	62.678462	115.458813	177.113035
2	1.988675	3.967780	7.810222	15.412376	30.590654	59.224599	90.073101
4	0.988316	1.969736	3.860258	7.504782	15.280334	25.072342	45.581149
8	0.438223	0.979623	1.921544	3.711172	6.605146	13.536398	24.000350
16	—	0.415439	0.961914	1.870153	3.157265	6.211023	11.198788
32	—	—	0.392416	0.920219	1.598431	2.998286	5.543048
64	—	—	—	0.417919	0.777768	1.508442	2.815094
128	—	—	—	—	0.397742	0.845979	1.379412
250	—	—	—	—	—	—	0.698539
256	—	—	—	—	—	0.381570	0.659975
500	—	—	—	—	—	—	0.343848

Fig. 6.2: Tables of r_e for $m = 100, 500, 1000$ dashes indicate the test was not run

increase for various conditions. Upon implementing the algorithms and comparing their performances, our experimental results from corroborated the theory we had presented.

REFERENCES

[1] G. H. Golub and C. F. Van Loan, *Matrix Computations*, Second ed., The Johns Hopkins University Press, Baltimore, MD, 1989.

[2] Daniel, James W., et al., *Reorthogonalization and stable algorithms for updating the Gram-Schmidt QR factorization.* in Mathematics of Computation 30.136 (1976): 772-795.

[3] Gill, Philip E., et al., *Methods for modifying matrix factorizations.* Mathematics of Computation 28.126 (1974): 505-535.

m	100	500	1000
16	1.216168	3.922943	3.871414
32	2.235173	7.657692	7.648598
64	3.800537	14.374808	15.051345
100	5.437297	—	—
128	—	26.139462	29.736792
256	—	42.204666	55.022052
500	—	66.675017	—
512	—	—	104.479743
1000	—	—	178.217056

Fig. 6.3: Experimentally determined breakeven points for $m = 100, 500, 1000$. Dashes indicate the test was not run.

m	100	500	1000	100	500	1000
n						
16	1	2	2	2	4	4
32	2	4	4	4	8	8
64	2	8	8	4	16	16
100	4	—	—	8	—	—
128	—	16	16	—	32	32
256	—	32	32	—	64	64
500	—	64	—	—	125	—
512	—	—	64	—	—	128
1000	—	—	128	—	—	250

Fig. 6.4: Tables of bounds on the k where $r_e = 1$. lower bounds (left) followed by upper bounds (right), dashes indicate the test was not run