References

[1] Anderson, E., Bai, Z., Bischof, C., Demmel, J., Dongarra, J., Du Croz, J., Greenbaum, A., Hammarling, S., McKenney, A., Sorenson, D., LAPACK Users' Guide, SIAM Publications, 1992

[2] Bai, Z., Demmel, J., *On a Block Implementation of Hessenberg Multishift QR Iteration*, International Journal of High Speed Computing, Vol. 1, 1989, pp. 97-112, Formerly: ANL, Technical Report MCS-TM-127, LAPACK Working Note 8, January 1989.

[3] Dongarra, J. J., Du Croz, J., Hammarling, and Duff, I. S., 1988 *A set of Level 3 basic linear algebra subprograms*, Report AERE R 13297. Oxford: Computer Science and Systems Division, Harwell Laboratory.

[4] Dongarra, J. J., Hammarling, S., J., Sorensen, D. C., *Block Reduction of Matrices to Condensed Forms for Eigenvalue Computations*, ANL, Technical Report MCS-TM-99, LAPACK Working Note 2, September 1987

[5] Dongarra, J. J., Kaufman, L., Hammarling, S., *Squeezing the Most out of Eigenvalue Solvers on High-Performance Computers* Linear Algebra and Its Applications, Vol. 77, 1986, pp. 113-136

[6] Elwood, D., *Private Communication*

[7] Francis, J. G. F., *The QR Transformation: A unitary analogue to the LR Transformation, Parts 1 and 2*, Comp. J. 4, 1961, pp. 265-272, 332-45

[8] Golub, G., Van Loan, C., Matrix Computations, 2nd Ed., 1989, *The John Hopkins University Press.*

[9] Henry, G., *BLAS Based on Block Data Structures*, Theory Center Technical Report, CTC92TR89, 1/92.

[10] IBM, Engineering and Scientific Subroutine Library, "Guide and Reference", Release 4

[11] IBM, *IBM RISC System/6000 NIC Tuning Guide For Fortran and C*, Document GG24-3611-01

[12] IBM, *Predicting Execution Time on the IBM RISC System/6000*, Document GG24-3711

[13] Kågström, B., Van Loan, C.F., *GEMM-Based Level-3 BLAS*, Theory Center Technical Report, CTC91TR47, 1/91.

[14] Moler, C.B., *Matlab User's Guide* Technical Report CS81-1, Department of Computer Science, University of New Mexico, Albuquerque, New Mexico, 1980

[15] Smith, B., Boyle, J., Dongarra, J., Garbow, B., Ikebe, Y., Klema, V., Moler, C. *Matrix Eigensystem Routines - EISPACK guide, 2nd Ed.* Lecture Notes in Computer Science 6 Springer-Verlag, 1976

[16] Watkins, D.S., *Shifting Strategies for the Parallel QR Algorithm*, Presentation at the NATO Advanced Study Institute on Linear Algebra for Large-Scale and Real-Time Applications. May 1992 draft.

[17] Watkins, D.S., Elsner L., *Chasing Algorithms for the Eigenvalue Problem*, SIAM J. Matrix Anal. Appl., April 1991, pp. 374–384.

[18] Wilkinson, J.H., The Algebraic Eigenvalue Problem, Oxford University Press, Oxford, 1965

n	*New Algorithm*	*LAPACK*	*ESSL*
100	0.594	0.931	0.746
200	3.745	5.179	5.161
300	11.85	16.65	19.16
400	24.82	34.93	44.82
500	48.48	71.71	77.70

Table 3: IBM RS/6000 Timings for the Nonsymmetric Eigenvalue Problem

monotonically as the matrix grows.

We have attempted to provide a general framework for the unsymmetric QR algorithm on a machine with a memory hierarchy. This paper has modeled data movement, and stride, and gone through various algorithms showing how each new strategy can further improve the problem. We have studied the various effects of blocking, delaying row reflections, and using alternate data structures. We have presented timing results to verify our claims, and then considered some machine dependent concepts for improving the results that much further.

6 Acknowledgements

This research was partially supported by the Cornell Theory Center, which receives major funding from the National Science Foundation and IBM Corporation, with additional support from New York State and members of its Corporate Research Institute. The research was conducted using several different machine architectures. The primary work was done on two IBM RISC System/6000, models 530 and 550, which is a resource of the Cornell Theory Center. We also used a Sun Sparcstation II to obtain analogous results, which is another resource of the Cornell Theory Center. In addition, most of the code was tested on the IBM 3090 VF/600J on site with the generous help of A. Hoisie. We also thank P. Freeman, of MIT Lincoln Laboratory, for his assistance in testing this work on a Cray X/MP and a Star 910/VP Compute Server with a Sun Sparcstation 1 motherboard. Although these last two timings were not illustrative enough to justify including them when the 3090 timings were there, they are included in a follow up paper dealing with some vector specific issues.

We are grateful for communication with Z. Bai, D. Watkins, D. Elwood, and C. Bischof regarding this and similar work. We are also grateful for the line of communication maintained with J. McComb, of the ESSL Development team, and F. Gustavson. Finally, we are grateful for the assistance and suggestions of C. Van Loan.

$n_b = 2$, $n_s = 2$, and $n_i = 1$ on a Model 530. The compiler predicted it would take 19 cycles per iteration of the inner loop, four of those being stores, with an ideal of 10 necessary calculations (ignoring the stores). The theoretical peak speed of a Model 530 is 50 Megaflops, so this algorithm could do no better than 26.3 Megaflops by the above reasoning. This was the actual performance. The same example with $n_i = 2$ has a theoretical peak speed of 32, but was timed at 28.8, with the additional loss due to data movement and stride problems. Note that this model predicted the peak speed of DHSEQR from LAPACK ($n_s = 6$) as 24 Megaflops. Note also that the model fails to consider how the stride or the cost of data movement affects the problem, which was the reason we included that in this paper.

During the presentation of **Alg. 7**, it was assumed that $n_s = 1$. For $n_s > 1$, indexing for the data structure is difficult. If $n_b = n_s > 1$, then the indexing is more manageable. A $n_b = n_s = n_i = 2$ version of **Alg. 7**, which still had the same theoretical peak speed of 32 Megaflops, was timed at 30.1 Megaflops. A $n_b = n_s = 3$, $n_i = 2$ version of **Alg. 7**, for both the row and column inner loops, was estimated to have a peak speed of 39 Megaflops and was timed at approximately 33 Megaflops for $n = 500$. Note that LAPACK's GEMM-based QR decomposition for $n = 500$ is rated at 34 Megaflops on this machine. **Alg. 7**'s efficiency for $n_s > 1$ is pleasing since this model ignores data movement. For these cases, we also incorporated cache prefetching.

5 Numerical Tests and Conclusions

To emphasize compatibility with LAPACK, we implement **Alg. 5** rather than **Alg. 6**, since the latter would involve restructuring significantly more of the LAPACK code. We also use shifts of size one or two since most of the eigenvalue literature uses this shift size.

We implemented a standard single and double shift combination LQ double precision algorithm. We used LAPACK's reduction to Hessenberg form, balancing, decoupling, stopping criterion, and overflow and underflow testing. We just replaced their multi-shift step with **Alg. 5** using explicit Householder vector generation (see §4.3). We used no other machine specific optimization tricks from §4. Included in the timings was transposition, since our eigenvalue finder required a lower Hessenberg matrix.

Table 3 contains the average number of seconds to solve multiple problems of the given problem size on an IBM RISC System/6000 Model 530. We see a worthwhile improvement, ranging around thirty to forty percent better. Note that this time includes the reduction to Hessenberg form, which we did not optimize on this machine. Since we used LAPACK's code for that step, it is interesting to compare the two results. Both our algorithm and the ESSL [10] algorithm used shifts of size one or two, whereas LAPACK uses shifts of size six. Notice also that the ratio of improvement with the new algorithm grows

4.4 Vector Architectures

If we employ some techniques in [5], then it is possible for all the algorithms in §2 to perform better on a vector architecture. We believe even with **Alg. 7**, which performed the worst on the 3090, it is possible to get the same performance as **Alg. 4**. However, some of the same techniques brought over to the scalar machines mean performance degradation. It seems increasingly unlikely to find a single algorithm that performs well in all environments.

In addition, there is another reason we prefer a lower Hessenberg matrix. On the vector machine, rotating columns is significantly faster than rotating rows. When we do an explicit single shift in forming LQ, we rotate two columns at a time, and only use the second column in the next turn. This observation allows a partial decomposition, where we only do half the work it takes to form LQ, which is one quarter of the total work it takes to form QL afterward, but still find the lower part of QL. We call this idea a "partial step", and it improves performance by reducing the total number of iterations. Because the matrix is lower Hessenberg, the partial step requires less time than a quarter of a full step.

4.5 A Note on Machine Specific Modeling

We model the ideal setting of no cache misses on an IBM RISC System/6000. The DMF we presented represents loads and saves, but in this ideal framework it is only the saves that are relevant. Since the DMF is half loads and half saves regardless of n_b or n_s, the ratio is still relevant even in this somewhat hypothetical modeling. Also, in the "ideal" setting, each multiply and add and each save costs a single cycle.

The theoretical peak speed of a particular algorithm is:

$$\text{Alg. Peak} = \frac{\text{Number of Multiply-Adds}}{\text{Number of Multiply-Adds plus Stores}} \times \text{Machine Peak Mflop Speed} \qquad (9)$$

Consider only the innermost loop(s) in this approximation. For the eigenvalue algorithm, the above equation can be substituted with

$$\text{Alg. Peak} \leq \frac{1}{1 + DMF} \times \text{Machine Peak Mflop Speed}. \qquad (10)$$

This result shows the importance of reducing the DMF on this machine.

The DMF does not indicate independence of Multiply-Adds (see §4.1). We consider greater detail since the above equations may be more generous than necessary. Suppose n_i is the unrolling depth of the inner loop. Then the total number of stores in the inner loop is $n_i(n_s + n_b)$. Modeling the number of combination multiply and adds as a function of n_i, n_s, and n_b is difficult. Since the compiler does this automatically with a special compile option, it may not even be necessary. We timed the inner loop of the column reflections with

so they cannot be used independently. But we have $n_b n_s n_i$ multiply-adds that are independent of the next loads. During these multiply-adds, we can start loading one or more of the $n_b n_i$ loads needed in the next run of the loop.

Cache prefetching on the column updates can be exploited, particularly if we are sweeping a block down the column ($n_i > 1$). Cache prefetching can be exploited if the matrix is upper Hessenberg as well.

4.3 Explicit Householder Vector Generation

For small n_s and/or n, there is a significant overhead due to Householder vector generation on the IBM RISC System/6000. On a matrix of size $n = 500$ and double shifts ($n_s = 2$) on the Model 530 over ten percent of the total time of the algorithm is spent in Householder vector generation. The corresponding figure on a Sun Sparcstation II is seven percent. If the time it takes to generate a single Householder vector that zeros out the last two elements of a three element vector, which is what is required for $n_s = 2$, requires one time unit, then there is a substantial overhead if underflow and overflow are considered, sometimes by a factor of five or more. Overflow and underflow are necessary to consider during timings (unless n is extremely large). If one time unit represents the amount of time it takes to generate a Householder vector of length three, ignoring numerical issues, then the LAPACK routine *DLARFG* ([1], ver. 1.0, 2/29/92) requires 4.5 time units. In our code, whenever we generate Householder vectors, we used the same DLARFG algorithm but we explicitly wrote out what the length three case was. The resulting algorithm is numerically identical, but requires only 1.7 time units. This idea of explicitly writing out the small cases of Householder accumulation is not new, this is what LAPACK did when applying Householder transformations.

For matrices of sizes around 100, over twenty five percent of the total time to compute the new Hessenberg form is spent in Householder vector generation (over seventeen percent for a Sun Sparcstation II). Note that the vector generation times grows linearly with the dimension, whereas the total time grows quadratically. Nevertheless, the IBM RISC System/6000 is so fast at doing the update, it was not until matrices of size 600 or more that the Householder vector generation time became irrelevant.

Loop unrolling and cache prefetching may lead to performance degradation in some environments. Explicit Householder vector generation cannot degrade performance, but its value is machine dependent. For this reason, we did not feel compelled to avoid this technique in our section on Numerical Tests (§5.) In a wide variety of settings, it should make no difference if the Householder vector generation is explicit. It happens to make a difference on the IBM RISC System/6000, for matrices of size 500 or less.

multiply and an add in a single cycle. If this multiply and add are dependent on the previous multiply and add, it takes two or more cycles. For example, if we wish to compute

$$sum = a + b * c + d * e,$$

then compiler first computes

$$sum = a + b * c$$

and then

$$sum = sum + d * e.$$

The second calculation is dependent on the first calculation because it cannot add *sum* in before the first calculation is done. Because of this dependency, it takes an additional cycle to find the result. The compiler usually can find ways of avoiding this by reorganizing the computation within a loop, but a straightforward implicit shift implementation runs into this problem. That is because the first part of a Householder update is the generation of a sum, and the next part is an update dependent on the sum. Furthermore, each calculation in the sum is dependent on whatever the sum is up to that point. Givens updates avoid this problem, but for shift size $n_s > 1$, this involves more work.

The solution is simple, but it is machine dependent. Unrolling the inner loop, which is the same as chasing a block down a column, allows the machine to interchange between two or more sums simultaneously, so that each calculation is independent of the previous. The inner loop should be unrolled to a depth at most $n_s + 1$, although a depth of two is usually sufficient. Since the outer loop represents the blocking size, n_b, this often has ideal values above two.

None of the timings in this paper reflect this idea, although we did implement the appropriate algorithms to verify the idea works in practice.

4.2 Cache Prefetching

On an IBM RISC System/6000, the cache can be prefetched using the pipelined floating and fixed point units. It is often possible to load a data element into the cache simultaneously with a series of independent floating multiply and adds. Loading an element into the cache from outside takes a minimum of 8 cycles. An element is loaded into the cache with its cache line and subsequent loads might be accessed in a single cycle. We did our testing on a Model 530 and Model 550, which each have cache lines of 128 bytes, or 16 double words.

During the row updates of the multishift analog of **Alg. 5**, most of the calculation is on temporary variables, which translate to registers after compilation. Let n_b still represent the column blocking size, n_s the shift size, and n_i the row blocking size. There are $n_b n_i$ loads and $n_b n_i$ saves in the inner loop of the row rotation part. However, there are $n_b(2n_s + 1)n_i$ adds and the same number of multiplies. Unfortunately, the $n_b n_i$ saves are dependent of half of these flops,

instead of A to compute the eigenvalues, it is also possible that the H in perfect precision would allow a decoupling, and the computed H would not.

D. Watkins [16] timed modified GRS favorably against LAPACK and EISPACK. He shows that one does not need to complete one full double shift before starting the next, and one does not need to complete all $n_s/2$ shifts before decoupling is possible. The modified GRS do remarkably better in terms of total number of iterations. But they are $5n_s n^2$ flops each, instead of $(2+4n_s)n^2$ flops for the multi-shift. Watkins shows that the greater accuracy makes up for the approximately twenty-five percent additional work.

Suppose we do a modified GRS of two double shifts. When the first double shift has completed its fourth update, clearing out the third column/row, the next double shift can start. To improve data reuse, a single loop could update both double shifts simultaneously for most of the problem. This is not as efficient as doing a shift of size four, but it might lead to faster convergence. In terms of data reuse, this algorithm has similar data loading properties to incorporating a shift of size five.

4 Machine Dependent Considerations for the IBM RISC System/6000 and other Architectures

The previous results have applicability over a wide range of architectures. Our goal has been to present a general framework for the unsymmetric QR algorithm on machines with a memory hierarchy. To achieve this goal, we have avoided machine specific ideas and did not pursue fine tuned optimization. The combinations of blocking size, shift size, the modified GRS, different data structures, and the new material in this section involve an implementation beyond the scope of this paper. The final result would be extremely machine specific, require lots of details in justification, and not of general interest.

However, our research points to a few machine dependent considerations that we now discuss. Note that the ideas in this section have not been exploited elsewhere in this paper (except for §4.3, for reasons made clear there.) Our preliminary calculations on the IBM RISC System/6000 show it is possible to get an additional thirty percent or greater improvement over the generic implementations found in §5. This means that the level-2 algorithms are achieving level-3 performance.

Some excellent references for §4.1, §4.2, and §4.5 include [11, 12].

4.1 Inner Loop Unrolling

On an IBM RISC System/6000, it is important that sequential calculations be independent. For our purpose, the architecture can in principle compute a

GEMM speeds on such small inner dimensions, especially since another one of the dimensions is on the order of $(n_s + n_b)$ (see [9] for details). If we scale the number of loads so that the DMF reflects the additional flops, then the DMF is still within the same range as it was with $n_b = 1$.

On some machines, even those with a memory hierarchy, this result may suggest small blocking sizes n_b. If we use GEMM-based updates, then the small sizes are necessary because of the additional flops. For nonGEMM-based updates, the small sizes are necessary to prevent poor column stride. We reiterate that poor column stride is often not as significant as the DMF, and so optimal efficiency may be achieved for a nonGEMM-based large block size.

If we use column-oriented code, and do the row reflections in a second pass of the algorithm (Type 2 blocking), then the problem of column stride is greatly diminished. More importantly, reflecting the columns is often a much faster process than reflecting the rows, sometimes by a factor of five or more. Type 2 blocking can gain many advantages of **Alg. 5** or **Alg. 6**.

3.1 Possibility of Large Multishifts

Even if the blocking size n_b must be small to gain benefits, there are advantages in allowing n_s to grow. In brief, the DMF gets better as n_s grows. For n_s large, the number of temporary loads and saves outweighs the advantages on reducing the loads and saves on the full matrix. When this happens, the matrix should be in upper Hessenberg form. This phenomena occurs at different times from machine to machine.

The question of large n_s may be a mute point anyway. Bai and Demmel observed that the first decoupling for large n_s occurs much later than one would hope. That is, to be consistent with EISPACK and $n_s = 2$, we should deflate in the first four or five iterations, which for large n and n_s is not something observed in practice.

An idea attributed to A. Dubrulle of IBM Research is that instead of doing a single shift of size n_s, if one does $n_s/2$ shifts of size 2, which should be numerically similar, that decoupling occurs at a rate competitive with EISPACK. We refer to this as modified Generalized Raleigh Shift, or just modified GRS.

Dubrulle's observation shows the problem is not the shifting strategy. Indeed, if one used the exact eigenvalues of H for the n_s shift values, rounding errors alone might make decoupling impossible. Suppose we have a matrix A with elements in the strict lower triangle. If we wish to find an upper Hessenberg H and an orthogonal Q such that $A = QHQ^T$, then usually we can expect that the computed H satisfies $H = P^T(A+E)P$ where $P^TP = I$ and $||E||_F \leq cn^2\epsilon||A||_F$ ([18]), where c is a small constant and ϵ is the machine epsilon. This backward error result can be somewhat improved when we chase a small bulge. The eigenvalues of H and A are close, even if H is not a good forward approximation to the true H. While it is usually reasonable to use H

below by 1/2. Unless blocking is implemented, or it is rewritten in a column oriented fashion, the DMF is unnecessarily high. For example, the DMF in DHSEQR version 1.0 from LAPACK is 7/12.

If n_b columns reflections of an upper Hessenberg matrix are blocked together, and the rows and columns are rotated simultaneously (Type 1), then the number of loads and saves per step is the same as if we had a bulge of size $n_b + n_s$. Since there are n/n_b steps, this puts the total number of loads and saves at $(2 + 2(n_b + n_s))n^2/n_b$. If $n_b >> n_s$, this yields a reasonable DMF. But the column stride goes up as well. In many architectures, this is not as important as data reuse of the cache, and so large block sizes perform well [6]. In addition, if storage is considered, and in our tests we allowed as much storage as needed, then large block sizes can help to avoid saving all the Householder transforms.

One way to prevent the column stride from becoming a problem is to do extra flops and make the innermost loops GEMM calls. Bai and Demmel in [2] introduced the application of this idea to the multishift QR algorithm. The idea from [2] is that if $P_i = I - u_i u_i^T$ is the Householder matrix that chases the bulge from column i to column $i+1$, then

$$H_{i+1} = P_i H_i P_i = H_i - u_i v_i^T - w_i u_i^T$$

where $y_i = H_i^T u_i$, $z_i = H_i u_i$, $v_i = y_i - (z_i^T u_i)u_i$, and $w_i = z_i$. They aggregate a sequence of these transformations so that

$$H_{n_b+i} = H_i - UV^T - WU^T$$

where $U, V, W \in \Re^{n \times n_b}$ and

$$y_j = (H_i^T - V(:,1{:}j-1)U(:,1{:}j-1)^T - U(:,1{:}j-1)W(:,1{:}j-1)^T)u_j$$
$$z_j = (H_i - U(:,1{:}j-1)V(:,1{:}j-1)^T - W(:,1{:}j-1)U(:,1{:}j-1)^T)u_j.$$

Here $U(:,j) = u_j$, $V(:,j) = y_j - .5(z_j^T u_j)u_j$, and $W(:,j) = z_j - .5(y_j^T u_j)u_j$. Structure should be exploited in the computation of y_j and z_j. Dongarra et. al. [4] introduced Householder transform aggregation in the Hessenberg reduction. See [2] for further details.

The update, following the computation of U, V, and W is $2(n_s + 1)n^2$ total multiplies and adds. When $n_b > 1$, we are updating H_i along $n_b + n_s$ rows and $n_b + n_s$ columns. The inner dimension of the multiply is n_b. The multiply is $2(n_s + n_b)n^2$ total flops. This reduces to $2(n_s + 1)n^2$ if some rows and columns are computed outside the GEMM call because of sparsity. For $n_b > 1$, additional work is involved in computing y_j and z_j. According to [2] this amounts to $(n_s + 1 + .5n_b)n^2$ extra flops. Our own calculation is that at least $(n_b)n^2$ extra flops are required ($.5n_b$ adds, $.5n_b$ multiplies), and this result seems consistent with [4]. If this last result is true, and considering this algorithm can be no more than 1.9 times as fast as **Alg. 6** or **Alg. 7** on the 550, then if $n_s = 6$, $n_b \leq 23$. If the former result is taken, then $n_b \leq 33$. It is difficult to get efficient

of no consequence. The IBM 3090 is a vector machine, and most of these results are geared toward scalar machines. It does provide an interesting comparison. Having a small column stride is more significant on a vector machine, as is having simple code that can be easily vectorized. This explains the improvement of **Alg. 4** and the poor performance of **Alg. 6**. Yet, it is possible to improve **Alg. 7**, the slowest of our methods on the 3090. Dongarra et. al. [5] suggest using temporary arrays to decrease data movement during the column rotations on vector machines. This is similar to the "reduced saves" for the row rotations in **Alg. 3**. Using this idea on the 3090, an equivalent version of **Alg. 7** ran with the same speed as **Alg. 4** (see §4.4).

Some key concepts are highlighted below:

- Use Column-Only Oriented Code (or Row-Only)
- Reduce the number of Saves with temporary Variables
- Use Lower Hessenberg Matrices
- Use Block Reductions to save on Data Movement
- Use Delayed Row Rotations if the architecture permits
- Allow Different Data Structures if Desired
- Allow Multi-shifts

These ideas can be implemented with implicit or explicit shifts, multiple or single shift steps.

3 Multishift Steps

Previously we have limited the discussion to single explicit shifts. For implicit shifts, the strategies are similar, but we must rotate a few of the rows (on the order of the constant bulge size) before rotating the next set of columns. These few rows then do not need rotating during the "second pass" of the algorithm. This amounts to a simple shift of the work within the loops, but does not change any of the major results and so we have limited the discussion to explicit shifts to ease the exposition.

Since we considered explicit shifts, all our examples were single shifts. For multiple shifts, we can still incorporate these data reuse techniques. We only highlight the important details to avoid redundancy.

As noted earlier, the number of flops for an implicit multishift of size n_s is $(6+4(n_s-1))n^2$. Since the local convergence rate is still usually quadratic [17], this makes for a theoretical improvement in complexity (up to a point). The number of loads and saves is $(1+n_s)n^2$ each. This places the initial DMF of the naive algorithm at $(2+2n_s)/(2+4n_s)$. This means that DMF is bounded

Alg	*DMF, Matrix*	*DMF, Temps*	*Col Stride*	*RS/6000*	*SparcII*	*3090*
1	.6667	.667/n	n/4	11.5	4.04	15.1
2	.6667	.1667	1.5	29.5	4.11	14.3
3	.5	.1667	1.67	31.2	4.64	17.0
4	.5	.1667	1.67	33.2	4.56	22.0
5, $n_b = 2$	.4167	.0833	2.6	37.8	5.23	22.6
6, $n_b = 2$	.25	.0833	3.0	44.2	6.10	12.6
7, $n_b = 3$	.3889	.0833	1.57	43.6-47.4	5.79-5.95	11.4

Table 2: QR Explicit Single Shift Algorithm Mflop Comparisons,$n = 500$

$\beta = s_i \tau_2 + c_i \tau_1$
$H(\tilde{i_1}) = s_{i+1} \beta - c_{i+1} \tau_3$
$H(\tilde{i_1} + 1) = s_{i+1} \tau_3 + c_{i+1} \beta$
$\tilde{i_0} = \tilde{i_0} + 2$, $\tilde{i_1} = \tilde{i_1} + 2$
endfor
$i_0 = i_1 + 1$
$i_1 = i_1 + ldh - 2i - 2$

The average column stride of **Alg. 7** is

$$CS(n_b) = \frac{3n_b + 2}{2n_b + 1}, \tag{8}$$

which is significantly less than **Alg. 5** from which it is derived.

We also could use a data structure to rewrite **Alg. 6** and still maintain a respectable DMF and the same column stride as above. So whenever **Alg. 7** appears in a table in this paper, we have given it a range of values.

There are several things that have remained constant from **Alg. 1** to **Alg. 6**. There has been $4n^2$ multiplies and $2n^2$ additions. Although the last three algorithms can be rewritten to change this, they all share the same Level-3 fraction of 0 (or Level-2 fraction of 1). In all cases we can have an equivalent result. The biggest difference between each algorithm is the DMF ratio and column stride. Each time we can decrease the DMF ratio without seriously hurting the column stride, the performance of the algorithm improved. Table 2 summarizes our results.

Table 2 contains results from three different computers: IBM RISC System/6000 Model 550, Sun Sparcstation II, and an IBM 3090 VF/600J. Column oriented code and removing extra flops, such as in **Alg. 3**, are well established optimization methods for the eigenvalue problem. It is interesting that at least a thirty to forty percent additional improvement is possible. On the Sun, **Alg. 4** did not do as well as **Alg. 3** because the reduction of temporary loads were

Therefore, the first column has stride n_b, and all the other columns are grouped sequentially with stride one. Except for that one point, we follow **Alg. 5**, but the column stride of this alternate data structure version is less.

To define more precisely the above alternate data structure, we give a brief algorithm that maps the data of the matrix into the data structure.

Alg. Map Data into Alt. Data Structure

Group n_b columns of lower Hessenberg H together into v
(Can be rewritten so that v and H occupy the same space.)
$m = 1$
for $i = 1{:}n_b{:}n_b\text{floor}(\frac{n}{n_b}) - 1$
  for $j = max([i-1, 1]){:}n$
    $v(m{:}m + n_b - 1) = H(j, i{:}i + n_b - 1)$
    $m = m + n_b$
  endfor
endfor
for $i = n_b\text{floor}(\frac{n}{n_b}) + 1{:}n$
  for $j = n_b\text{floor}(\frac{n}{n_b}){:}n$
    $v(m{:}m + (n - i)) = H(j, i{:}n)$
    $v(m + n - i + 1{:}m + n_b - 1) = \text{zeros}(n_b - 1 - n + i, 1)$
    $m = m + n_b$
  endfor
endfor

This storage scheme uses around half the space of storing A, presuming $n_b << n$. Also, the matrix can be overwritten in place so that the (Real) Schur form is available and no extra storage space is required.

Consider the same brief example for **Alg. 7** that was considered for **Alg. 5**. This is a nonGEMM-based Type 2 blocking, with $n_b = 2$ and an outer loop of $i = 1{:}2{:}n-2$. H is stored in a vector, reducing some superfluous zeros. We keep a few for simplicity of exposition, but otherwise this is similar to packed storage of a triangular matrix. In addition, we still keep around a "leading dimension" (ldh), otherwise deflation is impossible. The inner loop of **Alg. 7** is:

Inner Loop for Alg. 7 ($n_b = 2$)

Let i_0, i_1 point to positions $H(j, i)$ and $H(j, i+1)$ respectively. Assume that $H(i + 3{:}n, i + 1)$ and $H(i + 3{:}n, i + 2)$ are grouped together sequentially in an alternate data structure. Let ldh be the leading dimension.
$\tilde{i_0} = i_0$
$\tilde{i_1} = i_1$
for $j = i + 3{:}n$
  $\tau_1 = H(\tilde{i_0})$, $\tau_2 = H(\tilde{i_1} + 1)$, $\tau_3 = H(\tilde{i_1} + 2)$
  $H(\tilde{i_0}) = s_i\tau_1 - c_i\tau_2$

Inner Loop for Alg. 6 ($n_b = 2$)

(Assume that the rows of $H(:, i)$ have already been rotated. Rotate the rows of $H(:, i+1)$ and $H(:, i+2)$ while simultaneously rotating the columns.)

$\tau_2 = H(i+3, i+1)$
$\tau_3 = H(i+3, i+2)$
$\gamma = \tau_2$
$\delta = \tau_3$
for $j = i+3{:}n-1$
  $\tau_1 = H(j, i)$
  $\tau_{21} = H(j+1, i+1)$
  $\tau_{31} = H(j+1, i+2)$
  $\tau_2 = s_j\gamma - c_j\tau_{21}$, $\tau_{21} = s_j\tau_{21} - c_j\gamma$
  $\tau_3 = s_j\delta - c_j\tau_{31}$, $\tau_{31} = s_j\tau_{31} - c_j\delta$
  $H(j, i) = s_i\tau_1 - c_i\tau_2$
  $\omega = s_i\tau_2 + c_i\tau_1$
  $H(j, i+1) = s_{i+1}\omega - c_{i+1}\tau_3$
  $H(j, i+2) = s_{i+1}\tau_3 + c_{i+1}\omega$
endfor

We call the above idea *delayed row rotating*. There are no extra loads or saves of the matrix within the inner loop. The goal is to rotate rows of the previous iteration *simultaneously* with the loads necessary to rotate the columns of the current iteration. There are $1.5n^2$ loads and saves with $n_b = 2$ (see (5) and keep in mind we save n^2 loads and saves with this idea). This moves the original DMF from 2/3 to 1/4. Asymptotically the DMF for the single shift nonGEMM-based blocking is bounded below by 1/6. We can find no way of doing better than that without going to multiple shifts. We consider this last idea to represent **Alg. 6**.

The average column stride of **Alg. 6** is

$$CS(n_b) = \frac{n_b^2 + 2n_b + 1}{n_b + 1}, \qquad (7)$$

which is slightly more than the corresponding figure for **Alg. 5**. Since the improvement to the DMF is more dramatic, this algorithm was timed at 44.1 Mflops, which is a fifteen percent improvement over **Alg. 5**.

We further justify the examination of the DMF and the column stride, by introducing **Alg. 7** which is the same as **Alg. 5** except that the matrix is in an alternate data structure. This is not an entirely unreasonable assumption, since the eigenvalue routines typically overwrite the array H; we can transform the matrix to and from the data structure in place. The idea is to group n_b columns together in storage and store each element along the rows of those columns sequentially. When it is time to rotate the current group of $n_b + 1$ columns, the first column of the group is the last column of the previous group.

$H(j,i) = s_i\tau_1 - c_i\tau_2$
$\beta = s_i\tau_2 + c_i\tau_1$
$H(j,i+1) = s_{i+1}\beta - c_{i+1}\tau_3$
$H(j,i+2) = s_{i+1}\tau_3 + c_{i+1}\beta$
endfor

The above section of code combines two sets of column rotations. Normally, we have two loads from the matrix in the inner loop for each column rotation, or a total of four. Combining the two rotations together gives only three loads per pass in the inner loop. Then $n_b = 2$ has 3/4 of the loads and saves in the column rotations. The only difference in the row rotations is that the number of loads and saves on the temporary variables decrease. Instead of $3n^2$ loads and saves from **Alg. 4**, we have $2.5n^2$ total loads and saves, n^2 of which come from the row rotations. Of those $2.5n^2$ loads and saves, $0.75n^2$ loads come from the column rotations, and $0.5n^2$ loads come from the row rotations. This brings the DMF from 1/2 to 5/12. The column stride for this problem is 2.6. The performance for $n_b = 2$ and $n = 500$ on the Model 550 moved from 33.2 Mflops to 37.8 Mflops. We consider this a significant improvement.

Larger values of n_b make the data reuse increasingly better, but the stride problem gets in the way. That is, the greater n_b is, the more we hop around the matrix. Like any other blocking parameter, this must be optimized on each particular machine. The amount of loads and saves on the full matrix for block size n_b in algorithms three through five is

$$LS(n_b) = n^2 + (\frac{n_b+1}{2n_b})2n^2 \tag{5}$$

This shows a limit to how many loads and saves we can avoid.

We can choose different values of n_b for the row and column permutations when we use Type 2 non-simultaneous block rotations. For analysis only, assume that n_b represents the number of columns grouped together during *either* the row or column rotations. The column stride of the problem is approximately:

$$CS(n_b) = \frac{2n_b^2 + 2n_b + 1}{2n_b + 1} \tag{6}$$

The column stride gets proportionally worse as n_b grows.

We can cut the number of loads and saves $LS(n_b)$ in (5) even further. As mentioned, the first n^2 term of (5) represents the row rotations. Since we are delaying our row rotations until the end anyhow, and since we are doing them in a column oriented fashion, additional loads for the row rotations can be avoided if we do them simultaneously with the column rotations. This reduces the total number of loads and saves from the matrix by that n^2. Since the algorithms are getting complicated, we again give the same subsection of code representing the inner loop of the column rotations for another $n_b = 2$ example.

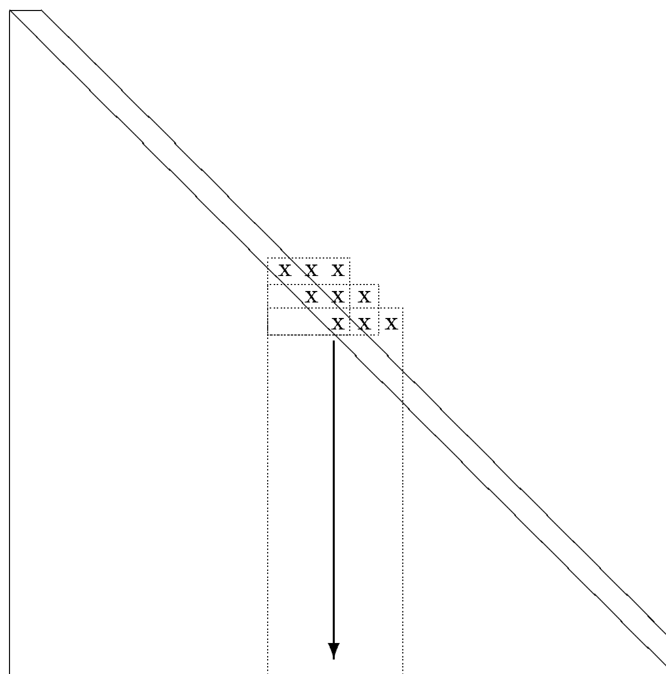

Figure 2: Non-Simultaneous Block Row/Column Updates

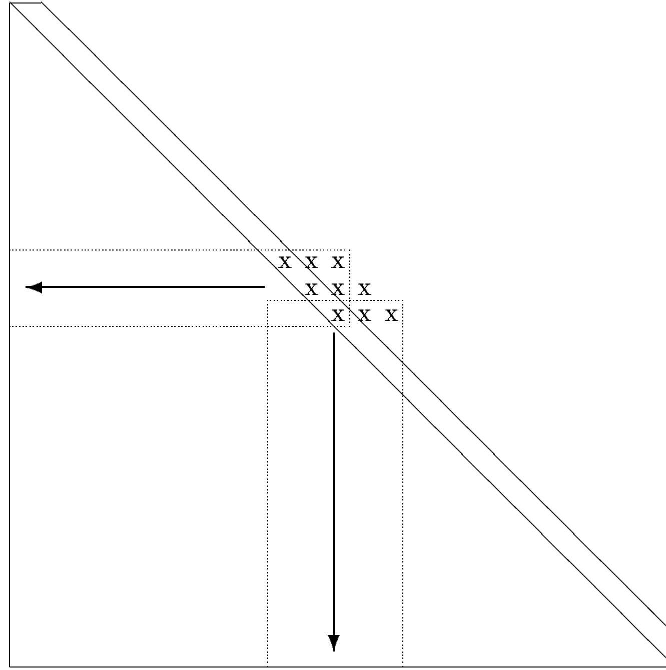

Figure 1: Simultaneous Block Row/Column Updates

- GEMM-based

Examples of Types 1 and 2 are in Figures 1 and 2 respectively. The arrows represent the part that is immediately updated after the block transforms are generated. Figures 1 and 2 depict lower Hessenberg matrices with $n_b = 3$ and an implicit double shift.

Chasing a block down a column or row, within the outer level of blocking, may be beneficial. This form of double level blocking is discussed in §4.1.

Consider a specific example to clarify **Alg. 5**. Suppose $n_b = 2$, implying two sets of Givens parameters are generated before the rotation. The outer loop during the formation of LQ is $i = 1{:}2{:}n-2$ with an inner loop on j. The inner loop of a nonGEMM-based blocking (Type 2) could look as follows:

Inner Loop for Alg. 5 ($n_b = 2$)
for $j = i + 3{:}n$
  $\tau_1 = H(j, i)$
  $\tau_2 = H(j, i+1)$
  $\tau_3 = H(j, i+2)$

Alg. 5: Block Reduced Saves Single Explicit Shift LQ

Use block size n_b

for $i = 1{:}n_b{:}n - n_b$

  Form LQ for $H(i{:}i + n_b, i{:}i + n_b)$ saving Givens parameters.

  Rotate columns $H(i + n_b + 1{:}n, i)$ through $H(i + n_b + 1{:}n, i + n_b)$

endfor

Finish the remaining columns through $n - 1$

Reform QL in the same block fashion.

The idea of using Block Methods for the Hessenberg QR algorithm is not new. Bai and Demmel discuss this in a LAPACK Working Note [2]. In this paper, the matrix is upper Hessenberg and additional flops permit a GEMM-based update (GEneral Matrix Matrix multiply- [3]). Clearly, a GEMM-based Householder update is more feasible than a GEMM-based Givens update, and this point is implicitly understood for the remainder of the paper. A more thorough analysis of this strategy is in the next section, but note that for small step sizes n_b it is often beneficial to avoid the extra flops of a GEMM-based update. Although GEMM-based strategies are preferred because of their efficiency and portability, for small dimensions efficiency is often compromised [9]. In addition, Elwood [6] has rewritten EISPACK codes [15] in terms of large block strategies for upper Hessenberg matrices in serial and parallel implementations. We are not sure whether this work was GEMM-based, or not, but like [2] it generalizes EISPACK and does not delay any row or column rotations till the end.

We classify blocking strategies into two types, each with two subcategories depending upon whether it is GEMM-based or not. Type two block reduction below is optimal for small blocks. For large blocks, optimality depends on how much temporary storage is available and the machine architecture.

Type 1. *Simultaneous Row and Column Rotations.* This makes sense in implicit shifts only. We do the minimum amount of work it takes to generate n_b transforms, and then apply them to both the rows and columns simultaneously. This is the natural generalization of EISPACK, and is done by both [2] and [6].

- NonGEMM-based
- GEMM-based

Type 2. *Nonsimultaneous Row and Column Rotations.* This is the default in the explicit shift, but also applies to the implicit shift. After doing the minimum amount of work to generate n_b transforms, we apply them to either the rows or the columns, but not both, and save the transform's data to finish the application at a later stage.

- NonGEMM-based

The improvement in performance over **Alg. 1** is impressive, and it appears that the only thing that improves performance at this point is the multi-shift steps, which we consider separately in the next section. To have stride one access, the temporary arrays are loaded in the same part of the problem, the row rotations, as they are saved. We found when timing each part that sometimes the algorithm is spending up to three quarters of the total time in the row rotation part.

If we could rotate columns first, then once c_k, s_k were defined, we could use them to finish rotating the columns with stride one access. We would have no need to load $c[1{:}k-1]$ and $s[1{:}k-1]$ to do this. The overall structure of the problem would be simpler.

Therefore, consider a *lower* Hessenberg matrix instead of upper Hessenberg. This is the same algorithm mathematically, but on the transpose. Instead of forming QR and then RQ, we form LQ and then QL. In the latter case, Q is Q^T of the former case, and then the two algorithms are equivalent.

This leads to **Alg. 4**.

Alg. 4: Reduced Saves Single Explicit Shift LQ

Do **Alg. 3** but on the transpose of H

The column stride and DMF are the same for this problem. This version saves $O(n)$ temporary loads and saves, but this should not account for much. The major factor is some computational time in the first part of the algorithm is shuffled into the second part, making for a more even balance between the parts. This algorithm solved the identical problem of $n = 500$ in 33.2 Mflops. A possible reason for this being slightly faster is that the compiler had more opportunities for optimization since the code was simplier. (An additional reason for the preference of lower Hessenberg matrices is given in §4.4).

There is $3n^2$ total loads and saves on the full matrix. Although this is an improvement over $4n^2$ in **Alg. 1**, there are still two flops for every load or save. It is this reason among many that we often go to multi-shift steps. The multi-shift steps also have the advantage of fewer flops. Whereas the single shift (implicit or explicit) requires $6n^2$ flops per iteration, a multi-shift of size n_s requires

$$\text{total flops}(n_s) = (6 + (n_s - 1) * 4)n^2 \tag{4}$$

Going to a multi-shift is not the only way to increase data reuse. We could, for example, generate a block version of algorithm **Alg. 4**. The idea is to do the minimum amount of work to determine the first $n_b - 1$ Givens or Householder parameters, and then update n_b rows and columns simultaneously. We give a loose description of this algorithm in **Alg. 5**:

The column stride of the row rotations of **Alg. 2** is now one instead of $n/4$. The column stride of the column rotations is two, since two columns are accessed for each Givens update. The average column stride is therefore 1.5.

A closer look at the inner loop of the row rotations shows that:

for $j = 1{:}k-1$
 $H(j,k) = c_j\tau_1 - s_j\tau_2$
 $H(j+1,k) = s_j\tau_1 + c_j\tau_2$
endfor

Once $H(j+1,k)$ is defined, it is saved and reloaded in the next step. Instead $H(j+1,k)$ can be placed in a temporary variable so that it is ready for the next iteration of loop. Dongarra et al. [5] use this idea to improve performance of the QR algorithm on vector machines. In general, whenever there is a sequence of Givens or Householder transforms, $G(1,2)G(2,3)G(3,4)...G(k-1,k)H$, we can reduce the number of loads and saves of the matrix or vector H with temporary variables. This observation leads to the next algorithm, which we call *Reduced Saves* because the number of saves and loads on the matrix is reduced.

Alg. 3: Reduced Saves Single Explicit Shift QR

$(c_1, s_1) = \text{givens}(H(1,1), H(2,1))$
$H(1,1) = c_1H(1,1) - s_1H(2,1)$, $H(2,1) = 0$
Save c_1, s_1 in a temporary array
for $k = 2{:}n-1$
 $\tau_1 = H(1,k)$
 for $j = 1{:}k-1$
  $\tau_2 = H(j+1,k)$
  Load c_j, s_j from a temporary array
  $H(j,k) = c_j\tau_1 - s_j\tau_2$
  $\tau_1 = s_j\tau_1 + c_j\tau_2$
 endfor
 $(c_k, s_k) = \text{givens}(\tau_1, H(k+1,k))$
 $H(k,k) = c_kH(k,k) - s_kH(k+1,k)$, $H(k+1,k) = 0$
 Save c_k, s_k in a temporary array
endfor
Rotate the columns as in **Alg. 1**

There is n^2 loads on the temporary array, and $1.5n^2$ loads and saves each on the full matrix. This saves n^2 loads and saves on the full matrix. This reduction of loads during the part that was stride one access already makes the column stride for **Alg. 3** a 5/3, which is still low. This algorithm was timed at 31.2 Mflops for the same $n = 500$.

Alg.	*Key Concept*	*Mflops*
1		11.5
2	Column Oriented	29.5
3	Reduced Saves	31.2
4	Lower Hessenberg	33.2
5	Block Method	37.8
6	Delayed Row Rotations	44.2
7	Alternate Storage	43.6-47.4

Table 1: QR Explicit Single Shift Concept Comparisons, $n = 500$

blocking size of interest (two). The range for **Alg. 7** represents whether the data structure used **Alg. 5** or **Alg. 6**. The timings for **Alg. 7** when using the strategy of **Alg. 6** include only the application over most of the problem, and not the remaining left-over columns that do not fit into the natural blocking.

2 Improving the Data Reuse

Consider a column oriented version of the same problem in **Alg. 2**.

Alg. 2: Column Oriented Single Explicit Shift QR

for $k = 1{:}n-1$
  for $j = 1{:}k-1$
    $\tau_1 = H(j,k)$, $\tau_2 = H(j+1,k)$
    Load c_j, s_j from a temporary array
    $H(j,k) = c_j\tau_1 - s_j\tau_2$, $H(j+1,k) = s_j\tau_1 + c_j\tau_2$
  endfor
  $(c_k, s_k) = \text{givens}(H(k,k), H(k+1,k))$
  $H(k,k) = c_k H(k,k) - s_k H(k+1,k)$, $H(k+1,k) = 0$
  Save c_k, s_k in a temporary array
endfor
Rotate the columns as in **Alg. 1**

The algorithm now has stride one access during the row rotations. For this reason alone, we expect **Alg. 2** to perform better than **Alg. 1**.

There are $4n^2$ multiplies, $2n^2$ adds, $2n^2$ loads and $2n^2$ saves on the full matrix. Instead of $O(n)$ loads and saves in the temporary work array, there are n^2. Any resulting disadvantage due to the extra data loads in minimal because there is now a better stride. The speed on the same 550, for the same matrix in the same test as mentioned in the discussion of **Alg. 1**, was 29.5 Mflops, over twice as fast.

timings in this paper refer to double precision. Since this is one seventh of the speed of GEMM (GEneral Matrix Multiply) [3] for the same size problem, this result is unsatisfactory. The Data Movement Fraction is as high as 2/3. Therefore, there are two loads or saves for every three flops. On the IBM RISC System/6000, loads can be pipelined with the floating point calculations (see §4), and once an element is in the cache, loading that element is usually free. The DMF is still significant since saves become the bottleneck, and the number of saves and loads within the inner loop of the QR algorithm is always identical.

To compute the column stride, ignore the column rotations because they are lower order compared to column stride along the rows. When rotating rows k and $k+1$, we access $n-k+1$ columns. The column stride amongst the rows is then $n/2$. The average column stride for **Alg. 1** is high at $n/4$.

Notice that **Alg. 1** is the standard method of considering the single explicit shift problem. Indeed, there are BLAS (Basic Linear Algebra Subroutines [3]) that rotate two rows. The rotation of rows, with the bad stride, takes seven times longer than the rotation of the columns although both amount to the same work. This unnatural balance suggests improvement.

This paper highlights some ideas for improving the DMF and column stride. Section two emphasizes five key concepts, and labels them **Alg. 2** through **Alg. 6** for the single shift QR algorithm. We then show further evidence of the usefulness of our model by introducing a QR algorithm based on an alternate storage scheme that has both an improved DMF and column stride (**Alg. 7**). Each algorithm builds upon the previous algorithms presented. Justification of each idea is done by showing an improvement in performance over the preceding algorithm. We show how to improve the DMF by a factor of four asymptotically before even considering multishifts. Then we discuss implicit and multishift steps in section three and how the same ideas can apply in that more complicated framework. Finally, we give some numerical results on performance of the entire algorithm.

The five key concepts of §2, which precedes our multishift work, are:

- Use Column-Only Oriented Code
- Reduce Saves with Temporary Variables
- Use Lower Hessenberg Matrices
- Use Block Reductions
- Delay Row Rotations

Before detailing each concept and how they affect data movement in our model, we preview Table 1 which shows some performance results on an IBM RISC System/6000, Model 550. The Table contains results for a double precision Hessenberg matrix of order 500. Algorithms five and six are block methods, and so performance depends on blocking size. For this reason, we use the smallest

EISPACK [15] avoids in *COMQR*. This amounts to a negligible improvement in performance since adding and subtracting the shift μ can be done during the computation avoiding extra data movement. Assume hereafter that explicit shifts are handled in this fashion.

On a machine with hierarchical memory, a blocked Level-3 algorithm is often preferable even when it involves marginally more arithmetic. Often this benefit is expressed in terms of a Level-3 Fraction [8]. In our context, this is unsatisfactory since the Level-3 Fraction is zero.

Yet it is still possible to improve **Alg. 1** in terms of data reuse. To quantify developments, consider the ratio of loads and saves to flops. We call this the *DMF*, for Data Movement Fraction, and define the ratio as:

$$DMF = \frac{\text{number of loads and saves}}{\text{number of total flops}} \tag{2}$$

Loads and saves refer to single scalar data movements, and not vector loads or saves [8]. A *flop* is an addition, subtraction, multiplication, or division. We show that the DMF is as important to Level-2 routines such as the Hessenberg QR Algorithm as the Level-3 Fraction is to cubic algorithms such as LU factorization and Hessenberg reduction. Two factors affecting performance are discussed:

Data Movement Fraction. Stride.

By *stride*, we mean the amount of space between two consecutive data loads or saves. We wish to quantify how column oriented an algorithm is. Consider any double loop Level-2 linear algebra algorithm:

for $k = k_1{:}k_2$
  for $j = j_1{:}j_2$

Suppose the number of columns accessed within the inner loop is $C_1(k)$. Let $C_2 = \sum_{k=k_1}^{k_2} C_1(k)$, the total number of column accesses if we counted duplications from one value of k to another. We define the average *column stride* as follows:

$$\text{Column Stride} = \frac{2C_2 n}{\text{number of loads and saves}} \tag{3}$$

The $2n$ term helps differentiate between cases when one element is accessed per column and cases where many elements are accessed. This definition indicates how columns are accessed on average in an algorithm, but does not show the stride within a column.

On an IBM RISC System/6000, Model 550 (theoretical peak speed of 83 Mflops), with a dimension of $n = 500$, **Alg. 1** was timed at 11.5 Mflops. All

1 Introduction

In this paper we examine the QR algorithm in terms of data reuse and develop new strategies for approaching the problem. For illustration, we first consider the standard explicit single shift, and show several ways to improve it. Then we show how the same results apply to the single or multiple implicit shifts.

Let A be a real unsymmetric matrix of order n, and H the orthogonal Hessenberg reduction similar to A. The QR algorithm involves the following two steps:

- $H \leftarrow PAP^T$ (Hessenberg reduction of A).
- Find the eigenvalues of H.

Consider the single explicit shift QR algorithm for finding the eigenvalues of H. First, form $QR = (H - \mu I)$ (QR factorization), where Q is not explicitly formed, and then overwrite H with $H \leftarrow RQ + \mu I$ [8, 18]. We choose the eigenvalue in the bottom 2×2 submatrix closest to $H(n,n)$, and if that is complex, we employ an implicit Francis double shift. In *exact* arithmetic, if μ is an eigenvalue of the unreduced H, then the bottom or the penultimate subdiagonal element of the new H is zero. This deflation creates a smaller problem in the next step. Assume throughout the paper there exists a routine **givens**(a,b) that returns (c,s) so

$$\begin{bmatrix} c & -s \\ s & c \end{bmatrix} \begin{bmatrix} a \\ b \end{bmatrix} = \begin{bmatrix} r \\ 0 \end{bmatrix}. \tag{1}$$

We shall refer to the orthogonal 2×2 matrix as $G(i,j)$. In $G(i,j)$, i and j are the positions of the rotated elements a, b. Summarizing:

Alg. 1: Single Explicit Shift QR step
(Algorithm 7.4.1 in [8])
for $k = 1{:}n-1$
  $(c_k, s_k) = \text{givens}(H(k,k), H(k+1,k))$
  Rotate rows $H(k, k{:}n)$ and $H(k+1, k{:}n)$
  Save c_k, s_k in a temporary array
endfor
for $k = 1{:}n-1$
  Load c_k, s_k from a temporary array
  Rotate columns $H(1{:}k+1, k)$ and $H(1{:}k+1, k+1)$
endfor

Alg. 1 has $4n^2$ multiplies, $2n^2$ adds, $2n^2$ loads, and $2n^2$ saves on the full matrix. It has $2n$ loads and $2n$ saves on the temporary array. Adjusting the shifts appropriately, μI does not need adding back when forming RQ, such as

Increasing Data Reuse in the Unsymmetric QR Algorithm

Greg Henry

July 28, 1992

Abstract

This paper models data use in the Unsymmetric QR Eigenvalue Algorithm to improve performance on machines with a memory hierarchy. Most of the algorithms and strategies presented can be implemented so that they are numerically similar to strategies found in such libraries as LAPACK and EISPACK ([1, 15]). We provide tests to show improvement of performance. Some strategies implemented include the use of block methods, transposing the matrix, reducing the average stride, reducing data movement with hybrid steps, and using block data structures.

Keywords: QR Algorithm, Memory Hierarchy, Data Reuse, Data Structures
AMS(MOS) subject classifications: 65F15, 65H15, 68P05

This research was partially supported by the Cornell Theory Center, which receives major funding from the National Science Foundation and IBM Corporation, with additional support from New York State and members of its Corporate Research Institute.