M. A. Saunders. The complexity of LU updating. In R. S. Anderssen and R. P. Brent, editors, *The Complexity of Computational Problem Solving*, pages 214–230. University of Queensland Press, 1976.

THE COMPLEXITY OF LU UPDATING IN THE SIMPLEX METHOD

M.A. Saunders

Applied Mathematics Division, DSIR, Wellington, New Zealand

1. Introduction

Two methods have become established as the most efficient implementations of the simplex method (Dantzig [3]) for solving the general large-scale linear programming problem

$$\begin{array}{ll} \text{minimize} & c^T x \\ \text{subject to} & Ax = b\ , \quad x \geq 0\ . \end{array}$$

These methods are based on the factorization $B = LU$, where B is the usual $m \times m$ basis matrix, L is a product of permutation and elimination matrices, and U is upper triangular. They are

Method BG, due to Bartels and Golub [1], [2], implemented by Reid [8], [9], and

Method FT, due to Forrest and Tomlin [4].

Their efficiency stems from the fact that the LU factorization of a general sparse matrix B normally requires less storage than any other factorization. Furthermore the sparsity is maintained remarkably well when L and U are updated to reflect a change of basis (when a column of B is replaced by some column from A). Experimental evidence to this effect has been given by Tomlin [10] and Reid [9], for both methods.

1.1. QUALITATIVE COMPARISON

Suppose that the p-th column of B is replaced by some column a_q from A, and let the "partially updated" vector u be computed from the equation

$$Lu = a_q .$$

The main features of either method for updating L and U are as follows:

1. the p-th column of U is deleted (giving $\overset{*}{U}$ say) and the vector u is appended as a new column, giving an intermediate matrix $H = [\overset{*}{U}\ u]$ which is upper Hessenberg;
2. some elimination transformation E is computed such that EH is either a triangle or a permuted triangle;
3. the new LU factors are $\overline{B} = \overline{L}\,\overline{U}$ where

$$\overline{L} = LE^{-1} \quad \text{(stored in product form)}$$

and

$$\overline{U} = EH \quad \text{(computed explicitly).}$$

The difference between the two methods is in the choice of E, and from an implementation point of view this implies a vital difference in the data structure used for storing U. In Method BG, E is chosen primarily to maintain numerical stability and is such that *additional nonzero elements* may be created in U. The simplest approach is to use an in-core linked list for U (e.g. Reid [8]), but problem size is then limited by the amount of real core storage available, since the random-access nature of linked lists could cause what is known as thrashing on a machine with a paged memory. An alternative is to use a simple ordered list and to make a new copy of U (with appropriate changes) each simplex iteration. This would allow use of either virtual memory or disk, but it effectively doubles the total storage required for U.

In contrast, Method FT was designed expressly to avoid insertion of nonzeros into U, and in fact nonzeros can only be *deleted*. This means U can be stored in a sequential disk file in what amounts to ordinary product form: new columns are added one by one to the end of the file, and no re-writing of previous parts of the file is necessary. The penalty for this significant advantage is numerical stability (although in practice this is by no means catastrophic). Also, the complexity of the method in terms of growth of nonzeros (see next section) appears from the results of Reid [9] to be non-optimal.

Hence interest continues in alternative methods. By investigating the complexity of Method BG, our aim here is to find ways of handling arbitrarily large U's on a machine with a paged memory. We avoid use of linked lists and prefer not to re-write unchanged parts of U .

1.2. COMPLEXITY

Since computational complexity involves the study of upper and lower bounds on the time required to execute an algorithm, our attention here focuses on the number of matrix elements which must be manipulated each iteration of the simplex method. Thus we may ask the following questions of any relevant method for updating LU factors.

C1: *How many nonzero elements (in E above) are added to L each iteration?*

C2: *What is the probable number of nonzeros in u , the new column of U ?*

For Method BG we may also ask

C3: *How much fill-in occurs at each iteration when H is reduced to the triangular form $\overline{U}$?*

C4: *In particular, is it possible to define* where *the fill-in occurs, so that storage for all modified U's can be allocated in advance?*

In posing question C4, we indicate that the complexity of Method BG will be reduced by avoiding random access to U .

1.3. SUMMARY

The main results of this paper are as follows:

1. Method MBG is suggested (Modified Bartels and Golub) in which the new columns of U need not be added to the end of U . Instead they may be inserted somewhere between existing columns. This reduces the growth of nonzeros in both L and U .

2. We can answer "Yes" to question C4 above. Thus an implementation scheme is suggested which should allow the method to operate efficiently in a virtual memory environment.

Our analysis is intimately linked with the "bump and spike" structure associated with the Preassigned Pivot Procedures of Hellerman and Rarick [5], [6], and represents a first attempt to estimate the effect of the spike distribution on iterations subsequent to "reinversion".

2. Notation

Matrices are denoted by capitals (A, B , etc) and vectors by lower case (u, v , etc). The following elementary transformations are used:

(a) *Cyclic permutations* P_{ij} $(i \leq j)$. In the product $P_{ij}v$ the element v_i is moved down to position j and elements v_{i+1} to v_j are moved up one place.

(b) *Interchanges* I_{ij} $(i \leq j)$. The product $I_{ij}v$ interchanges v_i and v_j .

(c) *Eliminations* M_{ij} $(i \neq j)$. The matrix M_{ij} is equal to the identity matrix except for its (i, j) element which is some multiplier m_{ij} .

The work *spike* is used repeatedly to mean a column of either B or U , specifically one which has some nonzero elements above the diagonal. A spike column of U when stored in packed form is commonly called an eta vector.

3. Bumps and spikes

Here we assume the reader to be familiar with the structure of a typical *LP* basis matrix, as described by Hellerman and Rarick [5], [6] in connection with their Preassigned Pivot Procedures. Briefly, the facts are that the rows and columns of the basis can be permuted in such a way that the resulting matrix B is block lower triangular, and furthermore each block (*bump*) is almost strictly lower triangular except for a few columns (*spikes*) which have some nonzeros above the diagonal.

This structure is well-suited to *LU* factorization for the following reasons:

1. Gaussian elimination with *column* interchanges preserves both stability and sparsity. The interchanges can be confined to each bump separately and seldom create more spikes than in the preassigned ordering. (In any one bump, if a column has a small pivot the only alternatives are the spikes in the same bump. A spike/spike interchange leaves the number of spikes unaltered, while an occasional non-spike/spike swap usually causes an increase of one.)

2. Fill-in occurs only in the spike columns of B . Hence L is essentially the

same as B and can be imbedded in the full matrix A as suggested by Kalan [7].

3. Most of the "weight" is in L, while U is extremely sparse. (For example, if B is triangular then $L = B$ and $U = I$.) Since storage of U is the main problem we prefer this approach to others which put more weight in U.

4. Modified LU factors

Here we describe a slightly generalized updating scheme which includes Methods BG and FT as special cases. Let a_q replace the p-th column of B, and since the vector $u = L^{-1}a_q$ will in general be sparse, let its first and last nonzero elements be in positions f and l, respectively. Clearly we have

$$f \le p \le l \le m .$$

(It will usually be true that $l < m$ and we take advantage of this in §7.)

Now the matrix $L^{-1}\overline{B}$ is the same as U except that its p-th column is replaced by u. A cyclic permutation of columns p through l gives the upper Hessenberg matrix

$$H = (L^{-1}\overline{B})P_{pl}^T$$

as shown in Figure 1a. Various methods for obtaining a new triangular matrix $\overline{U}$ are listed below. We use the notation

$$E(k) \equiv E_{k-1}\, E_{k-2} \cdots E_{p+1}\, E_p$$

to mean the product of $k - p$ matrices E_j $(j = p, \ldots, k-1)$.

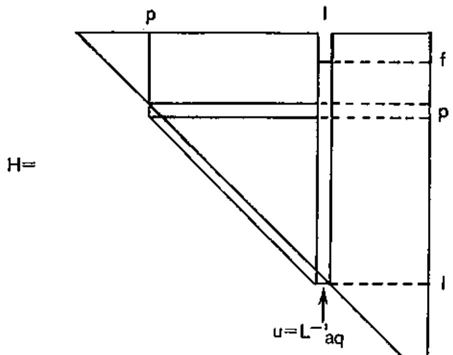

Fig 1a

The p–th column of U deleted

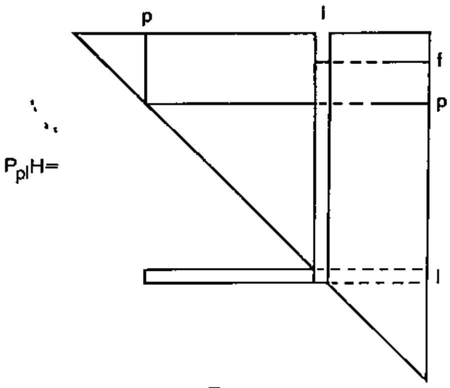

Fig 1b

The p–th row moved down to row l

1. Method BG. Bartels and Golub assume that $l = m$ and eliminate the sub-diagonal elements $h_{j+1,j}$ $(j = p, \ldots, m-1)$ using row interchanges to preserve stability. Thus $\overline{U} = E(m)H$ is a strict triangle and each E_j is one of three types:

$$E_j = I_{j,j+1} \tag{4.1a}$$

$$E_j = M_{j+1,j} \tag{4.1b}$$

$$E_j = M_{j+1,j}I_{j,j+1} \ . \tag{4.1c}$$

When U is sparse, most of the E_j will be simple interchange matrices (4.1a). Note however that $E_j = I$ is never true, so there are always $m - p$ transformations to be appended to L .

2. Method FT. Forrest and Tomlin again take $l = m$ and use elimination without interchanges to zero out the p-th row of H (except for the p-th element of u). Thus $\overline{U} = E(m)H$ is a permuted triangle and each E_j is one of two types:

$$E_j = I \tag{4.2a}$$

$$E_j = M_{p,j+1} \ . \tag{4.2b}$$

Most of the E_j will be the identity (4.2a) and hence considerably fewer than $m - p$ transformations are appended to L . (In practice, the numbers required to store the non-trivial E_j are exactly those required to store the nonzero elements of the sparse vector r where $U^T r = u_{pp}e_p$; e.g. see Tomlin [11].)

3. Method MBGa (Modified Bartels and Golub). A natural combination of the first two methods is to eliminate elements p to $l - 1$ of the p-th row of H using interchanges to preserve stability and/or sparsity. Thus $\overline{U} = E(l)H$ will be a permuted triangle, and in addition to (4.2a) and (4.2b) we may also have

$$E_j = M_{p,j+1}I_{p,j+1} \ . \tag{4.2c}$$

As with Method FT, most of the E_j will be identities and in general much fewer than $l - p$ transformations will be added to L .

4. Method MBGb. If the same cyclic permutation which produced H is applied to the *rows*

of H, the matrix $P_{pl}H$ of Figure 1b is obtained. The subdiagonal elements in the new l-th row can be eliminated using interchanges to give a strict upper triangle $\overline{U} = E(l)P_{pl}H$, where each E_j is one of three types:

$$E_j = I \tag{4.3a}$$

$$E_j = M_{l,j} \tag{4.3b}$$

$$E_j = M_{l,j}I_{j,l} . \tag{4.3c}$$

Again most of the E_j will be identities (4.3a), so there will be fewer than $l - p$ transformations. This method is equivalent to Method MBGa except for the permutation matrices involved.

Note that the transformation $P_{pl}H$ requires physically altering row indices in the packed columns of U. This is very inconvenient if U is stored on disk, and is avoided by Forrest and Tomlin by allowing $\overline{U}$ to be a permuted triangle. Similarly the first version of Method MBG is more efficient. However it is much easier to visualize what is happening if strict triangles are maintained, so we shall henceforth discuss Method MBGb.

5. The growth of L

As we saw in §3, Hellerman and Rarick's bump and spike structure leads to a very sparse LU after each refactorization. In particular, the bumps and spikes are inherited by U. We now look at the effect this structure has on L and U in subsequent iterations. In §§5 and 6 we assume that $l = m$ so that all new columns are added to the end of U.

Figure 2 shows a typical sparse U which originally came from a basis with three bumps containing 4, 5 and 2 spikes respectively. After k iterations there are k more spikes at the end of U ($k = 5$ is shown). The heavy dots may or may not be nonzeros in the spikes; they can be ignored until §6.

Recall that when U is updated at the k-th iteration, the p-th column of U is deleted and the p-th row of U is permuted to the bottom. In Figure 2, we suppose that the p-th column was near the front of the second bump (just before the first spike in that bump). Observe that any nonzeros in the new bottom row come from some of the spikes in the second bump and some of the k spikes at the end of U. Possible positions are marked by open dots. When Gaussian

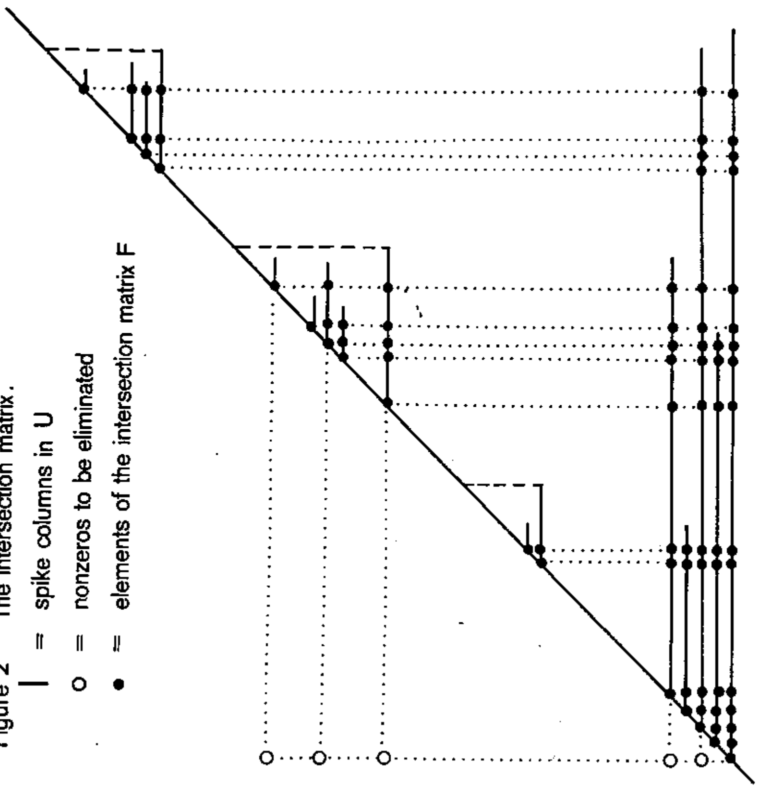
Figure 2 The Intersection matrix.
| = spike columns in U
○ = nonzeros to be eliminated
● = elements of the intersection matrix F

elimination with row interchanges is used to eliminate them, some additional nonzeros may be created in the bottom row, but again they can come only from the second bump and from the last k spikes. Hence the following result, which provides an answer to Complexity Question C1.

THEOREM 1. *At the k-th iteration, if the column deleted from* U *is in a bump containing* s_i *spikes, the number of elementary transformations added to* L *is bounded by* $s_i + k$.

Note that $s_i = 0$ is possible (if the p-th column was between two bumps).

The storage required for each transformation $M_{m,j}$ or $M_{m,j}I_{j,m}$ in (4.3b, c) is essentially one nonzero (a multiplier and the index j). Also, if $s = \sum s_i$ is the total number of spikes in the initial U , we have $s_i \leq s$. Hence Theorem 1 can be restated as follows, in a slightly looser form.

THEOREM 2. *The number of elements added to* L *at the k-th iteration is at most the current number of spikes, and is bounded by* $s + k$.

This bound could be reached only if all spikes in U were of a similar "height" and if p

preceded the first spike. At any rate $s \ll m$ is generally true and the growth of L is seen to be minimal.

6. Bounding the fill-in of U

Since the number of nonzeros in a vector $u = L^{-1}a_q$ is arbitrary except in the case of structured problems (e.g. see §7.2), it does not seem possible in general to bound the total size of U. Thus we put complexity question C2 aside and move on to question C3; i.e. we accept however many nonzeros there might be in a spike u when that spike is created, and obtain a bound on the *fill-in* that could occur in u during later iterations.

Let $t = s + k$ be the total number of spikes in U after k iterations. These define a subset of the m columns of U. We now consider the same subset of the *rows* of U. Thus we may write U as the sum of two matrices:

$$U = R + F$$

where F will be called the *intersection matrix* and is zero except for t rows and columns (including t diagonals) which have been extracted from U. Then we define $R = U - F$.

With the aid of Figure 2, the non-trivial columns of F may be thought of as parallel roads going North, and the rows of F are like cross-roads travelling East from the foot of each spike, crossing the North-bound roads in at most $\frac{1}{2}t(t+1)$ intersections. Note however that *not all intersections are present*, since the North-bound roads are all "blind", i.e. they stop at various points depending on the height of the corresponding spikes. In Figure 2, the relevant points of F are marked by heavy dots. Note that these points of F may be zeros *or* nonzeros in the spikes of U.

Thus R and F are both upper triangular matrices which are sparse in the same sense as U, as they have only a few spike columns which are themselves sparse.

Now recall the situation in §5 where a row of U was permuted to the bottom ready for elimination. In Figure 2, the nonzeros to be eliminated are marked by open dots. Close inspection of which rows of U are actually affected by the elimination leads directly to the following result:

THEOREM 3. *Except for nonzeros deleted from the p-th row each iteration, the only elements of U that can be altered by subsequent updates are those of the intersection matrix F; i.e. at most $\frac{1}{2}t(t+1)$ elements.*

This means that only in F can *Fill-in* potentially occur, while the matrix R is essentially *Read-only*. Moreover, when a new spike of U is created, the disposition of all previous spikes defines exactly which elements are in the new column of F. Some of these may already be nonzero; the remainder are subject to fill-in during later iterations.

6.1. PRACTICAL IMPLICATIONS

We can now answer "Yes" to complexity question C4, and an implementation scheme comes to mind in which storage is allocated in advance for all possible nonzeros in F. Each column of F can be stored in a segment of (virtual) memory of predetermined size, and there is no longer any need for linked-lists to cater for random access.

During solution of systems of the form $Ux = y$, corresponding columns of R and F must be accessed at the same time, but they need not be stored together; in fact they may be regarded as "split eta vectors" as commonly used in product-form algorithms when a transformation spans two or more physical *I/O* records. Separation of R from F would be beneficial in any virtual memory system which avoids writing unaltered pages to disk during overlays (e.g. *OS/VS*1 and *OS/VS*2 on the IBM System/370). Alternatively, R can be stored in product form on disk, as it is analogous to the full U of Method FT. For each iteration, provision must still be made for deleting elements in the p-th row, since these are usually in R. (They are in F only if the p-th column of U is a spike.)

Precise implementation details will depend on the relative sizes of R and F. In some cases, it would be simpler to keep R and F together in virtual memory and ignore the read-only property of R. (For example, the Burroughs B6700 operating system re-writes data segments to disk whether they have been altered or not.) Deletion of elements from U would then require no special treatment.

6.2. COMPLEXITY

Since R is a submatrix of U, we already know from the results of Tomlin [10] and Reid [9] that it is extremely sparse in practice. The main question to be asked now is C3, which might be rephrased as

$\overline{\text{C3}}$: *How many potential nonzeros must be allowed for in each column of* F, *and is this number considerably less than the obvious bound of* m ?

To refine the bound given in Theorem 3, we consider spikes in the initial U separately from those added during updates. If s_i is the number of spikes in the i-th bump we clearly have

THEOREM 4. *The storage required in* F *to allow for fill-in in the* i*-th bump of the initial* U *is bounded by* $f_i = \frac{1}{2}s_i(s_i+1)$.

Hellerman and Rarick [5] give statistics for five medium-scale problems, for which the average number of spikes per bump is in the range (1.5, 7.7) . Of course most of the total number of spikes (ranging from 23 to 108) could have been in one particular bump, but if we assume an even distribution and take these five problems to be typical, we might expect s_i to be of order 10 , so that $f_i \doteqdot 50$. Since the average number of bumps is of order 15 the fill-in in the initial U is minimal ($\doteqdot 15 \times 50 = 750$) *regardless of the number of subsequent iterations.*

During updating, if all new spikes are added to the end of U they tend to make U into one large bump. If $s = \sum_i s_i$ is the total number of spikes in the initial U we have

THEOREM 5. *The storage required in* F *to allow for alterations to the* k*-th new column of* U *is at most* $s + k$.

The value of s may not be large in general (the average is $s = 50$ in the examples of [5], with a median of $s = 32$), but if k is allowed to grow to 200 or 300 iterations, the cumulative effect starts to look significant:

THEOREM 6. *The storage required in* F *to allow for fill-in in* k *new columns of* U *is bounded by* $f(k) = \sum_{j=1}^{k} (s+j) \doteqdot ks + \frac{1}{2}k^2$.

With $s = 50$, this gives $f(50) \doteqdot 3750$, $f(100) \doteqdot 10\ 000$ and $f(200) \doteqdot 30\ 000$. However, if s were as large as 50 , refactorization would probably be performed for other efficiency reasons for some $k \le 100$.

6.3. RELEVANCE TO METHOD FT

When new columns of U are created, they could already be nonzero in previous pivot rows (i.e. in the rows of the intersection matrix F). If so, the bound $f(k)$ of Theorem 6 has meaning even with Method FT. In particular, the term $\frac{1}{2}k^2$ corresponds to the bottom triangle of U (see Figure 2) and it does seem likely that this part of U will be dense.

In this context it is interesting to note the growth rates given by Forrest and Tomlin [4] for three large-scale problems. After 70 iterations of Method FT the total increase in nonzeros in both L and U is 4097 , 6705 and 11398 , respectively. A large part of this growth is probably attributable to the term $\frac{1}{2}k^2$ ($\doteqdot 2500$ when $k = 70$). The term ks would account for all remaining growth, if s were

$$23 , \quad 60 \quad \text{and} \quad 127 \tag{6.1}$$

respectively. Certainly the true values of s could be larger if complete fill-in did not occur, but since some of the total growth reported must have been in L and R it seems reasonable to suppose that (6.1) gives a realistic estimate of the total number of spikes in the starting basis (and hence in the initial U) for each problem.

One might venture to call this discussion *backward eta analysis.*

NOTE. In [4] the list of columns of U is called the backward eta file.

6.4. CAN WE DO BETTER?

The bounds given in the previous theorems are conservative because they assume that the spikes in any one bump are all of similar height, and that the new spikes span all bumps of the initial U . This is unlikely to be the case in practice, and storage can be minimized accordingly.

For example, in the bound of Theorem 6, the ks term will be conservative if some of the new spikes are short. This does not matter because it is easy to avoid allocating storage for F above the top of such spikes.

On the other hand, it seems likely that a $\frac{1}{2}k^2$ growth rate *can* be expected with the standard Method BG (assuming bump and spike-type ordering for the initial LU). With a view to lowering this part of the bound, we now consider putting new columns of U somewhere other than at the end.

7. The MIDDLE rule

In §4, we saw that if the last nonzero of the new spike u is in position l then the number of updating transformations can be reduced by making u the l-th column of U . In practice this means that the previous spikes must be listed in some order and the new spike must be inserted somewhere in the middle of the list. We shall call this the MIDDLE rule: Merge (the new spike) Into the Disposition Defined by its Last Element.

7.1. IMPLEMENTATION ASPECTS

Regarding virtual memory (VM) as one large array of storage, the simplest scheme would be to pack spikes into storage one after the other (just as before) and to maintain an ordered list of pointers to the start and finish of each spike. If the current U occupies the first w words of store the next spike may occupy words $w + 1$ onwards. Alternatively, it may use storage vacated by deleted spikes, if a large enough gap is available. In either case, the pointers are easily updated to reflect the proper logical ordering and no moving of previous spikes is required.

Since the elements of any one spike occupy contiguous words of storage, this scheme satisfies a basic rule of thumb for economic use of VM, viz. that memory references be *clustered*. As such, it would be considerably more efficient than storing U in one large linked list.

Greater efficiency should be possible, if some attention is given to page boundaries. Note that a segment of 256 words on a Burroughs B6700 and a page of 4096 bytes on an IBM 370/168 could hold about 200 and 400 packed nonzeros respectively. Hence a particularly large spike may require 2 or 3 pages of store, but often there may be several spikes packed into a single page. In the latter case it would be ideal (i.e. page-swapping would be minimized), if those spikes were close together in the ordered sequence. With this in mind we suggest the following scheme:

1. For the initial U , pack spikes one after the other without regard to page boundaries. (Alternatively, the first spike in each bump could begin a new page.)
2. For each new spike, if less than one page is required find the nearest logical neighbour whose page(s) contain sufficient unused storage. Otherwise, or if no such neighbour exists, assign the spike to consecutives pages of its own.

This scheme will lead to some internal fragmentation (wasted space inside pages), but it extends the previously mentioned clustering of memory references.

One other problem with the MIDDLE rule is that it requires insertion of new rows into the intersection matrix. Strictly speaking, the k-th new spike should allow for fill-in of $s + k + (n-k)$ elements instead of the $s + k$ in Theorem 5, i.e. a constant $s + n$ elements, where n is the refactorization frequency. However from a probabilistic point of view this would be very conservative. The best means of economizing in this area must be left to experiment.

7.2. IS $l < m$?

Suppose that the matrix A has block-angular structure and that, when a basis B is factorized, the row and column permutations are chosen to maximize sparsity inside blocks without destroying the overall angular structure. The resulting structure of L and U is shown in Figure 3 (a 3-block example), where L_j, U_j and V_j arise from basis vectors in the j-th block of B . If column a_q enters the basis from the j-th block of A , it is clear that a_q and $u = L^{-1}a_q$ are of the form

$$a_q = \begin{bmatrix} 0 \\ a_j \\ 0 \\ c_j \end{bmatrix}, \quad u = \begin{bmatrix} 0 \\ u_j \\ 0 \\ w_j \end{bmatrix}$$

where all quantities are vectors. The best we can say is that if $c_j = 0$, then $w_j = 0$. In this way, it would certainly be advantageous to apply the MIDDLE rule and insert u_j somewhere amongst the columns of U_j , for then only L_j, U_j and V_j will be altered.

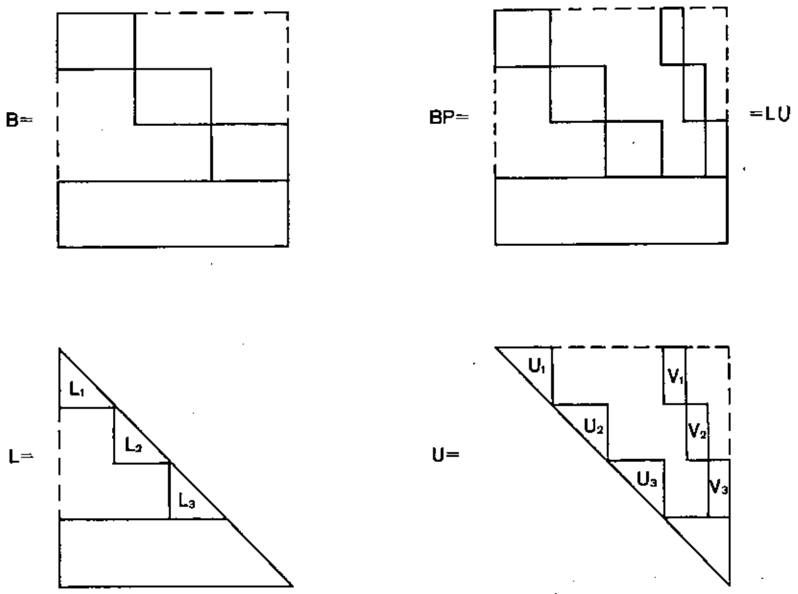

Figure 3 LU factors of a basis matrix from a problem with block–angular form

(P is a permutation matrix.)

In other words, if a variable entering the basis from block j is not coupled to the other blocks, the updating procedure will automatically affect only those parts of L and U belonging to block j. This seems to be a desirable property.

Similar statements can be made, if A has dual-angular structure.

For general problems, the distribution of the last nonzero of $L^{-1}a_q$ remains to be determined experimentally. We anticipate that use of the MIDDLE rule will spread the new spikes of U evenly among the initial bumps, so that the bumps will slowly expand and merge into one another, as suggested by the drawing in Figure 4.

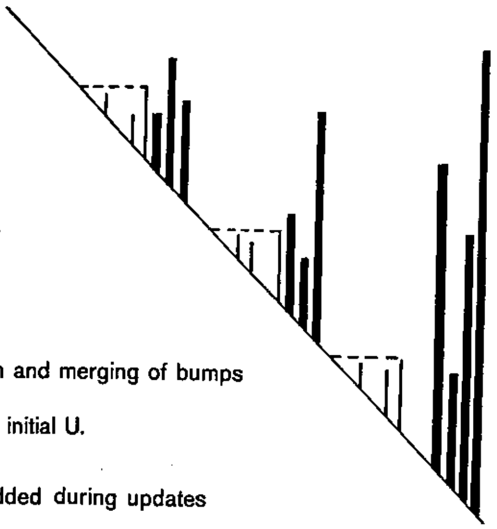

Figure 4 Expansion and merging of bumps

| = spikes in initial U.

▌ = spikes added during updates

8. Conclusions

The numerical results obtained by Reid [8], [9] show that, in Method BG, the freedom to interchange rows of U for both stability and sparsity reasons can lead to significantly lower growth of nonzeros than in Method FT. This is an instance of a rather rare situation, wherein the search for *increased numerical reliability* (of LU updating) has actually led to an algorithm which has *greater efficiency*. More often the two are conflicting desiderata.

Likewise, the motive for obtaining bounds on the complexity of Method BG (how serious *is* the fill-in of U ?) was provided by a wish to extend the method to virtual memory machines. This led to the realization that potential storage requirements are known in advance, so that implementation should be straightforward. It also led to the MIDDLE rule of §7 which promises to reduce the growth of nonzeros below that of both Method FT and the original Method BG. In this case, the search for *increased applicability* of a reliable method has led to an algorithm of greater efficiency.

To summarize, we suggest column-wise storage of U , along with explicit storage of the intersection matrix of §6, as a natural means of implementing the method of Bartels and Golub in a paging environment. We also suggest use of the MIDDLE rule to reduce the size of the intersection matrix, and to minimize the growth of L and U in any other implementation of the original method. Finally we hope that the actual benefits obtainable can be investigated experimentally in the near future.

Acknowledgements

I am very grateful to Dr John Reid for providing details [9] of his work on sparse linear programming.

References

[1] Bartels, R.H., A stabilization of the simplex method. *Num. Math.* 16, 414-434, 1971.

[2] Bartels, R.H. and Golub, G.H., The simplex method of linear programming using LU decomposition. *Comm. ACM* 12, 266-268, 1969.

[3] Dantzig, G.B., *Linear Programming and Extensions*. Princeton University Press, Princeton, N.J., 1963.

[4] Forrest, J.J.H. and Tomlin, J.A., Updating triangular factors of the basis to maintain sparsity in the product form simplex method. *Mathematical Programming* 2, 263-278, 1972.

[5] Hellerman, E. and Rarick, D., Reinversion with the preassigned pivot procedure. *Mathematical Programming* 1, 195-216, 1971.

[6] Hellerman, E. and Rarick, D., The partitioned preassigned pivot procedure (P^4) , 67-76; in *Sparse Matrices and their Applications*, D.J. Rose and R.A. Willoughby (eds). Plenum Press, N.Y., 1972.

[7] Kalan, J.E., Aspects of large-scale in-core linear programming. Proceedings of ACM Annual Conference, Chicago, Illinois, 1971.

[8] Reid, J.K., Sparse linear programming using the Bartels-Golub decomposition, presented at the VIII International Symposium on Mathematical Programming, Stanford University, August 1973.

[9] Reid, J.K., *Lecture Notes on Sparse Linear Programming*, 1974.

[10] Tomlin, J.A., Modifying triangular factors of the basis in the simplex method, 77-85; in *Sparse Matrices and their Applications*, D.J. Rose and R.A. Willoughby (eds). Plenum Press, N.Y., 1972.

[11] Tomlin, J.A., On pricing and backward transformation in linear programming, *Mathematical Programming* 6, 42-47, 1974.