Computing a Sparse Basis for the Null Space*

John R. Gilbert†

Michael T. Heath‡

TR 86-730
January 1986

Department of Computer Science
Cornell University
Ithaca, NY 14853

* This work was supported by the Applied Mathematical Sciences Research Program of the Office of Energy Research of the U.S. Department of Energy under contract DE-AC05-84OR21400 with Martin Marietta Energy Systems, Inc. Publication of this report was partially supported by the National Science Foundation under grant DCR-8451385.

† Department of Computer Science, Cornell University, Ithaca, New York, 14853.

‡ Mathematical Sciences Section, Engineering Physics and Mathematics Division, Oak Ridge National Laboratory, Oak Ridge, Tennessee, 37831.

COMPUTING A SPARSE BASIS FOR THE NULL SPACE*

John R. Gilbert† and Michael T. Heath‡

January 1986

Abstract. We present algorithms for computing a sparse basis for the null space of a sparse underdetermined matrix. We describe several possible computational strategies, both combinatorial and noncombinatorial in nature, and we compare their effectiveness for several test problems.

Key words. null basis, null space, sparse matrix, bipartite graph, matching

1. Introduction. Let A be an $m \times n$ matrix of rank r. (Without loss of generality, we will assume throughout that $r \leq m \leq n$.) If B is an $n \times (n-r)$ matrix of rank $n-r$ such that

$$AB=0,$$

then the columns of B form a basis for the $(n-r)$-dimensional null space of A. For brevity, we will refer to such a matrix B as a *null basis*. We will refer to the individual columns of B as *null vectors*, each of which corresponds to a set of columns of A whose linear combination is equal to zero. Obviously such a matrix B is not unique, not only in the relatively trivial sense of different possible scalings and column permutations, but also in the sense that there may be structurally distinct null bases for the same A (i.e., involving different combinations of columns of A).

We are concerned in this paper with computing the elements of the matrix B explicitly, although other representations for a null basis are possible (e.g., product forms). More specifically, if the matrix A is sparse, we wish to compute a suitably sparse null basis B. It is difficult to define precisely what we mean by a "suitably" sparse null basis. For one thing, there may be no such sparse B. For example, the matrix

$$[I, e],$$

where I is the identity matrix and e is the column vector all of whose components are equal to 1, is quite sparse but has no explicit sparse representation for its one-dimensional null space. Moreover, even if a sparse null basis exists, the problem of computing a sparsest representation for it has been shown to be NP-hard [2]. As in many sparse matrix computations, we will therefore content ourselves with developing heuristic computational strategies that find a "good" sparse null basis, though not necessarily the sparsest possible.

The sparse null basis problem has at least one important feature that distinguishes it from most other sparse matrix problems. The analysis of most sparse matrix problems is simplified by ignoring any zeros that might be created through exact cancellation as a result of some arithmetic operation on nonzeros (see, e.g., [4, p. 27]). In computing a sparse null basis, however, we are specifically seeking nontrivial linear combinations of nonzeros that give a zero result (i.e., arithmetic cancellation). For this reason we will necessarily employ numerical techniques along with some standard combinatorial methods, such as bipartite matching.

* This work was supported by the Applied Mathematical Sciences Research Program of the Office of Energy Research of the U.S. Department of Energy under contract DE-AC05-84OR21400 with Martin Marietta Energy Systems, Inc. Publication of this report was partially supported by the National Science Foundation under grant DCR-8451385.

† Department of Computer Science, Cornell University, Ithaca, New York, 14853.

‡ Mathematical Sciences Section, Engineering Physics and Mathematics Division, Oak Ridge National Laboratory, Oak Ridge, Tennessee 37831.

Before proceeding with a discussion of the algorithms we have developed, we will first give some applications that justify our interest in the sparse null basis problem and review other work on it. We then state our basic strategy for computing a sparse null basis and explore in detail several possible variations. The results of extensive empirical testing and some final observations conclude the paper.

2. Applications. There are numerous applications in which a null basis is important. The fundamental fact on which most of these applications are based is that the general solution of an underdetermined system of linear equations

$$Ax = b \tag{2.1}$$

can be expressed as

$$x = \hat{x} + By \tag{2.2}$$

for some vector y, where $\hat{x}$ is any particular solution to the system and B is a null basis. In constrained optimization problems, for example, if a set of linear (or linearized) equality constraints is expressed in the form (2.1), then every feasible point can be expressed in the form (2.2), thereby allowing the constrained problem to be solved by means of an unconstrained problem in the variable y. See [1, pp. 99-104] or [5, pp. 155-163] for further discussion of such null space methods in optimization.

The specific application that motivated our own interest in the null basis problem is the force method of structural analysis. Here A is the equilibrium matrix of a structure and b is a vector of applied loads, so that (2.1) expresses a constraint on the system force vector x, which is to be determined. (See [8] and references therein for further details of the discussion to follow). The locality of connections within the structure causes the matrix A to be quite sparse.

Minimizing the potential energy requires that x minimize the quadratic form

$$\tfrac{1}{2}\ x^T D x$$

subject to the constraint (2.1), where the $n \times n$, symmetric, block-diagonal matrix D is the element flexibility matrix of the structure. Using (2.2), we see that y must satisfy the symmetric linear system

$$B^T D B y = -B^T D \hat{x}. \tag{2.3}$$

In this context the null basis B is called the self-stress matrix. Thus, having computed a particular solution $\hat{x}$ and the redundant force vector y, the desired system force vector x is given by (2.2).

One of the principal virtues of the force method is that it separates the computation into two somewhat independent phases:

1. Compute a null basis B and a particular solution $\hat{x}$.
2. Solve the linear system (2.3).

The importance of this separation becomes apparent when solving a sequence of problems having a fixed layout but differing material properties, such as multiple redesign problems or nonlinear elastic analysis. In such cases the matrix A is fixed, but the matrix D changes from problem to problem. Thus the first phase need be done only once for the entire sequence of problems, and only the second phase is repeated for each problem. A further implication is that it may be worth considerable effort producing a sparse B, since this one-time cost will be amortized over the whole sequence of problems.

Another important conclusion we can draw from the various uses of the null basis is that it should be as well conditioned as possible (i.e., the columns of B should not be nearly linearly dependent numerically). For example, a poorly conditioned B would make the conditioning of the linear system (2.3) extremely poor and might therefore yield a highly inaccurate solution. For numerical purposes, an orthogonal null basis would be highly desirable, but in many cases this

goal would conflict too greatly with sparsity considerations.

3. Methods for computing a null basis. Many mathematical programming algorithms use a variable-reduction technique to compute a null basis B (see, e.g., [5, p. 163]). Assume for the moment that A has full row rank m, and let A be partitioned so that

$$AP=[A_1, A_2],$$

where A_1 is $m \times m$ and nonsingular, and P is a permutation matrix that may be required in order to ensure that A_1 is nonsingular. We may then take

$$B=P\begin{bmatrix} -A_1^{-1}A_2 \\ I \end{bmatrix}. \tag{3.1}$$

A permutation P that yields a structurally nonsingular A_1 can be chosen purely symbolically (see, e.g., [3]), but this says nothing about the possible numerical conditioning of A_1 and the resulting B.

In order to control numerical conditioning, numerical pivoting must be employed. Several such methods have been proposed based on various matrix factorizations, including LU, QR, LQ, SVD, and Gauss-Jordan elimination (see [8] for a survey). For example, QR factorization with column pivoting (see, e.g., [6, p.165]) yields

$$AP=Q[R_1, R_2],$$

where P is again a permutation matrix, and R_1 is an upper triangular matrix of order m. We may now take

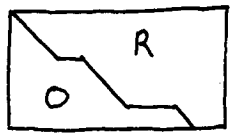
$$B=P\begin{bmatrix} -R_1^{-1}R_2 \\ I \end{bmatrix}. \tag{3.2}$$

We note that if the permutation matrix P were the same in both cases, then the null bases given by (3.1) and (3.2) would be the same. Thus, the QR approach can be viewed simply as a means of choosing a permutation P on numerical grounds. Of course, numerical considerations may be at odds with sparsity considerations, and a compromise may have to be made between the two. In any case, with either (3.1) or (3.2) there may be a great deal of intermediate fill during the computation. Moreover, forcing B to contain an embedded identity matrix may restrict us to a considerably less sparse null basis than might otherwise be possible.

When A is banded, a method for computing a banded null basis B has been developed by Topcu [13] and Kaneko, Lawo and Thierauf [9]. Their method is based on LU factorization and is called, for reasons that will become obvious, the "turnback" method. Heath, Plemmons, and Ward [8] extended and adapted this method for use with QR factorization. Our algorithms, described in Sections 4 and 5, were motivated by turnback; thus we describe this method in some detail below.

Write $A=(a_1,a_2,\ldots,a_n)$ by columns. A *start column* is a column a_s such that the ranks of $(a_1,a_2,\ldots,a_{s-1})$ and $(a_1,a_2,\ldots,a_s)$ are equal. Equivalently, a_s is a start column if it is linearly dependent on lower-numbered columns. The coefficients of this linear dependency give a null vector whose highest-numbered nonzero is in position s. It is easy to see that the number of start columns is $n-r$, the dimension of the null space of A.

The start columns can be found by doing a QR factorization of A, using orthogonal transformations to annihilate the subdiagonal nonzeros. Suppose that in carrying out the QR factorization we do not perform column interchanges but simply skip over any columns that are already zero (or numerically negligible) on and below the diagonal. The result will be a factorization of the form

$A=Q$ 

.

The start columns are the columns where the upper triangular structure jogs to the right; that is, a_s is a start column if the highest nonzero position in column s of R is no larger than the highest nonzero position in earlier columns of R.

Turnback finds one null vector for each start column a_s by "turning back" from column s to find the smallest k for which columns $a_s, a_{s-1}, \ldots, a_{s-k}$ are linearly dependent. The null vector has nonzeros only in positions $s-k$ through s. Thus if k is small for most of the start columns, then the null basis will have a small profile. Note that turnback operates on A, not R. The initial QR factorization of A is used only to determine the start columns, and is then discarded.

As described above, the null vector that turnback finds from start column a_s may not actually be nonzero in position s. Therefore, turnback needs to have some way to guarantee that its null vectors are linearly independent. Heath, Plemmons, and Ward accomplish this by forbidding the leftmost column of the dependency for each null vector from participating in any later dependencies. Thus, if the null vector for start column a_s has its first nonzero in position $s-k$, every null vector for a start column to the right of a_s will be zero in position $s-k$.

4. Overview of the algorithms. The four algorithms we compare in this paper all fit the following framework, which is based on turnback.

```
Preorder the columns of A;
Perform QR factorization of A to get start column numbers s_1, s_2,...,s_{n-r};
for j:=1 to n-r do
    Find a null vector whose highest nonzero position is s_j
```

The initial QR factorization is done by the George-Heath algorithm as described in [7]. As in turnback, the factorization is used only to find the start columns, and is then discarded. Preordering the columns of A may be necessary to make the initial QR factorization sparse. We experimented with several preordering strategies, as described in Section 6.

Each start column is the rightmost member of some dependent set of columns. Thus, each start column corresponds to a null vector whose highest-numbered nonzero is in that column. Each such null vector is found independently.

The algorithm maintains a set of *active columns*, initially containing only the current start column a_s. It adds lower-numbered columns to the active set, one at a time. If a lower-numbered column is dependent on some active columns not including the start column, that column is not added to the set. When the active set becomes linearly dependent, its columns correspond to the nonzero positions of the desired null vector.

The *active rows* are the rows of A in which some active column is nonzero. The *active submatrix* is the matrix of active rows and columns. Thus, the algorithm keeps adding columns to the active submatrix until it becomes deficient in column rank. In order to produce a sparse null vector, we want the active submatrix to grow as little as possible before a dependency is found.

The algorithm for finding one null vector is summarized in the following pseudocode.

```
ActiveColumns := {a_s};
repeat
    Choose an inactive column a_c, c < s;
    if a_c is independent of ActiveColumns - {a_s}
        then ActiveColumns := ActiveColumns + {a_c}
until ActiveColumns is linearly dependent
```

The algorithm determines linear dependence or independence by maintaining a QR factorization of the active submatrix. Since the active submatrix contains only the active rows and columns, it is likely to be quite dense. Therefore, the QR factorization is stored in a dense data structure, each column of which contains a column of R above the diagonal and a Householder transformation below the diagonal. It is updated as follows. Suppose there are k active columns, and a new column is being considered as a potential active column. The first k Householder

transformations are applied to the new column. If the result has any nonzeros below position k, a new Householder transformation is computed to zero the new column below position k (thus updating the QR factorization) and the new column becomes active.

If, on the other hand, the result is zero below position k, then the new column is dependent on the other active columns. Either the dependency includes the start column, in which case the desired null vector has been found; or the dependency excludes the start column, in which case the new column does not become active and the QR factorization is not updated. Once a dependency has been found, the numerical values of the nonzero entries in the corresponding null vector are computed by back substitution with the triangular matrix R.

This procedure guarantees that the active columns are always linearly independent, so when we find the null vector it will be nonzero in position s as desired. Notice that this part of the algorithm is purely numerical; we use no information about the nonzero structure except in the definition of an active row as one in which an active column is nonzero.

The size of the active submatrix is crucial to the efficiency of the algorithm in three ways. First, its QR factorization (stored in dense format) dominates the space required by the algorithm. Second, updating this QR factorization dominates the total time required. Third, the number of columns in the active submatrix is at least as large as the number of nonzeros in the current null vector, so small active submatrices will lead to a sparse null basis. We want somehow to select columns for the active submatrix in a way that will keep the active submatrix small. The following section considers several strategies for selecting columns.

5. Details of column selection strategy. The heart of the algorithm is the strategy for choosing columns to add to the active submatrix. Since finding the sparsest null vector of a matrix is NP-hard [2], we do not hope to find the best possible choice of columns. Rather, we consider several heuristics.

5.1. Closest column next (Turnback). The simplest strategy is to choose columns in right-to-left order from the start column a_s. This is the "turnback" strategy described in Section 3 above, with a minor difference in the way it avoids finding linearly dependent null vectors. The turnback algorithm in [8] never adds to the active submatrix the lowest-numbered nonzero column of any earlier null vector; our implementation may add such a column, but it never adds a column that would create a dependency that does not include a_s.

Turnback performs well when the columns in a dependent set are close together in A, which happens when A is banded. Turnback tries to minimize the bandwidth of the current null vector, so it tries to produce a banded null basis.

Our experience is that turnback usually produces a basis with a sparse band, even for the problems on which it performs best. Therefore, a general sparse data structure may be more compact than a band or envelope data structure. Our implementation stores both A and the null basis B by columns in the general sparse data structure used in Sparspak [4].

5.2. Cheapest column next. Turnback tries to find null vectors with small bandwidth by choosing columns close to the start column. In a general sparse setting we want to choose columns on grounds of sparsity rather than bandwidth. The next algorithm assigns each column a *cost* that measures the growth it would cause in the active submatrix; then the algorithm chooses the cheapest column.

Let n be the number of columns in A. The cost of column a_j is defined to be

$$\begin{aligned} cost(a_j) = &\text{(number of nonzeros in inactive rows of } a_j\text{)} \\ &- \text{(number of nonzeros in active rows of } a_j\text{)}/n \\ &- j/n^2. \end{aligned}$$

This definition makes a column cheaper if it adds fewer rows to the active submatrix. The null vector is sparse if the final active submatrix has few columns. The submatrix has few columns if

it has few rows, since the null vector is complete when the submatrix becomes deficient in column rank.

In case of a tie in the number of nonzeros in inactive rows, we make a column cheaper if it has more nonzeros in active rows. The heuristic reason for this is that we hope to encounter numerical cancellation that will make the active submatrix deficient in column rank while it still has more rows than columns. Our experience is that such cancellation is more likely if the active submatrix is denser.

If ties in cost still remain, we make a column cheaper if it is farther to the right, that is, closer to the start column. All else being equal, this tries to minimize bandwidth. Our experience is that this tiebreaking rule usually makes little difference in the sparsity of the basis, but on some banded problems it helps significantly. Cheapest-column-next with cost defined only by this tiebreaking rule is the same as turnback.

5.3. Choosing a column by matching. A more sophisticated way to choose columns for the active submatrix is based on the combinatorial structure of the matrix. In this section we describe two versions of a heuristic that uses matchings in bipartite graphs to guide the search for a good column. Appendix A contains the necessary definitions and lemmas from bipartite matching theory.

In combinatorial terms, the matrix A is a bipartite graph whose two disjoint sets of vertices are its rows and its columns, and whose edges are its nonzeros. The start column a_s is a vertex of A. The active submatrix is the subgraph containing the active columns, the active rows (which are the vertices adjacent to active columns), and the edges between them. The active columns less a_s are always independent, so by Lemma 3 there is a matching that covers all the active columns except a_s. The new column to be added will increase the size of the matching by one. The algorithm searches for a column to add by following alternating paths. We give details below, followed by a proof that the algorithm will find the desired null vector.

Though we use matchings and alternating paths to guide the algorithm, we still use the numerical QR factorization of the active submatrix to decide when sets of columns are dependent. This lets us avoid any no-cancellation assumptions, and it lets us find null vectors that could not be predicted from the structure alone of A. It means that we must be careful to distinguish numerical and structural notions both in the algorithm and in its correctness proof: "dependent" and "independent" are numerical; "matching", "path", and "cover" are structural.

The algorithm. Given a start column a_s, this algorithm finds a null vector whose highest nonzero position is s, if there is such a null vector. In the algorithm, C is the set of active columns. The active rows are all those rows in which some active column is nonzero.

$C := \{a_s\}$;
Start with the empty matching;
repeat
    Find an alternating path from some uncovered active row, number r, to some inactive column a_c to the left of a_s that is independent of $C - \{a_s\}$;
    Alternate along the path, increasing the size of the matching by one and covering row r and column c;
    $C := C + \{a_c\}$;
until either a_s is dependent on $C - \{a_s\}$
    or no such alternating path exists;
if a_s is dependent on $C - \{a_s\}$
    then null vector := coefficients of the dependency
    else report "no such null vector"

An invariant that is true at the beginning and end of the main loop is: The columns in $C - \{a_s\}$ are independent, and the matching covers all columns in $C - \{a_s\}$ and only active rows. See Figure 1 for a sketch. (The active columns and rows may not actually be contiguous in the matrix.)

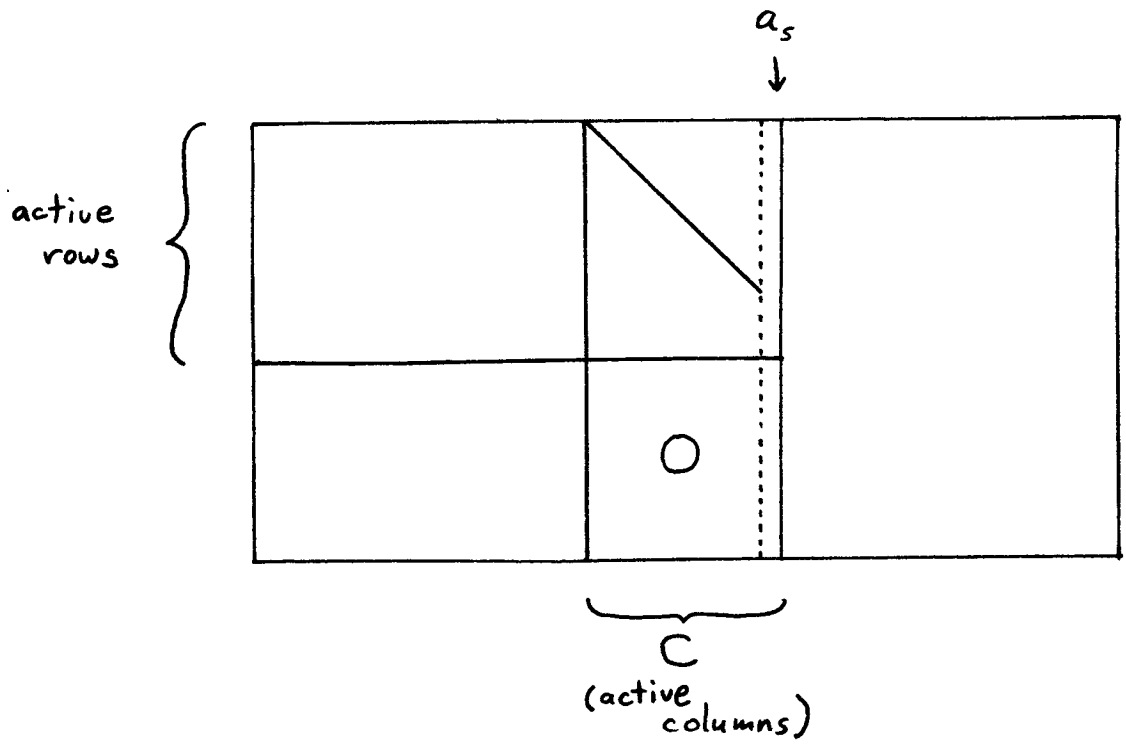


Figure 1. Computing one null vector. Columns in $C-\{a_s\}$ are independent and matched to active rows. Active rows and columns may not be contiguous.

If there is no numerical cancellation, so that the rank of every matrix involved is equal to its maximum matching size, then the algorithm will find a null vector when the active columns first become more numerous than the active rows. Then there will be exactly one more active column than active row, and the matching will cover all the active rows. If there is cancellation, the algorithm may find a null vector when there are more active rows than columns. This will be a null vector whose nonzero structure could not have been predicted from the nonzero structure of A.

Correctness of the algorithm.

Theorem. *If there is a null vector whose highest nonzero position is s, this algorithm stops with a_s dependent on $C-\{a_s\}$; otherwise it stops with a_s independent of $C-\{a_s\}$.*

Proof. Each iteration of the loop makes C larger, so the algorithm must stop eventually. If there is no null vector, a_s is independent of all the earlier columns, so it is independent of $C-\{a_s\}$. Thus we need only prove that if the null vector exists then the algorithm will not stop early; that is, if the null vector exists and a_s is independent of $C-\{a_s\}$, then there is an alternating path from some uncovered active row to some inactive column to the left of a_s.

Assume that the desired null vector exists. Then a_s is a linear combination of the columns to the left of a_s. It is not a linear combination of $C-\{a_s\}$, so there is some $c<s$ such that a_c is independent of $C-\{a_s\}$, even considering only the active rows. Then $C-\{a_s\}+\{a_c\}$ is independent, even considering only the active rows. Then Lemma 3 says that $C-\{a_s\}+\{a_c\}$ has a matching that covers all its columns and covers only active rows.

The current matching is thus not a maximum matching on columns $C-\{a_s\}+\{a_c\}$ and the active rows. Therefore, by Lemma 2, there is an alternating path from some uncovered active row to some uncovered column. The only uncovered column in $C-\{a_s\}+\{a_c\}$ is the inactive column a_c. □

Finding an alternating path. The algorithm maintains a queue of uncovered active rows. At each iteration, it takes a row from the head of the queue and searches for an alternating path to an inactive column. If no such path exists, it proceeds to the next row on the queue. When it chooses a new column, it adds any newly active rows to the tail of the queue.

There are two versions of the search for an alternating path from a particular uncovered active row. The "DFS matching" version performs a depth-first search through alternating paths from the row, visiting every inactive column that can be reached by such a path. It chooses the cheapest of those columns according to the cost criterion of Section 5.2. The "greedy matching" first looks for an inactive column that can cover the uncovered active row, and chooses the cheapest such column if there is one. If there is no such column, it performs a depth-first search. Thus it first tries to find an alternating path of length one, and spends the time to search all alternating paths only if that fails.

The greedy algorithm is based on Duff's code MC21A for finding a nonzero diagonal of a matrix [3]. MC21A finds a matching by repeatedly finding alternating paths, and the greedy heuristic speeds up MC21A very significantly in practice. We expected the DFS version to find sparser null bases than the greedy version, but to take longer. However, in the experiments we report in Section 6, the DFS version was usually better than the greedy version in running time and storage as well as in sparsity of the null basis. Presumably this is because it is more successful in keeping the active submatrix small; the time spent doing depth-first searches is saved in updating the QR factorization.

6. Experimental results. We experimented with the four algorithms described in Section 5: turnback, cheapest column next, greedy matching, and DFS matching. Table 1 describes nine sample problems, from various sources, that we used for testing.

Our code gives the option of preordering the columns of A before beginning the null basis computation. Preordering may be necessary to keep the initial QR factorization, which is used to find start columns, sparse; for details see [7]. We found little overall correlation between

preordering method and null basis density, though some problems did show a marked preference for one ordering or another. For the results in Tables 2 through 5 we used, for each algorithm and each problem, the preordering that gave the sparsest null basis for that algorithm on that problem. To support our contention that this compares the algorithms fairly, Table 6 gives results with no preordering at all.

For all but one of the problems we tried three orderings: the original order in which the matrix was presented, reverse Cuthill-McKee, and nested dissection. (The last two were applied to the structure of A^TA.) One matrix, WHEEL, was presented in four different orders; we tried all four, for a total of six in all. It should be noted that for most of the problems the original ordering had already been carefully chosen to reflect certain structural characteristics. In general, one of the automated orderings would be necessary in order to make the initial sparse QR factorization feasible.

Table 2 reports the numbers of nonzeros in the null bases found by the various algorithms. The last column normalizes these numbers: It gives the ratio of the density of the particular null basis to that of the sparsest null basis any algorithm could find for the same problem. Thus, for example, the turnback null basis for FRAME3D had 1.43 times as many nonzeros as the DFS matching null basis, which was the sparsest one found.

Table 3 reports running times. The time reported excludes the time to preorder the columns, perform the initial QR factorization, and find start columns. The excluded times do not depend on which of the four algorithms is being used, and they account for much less than 10% of the total in the larger problems. Again, the last column normalizes each time to the fastest time for the same matrix.

Table 4 reports the maximum size of the active submatrix during the computation. The active submatrix dominates the storage requirements of all four algorithms. Its QR factorization is stored in a dense rectangular array, using the lower triangle to store the Householder transformations whose product is Q. The size reported is the product of the number of active rows and active columns. The last column normalizes the results.

Table 5 gives, for each algorithm, the average over all nine problems of the normalized basis density, running time, and submatrix size.

As we mentioned above, for each problem and algorithm we chose the preordering that gave the sparsest null basis. Table 6 gives the same figures as Table 5, with no preordering; that is, it gives the results when each problem is solved in the column order it was first presented in.

Our experimental code is written in Fortran 77 and was run on a lightly loaded Vax 780 (with floating point accelerator), under Berkeley 4.2 Unix. While we coded it carefully, we did not go to extraordinary lengths to minimize runtime. We have not compared our code to any other implementation of turnback. Our running times were reproducible to within about 15%.

7. Conclusions. Finding a sparse null basis is a problem that is partly combinatorial, partly numerical. We have experimented with algorithms that use the combinatorial structure of the matrix to guide a search for sparse null vectors, but use numerical computation to decide linear dependence. They range in combinatorial sophistication from turnback (which uses none of the structure of the matrix), through cheapest-column-next (which uses the nonzero counts of the rows and columns), to the depth-first search matching algorithm (which uses matchings and alternating paths in the bipartite graph of the matrix).

The results in Section 6 are somewhat mixed, but we can draw some rough conclusions. The DFS matching algorithm looks promising. It has a small but consistent advantage in sparsity; indeed, for only one problem (PLANE) did it fail to come within 1% of the sparsest basis we could find. On the other hand, all the algorithms found pretty good null bases; the worst basis of the four was rarely more than 25% denser than the best. (This is counting only actual nonzeros; a band-oriented approach like the original turnback algorithms might also require storage of many zero entries in the null basis.) The differences in runtime and storage were greater: The matching methods usually ran 2 to 5 times as fast as the non-matching methods, and used correspondingly

smaller active submatrices. There was wide variation: DFS matching was never slower than turnback, but the ratio between them ranged from 1.4 to 22.

Cheapest-column-next generally did better than turnback but worse than DFS matching. On the whole, we conclude that more combinatorial sophistication seems to help, both in sparsity of the null basis and in effort to find it.

All these algorithms are limited by the storage needed for the active submatrix. The QR factorization of the active submatrix is quite dense. We know of no way to avoid this while adding columns to the submatrix in unpredictable order. The other storage bottleneck is the initial QR factorization of A. Here we use Heath's technique of withholding any dense rows if necessary [7]. This approach may lead to a few extra "spurious" start columns, but these are detected correctly by the subsequent null vector algorithm and do not affect the ultimate null basis (although they may incur extra computation). We do not have detailed statistics, but when the program ran out of space it was always because of the active submatrix rather than the initial QR factorization.

We made some tests of the numerical quality of the computed null basis B. We estimated $\|AB\| / \|A\| \|B\|$ to see how nearly orthogonal A and B were, and the answer was always near machine epsilon. We estimated the condition number of B for six of the problems (all but MIXED1, ADLITTLE, and SHARE1B). The answer was almost always reasonably small, but on PLANE and WHEEL there were just a few bases with conditions as high as 10^8. The ill-conditioned bases do not seem to correlate with choice of algorithm or choice of preordering. We think these results on condition number are acceptable--at least, for every problem the majority of the algorithms and preorderings produced well-conditioned bases--but we don't know how to guarantee good conditioning. Coleman and Pothen [2] gave an algorithm for finding an orthogonal basis for the null space, but they also showed that the sparsest orthogonal null basis may be very much denser than an arbitrary null basis. How to trade off sparsity for conditioning is an interesting open question.

As mentioned above, some of the theory behind the matching methods we used comes from Alex Pothen's thesis [2]. Pothen is experimenting with the null basis algorithms from his thesis, but we do not yet have any comparisons with our algorithms. Pothen [12] has also recently suggested an interesting class of heuristics for null basis problems from structural analysis, based on the structure of the object being analyzed in addition to the structure of the matrix.

Appendix. Bipartite graphs, matchings, and rank. Let A be a matrix. The *bipartite graph of* A is the graph whose vertices are the rows of A and the columns of A, with an edge between a row vertex and a column vertex if and only if the corresponding entry of A is nonzero. We do not distinguish between a row of A and a row vertex of the graph of A. Informally, we do not distinguish between A and its graph.

A *matching* on A is a set of edges, no two of which have a common endpoint. (Equivalently, it is a set of nonzeros, no two of which are in the same row or column.) A vertex is *covered* by a matching if it is the endpoint of some matching edge, and *uncovered* otherwise. A *maximum matching* is a matching such that no matching on A has more edges.

A *path* in A is a sequence of distinct vertices $v_0,v_1,\ldots,v_k$ such that $\{v_{i-1},v_i\}$ is an edge for $1 \le i \le k$. The *length* of the path is k. If M is a matching on A, an *alternating path* (with respect to M) is a path whose edges are alternately in M and not in M. If $P=v_0,\ldots,v_k$ is an alternating path from an uncovered vertex v_0 to an uncovered vertex v_k, we can modify the matching by removing edges $\{v_1,v_2\},\{v_3,v_4\},\ldots,\{v_{k-2},v_{k-1}\}$ and adding edges $\{v_0,v_1\},\{v_2,v_3\},\ldots,\{v_{k-1},v_k\}$. This is called *alternating along the path* P. It increases the size of the matching by one edge, covers v_0 and v_k, and leaves all previously covered vertices covered.

For proofs of the following lemmas and more background on bipartite matching, see Lawler [10] or Papadimitriou and Steiglitz [11].

Lemma 1 (Hall's Theorem). *Matrix A has a matching that covers every column if and only if every set of columns of A intersects a set of rows of A that is at least as large.*

Lemma 2. *If M is a matching on A whose size is not maximum, then there is an alternating path from some uncovered row of A to some uncovered column of A.*

Lemma 3. *Every matrix has a matching that is at least as large as its numerical rank.*

As a consequence of Lemma 3, if the columns of A are linearly independent, then there is a matching that covers every column. The combinatorial notion of maximum matching size corresponds closely to the numerical notion of rank. It can be shown that if we fix the nonzero structure of A and assign values to those nonzeros at random, then with probability 1 the rank is equal to the maximum matching size.

References

[1] T. F. Coleman, *Large Sparse Numerical Optimization*, Springer-Verlag, New York, 1984.

[2] T. F. Coleman and A. Pothen, *The sparse null space basis problem*, Tech. Rept. 84-598, Dept. of Computer Science, Cornell University, Ithaca, NY, July 1984. To appear in SIAM J. Alg. Disc. Methods.

[3] I. S. Duff, *On algorithms for obtaining a maximum transversal*, ACM Trans. Math. Software, 7 (1981), pp. 315-330.

[4] A. George and J. W. H. Liu, *Computer Solution of Large Sparse Positive Definite Systems*, Prentice-Hall, Englewood Cliffs, NJ, 1981.

[5] P. E. Gill, W. Murray, and M. H. Wright, *Practical Optimization*, Academic Press, New York, 1981.

[6] G. H. Golub and C. F. Van Loan, *Matrix Computations*, Johns Hopkins University Press, Baltimore, 1983.

[7] M. T. Heath, *Some extensions of an algorithm for sparse linear least squares problems*, SIAM J. Sci. Stat. Comput., 3 (1982), pp. 223-237.

[8] M. T. Heath, R. J. Plemmons, and R. C. Ward, *Sparse orthogonal schemes for structural optimization using the force method*, SIAM J. Sci. Stat. Comput., 5 (1984), pp. 514-532.

[9] I. Kaneko, M. Lawo, and G. Thierauf, *On computational procedures for the force method*, Int. J. Numer. Meth. Engrg., 18 (1982), pp. 1469-1495.

[10] E. L. Lawler, *Combinatorial Optimization: Networks and Matroids*, Holt, Rinehart and Winston, New York, 1976.

[11] C. H. Papadimitriou and K. Steiglitz, *Combinatorial Optimization: Algorithms and Complexity*, Prentice-Hall, Englewood Cliffs, NJ, 1982.

[12] A. Pothen, personal communication, August 1985.

[13] A. Topcu, *A contribution to the systematic analysis of finite element structures using the force method* (in German), Doctoral Thesis, University of Essen, Germany, 1979.

	Rows	Columns	Nonzeros
FRAME2D	27	45	93
PLANE	40	80	168
ADLITTLE	57	97	465
PLATE	59	144	364
FRAME3D	72	144	304
WHEEL	96	120	420
WRENCH	112	216	490
SHARE1B	118	225	1182
MIXED1	171	320	906

Table 1. Description of test problems. ADLITTLE, SHARE1B, and MIXED1 are linear programming bases. The remaining problems are from structural analysis.

Problem	Algorithm	Best Preorder	Nonzeros in Null Basis	Normalized Density
FRAME2D	Turnback	none	76	1.00
	Cheapest Column	none	76	1.00
	Greedy Matching	none	87	1.14
	DFS Matching	none	76	1.00
PLANE	Turnback	none	166	1.00
	Cheapest Column	none	166	1.00
	Greedy Matching	RCM	183	1.10
	DFS Matching	none	177	1.07
ADLITTLE	Turnback	RCM	391	1.07
	Cheapest Column	ND	385	1.05
	Greedy Matching	RCM	387	1.05
	DFS Matching	ND	367	1.00
PLATE	Turnback	RCM	326	1.05
	Cheapest Column	RCM	310	1.00
	Greedy Matching	RCM	313	1.01
	DFS Matching	RCM	311	1.00
FRAME3D	Turnback	none	452	1.43
	Cheapest Column	none	338	1.07
	Greedy Matching	none	369	1.16
	DFS Matching	none	317	1.00
WHEEL	Turnback	order 3	503	1.03
	Cheapest Column	order 2	516	1.06
	Greedy Matching	RCM	625	1.28
	DFS Matching	order 3	488	1.00
WRENCH	Turnback	none	544	1.05
	Cheapest Column	none	549	1.06
	Greedy Matching	none	590	1.14
	DFS Matching	none	518	1.00
SHARE1B	Turnback	none	1531	1.12
	Cheapest Column	ND	1567	1.15
	Greedy Matching	RCM	1604	1.18
	DFS Matching	RCM	1363	1.00
MIXED1	Turnback	none	1518	1.38
	Cheapest Column	RCM	1323	1.20
	Greedy Matching	none	1161	1.05
	DFS Matching	none	1101	1.00

Table 2. Density of null basis.

Problem	Algorithm	Best Preorder	Running Time (Seconds)	Normalized Time
FRAME2D	Turnback	none	2.30	3.65
	Cheapest Column	none	1.12	1.78
	Greedy Matching	none	0.72	1.14
	DFS Matching	none	0.63	1.00
PLANE	Turnback	none	7.05	2.04
	Cheapest Column	none	5.25	1.52
	Greedy Matching	RCM	3.45	1.00
	DFS Matching	none	4.88	1.41
ADLITTLE	Turnback	RCM	28.02	3.19
	Cheapest Column	ND	17.47	1.99
	Greedy Matching	RCM	8.78	1.00
	DFS Matching	ND	10.50	1.20
PLATE	Turnback	RCM	13.02	3.28
	Cheapest Column	RCM	6.65	1.68
	Greedy Matching	RCM	3.97	1.00
	DFS Matching	RCM	4.12	1.04
FRAME3D	Turnback	none	66.12	22.41
	Cheapest Column	none	16.10	5.46
	Greedy Matching	none	2.95	1.00
	DFS Matching	none	2.95	1.00
WHEEL	Turnback	order 3	17.32	1.66
	Cheapest Column	order 2	83.35	7.98
	Greedy Matching	RCM	16.83	1.61
	DFS Matching	order 3	10.45	1.00
WRENCH	Turnback	none	58.38	2.16
	Cheapest Column	none	57.18	2.12
	Greedy Matching	none	30.52	1.13
	DFS Matching	none	26.98	1.00
SHARE1B	Turnback	none	773.15	7.85
	Cheapest Column	ND	365.90	3.72
	Greedy Matching	RCM	98.45	1.00
	DFS Matching	RCM	103.68	1.05
MIXED1	Turnback	none	288.87	4.75
	Cheapest Column	RCM	584.68	9.62
	Greedy Matching	none	70.73	1.16
	DFS Matching	none	60.80	1.00

Table 3. Time to find null basis (excluding preordering and initial QR factorization).

Problem	Algorithm	Best Preorder	Size of Active Submatrix	Normalized Size
FRAME2D	Turnback	none	288	4.00
	Cheapest Column	none	182	2.53
	Greedy Matching	none	110	1.53
	DFS Matching	none	72	1.00
PLANE	Turnback	none	506	1.00
	Cheapest Column	none	506	1.00
	Greedy Matching	RCM	600	1.19
	DFS Matching	none	650	1.28
ADLITTLE	Turnback	RCM	2550	2.34
	Cheapest Column	ND	1680	1.54
	Greedy Matching	RCM	1089	1.00
	DFS Matching	ND	1680	1.54
PLATE	Turnback	RCM	812	4.46
	Cheapest Column	RCM	182	1.00
	Greedy Matching	RCM	210	1.15
	DFS Matching	RCM	182	1.00
FRAME3D	Turnback	none	2064	12.29
	Cheapest Column	none	1089	6.48
	Greedy Matching	none	288	1.71
	DFS Matching	none	168	1.00
WHEEL	Turnback	order 3	1560	2.06
	Cheapest Column	order 2	9312	12.32
	Greedy Matching	RCM	2256	2.98
	DFS Matching	order 3	756	1.00
WRENCH	Turnback	none	5112	1.60
	Cheapest Column	none	3782	1.18
	Greedy Matching	none	3192	1.00
	DFS Matching	none	3782	1.18
SHARE1B	Turnback	none	13570	1.66
	Cheapest Column	ND	9310	1.14
	Greedy Matching	RCM	8742	1.07
	DFS Matching	RCM	8160	1.00
MIXED1	Turnback	none	10506	2.97
	Cheapest Column	RCM	15500	4.37
	Greedy Matching	none	4356	1.23
	DFS Matching	none	3540	1.00

Table 4. Maximum size of active submatrix during null basis computation.

Algorithm	Nonzeros in Null Basis	Running Time	Size of Active Submatrix
Turnback	1.13	5.67	3.60
Cheapest Column	1.07	3.99	3.51
Greedy Matching	1.12	1.12	1.43
DFS Matching	1.01	1.08	1.11

Table 5. Normalized performance measures, averaged over all nine problems.

Algorithm	Nonzeros in Null Basis	Running Time	Size of Active Submatrix
Turnback	1.26	6.70	4.25
Cheapest Column	1.10	4.61	3.36
Greedy Matching	1.17	1.24	1.64
DFS Matching	1.01	1.10	1.05

Table 6. Normalized performance measures, averaged over all nine problems (no preordering).