1990

The Linear Least Squares Problem of Bundle Adjustment

Joseph Walker Woodard
University of North Florida

THE LINEAR LEAST SQUARES PROBLEM OF BUNDLE ADJUSTMENT

by

Joseph Walker Woodard

A thesis submitted to the Department of Mathematics and Statistics in partial fulfillment of the requirements for the degree of Master of Arts in Mathematical Sciences

UNIVERSITY OF NORTH FLORIDA

COLLEGE OF ARTS AND SCIENCES

AUGUST, 1990

The thesis of Joseph Walker Woodard is approved:

(date)

Signature Deleted

_____ 8/1/90

Signature Deleted

_____ 8/1/90

Signature Deleted

_____ 8/1/90

Committee Chairperson

Accepted for the Department:

Signature Deleted

_____ 8/1/90

Chairperson

Accepted for the College of Arts and Sciences:

Signature Deleted

_____ 8/3/90

Dean

Accepted for the University:

Signature Deleted

_____ 8-7-90

Vice-President for Academic Affairs

Acknowledgements

This work would not have been possible without the support and encouragement of Palmer Kinser, Director of the Wetland Mapping Project, John Griffith, Director of the Data Systems Division, and Adel Boules, my Major Professor. I also wish to express my gratitude to Nick Eckhardt, who typed this document and assisted with many of the figures, and to Wayne King, for his help and support in ways too numerous to mention.

Table of Contents

List of Figures

Abstract

A method is described for finding the least squares solution of the overdetermined linear system that arises in the photogrammetric problem of bundle adjustment of aerial photographs. Because of the sparse, blocked structure of the coefficient matrix of the linear system, the proposed method is based on sparse QR factorization using Givens rotations. A reordering of the rows and columns of the matrix greatly reduces the fill-in during the factorization. Rules which predict the fill-in for this ordering are proven based upon the block structure of the matrix. These rules eliminate the need for the usual symbolic factorization in most cases. A subroutine library that implements the proposed method is listed. Timings and populations of a range of test problems are given.

Section 1 - Introduction

This paper describes a method for finding the least squares solution of a large, sparse, overdetermined system of linear equations that arise while performing bundle adjustment on a set of aerial photographs. This method takes advantage of the sparsity and block structure of the system of equations to reduce the computer time and memory requirements for the solution. The method stores and calculates only with those elements of the matrix that are potentially non-zero at some point during the calculations. The block structure of the matrix allows the locations of those non-zero elements that arise in the matrix calculations ("fill-in elements") to be predicted *a priori*. The data structure used during the solution provides storage locations for these fill-in elements as well as the non-zero elements of the original matrix. The ordering of the rows of the matrix is carefully chosen to minimize the number of these fill-in elements, reducing both the storage requirements and the number of calculations to be performed.

Photogrammetry is the science of obtaining reliable measurements from photographs or other imagery of the real world. Bundle adjustment is a mathematical technique developed by photogrammetrists for accurately projecting

information from multiple aerial photographs onto an existing map. It involves setting up a large system of nonlinear equations and solving this system iteratively using Newton's method. The large, sparse, overdetermined system of linear equations that is the subject of this paper arises as the linearized system of equations that must be solved at each iteration of the Newton's method solution.

Aerial photographs, measurements made on them, and bundle adjustment are described in Section 2. Particular attention is given to certain measured points called control points and tie points that are used to locate the photographs in space. The interrelationships of the photos, control points, and tie points give a block structure to the system of linear equations to be solved. These interrelationships and the resulting sparse block matrix structure are explored in Section 3.

The least squares solution of an overdetermined linear system is reviewed briefly in Section 4. The normal equations and QR factorization approaches to solving this linear least squares problem (LLSP) are described. QR factorization using Givens rotations is the technique used to solve the linear system in this paper. Section 5 describes Givens rotations and explains their beneficial properties in preserving the sparsity of a matrix during factorization. The subroutine library GIVENS2 that implements these ideas in FORTRAN is also described in

Section 5 with the data structures that allow these subroutines to store and calculate only with the non-zero entries of the matrix.

Section 6 applies the properties of Givens rotations to factorization of the bundle adjustment matrix to predict where fill-in will occur and prescribe a row ordering that will reduce this fill-in. Section 7 contains some concluding remarks and timings of test runs of the software. The method developed in this paper is briefly compared with other methods from the recent literature.

Section 2 - Bundle Adjustment

Figure 1 shows an aerial photograph that is typical of those taken for many natural resource mapping purposes such as wetland mapping. This photo is a "vertical" photograph, i.e., one whose axis is intended to be vertical and is usually not tilted more than three degrees. Figure 2 shows the same area in an oblique photograph. Features of interest are delineated on such a photo (or on a sheet of mylar overlaying the photo) by a skilled photo interpreter who examines overlapping pairs of photos on a stereoscopic viewing instrument. The delineated features are then digitized into a computer as a series of x-y coordinates using a digitizing table. These measurements are in an essentially arbitrary two-dimensioned coordinate system. The mathematical challenge of photogrammetry lies in projecting these measurements accurately onto a map.

The position of a feature on a map may be specified in terms of its State Plane coordinates and its elevation above sea level. State Plane coordinates are a set of cartesian coordinates that are defined for a region of a state that is small enough that the error in the location of a given point due to the curvature of the earth is less than one part in ten thousand. Florida, for example, is divided

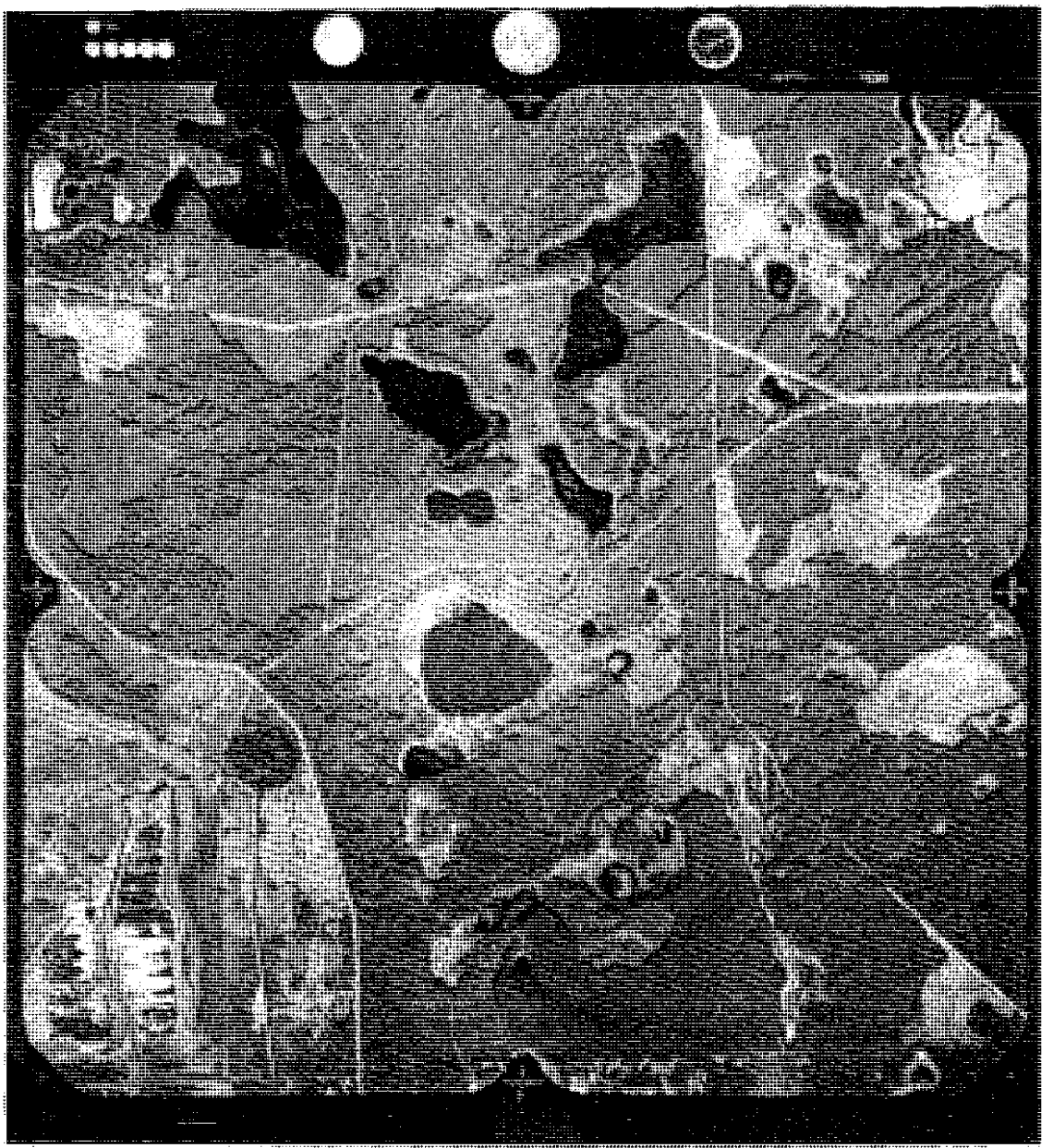


Figure 1 - Vertical Aerial Photograph

Figure 2 - Oblique Aerial Photograph -
Includes Area of Figure 1

into three State Plane zones, separated along county boundaries. The state plane coordinates and elevation above sea level provide a three-dimensional, real-world cartesian coordinate system for such purposes as resource mapping or specifying the location of an aircraft at the moment an aerial photograph was taken.

Each aerial photograph includes a set of fiducial marks that allow points in the photo to be referenced to a set of axes fixed in the camera that took the photo. In figures 1 and 2 these fiducial marks are the small crosses on the points that project into the center of each edge of the photo. These fiducial marks define a set of x-y "photo coordinates" with their origin at the center of the photo and coordinate axes passing through the fiducial marks as in Figure 3. The origin of the photo coordinate system represents the principal point, the point at the center of the photo where the axis of the camera intersects the focal plane. The focal point of the camera lies on the camera axis a precisely known distance above the principal point.

The fiducial marks allow measurements made on an aerial photo to be transformed to the photo coordinate system, which is fixed with respect to the focal point, axis, and focal plane of the camera. The equations of perspective projection then allow measured points on the photo to be projected onto a map based on six coordinates that fix the location and orientation of the camera at the moment the

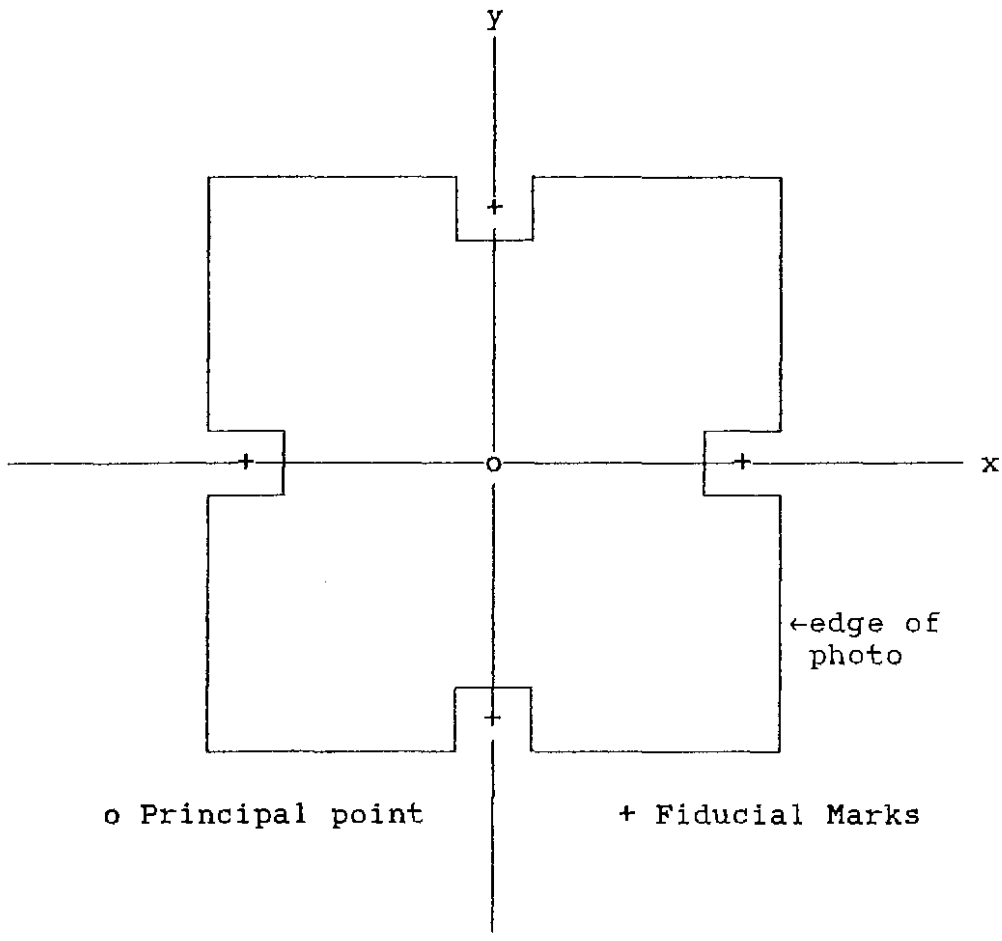
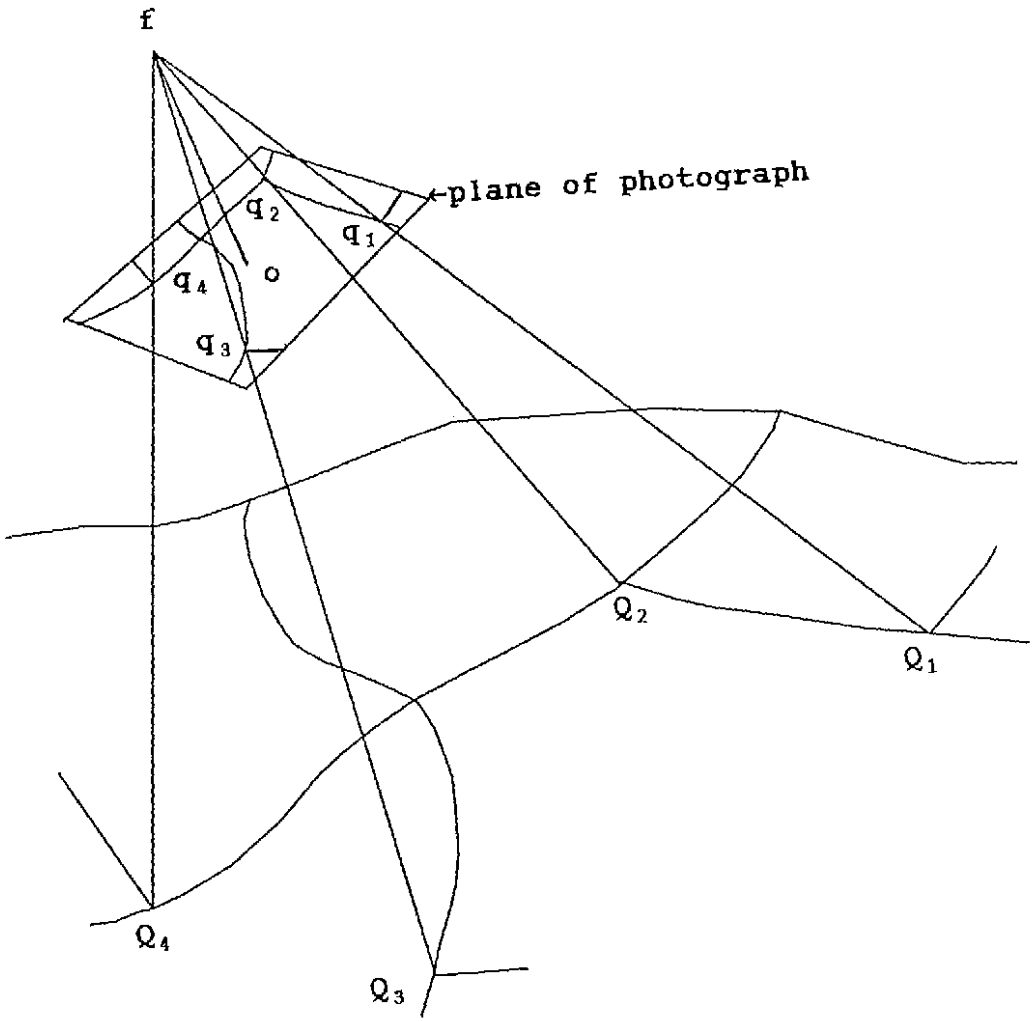


Figure 3 Diagram of Fiducial Marks and Photo Coordinates

photograph was taken. These six coordinates are called the "elements of exterior orientation," or "photo elements," and consist of the three real world x-y-z coordinates of the focal point of the camera and three angles that specify the orientation of the camera axis with respect to the real-world coordinate system. The perspective equations that project from photo coordinates to real coordinates are called the equations of collinearity. There are two collinearity equations for each photo point projected. Their derivation is based on the fact that the image point and the real-world object point are collinear with the focal point. They are developed, for example, in Wolf, 1983, Appendix C. They are, of course, nonlinear equations.

The collinearity equations can be used to project photo data onto a map provided the six photo elements are known for the photo. These unknown elements are found using several points on the photo whose locations are accurately known on both the photo and the existing map. These points are called control points, and are usually road intersections that show clearly in the photo and are plotted accurately on the map. Figure 4 shows the geometry of control points. The system of collinearity equations for the control points is formed and solved for the unknown elements of exterior orientation. This method of finding the photo elements is called "space resection by collinearity."



f - focal point
o - principal point
Q - object on ground
q - image on photo

Figure 4 Diagram of Projection of Photo Points onto Map

Since there are six unknown elements of exterior orientation for each photo, at least three control points are required for each photo, giving a system of six nonlinear equations in six unknowns. Usually four or more control points are used, resulting in an overdetermined system of equations in which inaccurate control points can be detected by the increased residuals. As the equations are nonlinear, they are solved iteratively using Newton's method. The linearized system of equations that must be solved at each iteration of the Newton's method solution is solved in the least squares sense as explained in Section 4. See Wolf, 1983, or Burnside, 1985, for a derivation of the linearized equations.

An aerial photography mission usually involves several closely-spaced, parallel flight lines with photos taken at short distances along each flight line. This produces a large block of overlapping photos. Each photo overlaps its neighbor by sixty percent or more. This overlap allows stereoscopic viewing of adjacent photos for greatly enhanced accuracy of photo interpretation. (Features are delineated on only one of each stereo pair of photographs - usually every other photo along the flight path.) The overlap of adjacent photos also improves the geometric control of the projection process by allowing tie points. Tie points are points whose locations are known accurately on two or more overlapping photos but not on the existing map. They allow

the computer program that finds the unknown elements of exterior orientation to bring the adjacent photos into better alignment with one another by minimizing the displacement of tie points on adjacent photos.

The generalization of space resection by collinearity to multiple photos using tie points is called bundle adjustment. It is the process by which the computer can bring multiple photos into optimal alignment with the existing map (using control points) and with one another (using tie points). The unknown x and y map coordinates for each tie point are additional unknowns of the "bundled" problem. Each tie point results in two additional collinearity equations for each photo in which it appears, the same as for control points. Figure 5 shows a small bundle of four photos with a typical distribution of control and tie points. This bundle of four photos and its coefficient matrix will be used as an example throughout this paper.

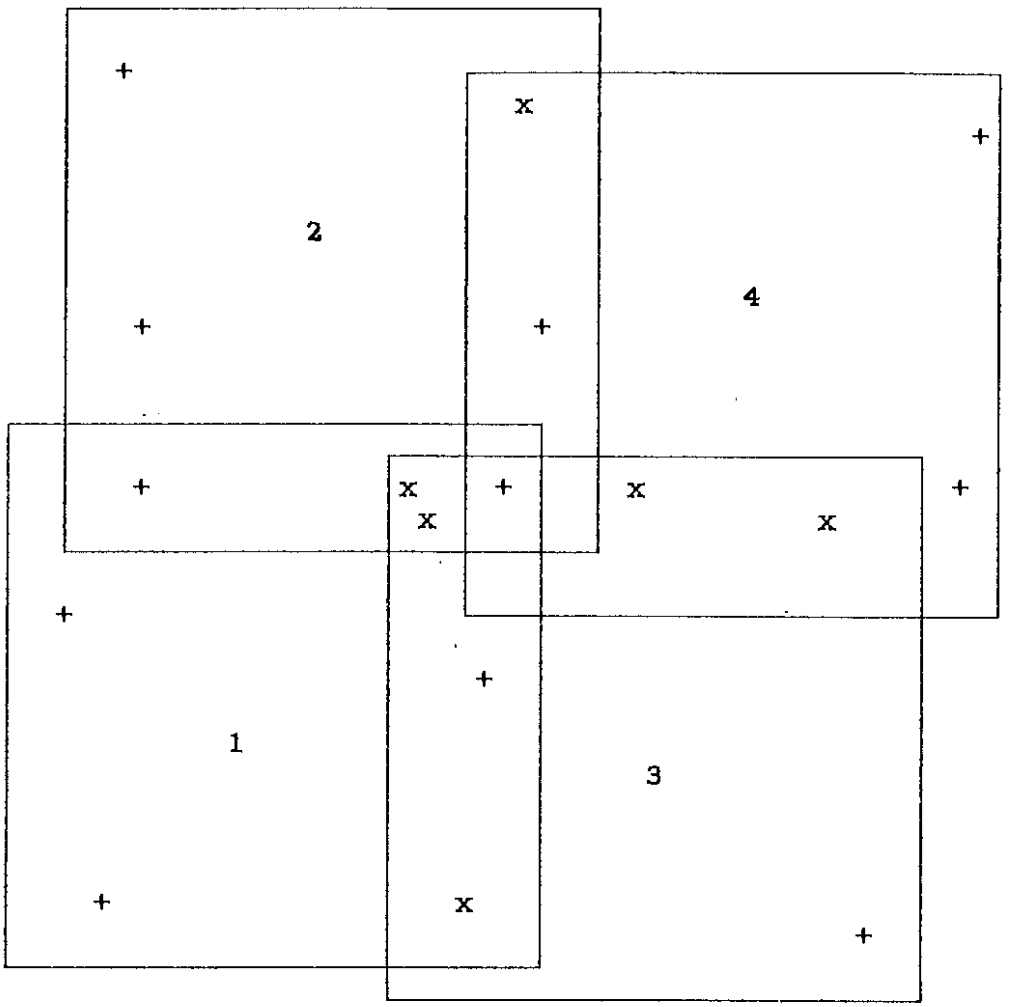


Figure 5 Bundle of Four Photographs Showing Control, Tie Points

Section 3 - The Structure of the Coefficient Matrix

As explained in the previous section, the unknown photo elements and tie point coordinates for a bundle of photos are found by solving a system of nonlinear equations (Wolf, 1983; Burnside, 1985). This system has two nonlinearity equations for each occurrence of a tie point or control point in a photo. This system of nonlinear equations is solved iteratively using Newton's method. At each iteration of the Newton's method solution, a system of linear ("linearized") equations must be solved. The solution to this linear system is the vector of corrections to the solution found during the previous iteration.

This linearized system will in general have different values for the elements of its coefficient matrix (and right hand side vector) at each iteration, but the zero-nonzero structure of the coefficient matrix will remain the same through all iterations. The interrelationships of the photos, control points, and tie points cause the coefficient matrix to take on a regular block structure. For large bundles of photos, the coefficient matrix is quite sparse, that is, it has a very low percentage of nonzero elements. For example, for a bundle of thirty three photos, the matrix has only about 2½% of its elements nonzero.

The sparsity of the coefficient matrix is the feature that, more than any other, determines the choices made in designing the software to solve the bundle adjustment problem. This is because of the great potential for speeding up the solution process by using software that stores and calculates only with the small percentage of matrix elements that are nonzero. Not only is sparse software much faster for a given size of matrix, its asymptotic behavior as the size of the matrix increases is much better. Dense QR factorization has asymptotic behavior proportional to the n^3 (Stewart, 1973, p237). While no such simple limit is known for sparse QR factorization, it tends to be not much worse than proportional to n for large problems.

In addition to those elements of the matrix that are originally nonzero, other elements will become nonzero during the process of the calculations. These are called fill-in elements. The number of fill-in elements that occurs depends critically upon the design of the software used and upon the order in which the rows and columns of the coefficient matrix are processed. The total number of original elements plus fill-in elements is called the population of the matrix. For a large bundle, efficient software and a carefully chosen ordering of the rows of the matrix can hold the population to less than six percent. Less efficient software or a poor row ordering can cause so

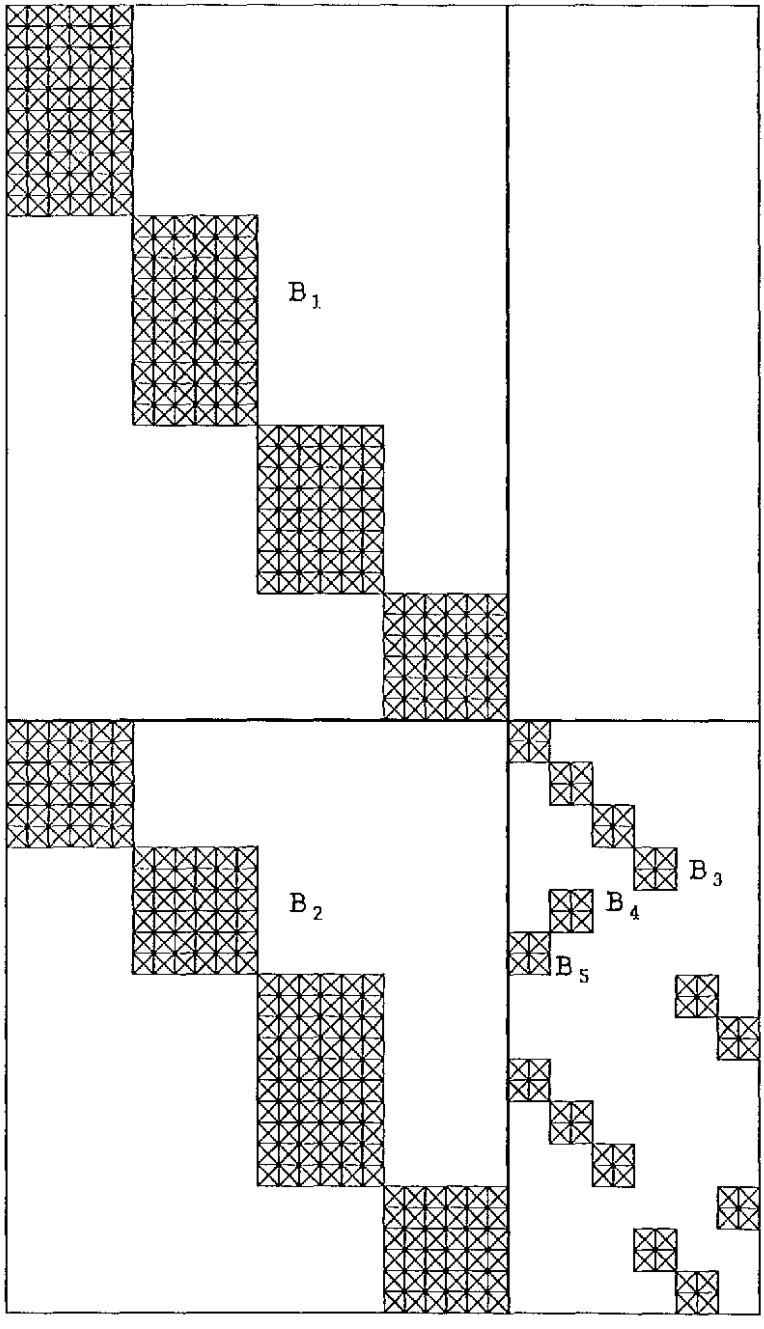
much fill-in as to convert the matrix to a dense matrix - one in which there are too few zero elements to be exploited.

Figure 6 shows the zero-nonzero structure of the coefficient matrix for the bundle of four photos in Figure 5. This matrix will be used as a continuing example in this and the following sections.

The system of linear equations for one iteration has the form:

$$Ax = b$$

where A is the m x n coefficient matrix, x is the n-vector of unknown photo elements and tie point coordinates, and b is the right-hand-side (RHS) n-vector. This system will include two equations for each occurrence of a control point in a photo, and two equations for each occurrence of a tie point in a photo. The control point equations and the tie point equations will have a different structure. In Figure 6 the control point equations are above the horizontal line, the tie point equations are below. The x-vector includes two different types of variables, photo elements and tie point coordinates, whose coefficients in the equations have different structures. In Figure 6 the coefficients of the photo elements are to the left of the vertical line, the coefficients of the tie point coordinates are to the right.



⊠

Non-zero Element

Figure 6 - Initial Pattern of the Coefficient Matrix for the Bundle of Four Photos

Each photo must include a minimum of four control/tie points unless it includes only tie points, in which case it must have at least five. So each photo generates a minimum of eight or ten lines of matrix A. In addition there is a minimum requirement of at least four control points per bundle. For example our bundle of four photos could have been done using only one control point for each outside corner of the entire bundle. This is one reason for requiring more points when only tie points are used.

The collinearity equations express the relationship between the photo coordinates of a point and the real-world coordinates of that point. The relationship depends on the six orientation elements of the photo in which the point appears, but it does not depend on the orientation elements of any other photo. Consequently the photo elements for a particular photo appear with nonzero coefficients only in the equations of points that appear in that particular photo. The photo elements for that photo appear with zero coefficients in all other equations. Thus the coefficient matrix will have a block of nonzero coefficients six elements wide for each photo. The block appears in each pair of rows corresponding to a control or tie point in that photo. In Figure 6, the six wide block for the control points of the second photo in the bundle is labeled B_1. The six wide block for the tie points of the second photo is labelled B_2. In all the other rows the matrix has zero

values in these six columns since the photo elements for the second photo do not enter into the equations for points in any other photo.

For the tie points, the collinearity equations involve not only the six unknown photo elements, but also the two x-y coordinates of the tie point. These two tie point coordinates appear as unknowns in two collinearity equations in each photo in which the tie point appears. (The photo coordinates of the point also appear in the linearity equations for each control point, but they are known values and are included in the constant on the right-hand-side.) Thus each tie point requires two equations that include a two element by two element block of coefficients for the unknown tie point coordinates. These 2x2 tie point blocks appear to the right of the vertical line and below the horizontal line in Figure 6. The three 2x2 blocks for the second photo in the bundle are labelled B_3, B_4, and B_5. Note that each 2x2 block also appears in at least one other photo since each tie point must necessarily appear in at least two photos. These two columns are zero in all other rows that do not involve this tie point.

The vertical line in Figure 6 that separates the photo element blocks from the tie point blocks divides the matrix into two regions that differ not only in the size and shape of their blocks but also in the way these blocks relate to one another. To the left of the line, the photo element

blocks for each photo are all disjoint from one other in the sense that not one of them intrudes into the rows or columns used by the blocks for another photo. To the right of this line each tie point block is guaranteed to have a corresponding block in the same columns of the rows used by at least one other photo. This dichotomy of structure will have a profound effect upon the factorization process in the least squares solution of the system of linearized equations. The fill-in elements that appear when the elements to the left of the vertical line are processed will take on a surprisingly simple and regular block structure. To the right of this line we shall be reduced to proving min-max rules that merely put bounds on the fill-in in each row, without specifying its structure between these limits.

Section 4 - The Linear Least Squares Problem

The system of equations for one iteration of the Newton's method solution has the form:

$$A\,x = b$$

where A is the m x n block structured coefficient matrix described in Section 3, x is the n-vector of unknown photo elements and tie point coordinates, and b is the m-element constant RHS vector. Since m is in general larger than n, this system has more equations than unknowns and there will in general be no vector x that satisfies all the equations. Thus we must seek a best-fit solution in the least squares sense, that is we seek the vector x for which the square of the two norm of the residual:

$$\| A\,x - b \|_2^2$$

is a minimum. That an overdetermined linear system has such a solution and that it is unique when the matrix A is of full rank is shown, for example, in Stewart, 1973, Chapter 5, or Hager, 1988, Chapter 5. These texts (and Ortega, 1987) are good references on the matrix theory described briefly in the following paragraphs.

Various methods exist for finding the least squares solution of an overdetermined linear system. One method that is of great theoretical importance is the use of

Cholesky decomposition on the system of normal equations. The normal equations are formed by multiplying the original system by the transpose of its coefficient matrix:

$$A^TAx = A^Tb$$

This produces an n x n square system of equations that will have a unique solution whenever A is of full rank. This solution is in fact the least squares solution of the original overdetermined system.

The coefficient matrix A^TA of the system of normal equations is square, symmetric, and positive definite, and can be expressed in the form:

$$A^TA = LL^T$$

where L is a unique n x n lower triangular matrix with positive diagonal elements. This factorization can be found using the Cholesky decomposition, a specialized form of Gaussian elimination for symmetric matrices. The upper triangular matrix L^T, which is unique for a fixed ordering of the columns of A, will be called the unique Cholesky factor in this paper.

This factorization allows the overdetermined linear system to be solved by successively solving two triangular systems. We have

$$A^TAx = LL^Tx = A^Tb = b_*$$

or, letting $y = L^{-1}b_*$,

$$L^Tx = L^{-1}b_* = y.$$

The second equality is equivalent to

$$Ly = b.$$

This is a lower triangular system which can be solved by forward substitution. Then the upper triangular system

$$L^T x = y$$

can be solved by backsubstitution.

The normal equations method is unfortunately unstable, resulting in an inaccurate solution vector x unless the coefficient matrix is extremely well conditioned (G.W. Stewart, 1973, pp 225-230). It usually requires auxiliary calculations to detect an ill-conditioned matrix and iteratively refine the solution vector. Nevertheless, it has been used extensively in the sparse matrix setting. See Duff, Erisman, and Reid, 1986 and George and Liu, 1981 for an overview.

The method used to find the least squares solution in this paper involves the QR factorization of the matrix A itself. This method is extremely stable, resulting in an accurate solution vector even when the coefficient matrix is quite ill-conditioned (G.W.Stewart, 1973, p237). This is true regardless of the order of the rows or columns of the matrix. This fact will allow us to reorder the rows and columns of the coefficient matrix to preserve the sparsity of the matrix during factorization. (George and Heath, 1980.) QR factorization involves expressing A as:

$$A = QR$$

where Q is an m x n matrix having orthogonal columns and R is an n x n upper triangular matrix. In fact, except for the possible multiplication of some rows by -1, R is the unique Cholesky factor of A^TA.

The orthogonal matrix Q is not found explicitly in performing the QR factorization of A. Instead, a sequence of orthogonal matrices $Q_1, Q_2, \ldots, Q_p$ such that

$$Q_1^T Q_2^T \ldots Q_p^T = Q$$

is found, so that

$$Q_p Q_{p-1} \ldots Q_1 A = Q^{-1} QR = R$$

This sequence of orthogonal matrices is chosen to introduce zero elements below the diagonal of A, and applied to the coefficient matrix A where it is stored in the computers memory, so that A is gradually replaced by R.

This sequence of orthogonal matrices is applied simultaneously to the RHS vector b to give

$$y = Q_p Q_{p-1} \ldots Q_1 b$$

The Q_i matrices are not stored, but are simply discarded after being applied to both sides of the matrix equation.

The triangular system

$$R x = y$$

that results can be quickly solved by back substitution. The residual vector

$$r = Ax - b$$

can then be calculated provided a copy of the original matrix A has been retained. If only the 2-norm of the residual is needed, it can be found from

$$\|y\|_2 = \|Ax - b\|_2$$

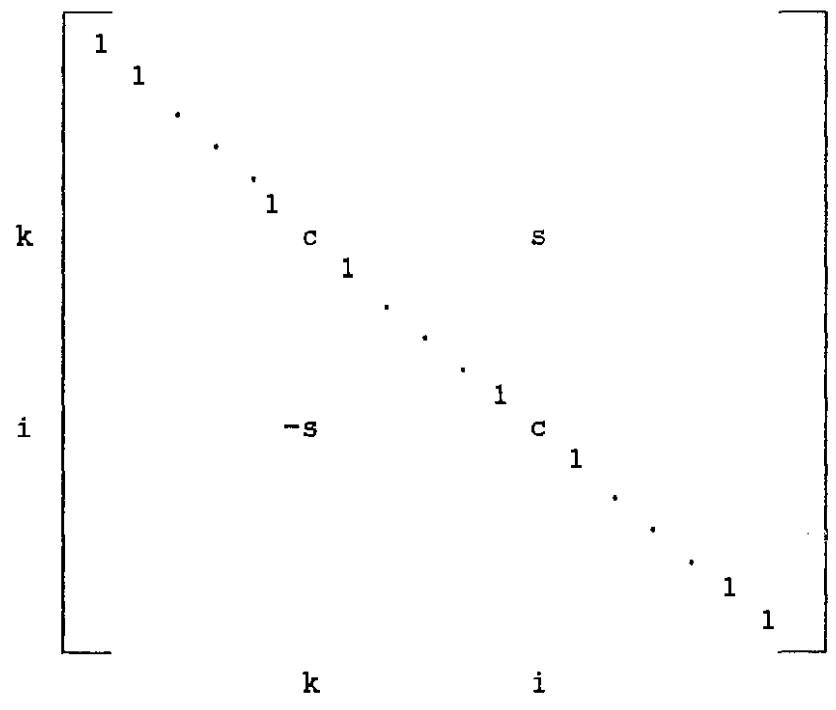
and A need not be stored.

The orthogonal matrices Q_i are chosen to introduce zeroes below the diagonal of the matrix A. Two popular choices for these orthogonal matrices are Householder reflections and Givens rotations. A Householder reflection replaces all the subdiagonal elements of a particular column of A with zeroes at once. A Givens rotation replaces a single subdiagonal element with zero. Householder reflections are inherently much faster than Givens rotations (Björck, 1976, Hager, 1988) and are the appropriate method when the coefficient matrix A is dense. But, for sparse matrices, the Householder reflections produce an unacceptable amount of fill-in (George and Heath, 1980). Givens rotations, on the other hand, are well suited for use with sparse matrices because they can be used to zero out individual elements selectively and they cause only a moderate amount of fill-in while doing so (Duff, 1974; Gill and Murray, 1976). This ability to preserve sparsity more than overcomes the difference in speed for large matrices.

Section 5 - Givens Rotations and the GIVENS2 Library

A Givens rotation is a sparse matrix that differs from the identity matrix by the addition of a matrix of rank 2 and has this form:

$$G_{ik} =$$



where $c^2 + s^2 = 1$ (Hager, 1988). A Givens rotation is an orthogonal matrix which, when it multiplies a vector x, rotates the k^{th} and i^{th} components of x through the angle

$$\theta = \sin^{-1}(s) = \cos^{-1}(c)$$

and leaves the remaining components of x unchanged. A Givens rotation can be chosen to reduce the subdiagonal element a_{ik} of the matrix A to zero by taking $c = a_{kk}/(a_{kk}^2 + a_{ik}^2)^{1/2}$ and $s = a_{ik}/(a_{kk}^2 + a_{ik}^2)^{1/2}$ where a_{kk} is the diagonal element of A in the same column as a_{ik}. The usual scheme for transforming A to upper triangular form using Givens rotations is to work down each column in turn, starting with the first column, using the diagonal element in that column as the pivot element in zeroing out each subdiagonal target element.

Zeroing out all the subdiagonal elements in a single column is referred to as a major step; zeroing out a single subdiagonal element using a single Givens rotation is referred to as a minor step. Other variations are possible. For example, one can zero out all the elements of a single row that lie to the left of the diagonal as the major step, using the diagonal element in a different column as the pivot at each minor step. Or one can do variable row pivoting, i.e. interchange the pivot row with the sparsest row having a leading nonzero before each minor step (Duff, 1974). Incidentally, some authors define the Givens rotation with the submatrix:

$$\begin{bmatrix} c & s \\ s & -c \end{bmatrix}$$

so that it is symmetrical and $G^{-1} = G^T = G$ (Gill and Murray, 1976). There is also a "fast" Givens rotation that doesn't

require taking a square root, but it is much more complicated and not much faster (Hager, 1988; Gentleman, 1973; George and Heath, 1980, p78).

When a Givens rotation G_{ik} multiplies a matrix A it leaves all the rows except the k^{th} and i^{th} rows unchanged. These two rows are replaced by a linear combination of their previous values where the coefficients of the linear combination are c and s. That is:

$$a_{kj} = ca_{kj} - s\ a_{ij}$$

and

$$a_{ij} = sa_{kj} + c\ a_{ij}.$$

So each element in the pivot row is replaced by a linear combination of itself and the corresponding element in the target row; each element in the target row is replaced by a linear combination of itself and the corresponding element in the pivot row. This is the secret of the Givens rotations' ability to preserve sparsity. For at each minor step in the factorization each element of A is replaced with a linear combination of only two elements of A. So if both of these elements are zero, they will remain zero. Since A is sparse, this will be the usual case, and vast areas of A will remain zero.

This also explains why the pivot element is always the diagonal element and all elements to the left of the target element in the two rows involved are zeroed-out before a particular element is selected as the target. With this

convention, there will be no fill-in to the left of the target element in either row and the previously annihilated elements will remain annihilated.

Unfortunately, if either a_{kj} or a_{ij} is nonzero before the multiplication, both will be nonzero after the multiplication and there will be some fill-in. In fact the target row and pivot row both become nonzero wherever either was nonzero. This is called the "local Givens rule" for fill-in (Coleman, Edenbrandt, and Gilbert, 1983). It is usually expressed by saying that the set of subscripts of nonzero elements of the target row and pivot row become the union of their former sets at each minor step.

Any fill-in elements which appear below the diagonal will be zeroed out by some later pivot element before the factorization is complete. For this reason, fill-in elements below the diagonal are sometimes called transient or intermediate fill-in.

The local Givens rule allows the location of fill-in elements to be predicted, but usually only by a very laborious process. Say there was a nonzero element, a_{ij}. When some earlier element in row i is annihilated, a_{ij} will cause fill-in in column j of the pivot row. That fill-in element will then cause fill-in in some other target row, which will cause fill-in in some later pivot row, etc., etc.

The usual method of locating the fill-in is to do a symbolic factorization. This is a process in which one goes

through an entire factorization, keeping track of where every nonzero element appears, but disregarding the actual values of the elements. The object of the symbolic factorization is to set up a data structure with a storage location for every nonzero element a_{ij} that will appear during the factorization. This data structure can then be used to store the matrix elements during the real factorization that follows. Since each iteration uses a coefficient matrix with the same sparsity pattern, this data structure can be used repeatedly, saving time.

It is the fact that the data structure can be used repeatedly that justifies separating the symbolic factorization from the real factorization. The symbolic factorization must set up a large data structure that must necessarily allow new elements to be added at random. It typically takes longer than a single iteration of the real factorization, and doing it only once results in significant time saving.

In this paper, we do even better. We shall show that, with some rare exceptions, most bundle adjustment problems produce a coefficient matrix whose rows can be reordered so that the fill-in can be predicted a priori. This allows the fill-in elements to be added without going through the laborious copy-up, copy-down process of the symbolic factorization. In the following section, we show how that can be done.

The GIVENS2 subroutine library implements in FORTRAN the algorithms described so far in this paper. It includes subroutines for performing the symbolic and real QR factorizations on a sparse matrix using Givens rotations. It also includes subroutines for solving the resulting triangular system by backsubstitution and for calculating the residual vector using a copy of the original matrix. The subroutine SROTG which calculates the constants of the required Givens rotation is a standard LINPACK routine. (Dongarra, Bunch, Moler, and Stewart, 1979.)

Since the data structure used to store the nonzero elements of the coefficient matrix is not a simple array, the GIVENS2 library also provides subroutines for storing, retrieving, and modifying elements of the matrix. Other subroutines efficiently locate the next item in a particular row or column of the matrix.

Two different data structures are used in the GIVENS2 library. The equation building and symbolic factorization phases use a linked list data structure (Duff, Erisman, and Reid, 1986; Schendel, 1989). The linked list is then read into a static sequential list for the real factorization and backsubstitution phases (George and Heath, 1980; Gill and Murray, 1976). The linked list is used again in the calculation of the residual since it retains a copy of the original matrix.

The linked list data structure is used during the symbolic factorization because it allows new elements to be added to the matrix at random. This results in a data structure that includes every nonzero element of the matrix, but stores them in a very disorderly manner in memory. This linked list is then used to set up a sequential list that allows the data to be accessed very efficiently during subsequent calculations. The linked list can accept rows of a matrix in any order, while the sequential list necessarily puts the rows into their proper order.

The linked list data structure consists of the following variables:

W - a real array that contains the list of values of the elements of the matrix A

JW - an integer array that contains the column number J of each entry in W

ISTART - an integer array whose I^{th} entry contains the array subscript in W of the first nonzero element of row i of A

NEXT - an integer array that contains the array subscript in W of the next element of A in row order

LAST - an integer variable that contains the integer subscript in W of the final element in A

For example, suppose for a particular small matrix A, the fifth row of A contained the three nonzero elements

$$a_{5,2}=37.65,\ a_{5,5}=10.21,\ a_{5,9}=0.765$$

these might be stored in the linked list as follows:

W(75)=37.65, JW(75)=2, NEXT(75)=81

W(81)=10.21, JW(81)=5, NEXT(81)=99

W(99)=0.765, JW(9)=9, NEXT(99)=0

ISTART(5)=75, LAST=376

To locate the elements of row 5 of A we would proceed as follows. The fifth element in the array ISTART is 75, which indicates that the first nonzero element of the fifth row of A is stored as W(75), the seventy fifth element of W. The value of W(75) is 37.65 and that of JW(75) is 2, so we know the first nonzero element in this row is $a_{5,2}$=36.75. NEXT(75) is 81 so we know the second nonzero element in the row is stored as W(81). W(81) is 10.21 and JW(81) is 5 so we know that the next nonzero element is $a_{5,5}$=10.21. Similarly, the third element, $a_{5,9}$=0.765, is stored at W(99). Now NEXT(99) is zero so we know there are no further nonzero elements in row 5 of matrix A.

Suppose a fourth nonzero element is to be added to row 5 of this matrix, say

$$a_{5,7}=18.03.$$

The data structure allows us to add the new element at the end of the list and still follow the "thread" of NEXT pointers to reconstruct the row from the beginning when needed. In our example, LAST=376 so we know that 376 elements of W and JW have been used and that the 377th entry is available for use. To install the new element, we set

LAST=LAST+1=377, W(LAST)=18.03 AND JW(LAST)=7. The seventh element in the row must logically follow the fifth entry, which is stored as W(81) and precedes the ninth entry, which is stored as W(99). So to fix the pointers, we must now set NEXT(81)=377, NEXT(LAST)=NEXT(377)=99. The entire row is now stored as:

```
W(75)=37.65,   JW(75)=2,   NEXT(75)=81
W(81)=10.21,   JW(81)=5,   NEXT(81)=377
W(99)=0.765,   JW(9)=9,    NEXT(99)=0
W(377)=18.03,  JW(377)=7,  NEXT(377)=99
  ISTART(5)=75,            LAST=377
```

and the thread of the next pointers can again be followed to recreate the entire row.

Suppose we need to locate a particular element a_{ij} of the matrix A. There is only one way to do it - start at ISTART(I) and follow the thread. The subroutine FINDITEM below illustrates how to do this. (This listing is not a complete subroutine. The dimension statements are not shown in this illustration. See Appendix B for the complete listing.)

```
      SUBROUTINE FINDITEM (I,J,ELEMENT,FOUND,LOC)
C
C     THIS SUBROUTINE LOCATES AN ELEMENT A(I,J) OF A MATRIX A
C     THAT IS STORED IN A ROW-ORIENTED LINKED LIST.
C
      FOUND = .FALSE.
      ELEMENT = 0.0
      IF (LOC .GE. 0) THEN
            L = ISTART(I)
            IF (LOC .GT. 0) L = LOC
            LPREV = 0
 1000       IF (JW(L) .EQ. J) THEN
                  FOUND = .TRUE.
                  ELEMENT = W(L)
                  LOC = L
                  GO TO 9999
            ENDIF
            IF (JW(L) .GT. J) THEN
                  LOC = LPREV
                  GO TO 9999
            ENDIF
            LPREV = L
            IF (NEXT(L) .NE. 0) THEN
                  L = NEXT(L)
                  GO TO 1000
            ENDIF
            LOC = -L
      ENDIF
C
C     RETURN TO CALLER
C
 9999 RETURN
      END
```

Notice that FINDITEM returns the logical variable FOUND to tell the caller whether the needed element existed. It also returns the location (in W and JW) where the element was found as the value of the variable LOC. If the element was not found, LOC is set to the location of the item that would precede it in row order. This is so the item can be quickly added if need be without following the thread again. The following listing of subroutine ADDITEM illustrates this use of LOC. (Again, this is not a complete listing of the subroutine. See Appendix B for the complete listing.) If ADDITEM is called with LOC=0, ADDITEM follows the thread for the row to find the correct location to insert a_{ij}. If a nonzero value of LOC is provided, ADDITEM inserts a_{ij} after that location.

Note that another piece of information is returned in LOC. If the last existing nonzero element in the row has been passed in the search for a_{ij}, LOC is set negative to signal that the end of the row has been reached.

```
      SUBROUTINE ADDITEM (I,J,ELEMENT,LOC)

C
C     THIS SUBROUTINE ADDS AN ELEMENT A(I,J) TO A MATRIX A
C     THAT IS STORED IN A ROW-ORIENTED LINKED LIST.
C
C     START SEARCH AT BEGINNING OF THIS ROW
C     OR LATER IF A LATER LOCATION IS PROVIDED
C
      L = ISTART(I)
      IF (LOC .NE. 0) L = ABS(LOC)
C
C     DONE IF ITEM ALREADY EXISTS
C
 1000 IF (JW(L) .EQ. J) GO TO 9999
C
C     ADD ITEM IF A LATER ELEMENT IN THE ROW IS REACHED
C
      IF (JW(L) .GT. J) THEN
            LAST = LAST + 1
            JW(LAST) = J
            W(LAST) = ELEMENT
            NEXT(LAST) = L
            LOC = LAST
            NEXT(LPREV) = LAST
            GO TO 9999
      ENDIF
C
C     FOLLOW THE THREAD IF THIS IS NOT THE END OF THE ROW
C
      LPREV = L
      IF (NEXT(L) .NE. 0) THEN
            L = NEXT(L)
            GO TO 1000
      ENDIF
C
C     ADD ITEM TO END OF ROW IF END IS REACHED
C
      LAST = LAST + 1
      JW(LAST) = J
      W(LAST) = ELEMENT
      NEXT(LAST) = 0
      NEXT(LPREV) = LAST
      LOC = -LAST
      GO TO 9999
C
C     RETURN TO CALLER
C
 9999 RETURN
      END
```

The sequential list is similar to the linked list, but simpler.

V - a real array that contains the list of values of the elements of A

JV - an integer array that contains the column number J of each entry on V

LOCI - an integer array whose I^{th} entry contains the array subscript in V of the first nonzero element in row i of A

Elements of the matrix A are stored sequentially, in row order, within V. LOCI(I) tells the starting location of the i^{th} row of A, so LOCI(I+1)-1 is the ending location of the i^{th} row. An $(M+1)^{st}$ element of LOCI is provided so this trick will work for the m^{th} or last row of A. No LAST variable is needed.

Notice that the sequential list includes all those matrix elements that appear at some point during the symbolic factorization, with no indication which should be used with what pivot row or target row. This is inferred from the following simple rule. At each minor step during the real factorization, process all those columns which have an entry in both the target and pivot rows. Since the target row and pivot row took on the structure of their union after each minor step of the symbolic factorization, this will include all the elements changed during this minor step. Any pivot row elements or target row elements that

are skipped over in this process appeared later in the symbolic factorization and shouldn't be changed during this minor step.

It may happen that the target row and pivot row have later acquired an entry in the same column, but if so they will be zero initially. So, processing them at this minor step will leave them zero and no error occurs.

The pivot row will generally start at a later column than the target row. So the search for corresponding elements is most efficient if done by scanning the pivot row sequentially from left to right and checking the target row to see if the corresponding element exists. Since the pivot row contains the union of all its target rows, a pivot row will in general go further to the right than the target row. Thus it is important for the software to end the minor step immediately when the end of the target row is reached. This is why the subroutines flag the end of a row by setting LOC negative.

The simplicity of performing QR factorization using Givens rotations can be appreciated by glancing over the subroutine GIVENS in Appendix A. This is a dense matrix implementation of QR factorization using Givens rotations. The listing of the factorization subroutine takes less than a page. The much larger GIVENS2 subroutine library in Appendix B performs the same calculations in the sparse matrix setting. Appendix C is a listing of the subroutine

LEAST that organizes the QR factorization by calling the subroutines of the GIVENS2 library.

The major subroutines of the GIVENS2 library are SYMBOL, which performs the symbolic factorization using the linked list data structure, and FACTOR, which performs the real factorization using the sequential list. Subroutine BACKSOLVE solves the resulting triangular system using backsubstitution. Subroutine RESIDUAL calculates the residual vector using the copy of the original matrix that is retained in the linked list. The library also includes various subroutines to add, locate, or modify matrix elements in each data structure. The functions of these are explained in the numerous comments included in their listings.

Section 6 - Predicting the Fill-in

In this section we look at the fill-in caused by Givens rotations in detail, describing several rules about how this fill-in happens and how it can be minimized and predicted. We then apply these rules to the QR factorization of the bundle adjustment problem. We derive a set of rules for ordering the rows of the matrix so as to minimize the fill-in. We then show how this ordering of the rows allows the fill-in to be predicted a-priori, eliminating the need for the time-consuming symbolic factorization.

The most fundamental rule in performing QR factorization using Givens rotations is to process the most complex rows and columns last (Björck, 1976; Duff, 1974). The pivot and target rows take on the union of one another many times during the symbolic factorization. If fill-in starts too early in the process it will grow explosively before the factorization is complete. So the most complex rows must be held out until the end, keeping the fill-in to a minimum as much as possible.

As for reordering the columns, the order in figure 6 where the photo element blocks are to the left, the tie point blocks to the right, is already a workable solution. The problem of choosing an optimal ordering for the columns

to make the bandwidth of the tie point blocks as narrow as possible is well known to be NP-complete (Duff, Erisman, and Reid, 1986, p127.) Except for putting the tie point blocks to the right, we shall take the order in which the columns occur for granted and look for a row ordering that reduces the fill-in as much as possible for that column order.

Another fundamental observation is that, since any nonzero elements initially present in a pivot row will be reproduced in every target row it processes, it is very helpful to provide pivot rows that are all zero initially. Another, more subtle idea, that takes this a big step further is to provide pivot rows that include some nonzero elements they would otherwise process as target elements. This reduces the population even further by moving some initial nonzero elements into locations where fill-in would inevitably occur anyway.

We have already observed that fill-in never occurs to the left of the target element. This implies that if, at a certain stage of the factorization, row i has its first nonzero subdiagonal element in column j, then during the rest of the factorization no fill-in will ever appear in row i before column j. This is because no earlier pivot element than a_{ij} will be used on row i since it has no nonzero elements before column j to be annihilated, and row j will be zero to the left of a_{ij} when that pivot is used. This places an earliest-column bound on the fill-in in row i.

A similar rule can be proven for the latest-column bound on the fill-in in a row i provided that the pivot row is initially all zero. If the pivot row is initially all zero, then the latest column that can be filled-in in row i is just the maximum of the latest column of any nonzero element in the target rows up to and including row i. This is because a nonzero element could only occur in column j of row i if it was there initially or was in the pivot row when row i was processed. It could only be in the pivot row if it appeared during the processing of some earlier target row, since the pivot row was initially zero.

Our final observation requires some new terminology. It is the statement that Givens rotations preserve row/pivot disjoint submatrices.

Recall that a submatrix of a matrix A is a matrix made up of those elements of A that lie in the intersections of a specified subset of the rows and columns of A. One can think of creating the submatrix by deleting all but certain rows and columns of A. The submatrix is what's left. Two submatrices, A_1 and A_2 are said to be disjoint if they have no element in the same row or column of A. In other words, when deleting rows or columns of A to find A_1, all of A_2 is deleted, and vice versa. We observed in Section 3 that the photo element blocks in the matrix of Figure 6 are disjoint in this sense, while the tie point blocks are not.

Two submatrices A_1 and A_2 of A are row disjoint if they have no element in the same row of A. They might have elements in the same column or columns of A. Two submatrices are pivot disjoint with respect to a certain pivot if that pivot column contains only elements of A_1 or A_2 but not both.

Two submatrices A_1 and A_2 are row/pivot disjoint with respect to a certain pivot a_{kk} if the two submatrices are row disjoint, they are pivot disjoint with respect to a_{kk}, and if a_{kk} can annihilate elements of A_1 (say) then there are no elements of A_2 in the pivot row k (or vice versa). Figure 7 shows two submatrices that are row/pivot disjoint. The original elements of these two matrices have solid outlines, the areas with dashed outlines are regions where fill-in for each might occur. Note that being row/pivot disjoint is more than being pivot disjoint and row disjoint since it requires that the elements of each submatrix not intrude into the pivot row(s) of the other submatrix.

The reason we are interested in row/pivot disjoint submatrices is that if a submatrix is row/pivot disjoint with respect to a certain set of pivot elements from the rest of A, then processing with those pivots will leave it that way. That is, processing elements of a submatrix A_1 will only produce fill-in among the elements of A_1.

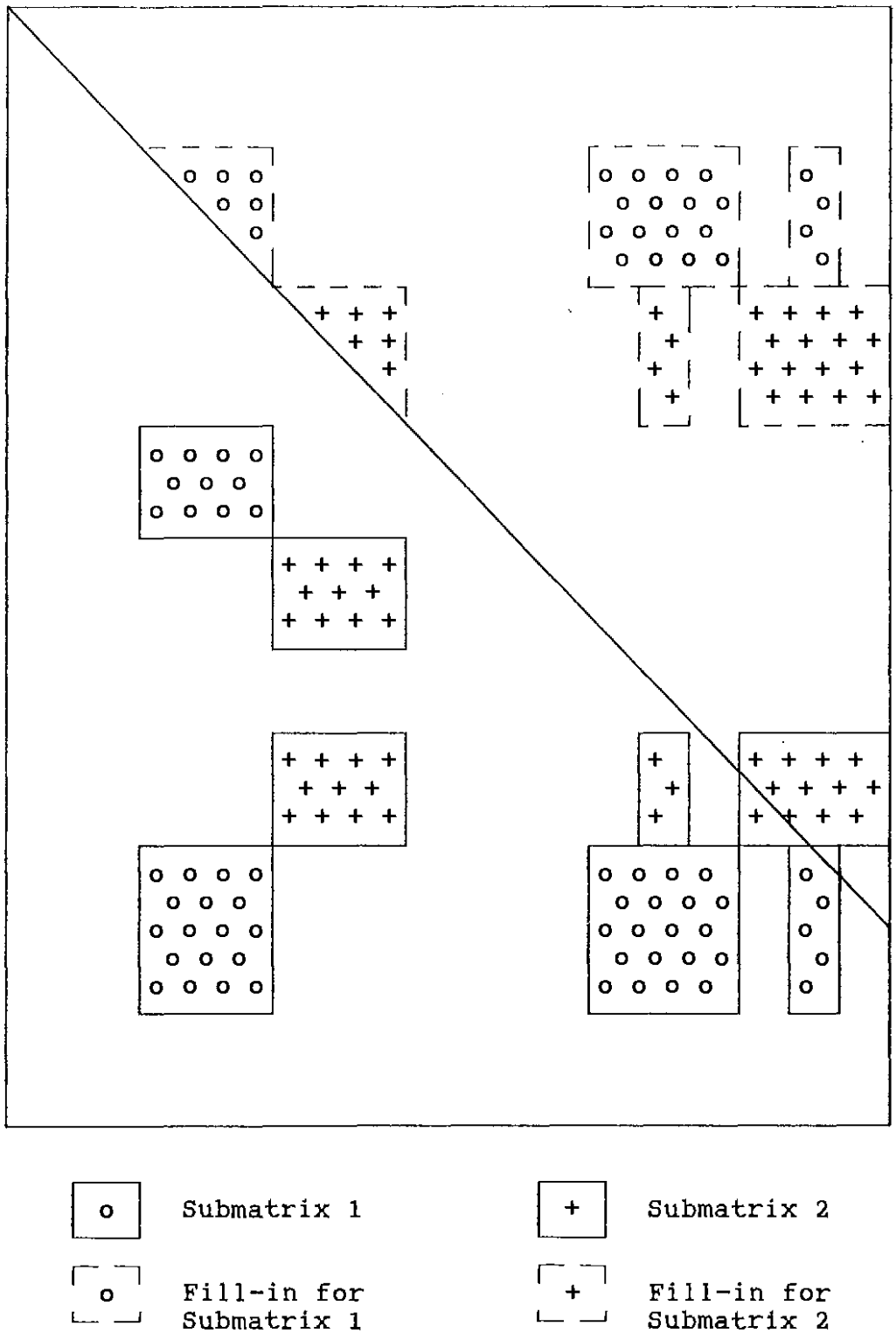


Figure 7 - Two Row/pivot Disjoint Submatrices

If we can order the rows of A so that the rows for each photo form a submatrix that is row/pivot disjoint from the rows of all other photos (And we can.), then we can predict very closely where the fill-in will occur for each photo.

To see that Givens rotations preserve a row/pivot disjoint submatrix, consider a four-element submatrix A_1 consisting of a pivot element a_{kk}, a target element a_{ik}, and two elements in the same column j of the pivot and target rows, a_{kj} and a_{ij}. Suppose this four-element submatrix A_1 is row/pivot disjoint from all other submatrices of A. From this we know that the pivot row k, the target row i, and the pivot column k contain no nonzero elements outside of the four elements of A_1. So when a_{kk} is used as a pivot to annihilate a_{ik}, no fill-in will occur outside of the four elements of A_1 since both the pivot row and target row are all zero outside of A_1. The pivot element a_{kk} will not be used to annihilate any target element outside of A_1 since the pivot column is all zero outside of A_1. So any fill-in caused by the pivot element of A_1 is confined to the four elements of A_1. Now consider a submatrix A_2 of A which is row/pivot disjoint from the rest of A with respect to a certain set of pivot elements. A_2 and its pivot row(s) consist of the union of all such four-element submatrices for that set of pivots. So any fill-in caused by these pivots is confined to the row/pivot disjoint

submatrix A_2 and the intersection of the columns of A_2 with its pivot row(s).

The remainder of this section of the paper will describe in detail where the fill-in occurs during the QR factorization of the bundle adjustment matrix. There are two distinct phases to the factorization that produce quite different patterns of fill-in. The use of the photo element pivot rows that annihilate the B_1 and B_2 blocks (see Figure 6) for each photo produce fill-in in neat, orderly blocks. The use of the tie point pivots is less orderly. After we have seen where the fill-in occurs, we shall list some rules for reducing the size of these regions and for combining some of them with the initial blocks.

Figure 8 shows the nonzero blocks for a typical aerial photo. We shall use this diagram to examine the fill-in that will result from the action of the photo element pivots for this photo. These pivots are the diagonal elements in the rows k_1 through k_2 that will be used to annihilate the photo element coefficients in blocks B_1 and B_2. Let us suppose that no initial elements for any other photo lie in these pivot rows, so the blocks for this photo are row/pivot disjoint from the initial blocks for all other photos in this bundle.

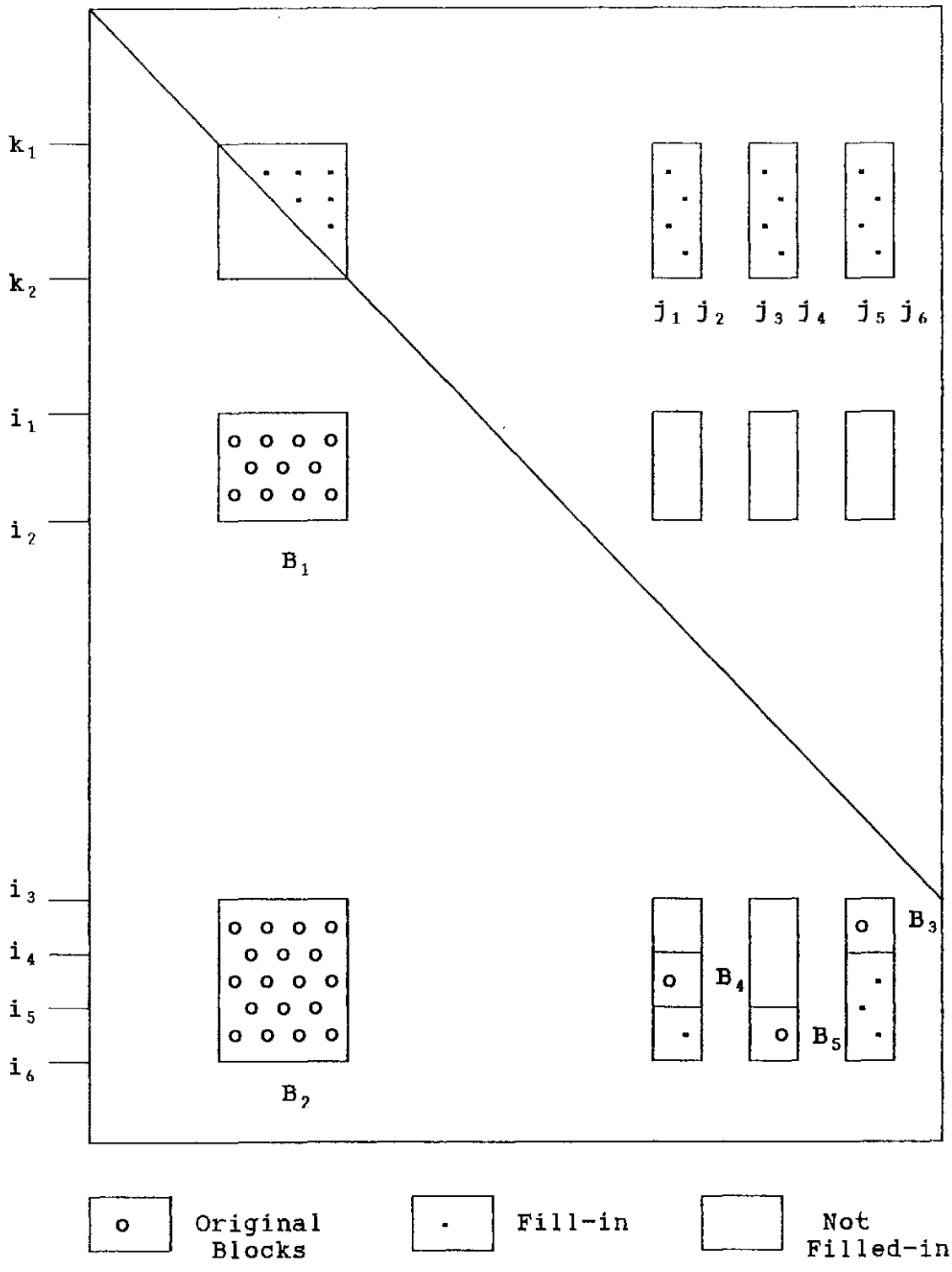


Figure 8 - Fill-in For a Single Photo
Using Photo Element Pivots

We can see immediately that the fill-in that we're looking for can only happen within the pivot rows and the rows and columns that contain the initial blocks B_1 thru B_5. By considering the order in which the pivot rows process the target rows, we can refine this further. We can show that the regions outlined but not shaded in Figure 8 will not be filled in.

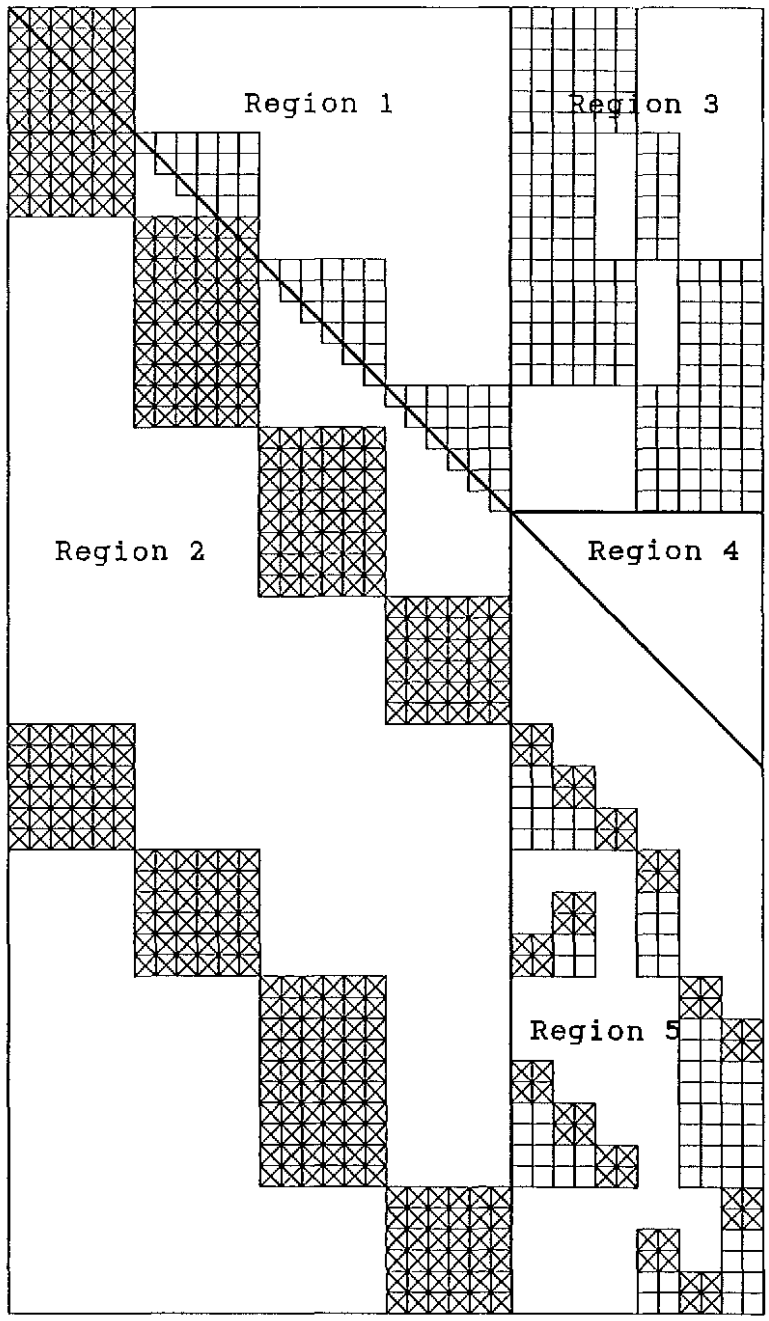
The pivot rows k_1 through k_2 are initially all zero, so when the first pivot row k_1, processes the B_1 block it will get filled-in in columns k_1 through k_2, but will be entirely zero outside these columns. This will remain true as this pivot row processes each target row in block B_1. So the dotted regions to the right of block B_1 will not be filled-in by pivot row k_1. When pivot row k_1 processes row i_3, the first row of blocks B_2 and B_3, this pivot row will be filled-in in columns j_5 through j_6 by the nonzero elements in block B_3. When this pivot row processes target row i_4 it will be filled-in in columns j_1 through j_2 by the nonzero elements in block B_4. The pivot row has already been filled-in in the columns of block B_3, so these columns will be filled-in in each target row until there are no more target elements for pivot row k_1 to process. This is the copy-up, copy-down phenomenon at work. Each tie point block will cause a "tail" of filled-in elements that continues downward from the tie point block to the last row of this

photo where there are no more nonzero elements in column k_1 to be processed by pivot row k_1.

The next major step begins with pivot row k_1+1, still all zero, processing target rows of B_1. The action of this pivot row is similar to that of pivot row k_1, except that it will not be fill-in to the left of column k_1+1 since column k_1 is already annihilated. This pivot row also will be filled-in by the tie point blocks when it reaches them. Similarly for the remaining pivot rows k_1+2 through k_2.

So the only fill-in that occurs in annihilating blocks B_1 and B_2 is of three types. A triangular block consisting of elements k through k_2 in each pivot row k results from the nonzero elements of blocks B_1 and B_2 themselves. A rectangular block where the pivot rows intersect the columns of the tie point blocks results from the nonzero elements of the tie point blocks. And a rectangular "tail" below each tie point block to the end of the photo results from the marking of the pivot rows by the tie point blocks.

The situation after all B_1 and B_2 blocks have been annihilated is shown in Figure 9 for the four-photo bundle. The regions above the diagonal marked region 1 and region 3 contain fill-in blocks. Regions 2 and 4 are entirely zero. Region 5 contains original tie point blocks with fill-in "tails" below each block to the end of its photo. The nonzero elements in this region, original or fill-in, must



⊠ □

Original Block Element Fill-in Element

Figure 9 - Fill-in After All Photo Element Pivots are Processed for the Bundle of Four Photos

now be zeroed out using the all-zero rows of region 4 as pivot rows.

Let k_* be the column number of the first tie point coordinate. Then row k_* is the first pivot row in region 4.

Bounds on the fill-in in each row in regions 4 and 5 due to annihilating the nonzero elements in region 5 can now be inferred from the distribution of these nonzero elements and our knowledge of the behavior of Givens rotations.

For each row in region 5, we know that no fill-in will occur to the left of the earliest column that is already nonzero in row i. Further, since the pivot rows are initially all zero, we know that no fill-in will occur to the right of the latest column that is nonzero in any target row up to and including row i.

Let f_i be the column number of the first nonzero entry in each row i of region 5. Let l_i be the maximum of the column number of the last nonzero entry in each row for rows k_* through i.

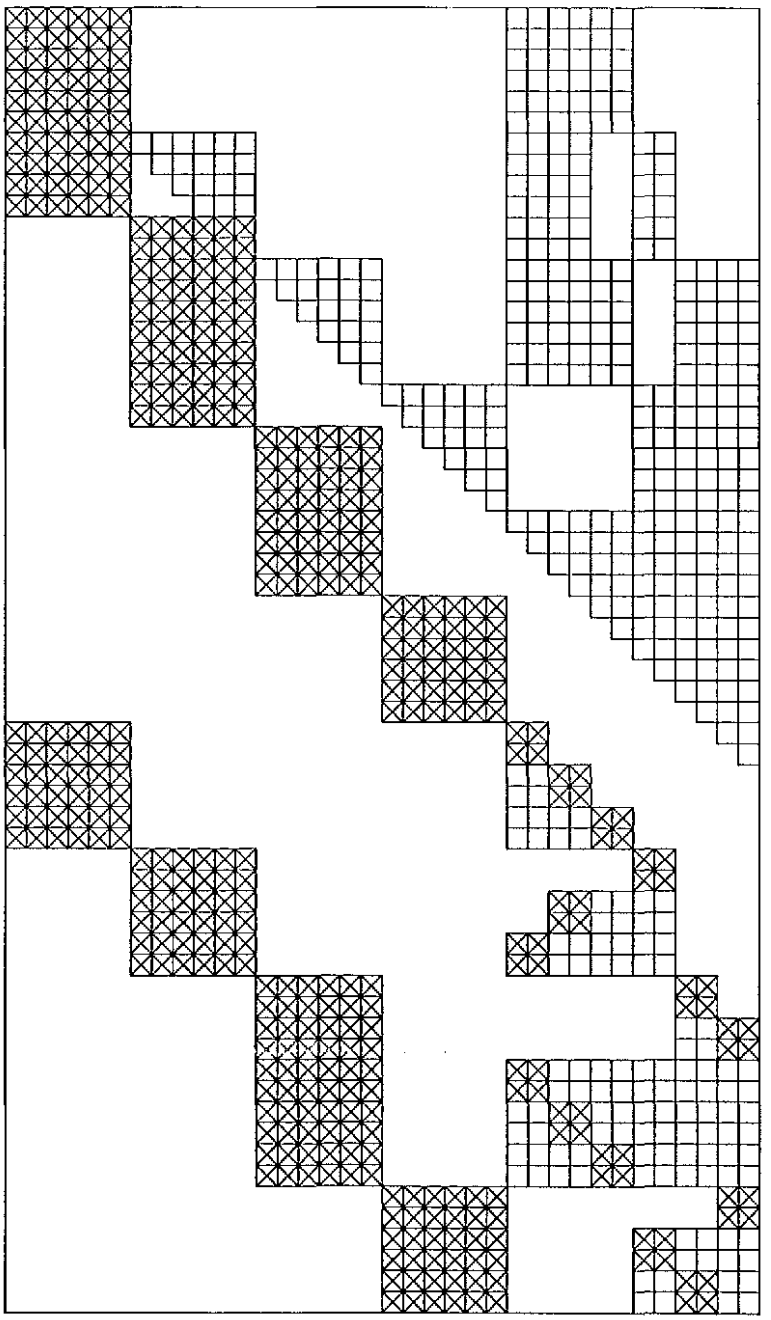
Then the fill-in in row i is confined to the columns f_i through l_i for each row i in region 5. The nonzero elements that are already present in region 5 that determine the starting and ending column numbers f_i and l_i will be either the original 2 x 2 tie point blocks or the two-column "tails" of fill-in below them that appeared earlier during the annihilation of photo element blocks. The fill-in that appears during the use of the tie point pivots will tend to

be horizontal rows of nonzero elements that connect the vertical columns of earlier fill-in. Note that these rows are not necessarily all filled-in. We have shown that fill-in cannot occur outside l_i and f_i, not that it must occur everywhere between these limits.

These bounds on the fill-in in each target row place bounds on the fill-in in each pivot row in region 4. For each pivot row k, let $t_k = \max \{l_i: \ f_i \leq k \leq l_i \ , \ i \geq k\}$. Then the fill-in in pivot row k will be confined to columns k through t_k, since the fill-in cannot occur to the left of the pivot, and pivot a_{kk} will not be used to operate on any row for which l_i is greater than t_k.

Having explained in detail the effects of both photo element pivots and tie point pivots, we can now understand how the fill-in occurs during the entire factorization. This will enable us to include the fill-in in the data structure for the coefficient matrix without having to do the laborious symbolic factorization. We can also use this knowledge to reorder the rows of the coefficient matrix to reduce the fill-in.

Figure 10 shows the coefficient matrix for the four-photo bundle of Figure 5, complete with all fill-in elements that occur during the complete factorization. Figure 10 can be compared with Figure 6, which shows the coefficient matrix for this bundle before any fill-in occurs. A careful examination of this figure confirms that



⊠ □

Original Block Element Fill-in Element

Figure 10 - Fill-in for the Coefficient Matrix for the Bundle of Four Photos

the fill-in locations for the entire matrix can be inferred a priori from the locations of the initial nonzero blocks.

In region 1, above the diagonal in the photo element region, the only fill-in is the 6 x 6 upper triangular block that includes the pivot elements. In region 2, below the diagonal in the photo element region, there is no fill-in at all. In region 3, to the right of region 1, the only fill-in is the 6 x 2 rectangular block where the pivot rows for each of the four photos intersect the columns of the tie point blocks for that same photo. The triangular Region 4, initially all zero, that provides the pivot rows for annihilating the tie point blocks is entirely filled-in in this small bundle. In reality this fill-in is of finite bandwidth and, for much larger bundles, will form a band to the right of the diagonal in region 4, with a jagged right hand edge. In region 5 the fill-in is the vertical "tails" of fill-in below each tie point block and the horizontal rows of fill-in between them. These "rows" are rather abbreviated for this small bundle. In larger bundles the two types of fill-in in region 5 show quite distinctly. We now list some rules which will help to reduce the fill-in described here.

The way the equations are set up for bundle adjustment assures that the B_1 and B_2 blocks for a particular photo are in separate rows and columns from those blocks for any other photo. Thus the pivot elements for that photo are in

separate rows and columns from the pivot elements for any other photo. But it can happen that some elements of the B_1 block for a particular photo fall in the pivot rows of the previous photo and the photos will not be row/pivot disjoint. This might happen, for example, if the first photo in the bundle lacked control points. Then the control point equations for the second photo would begin in row one, which is a pivot row for the B_2 block of the first photo.

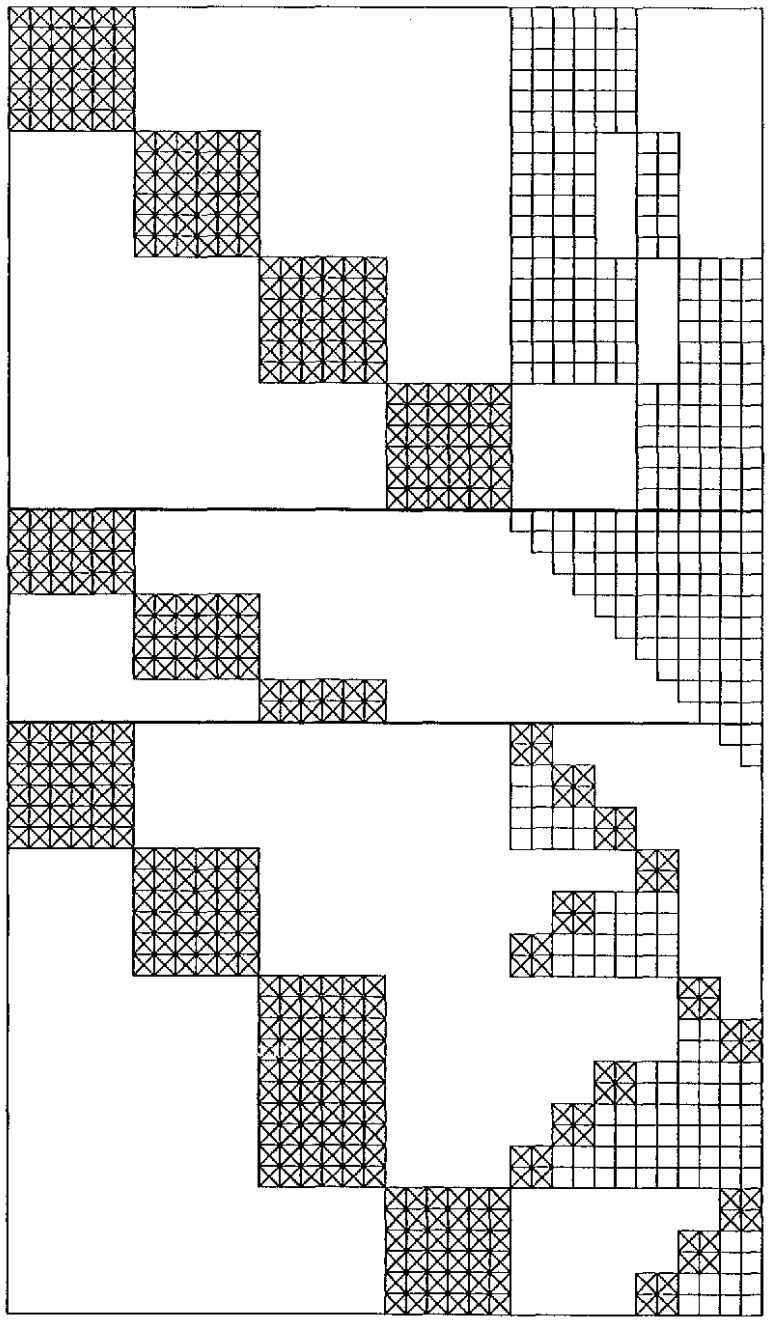
A judicious choice of the six rows of the matrix that make up the photo element pivot rows for each photo can guarantee that the photos are all mutually row/pivot disjoint. The required rule is simple: choose each group of six rows to be the first six control point equations for that photo; if a photo has less than three control points, use as many of its tie point equations as needed instead. This rule cannot fail since the rules of bundle adjustment require each photo to have at least four control/tie points so each photo will have at least eight equations.

This rule also results in a reduction of the population of the matrix since six rows of the B_1 and/or B_2 blocks will be combined with the 6 x 6 triangular fill-in block on the diagonal in the pivot rows. If tie point equations must be used the saving is even better since some 2 x 2 tie point blocks will be combined with the 6 x 2 fill-in blocks they cause in the pivot rows. Note that tie point equations must not be used before the available control point equations for

the photo are exhausted to get this effect artificially. This would cause unnecessary fill-in in the remaining control point equations for that photo.

The remaining control point equations for the entire bundle should be placed together in the rows immediately below the photo element pivot rows described above. These rows are the first of the tie point pivot rows and putting control points first here will help to push the remaining tie point blocks below the diagonal. This is necessary to insure that the tie point pivot rows are initially all zero as in the derivation above.

Figure 11 illustrates this improved row ordering for the four-photo bundle. The horizontal lines in Figure 11 divide the matrix into the three regions described above. In the upper region are the photo element pivot rows. These are formed by using first the control point equations for the photo and then, if there are not enough of these, using the tie point equations for the same photo. The middle region contains all the remaining control point equations. Notice that, for this bundle, this causes region 4 to be initially all zero. The lower region contains all the remaining tie point equations.



⊠ □

Original Block Element Fill-in Element

Figure 11 - Fill-in for the Coefficient Matrix
for the Bundle of Four Photos with Improved Row Ordering

It may not be possible to keep the tie point blocks below the diagonal and out of region 4. Recall that a bundle can be done with as few as four control points and many tie points. This is the source of those rare exceptions where the fill-in may not be so nice, although even this case may not be too bad. Each tie point block would cause a two-element-wide streak of fill-in down through region 5 until cut off by the f_i limit. Generally, when three or more control points are used in each photo this case will not happen. Our software does not try to add this type of fill-in a priori, but simply stops adding a priori fill-in and sets the flag for the symbolic factorization instead.

After the remaining control point equations come all the remaining tie point equations. These should be placed in order according to the following rules to reduce the fill-in prescribed by the f_i to l_i limits described above:

1) put the photos in increasing order by their last (furthest to the right) tie point number

2) within each photo, put the tie point equations for "new" tie points in increasing order by the tie point number, followed by

3) the tie point equations for the "old" tie points in decreasing order according to the tie point number.

Here a "new" tie point is one that hasn't appeared in a photo before, and an "old" tie point is one that has. This ordering of the tie point equations is used in Figure 11.

These rules will give the filled-in blocks for each photo a stair step appearance with the "new" tie point blocks going down and to the right followed by the "old" tie point blocks going down and to the left. Each photo will be offset a little to the right by the number of "new" tie point blocks it has. These rules will insure that the filled-in rows within each photo strictly increase in length and that the rows that go further to the right are later (further down) in the matrix. In strict accordance with the principle of saving the worst for last, this ordering of the tie point equations will insure that the lengths of the filled-in rows increase as slowly as possible for this ordering of the columns.

These rules also simplify predicting the fill-in in the tie point pivot rows in region 4. We described earlier how the right-hand limit of the fill-in in each pivot row can be found as the maximum l_i for all those rows on which the pivot row is used. With the target rows ordered according to our rules, the fill-in in each target row is a subset of the fill-in in the last row of each photo. That row is nonzero from the first tie point block for that photo to the last tie point block for that photo. Thus the fill-in in each pivot row can be found by examining the list of tie

point numbers for each photo. This can be done as each pivot row is formed, regardless of whether the complete set of target rows has yet been formed.

The working of these rules can be observed in Figure 12 that shows the original blocks and fill-in for a thirty three photo bundle. Figure 13 shows the same matrix with lines drawn between the different regions of the matrix for clarity. Figure 14 shows this matrix with the fill-in as predicted a priori. Notice that in Figure 14 the software has marked all nonzero elements as original, not fill-in, since they are provided by the calling program.

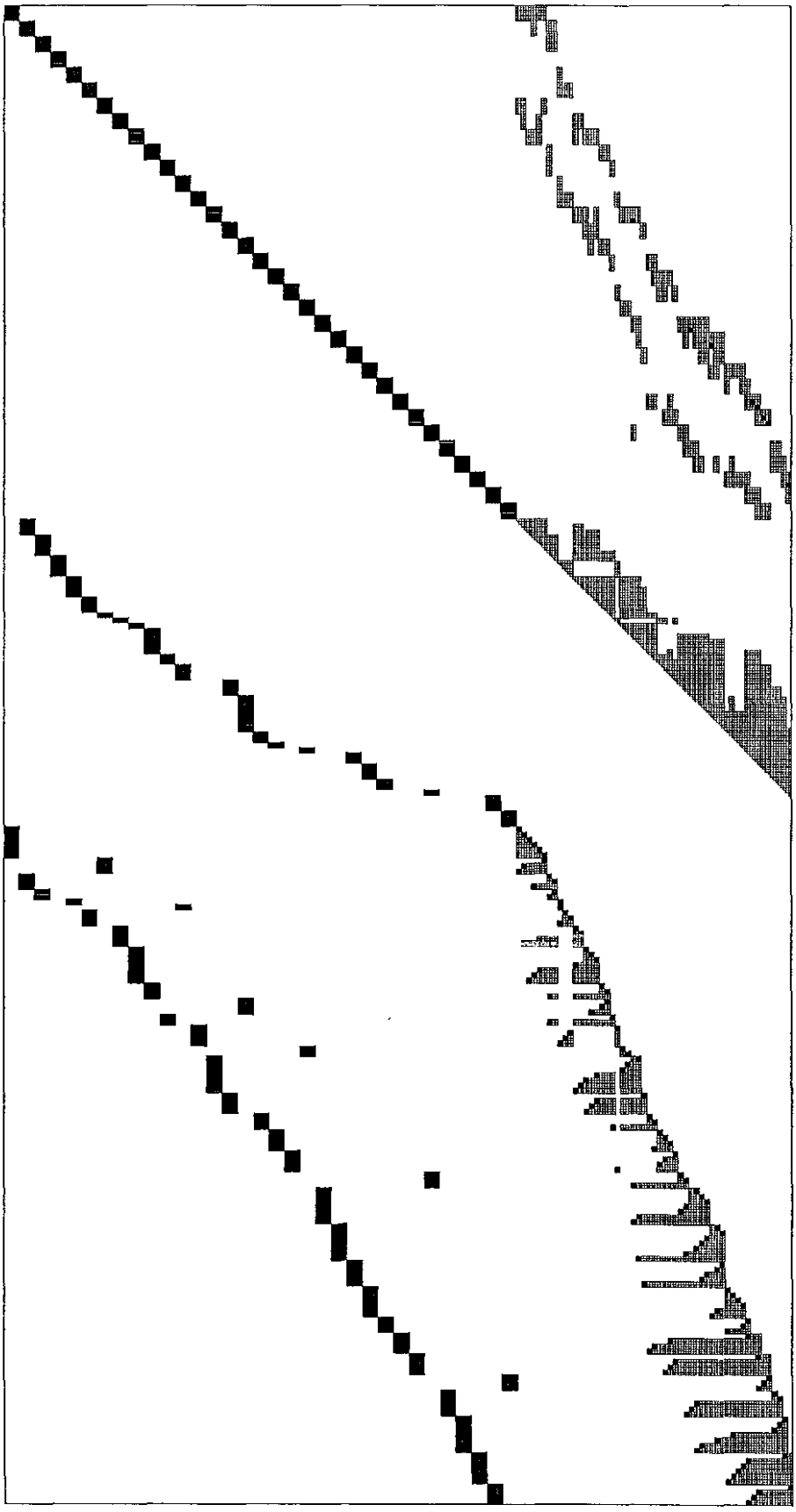
Below are titles of figures on following pages:

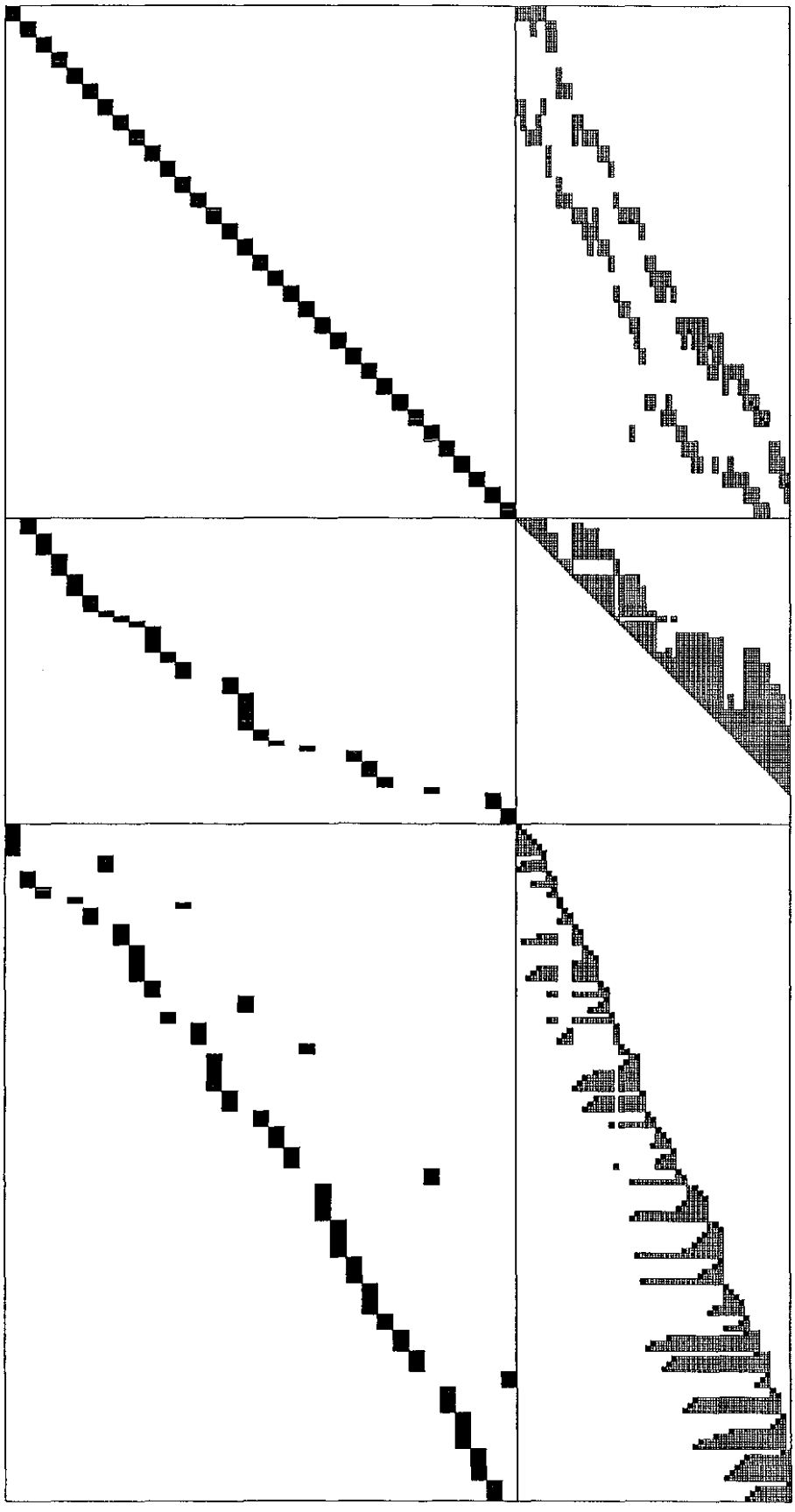
Figure 12 - Fill-in for the Coefficient Matrix for 33 Photo Bundle

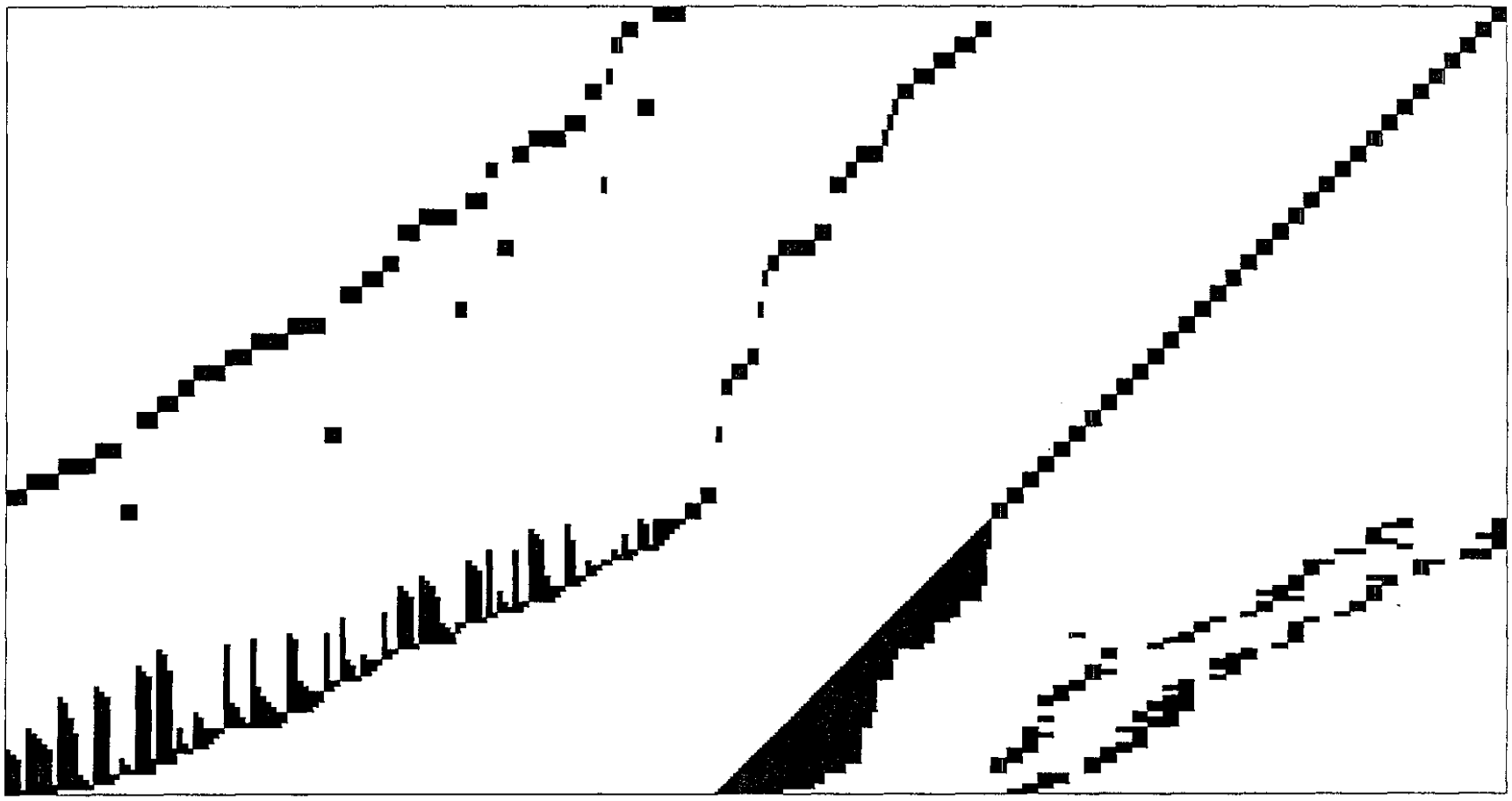
Figure 13 - Fill-in for the Coefficient Matrix for 33 Photo Bundle With Clarifying Lines Added

Figure 14 - Fill-in for the Coefficient Matrix for 33 Photo Bundle as Predicted A Priori

Note: Solid Squares Represent Original Block Elements.
Open Squares Represent Fill-in Elements.







Section 7 - Conclusion

To review briefly, the method of solving the LLSP of bundle adjustment developed in this paper is based upon sparse QR factorization using Givens rotations. The GIVENS2 subroutine library performs the required factorization. A row ordering for the bundle adjustment matrix is prescribed that significantly reduces the amount of fill-in and the time required for the factorization. Based upon this improved row ordering, a set of rules are provided for predicting where the fill-in will occur. For most bundle adjustment problems, these rules allow the time-consuming symbolic factorization step to be eliminated.

Figure 15 is a table showing the CPU times for bundle adjustment problems of various sizes. The column headed "dense factorization" lists timings using the LINPACK routines SQRDC and SQRSL. (Dongarra, Bunch, Moler, and Stewart, 1979; Coleman and Van Loan, 1988.) Figure 16 shows the populations for these test problems.

In the following paragraphs, the method developed in this paper is briefly contrasted with other LLSP solution methods from the recent literature. Opportunities for future research are noted where appropriate.

Problem Size	Factorization Timings in CPU Seconds			
	Dense One Iteration	Symbolic Done Only Once	Sparse After Symbolic One Iteration	Sparse w/ Estimated Fill-in One Iteration
4 Photos 62x36	0.15	0.16	0.15	0.17
15 photos 260x124	5.47	1.09	0.89	0.91
33 Photos 576x304	59.85	5.68	4.16	4.39
73 Photos 974x554	311.15	7.29	6.22	7.73
113 Photos 1478x870	1163.89	16.16	14.33	15.61

Figure 15 - Table of Timings of Test Runs

Problem Size	Number of Non-zero Elements			
	Original	Fill-in	Total	Estimated
4 Photos 62x36 =2236	428 (19.18%)	342 (15.32%)	770 (34.50%)	770 (34.50%)
15 photos 260x124 =32240	1748 (5.42%)	1522 (4.72%)	3270 (10.14%)	3331 (10.33%)
33 Photos 576x304 =175104	4008 (2.29%)	6223 (3.55%)	10231 (5.84%)	11007 (6.28%)
73 Photos 974x554 =539596	6480 (1.20%)	3966 (0.73%)	10446 (1.94%)	13334 (2.47%)
113 Photos 1478x870 =1285860	9916 (0.77%)	6688 (0.52%)	16604 (1.29%)	19932 (1.55%)

Figure 16 - Table of Populations of Test Runs

The method of solving the LLSP of bundle adjustment which is favored by photogrammetrists is recursive partitioning of the normal equations (Slama, 1980; Burnside, 1985; and Mikhail, 1976). In this method, the normal equations are formed and their coefficient matrix is partitioned into submatrices based upon the structure of the bundle, i.e. what control and tie points appear in what photographs. These submatrices are then partitioned based upon their structure and the process is repeated until most of the nonzeroes of the original matrix have been isolated in many small submatrices. The resulting small matrix equations are solved by dense Gaussian elimination. A direct comparison of recursive partitioning with the method of this paper is an important avenue of further research.

A method which is much-discussed in the recent literature of numerical analysis (see Schreiber and Van Loan, 1989, and Bischof and Van Loan, 1987, for example) is called the WY representation. This is a QR factorization using Householder reflections in which the product of Householder reflections Q is expressed as $Q = I + WY^T$ where W and Y are each m x n. This method is rich in matrix - matrix multiplications and so is of interest mainly in the parallel architecture realm where these operations are performed very quickly.

A block reflector is a generalization of a Householder reflection which can zero out elements of multiple columns of a matrix at once. Each block reflector differs from the identity matrix by a matrix of rank more than one. Block reflectors were first proposed by Brønlund and Johnsen in 1974 and have recently been rediscovered by the numerical analysis community in the search for efficient QR factorization algorithms for parallel computers. (Schreiber and Parlett, 1988.) Although at least one author has tried to apply block reflectors to sparse factorization (Kaufman, 1987) the algorithm proposed is limited to very well-conditioned matrices. The technology of block reflectors can be expected to improve rapidly in the next few years.

In the past decade several authors have looked at minimizing the fill-in in orthogonal factorization of a general, sparse matrix. The paper "Predicting the Fill for Sparse Orthogonal Factorization" by Coleman, Edenbrant, and Gilbert, (1983), is of fundamental importance. The authors use the theory of bipartite graphs and the local Givens rule to draw some important conclusions about fill-in. The bipartite graph of an m x n matrix A consists of two sets of vertices $V = \{v_1, \ldots, v_m\}$ and $W = \{w_1, \ldots, w_n\}$ with an edge connecting v_i and w_j whenever a_{ij} is not equal to zero. The major difficulty in the LLSP is that while the Cholesky factor is unique regardless of the row ordering, the amount

of intermediate fill-in can vary tremendously for different orderings of the rows. The paper, "Row Ordering Schemes for Sparse Givens Transformations: I. Bipartite Graph Model" by George, Liu, and Ng (1984), looks at this question in detail.

Growing out of the effort to predict fill-in for QR factorization have been a series of papers which propose algorithms based on the knowledge of where the fill-in will occur. In 1980, George and Heath proposed using the methods for predicting the structure of the Cholesky factor when it is formed by Gaussian elimination on the normal equations to set up a data structure which would then be used during QR factorization using Givens rotations. They avoided the problem of predicting the intermediate fill-in by working with only one row at a time, storing that row in a dense vector and zeroing out the row as the major step. In 1986, Liu proposed a generalization of the George and Heath method to variable pivoting. Liu not only chooses the target and pivot rows dynamically, but maintains several upper triangular matrices into which rows can be merged. The final Cholesky factor results from merging these smaller triangular matrices. He calls this method the "general row merging scheme."

In a later paper with Alan George entitled "Householder Reflections Versus Givens Rotations in Sparse Orthogonal Decomposition" (George and Liu, 1987) he describes how his

general row merging scheme results in small, essentially dense submatrices upon which Householder reflections can be used effectively. This allows the superior speed of Householder reflections to be applied to a sparse matrix problem. A detailed comparison of the method of George and Liu with that developed in this paper is another promising area for further research.

APPENDIX A - Dense Givens Routine

```
      SUBROUTINE GIVENS (A,MA,M,N,B)
C     THIS SUBROUTINE REPLACES THE M BY N MATRIX A
C     WITH ITS UPPER TRIANGULAR QR FACTOR USING GIVENS ROTATIONS.
C
      REAL A(MA,1),B(1)
C
C     CHOOSE THE COLUMN TO BE ZEROED OUT
C
      DO 5000 K = 1,N
      KP1 = K + 1
C
C     CHOOSE THE ELEMENT TO BE ZEROED OUT
C
      DO 4000 I = KP1,M
      R1 = A(K,K)
      R2 = A(I,K)
C
C     CALCULATE THE GIVENS ROTATION NEEDED AND APPLY
C
      CALL SROTG (R1,R2,C,S)
      A(K,K) = R1
      A(I,K) = 0.0
C
C     APPLY THIS ROTATION TO REMAINING COLUMNS
C
      DO 3000 J = KP1,N
      TEMP = C * A(K,J) + S * A(I,J)
      A(I,J) = C * A(I,J) - S * A(K,J)
      A(K,J) = TEMP
 3000 CONTINUE
C
C     APPLY THIS ROTATION TO THE RHS VECTOR
C
      TEMP = C * B(K) + S * B(I)
      B(I) = C * B(I) - S * B(K)
      B(K) = TEMP
C
C     GO BACK FOR NEXT ELEMENT IN THIS COLUMN
C
 4000 CONTINUE
C
C     GO BACK FOR NEXT COLUMN
C
```

```
 5000 CONTINUE
C     RETURN TO CALLER
C
      RETURN
      END
C
      SUBROUTINE BACKSOLVE (A,MA,M,N,B,X)
C
C     THIS SUBROUTINE SOLVES A TRIANGULAR SYSTEM OF LINEAR EQUATIONS
C     BY ROW-ORIENTED BACKSUBSTITUTION
C
      REAL A(MA,1),B(1),X(1)
C
      X(N) = B(N) / A(N,N)
      NM1 = N - 1
C
      DO 2000 IB = 1,NM1
      I = N - IB
      IP1 = I + 1
      TEMP = B(I)
C
      DO 1000 J = IP1,N
      TEMP = TEMP - A(I,J) * X(J)
 1000 CONTINUE
C
      X(I) = TEMP / A(I,I)
 2000 CONTINUE
C
C     RETURN TO CALLER
C
      RETURN
      END
C
      SUBROUTINE RESIDUAL (A,MA,M,N,B,X,R)
C
C     THIS SUBROUTINE CALCULATES THE RESIDUAL VECTOR OF A LINEAR SYSTEM.
C
      REAL A(MA,1),B(1),X(1),R(1)
C
      DO 2000 I = 1,M
      R(I) = B(I)
C
      DO 1000 J = 1,N
      R(I) = R(I) - A(I,J) * X(J)
 1000 CONTINUE
C
 2000 CONTINUE
C
C     RETURN TO CALLER
C
      RETURN
      END
```

APPENDIX B - GIVENS2 Sparse Matrix Library

```
C     GIVENS2 - THE SJRWMD SPARSE MATRIX PACKAGE
C
C     THIS PACKAGE OF SUBROUTINES CALCULATES THE LEAST SQUARES SOLUTION
C     OF AN OVERDETERMINED SYSTEM OF LINEAR EQUATIONS BY ORTHOGONAL
C     TRIANGULARIZATION USING GIVENS ROTATIONS.
C
C     ONLY THOSE ELEMENTS OF THE MATRIX WHICH ARE NONZERO(OR POTENTIALLY
C     NONZERO) ARE STORED.  TWO DIFFERENT DATA STRUCTURES ARE USED.  THE
C     EQUATION BUILDING AND SYMBOLIC FACTORIZATION PHASES USE A LINKED
C     LIST DATA STRUCTURE.  THE LINKED LIST IS THEN READ INTO A
C     SEQUENTIAL LIST FOR USE IN THE FACTORIZATION AND BACKSUBSTITUTION
C     PHASES.
C
C     PROGRAM AUTHOR:
C           JOSEPH W. WOODARD    PROGRAMMER/ANALYST
C           ST. JOHNS RIVER WATER MANAGEMENT DISTRICT
C           PALATKA, FL 32178      904-329-4280
C
      SUBROUTINE ADDROW (I,JLIST,VLIST,NVALS,MARK,LAST1)
C
C     THIS SUBROUTINE ADDS THE NONZERO ELEMENTS OF A SINGLE ROW OF
C     MATRIX A TO A ROW-ORIENTED LINKED LIST OF NONZERO ELEMENTS.
C     THE DATA STRUCTURE CREATED IS AS FOLLOWS:
C
C           W CONTAINS THE LIST OF VALUES OF A
C           JW CONTAINS THE COLUMN NUMBER J FOR EACH ENTRY IN W
C           ISTART CONTAINS THE ARRAY SUBSCRIPT IN W OF THE FIRST
C                NONZERO ELEMENT OF ROW I OF A
C           NEXT CONTAINS THE ARRAY SUBSCRIPT OF THE NEXT
C                ELEMENT IN ROW ORDER
C           LAST CONTAINS THE ARRAY SUBSCRIPT OF THE FINAL ELEMENT
C                (LAST MUST BE SET TO ZERO BY THE CALLER BEFORE
C                THE INITIAL CALL TO ADDROW FOR THE LIST)
C
C     THE ROW TO BE ADDED IS DEFINED BY:
C
C           I IS THE ROW NUMBER.  ROWS NEED NOT BE ADDED IN SEQUENCE.
C           JLIST IS THE LIST OF COLUMN SUBSCRIPTS FOR THIS ROW.
C                THESE SUBSCRIPTS MUST (!) BE IN ASCENDING ORDER.
C           VLIST IS THE LIST OF VALUES A(I,J) CORRESPONDING TO JLIST.
C           NVALS IS THE NUMBER OF VALUES IN VLIST.
C           MARK IS A LOGICAL VARIABLE WHICH CONTROLS UPDATING OF
C                THE COLUMN NUMBERS IN THE LINKED LIST.
C                MARK = .TRUE.    USE JLIST TO SET COLUMN NUMBERS.
```

```
C                MARK = .FALSE.    DON'T.  COLUMN NUMBERS FROM A
C                                   PREVIOUS CALL WILL BE USED.
C
      REAL VLIST(1)
      INTEGER JLIST(1)
      LOGICAL MARK
      INCLUDE 'INCLUDE1.F77'
C
      IF (NVALS .GT. 0) THEN
            LAST = LAST1
            ISTART(I) = LAST + 1
            K = NVALS - 1
            DO 1000 J = 1,K
            LAST = LAST + 1
            W(LAST) = VLIST(J)
            IF (MARK) THEN
                  JW(LAST) = JLIST(J)
                  NEXT(LAST) = LAST + 1
            ENDIF
 1000       CONTINUE
            LAST = LAST + 1
            W(LAST) = VLIST(NVALS)
            IF (MARK) THEN
                  JW(LAST) = JLIST(NVALS)
                  NEXT(LAST) = 0
            ENDIF
            LAST1 = LAST
      ENDIF
C
C     RETURN TO CALLER
C
      RETURN
      END
C
      SUBROUTINE SYMBOL (M1,N1,LAST1)
C
C     THIS SUBROUTINE PERFORMS A SYMBOLIC FACTORIZATION ON A
C     MATRIX A WHICH IS STORED IN A ROW-ORIENTED LINKED LIST.
C
      LOGICAL FOUNDAKK,FOUNDAIK,FOUNDAKJ,FOUNDAIJ
      INCLUDE 'INCLUDE1.F77'
C
C
C     SET ARRAY SIZE IN COMMON BLOCK
C
      M = M1
      N = N1
      LAST = LAST1
C
C     CHOOSE THE COLUMN TO BE ZEROED OUT
C
      DO 5000 K = 1,N
```

```
C
C     MARK THE DIAGONAL ELEMENT
C
      LOCAKK = 0
      CALL FINDITEM (K,K,AKK,FOUNDAKK,LOCAKK)
      IF (.NOT. FOUNDAKK) THEN
            AKK = 0
            CALL ADDITEM (K,K,AKK,LOCAKK)
      ENDIF
C
C     CHOOSE THE ELEMENT TO BE ZEROED OUT
C
      I = K
 1000 CALL LINKI (I,K,ITARGET,AIK,FOUNDAIK,LOCAIK)
      IF (FOUNDAIK) THEN
            I = ITARGET
C
C     APPLY THIS ROTATION TO REMAINING COLUMNS
C
            J = K
            LOCAKJ = 0
            LOCAIJ = 0
 2000       CALL LINKJ (K,J,JPIVOT,AKJ,FOUNDAKJ,LOCAKJ)
            IF (FOUNDAKJ) THEN
                  J = JPIVOT
                  CALL FINDITEM (I,J,AIJ,FOUNDAIJ,LOCAIJ)
                  IF (.NOT. FOUNDAIJ) THEN
                        AIJ = 0.0
                        CALL ADDITEM (I,J,AIJ,LOCAIJ)
                  ENDIF
                  GO TO 2000
            ENDIF
            J = K
            LOCAKJ = 0
            LOCAIJ = 0
 3000       CALL LINKJ (I,J,JTARGET,AIJ,FOUNDAIJ,LOCAIJ)
            IF (FOUNDAIJ) THEN
                  J = JTARGET
                  CALL FINDITEM (K,J,AKJ,FOUNDAKJ,LOCAKJ)
                  IF (.NOT. FOUNDAKJ) THEN
                        AKJ = 0.0
                        CALL ADDITEM (K,J,AKJ,LOCAKJ)
                  ENDIF
                  GO TO 3000
            ENDIF
C
C     GO BACK FOR NEXT ELEMENT IN THIS COLUMN
C
            GO TO 1000
      ENDIF
C
C     GO BACK FOR NEXT COLUMN
```

```
C
 5000 CONTINUE
C
C     RETURN TO CALLER
C
      LAST1 = LAST
      RETURN
      END
C
      SUBROUTINE FINDITEM (I,J,ELEMENT,FOUND,LOC)
C
C     THIS SUBROUTINE LOCATES AN ELEMENT A(I,J) OF A MATRIX A
C     WHICH IS STORED IN A ROW-ORIENTED LINKED LIST.
C
C     THE LOCATION WHERE A(I,J) WAS FOUND IS RETURNED IN LOC.
C     IF A(I,J) IS NOT FOUND, LOC IS SET TO EITHER ZERO OR
C     THE LOCATION OF THE LAST NONZERO ELEMENT WHOSE COLUMN NUMBER
C     IS LESS THAN J, IF ANY SUCH ELEMENT EXISTS.
C     IF J IS GREATER THAN THE COLUMN NUMBER OF ALL NONZERO
C     ENTRIES IN ROW I OF A, LOC IS SET TO THE NEGATIVE OF THE
C     LAST STORAGE LOCATION FOR ROW I.
C
C     THE VALUE OF LOC IS USED TO IMPROVE THE SPEED OF LOCATING
C     THE NEXT NONZERO ELEMENT IN THE ROW ON THE NEXT CALL TO THIS
C     SUBROUTINE.  THE USER MUST BE CAREFUL TO SET LOC TO ZERO HIMSELF
C     BEFORE ANY CALL TO THIS ROUTINE WHICH IS NOT INTENDED TO
C     LOCATE A SUCCEEDING ELEMENT IN THE SAME ROW.
C
      LOGICAL FOUND
      INCLUDE 'INCLUDE1.F77'
C
      FOUND = .FALSE.
      ELEMENT = 0.0
      IF (LOC .GE. 0) THEN
            L = ISTART(I)
            IF (LOC .GT. 0) L = LOC
            LPREV = 0
 1000       IF (JW(L) .EQ. J) THEN
                  FOUND = .TRUE.
                  ELEMENT = W(L)
                  LOC = L
                  GO TO 9999
            ENDIF
            IF (JW(L) .GT. J) THEN
                  LOC = LPREV
                  GO TO 9999
            ENDIF
            LPREV = L
            IF (NEXT(L) .NE. 0) THEN
                  L = NEXT(L)
                  GO TO 1000
            ENDIF
```

```
            LOC = -L
        ENDIF
C
C       RETURN TO CALLER
C
 9999 RETURN
        END
C
        SUBROUTINE RESIDUAL (NSEQ,X,B,R)
C
C       THIS SUBROUTINE CALCULATES THE RESIDUAL R = B - A * X
C       OF THE OVERDETERMINED LINEAR SYSTEM A * X = B WHERE THE
C       MATRIX A IS STORED IN A ROW-ORIENTED LINKED LIST.
C
C       THE LINKED LIST DATA STRUCTURE IS USED BECAUSE IT RETAINS
C       A COPY OF THE ORIGINAL INPUT MATRIX.  ONLY THOSE ELEMENTS
C       WHICH ARE STORED SEQUENTIALLY ARE USED IN THE CALCULATION,
C       SINCE THE FILL-IN ELEMENTS ARE ALL ZERO.
C
C       NSEQ MUST CONTAIN THE NUMBER OF ELEMENTS OF W WHICH ARE
C       STORED SEQUENTIALLY, I.E. THE NUMBER OF ELEMENTS W CONTAINED
C       BEFORE SYMBOL WAS CALLED TO ADD THE FILL-IN ELEMENTS.
C
        REAL B(1),X(1),R(1)
        LOGICAL FOUND
        INCLUDE 'INCLUDE1.F77'
C
C
        DO 2000 I = 1,M
        R(I) = B(I)
        J = 0
        LOC = 0
 1000 CALL LINKJ (I,J,JELEMENT,ELEMENT,FOUND,LOC)
        IF (FOUND) THEN
              IF (LOC .LE. NSEQ) THEN
                    R(I) = R(I) - ELEMENT * X(JELEMENT)
              ENDIF
              J = JELEMENT
              GO TO 1000
        ENDIF
 2000 CONTINUE
C
C       RETURN TO CALLER
C
 9999 RETURN
        END
C
        SUBROUTINE ADDITEM (I,J,ELEMENT,LOC)
C
C       THIS SUBROUTINE ADDS AN ELEMENT A(I,J) TO A MATRIX A
C       WHICH IS STORED IN A ROW-ORIENTED LINKED LIST.
C       THE LOCATION WHERE A(I,J) WAS STORED IS RETURNED IN LOC.
```

```
C     IF A(I,J) IS THE LAST NONZERO ELEMENT IN ROW I, LOC IS SET
C     TO THE NEGATIVE OF ITS LOCATION.
C
C     THE VALUE OF LOC IS USED TO IMPROVE THE SPEED OF ADDING AN
C     ITEM WHEN A PREVIOUS CALL TO FINDITEM HAS INDICATED THE ITEM
C     IS NOT PRESENT.  THE USER MUST BE CAREFUL TO SET LOC TO ZERO
C     BEFORE ANY CALL TO THIS ROUTINE WHERE THIS IS NOT THE CASE.
C
      LOGICAL FOUND
      INCLUDE 'INCLUDE1.F77'
C
C     START SEARCH AT BEGINNING OF THIS ROW
C     OR LATER IF A LATER LOCATION IS PROVIDED
C
      L = ISTART(I)
      IF (LOC .NE. 0) L = ABS(LOC)
C
C     DONE IF ITEM ALREADY EXISTS
C
 1000 IF (JW(L) .EQ. J) GO TO 9999
C
C     ADD ITEM IF A LATER ELEMENT IN THE ROW IS REACHED
C
      IF (JW(L) .GT. J) THEN
            LAST = LAST + 1
            IF (LAST .GT. NDATASIZE) GO TO 9000
            JW(LAST) = J
            W(LAST) = ELEMENT
            NEXT(LAST) = L
            LOC = LAST
            NEXT(LPREV) = LAST
            GO TO 9999
      ENDIF
C
C     FOLLOW THE THREAD IF THIS IS NOT THE END OF THE ROW
C
      LPREV = L
      IF (NEXT(L) .NE. 0) THEN
            L = NEXT(L)
            GO TO 1000
      ENDIF
C
C     ADD ITEM TO END OF ROW IF END IS REACHED
C
      LAST = LAST + 1
      IF (LAST .GT. NDATASIZE) GO TO 9000
      JW(LAST) = J
      W(LAST) = ELEMENT
      NEXT(LAST) = 0
      NEXT(LPREV) = LAST
      LOC = -LAST
      GO TO 9999
```

```
C
C     HANDLE OUT OF ROOM ERROR
C
 9000 WRITE (1,9010)
      WRITE (14,9010)
 9010 FORMAT (//,'STORAGE FOR LINKED LIST EXCEEDED',/,
     &   'SEE YOUR PROGRAMMER TO INCREASE THIS (ADDITEM)')
C
C     RETURN TO CALLER
C
 9999 RETURN
      END
C
      SUBROUTINE LINKI (I,J,IELEMENT,ELEMENT,FOUND,LOC)
C
C     THIS SUBROUTINE RETURNS THAT ELEMENT A(IELEMENT,J) IN COLUMN J
C     WHICH FOLLOWS ELEMENT A(I,J) IN COLUMN ORDER.
C     MATRIX A IS STORED IN A ROW-ORIENTED LINKED LIST.
C     THE LOCATION WHERE A(IELEMENT,J) WAS FOUND IS RETURNED IN LOC.
C     IF A(IELEMENT,J) IS NOT FOUND, LOC IS SET TO ZERO.
C
      LOGICAL FOUND
      INCLUDE 'INCLUDE1.F77'
C
      FOUND = .FALSE.
      IELEMENT = 0
      ELEMENT = 0.0
      LOC = 0
      IP1 = I + 1
      DO 2000 K = IP1,M
      L = ISTART(K)
 1000 IF (JW(L) .GT. J) GO TO 2000
      IF (JW(L) .EQ. J) THEN
            FOUND = .TRUE.
            IELEMENT = K
            ELEMENT = W(L)
            LOC = L
            GO TO 9999
      ENDIF
      IF (NEXT(L) .NE. 0) THEN
            L = NEXT(L)
            GO TO 1000
      ENDIF
 2000 CONTINUE
C
C     RETURN TO CALLER
C
 9999 RETURN
      END
C
```

```
      SUBROUTINE LINKJ (I,J,JELEMENT,ELEMENT,FOUND,LOC)
C
C     THIS SUBROUTINE RETURNS THAT ELEMENT A(I,JELEMENT) IN ROW I
C     WHICH FOLLOWS ELEMENT A(I,J) IN ROW ORDER.
C     MATRIX A IS STORED AS A ROW-ORIENTED LINKED LIST.
C     THE LOCATION WHERE A(I,JELEMENT) WAS FOUND IS RETURNED IN LOC.
C     IF A(I,JELEMENT) IS NOT FOUND, LOC IS SET TO ZERO.
C
C     THE VALUE OF LOC IS USED TO IMPROVE THE SPEED OF LOCATING
C     THE NEXT NONZERO ELEMENT IN THE ROW ON THE NEXT CALL TO THIS
C     SUBROUTINE.  THE USER MUST BE CAREFUL TO SET LOC TO ZERO HIMSELF
C     BEFORE ANY CALL TO THIS ROUTINE WHICH IS NOT INTENDED TO
C     LOCATE A SUCCEEDING ELEMENT IN THE SAME ROW.
C
      LOGICAL FOUND
      INCLUDE 'INCLUDE1.F77'
C
      FOUND = .FALSE.
      JELEMENT = 0
      ELEMENT = 0.0
      IF (LOC .EQ. 0) THEN
            L = ISTART(I)
 1000       IF (JW(L) .GT. J) THEN
                  FOUND = .TRUE.
                  JELEMENT = JW(L)
                  ELEMENT = W(L)
                  LOC = L
                  GO TO 9999
            ENDIF
            IF (NEXT(L) .NE. 0) THEN
                  L = NEXT(L)
                  GO TO 1000
            ENDIF
            LOC = 0
            GO TO 9999
      ENDIF
      IF (LOC .GT. 0) THEN
            L = LOC
            IF (NEXT(L) .EQ. 0) THEN
                  LOC = 0
                  GO TO 9999
            ELSE
                  L = NEXT(L)
                  FOUND = .TRUE.
                  JELEMENT = JW(L)
                  ELEMENT = W(L)
                  LOC = L
                  GO TO 9999
            ENDIF
      ENDIF
```

```
C
C     RETURN TO CALLER
C
 9999 RETURN
      END
C
      SUBROUTINE SEQUENCE (NSEQ)
C
C     THIS SUBROUTINE SCANS THE LINKED LIST AND CREATES A SEQUENTIAL
C     LIST.
C
C     THE DATA STRUCTURE CREATED IS AS FOLLOWS:
C
C           V CONTAINS THE LIST OF VALUES OF A
C           JV CONTAINS THE COLUMN NUMBER J FOR EACH ENTRY IN V
C           KV CONTAINS AN INTEGER WHICH INDICATES WHETHER THIS
C                ELEMENT IS A FILL-IN ELEMENT:
C                KV(L) = 1 IF LOCATION IS ORIGINAL
C                KV(L) = 2 IF LOCATION IS FILL-IN
C           LOCI CONTAINS THE LOCATION IN V OF THE FIRST
C                NONZERO ELEMENT OF ROW I OF A
C
C     NSEQ MUST CONTAIN THE NUMBER OF ELEMENTS OF W WHICH ARE
C     STORED SEQUENTIALLY, I.E. THE NUMBER OF ELEMENTS W CONTAINED
C     BEFORE SYMBOL WAS CALLED TO ADD THE FILL-IN ELEMENTS.
C
      INCLUDE 'INCLUDE1.F77'
C
C
C     CYCLE THROUGH ARRAY BY ROWS
C
      ICURR = 0
      DO 5000 I = 1,M
      LOCI(I) = ICURR + 1
      L = ISTART(I)
 1000 ICURR = ICURR + 1
      V(ICURR) = W(L)
      JV(ICURR) = JW(L)
      KV(ICURR) = 1
      IF (L .GT. NSEQ) KV(ICURR) = 2
      IF (NEXT(L) .NE. 0) THEN
            L = NEXT(L)
            GO TO 1000
      ENDIF
 5000 CONTINUE
      LOCI(M+1) = ICURR + 1
C
C     RETURN TO CALLER
C
      RETURN
      END
```

```
C
      SUBROUTINE FACTOR (B)
C
C     THIS SUBROUTINE REPLACES THE M BY N MATRIX A
C     WITH ITS UPPER TRIANGULAR QR FACTOR USING GIVENS ROTATIONS.
C     THE MATRIX IS STORED AS A ROW-ORIENTED SEQUENTIAL LIST.
C
      REAL B(1)
      LOGICAL FOUND
      INCLUDE 'INCLUDE1.F77'
C
C
C     CHOOSE THE COLUMN TO BE ZEROED OUT
C
      DO 5000 K = 1,N
      LOCAKK = 0
      CALL FIND (K,K,AKK,FOUND,LOCAKK)
      IF (.NOT. FOUND) GO TO 9000
C
C     CHOOSE THE ELEMENT TO BE ZEROED OUT
C
      I = K
 1000 CALL NEXTI (I,K,IELEMENT,AIK,FOUND,LOCAIK)
      IF (FOUND) THEN
            I = IELEMENT
C
C     CALCULATE THE GIVENS ROTATION NEEDED AND APPLY
C
            CALL SROTG (AKK,AIK,C,S)
            V(LOCAKK) = AKK
C
C     APPLY THIS ROTATION TO REMAINING COLUMNS
C
            LOCAIJ = 0
            IBEGIN = LOCAKK + 1
            IEND = LOCI(K + 1) - 1
            DO 2000 LOCAKJ = IBEGIN,IEND
            J = JV(LOCAKJ)
            CALL FIND (I,J,AIJ,FOUND,LOCAIJ)
            IF (LOCAIJ .LT. 0) GO TO 3000
            IF (FOUND) THEN
                  AKJ = V(LOCAKJ)
                  TEMP = C * AKJ + S * AIJ
                  AIJ = C * AIJ - S * AKJ
                  V(LOCAIJ) = AIJ
                  AKJ = TEMP
                  V(LOCAKJ) = AKJ
            ENDIF
 2000       CONTINUE
C
C     APPLY THIS ROTATION TO THE RHS VECTOR
```

```
C
 3000       TEMP = C * B(K) + S * B(I)
            B(I) = C * B(I) - S * B(K)
            B(K) = TEMP
C
C     GO BACK FOR NEXT ELEMENT IN THIS COLUMN
C
            GO TO 1000
      ENDIF
C
C     GO BACK FOR NEXT COLUMN
C
 5000 CONTINUE
C
C     DONE
C
      GO TO 9999
C
C     UNEXPECTED STORAGE FAILURE IS A FATAL ERROR
C
 9000 WRITE (1,9010) K,K
 9010 FORMAT (//,'UNEXPECTED STORAGE FAILURE (FACTOR)',/,
     &           'FINDING PIVOT ELEMENT AT K = ',I10,'  K = ',I10)
      STOP
C
C     RETURN TO CALLER
C
 9999 RETURN
      END
C
      SUBROUTINE BACKSOLVE (B,X)
C
C     THIS SUBROUTINE SOLVES A TRIANGULAR SYSTEM BY BACKSUBSTITUTION.
C     THE TRIANGULAR MATRIX IS STORED AS A ROW-ORIENTED SEQUENTIAL LIST.

C
      REAL B(1),X(1)
      LOGICAL FOUND
      INCLUDE 'INCLUDE1.F77'
C
C
      LOC = 0
      CALL FIND (N,N,ANN,FOUND,LOC)
      IF (ANN .EQ. 0.0) GO TO 9100
      X(N) = B(N) / ANN
C
C     CHOOSE NEXT ROW
C
      NM1 = N - 1
      DO 2000 IB = 1,NM1
      I = N - IB
      TEMP = B(I)
```

```
      LOC = 0
      CALL FIND (I,I,AII,FOUND,LOC)
      IF (AII .EQ. 0.0) GO TO 9100
      IBEGIN = LOC + 1
      IEND = LOCI(I + 1) - 1
      DO 1000 L = IBEGIN,IEND
      J = JV(L)
      TEMP = TEMP - V(L) * X(J)
 1000 CONTINUE
      X(I) = TEMP / AII
 2000 CONTINUE
C
C     DONE
C
      GO TO 9999
C
C     FATAL ERROR
C
 9100 WRITE (1,9110) I
 9110 FORMAT ('ZERO DIAGONAL ELEMENT (BACKSOLVE)',/,
     &        'I = ',I10)
      STOP
C
C     RETURN TO CALLER
C
 9999 RETURN
      END
C
      SUBROUTINE FIND (I,J,ELEMENT,FOUND,LOC)
C
C     THIS SUBROUTINE RETURNS AN ELEMENT A(I,J) OF A MATRIX A
C     WHICH IS STORED AS A ROW-ORIENTED SEQUENTIAL LIST.
C
C     THE LOCATION WHERE A(I,J) WAS FOUND IS RETURNED IN LOC.
C     IF A(I,J) IS NOT FOUND, LOC IS SET TO EITHER ZERO OR
C     THE LOCATION OF THE LAST NONZERO ELEMENT WHOSE COLUMN NUMBER
C     IS LESS THAN J, IF ANY SUCH ELEMENT EXISTS.
C     IF J IS GREATER THAN THE COLUMN NUMBER OF ALL NONZERO
C     ENTRIES IN ROW I OF A, LOC IS SET TO -1.
C
C     THE VALUE OF LOC IS USED TO IMPROVE THE SPEED OF LOCATING
C     THE NEXT NONZERO ELEMENT IN THE ROW ON THE NEXT CALL TO THIS
C     SUBROUTINE.  THE USER MUST BE CAREFUL TO SET LOC TO ZERO HIMSELF
C     BEFORE ANY CALL TO THIS ROUTINE WHICH IS NOT INTENDED TO
C     LOCATE A SUCCEEDING ELEMENT IN THE SAME ROW.
C
      LOGICAL FOUND
      INCLUDE 'INCLUDE1.F77'
C
      FOUND = .FALSE.
      ELEMENT = 0.0
      IF (LOC .GE. 0) THEN
```

```
            IBEGIN = LOCI(I)
            IF (LOC .GT. 0) IBEGIN = LOC
            IEND = LOCI(I+1) - 1
            LPREV = 0
            DO 1000 L = IBEGIN,IEND
            IF (JV(L) .EQ. J) THEN
                  FOUND = .TRUE.
                  ELEMENT = V(L)
                  LOC = L
                  GO TO 9999
            ENDIF
            IF (JV(L) .GT. J) THEN
                  LOC = LPREV
                  GO TO 9999
            ENDIF
            LPREV = L
 1000       CONTINUE
            LOC = -1
      ENDIF
C
C     RETURN TO CALLER
C
 9999 RETURN
      END
C
      SUBROUTINE NEXTI (I,J,IELEMENT,ELEMENT,FOUND,LOC)
C
C     THIS SUBROUTINE RETURNS THAT ELEMENT A(IELEMENT,J) IN COLUMN J
C     WHICH FOLLOWS ELEMENT A(I,J) IN COLUMN ORDER.
C     MATRIX A IS STORED AS A ROW-ORIENTED SEQUENTIAL LIST.
C     THE LOCATION WHERE A(IELEMENT,J) WAS FOUND IS RETURNED IN LOC.
C     IF A(IELEMENT,J) IS NOT FOUND, LOC IS SET TO ZERO.
C
      LOGICAL FOUND
      INCLUDE 'INCLUDE1.F77'
C
      FOUND = .FALSE.
      IELEMENT = 0
      ELEMENT = 0.0
      LOC = 0
      IP1 = I + 1
      DO 2000 K = IP1,M
      IBEGIN = LOCI(K)
      IEND = LOCI(K+1) - 1
      DO 1000 L = IBEGIN,IEND
      IF (JV(L) .EQ. J) THEN
            FOUND = .TRUE.
            IELEMENT = K
            ELEMENT = V(L)
            LOC = L
            GO TO 9999
      ENDIF
```

```
      IF (JV(L) .GT. J) GO TO 2000
 1000 CONTINUE
 2000 CONTINUE
C
C     RETURN TO CALLER
C
 9999 RETURN
      END
```

THE FOLLOWING IS THE CONTENTS OF THE INCLUDE FILE INCLUDE1 THAT SETS THE DIMENSIONS OF THE ARRAYS FOR THE LINKED LIST AND SEQUENTIAL LIST:

```
      PARAMETER NROWSIZE = 10000
      PARAMETER NDATASIZE = 400000
      REAL V(NDATASIZE),W(NDATASIZE)
      INTEGER JV(NDATASIZE),KV(NDATASIZE),LOCI(NROWSIZE)
      INTEGER JW(NDATASIZE),ISTART(NROWSIZE),NEXT(NDATASIZE)
      COMMON /SM1COM/ M,N,V,JV,KV,LOCI,W,JW,ISTART,NEXT,LAST
```

APPENDIX C - LEAST Subroutine

```
      SUBROUTINE LEAST (IROW,JARRAY,ARRAY,NVALS,M,N,BI,X,R,DUMP,ISYM)
C
C     THIS SUBROUTINE CALLS GIVENS2 ROUTINES TO GET THE LEAST SQUARES
C     SOLUTION OF AN OVERDETERMINED SYSTEM OF LINEAR EQUATIONS.
C     EACH EQUATION OF THE SYSTEM IS PASSED TO THIS ROUTINE IN
C     A SEPARATE CALL.  A FINAL CALL WITH IROW SET TO ZERO
C     GENERATES THE SOLUTION.
C
C     THE CHOICE OF DOING A SYMBOLIC FACTORIZATION IS CONTROLLED
C     BY THE PARAMETER ISYM AS FOLLOWS:
C
C           ISYM = 1      DO SYMBOLIC FACTORIZATION TO ADD THE NEEDED
C                         FILL-IN LOCATIONS TO THE LINKED LIST AFTER
C                         THE SEQUENTIAL DATA PROVIDED BY THE CALLER.
C
C           ISYM = 2      DO NOT PERFORM THE SYMBOLIC FACTORIZATION.
C                         THE LINKED LIST ALREADY CONTAINS THE NEEDED
C                         FILL-IN.
C
C           ISYM = 3      DO NOT PERFORM THE SYMBOLIC FACTORIZATION.
C                         ALL FILL-IN ELEMENTS WHICH WILL BE NEEDED ARE
C                         INCLUDED IN THE SEQUENTIAL DATA PROVIDED
C                         BY THE CALLER.
C
C     THE PARAMETER MARK CONTROLS WHETHER THE ARRAYS JW AND NEXT
C     WHICH DESCRIBE THE NONZERO STRUCTURE OF THE ROW SHOULD BE
C     UPDATED.  FOR A FIRST ITERATION, WITH OR WITHOUT THE
C     SYMBOLIC FACTORIZATION, THESE ARRAYS MUST BE SET.  FOR A
C     SECOND OR LATER ITERATION THE STRUCTURE IS THE SAME AS THE
C     FIRST ITERATION SO THEIR PREVIOUS VALUES CAN BE USED.
C
C           MARK = .TRUE.         JW AND NEXT ARRAYS ARE UPDATED
C
C           MARK = .FALSE.        JW AND NEXT ARRAYS ARE LEFT AS IS
C
C     AUTHOR:
C           JOSEPH W. WOODARD     PROGRAMMER/ANALYST
C           ST. JOHNS RIVER WATER MANAGEMENT DISTRICT
C           PALATKA, FL 32178       904-329-4280
C
      SAVE
      PARAMETER NROWSIZE = 10000          /* ALSO IN INCLUDE1
      PARAMETER NDATASIZE = 800000        /* ALSO IN INCLUDE1
      REAL ARRAY(1),X(1),R(1)
```

```
      INTEGER JARRAY(1)
      REAL B1(NROWSIZE),B2(NROWSIZE)
      LOGICAL DUMP,FOUND,MARK,FIRST
      DATA LAST /0/
      DATA FIRST /.TRUE./
C
C     CHECK DIMENSIONS OF PROBLEM AGAINST ARRAY SIZE
C
      IF (M .GT. NROWSIZE) GO TO 9200
      IF (IROW .GT. 0 .AND. LAST + NVALS .GT. NDATASIZE) GO TO 9200
C
C     ADD NONZERO ENTRIES FOR THIS ROW TO LINKED LIST
C
      IF (IROW .GT. 0) THEN
            IF (ISYM .EQ. 1) MARK = .TRUE.
            IF (ISYM .EQ. 2) MARK = .FALSE.
            IF (ISYM .EQ. 3) MARK = .TRUE.
            CALL ADDROW (IROW,JARRAY,ARRAY,NVALS,MARK,LAST)
            B1(IROW) = BI
            B2(IROW) = BI
            IER = 0
      ENDIF
C
C     SOLVE SYSTEM
C
      IF (IROW .EQ. 0) THEN
C
C           PERFORM SYMBOLIC FACTORIZATION, IF REQUESTED,
C           TO ADD NEEDED STORAGE LOCATIONS TO LINKED LIST
C
            NSEQ = LAST
            IF (ISYM .EQ. 1) THEN
                  CALL SYMBOL (M,N,LAST)
            ENDIF
            IF (ISYM .EQ. 2) THEN
                  LAST = NPREV
            ENDIF
C
C           COPY CONTENTS OF LINKED LIST TO SEQUENTIAL LIST
C
            CALL SEQUENCE (NSEQ)
C
C           PERFORM QR FACTORIZATION
C
            CALL FACTOR (B1)
C
C           BACKSOLVE
C
            CALL BACKSOLVE (B1,X)
C
C           CALCULATE THE RESIDUAL OF THE SYSTEM
C
```

```
            CALL RESIDUAL (NSEQ,X,B2,R)
C
C           REINITIALIZE FOR THE NEXT SOLUTION
C
            FIRST = .FALSE.
            NPREV = LAST
            LAST = 0
      ENDIF
      GO TO 9999
C
C     ERROR HANDLING SECTION
C
 9200 WRITE (1,9210) M,N
 9210 FORMAT ('DIMENSIONED ARRAY SIZE EXCEEDED (SUBROUTINE LEAST)',/,
     &        'SEE YOUR PROGRAMMER TO INCREASE THIS',/,
     &        'M = ',I10,'       N = ',I10)
      STOP
C
C     RETURN TO CALLER
C
 9999 RETURN
      END
```

REFERENCES

Bischof, C. and C Van Loan. (1987). "The WY Representation for Products of Householder Matrices." SIAM Journal of Scientific and Statistical Computation 8, 2-13.

Björck, Åke. (1976). "Methods for Sparse Linear Least Squares Problems." In Sparse Matrix Computations, edited by J.R. Bunch and D.J. Rose. New York: Academic Press, 177-199.

Brønlund, O.E. and Th. Lunde Johnsen. (1974). "QR Factorization of Partitioned Matrices." Computer Methods in applied Mechanics and Engineering 3, 153-172.

Burnside, C.D. (1985). Mapping From Aerial Photographs. New York: John Wiley and Sons.

Coleman, T. and C. Van Loan. (1988). Handbook for Matrix Computations. Philadelphia: Society for Industrial and Applied Mathematics.

Coleman, Thomas F., Anders Edenbrandt, and John R Gilbert. (1983). Predicting Fill for Sparse Orthogonal Factorization. Ithaca, New York: Department of Computer Science, Cornell University.

Dongarra, J.J., J.R. Bunch, C.B. Moler, and G.W. Stewart. (1979). LINPACK Users Guide. Philadelphia: Society for Industrial and Applied Mathematics.

Duff, I.S. (1974). "Pivot Selection and row Ordering in Givens Reduction On Sparse Matrices." Computing 13, 239-248.

Duff, I.S., A.M. Erisman, and J.K. Reid, (1986). Direct Methods For Sparse Matrices. 2ed. Oxford: Oxford University Press.

Gentleman, W. Morven. (1973). "Least Squares Computations by Givens Rotations Without Square Roots." Journal of the Institute for Mathematics and Its Applications. 12, 329-336.

George, Alan and Joseph W. H. Liu. (1987). "Householder Reflections Versus Givens Rotations in Sparse Orthogonal Decomposition." Linear Algebra and Its Applications 88/89, 223-238.

George, Alan and Michael T. Heath. (1980). "Solution of Sparse Linear Least Squares Problems Using Givens rotations"." Linear Algebra and Its Applications 34, 69-83.

George, Alan, Joseph Liu, and Esmond Ng. (1984). "Row Ordering Schemes for Sparse Givens Transformations. I. Bipartite Graph Model." Linear Algebra and Its Applications 61, 55-81.

Gill, Phillip E. and Walter Murray. (1976). "The Orthogonal Factorization of a Large Sparse Matrix." In Sparse Matrix Computations, edited by J.R. Bunch and D.J. Rose. New York: Academic Press, 201-212.

Hager, William W. (1988). Applied Numerical Linear Algebra. Englewood Cliffs, New Jersey: Prentice Hall.

Kaufman, Linda. (1987). "The Generalized Householder Transformation and Sparse Matrices." Linear Algebra and Its Applications 90, 221-234.

Liu, Joseph W.H. "On General Row Merging Schemes for Sparse Givens Transformations." SIAM Journal of Scientific and Statistical Computation 7, 1190-1211.

Mikhail, Edward M. (1976). Observations and Least Squares. New York: Dun-Donnelley Publishers.

Ortega, James M. (1987). Matrix Theory, a Second Course. New York: Plenum Press.

Schreiber, Robert and Beresford Parlett. (1988). "Block Reflectors: Theory and Computation." SIAM Journal of Numerical Analysis 25, 189-205.

Schreiber, Robert and Charles Van Loan. (1989). A Storage-Efficient WY Representation for Products of Householder Transformations." SIAM Journal of Scientific and Statistical Computation 10, 53-57.

Stewart, G.W. (1973). Introduction to Matrix Computations. New York: Academic Press.

Schendel, U. (1989). Sparse Matrices: Numerical Aspects with Applications for Scientists and Engineers. Chichester, West Sussex, England: Ellis Horwood Limited.

Slama, C.C. (1980). Manual of Photogrammetry, 4ed. Falls Church, Virginia: American Society of Photogrammetry.

Wolf, Paul R. (1983). Elements of Photogrammetry with Air Photo Interpretation and Remote Sensing, 2ed. New York: McGraw-Hill Book Company.

Vita

Name of Author: Joseph Walker Woodard

Place of Birth: Cincinnati, Ohio

Date of Birth:

Graduate and Undergraduate Schools Attended:

University of North Florida, Jacksonville FL

Florida State University, Tallahassee, FL

University of Arizona, Tucson, AZ

Degrees Awarded:

Master of Arts in Mathematical Sciences, 1990,
University of North Florida

Bachelor of Science in Physics, 1976,
University of Arizona

Areas of Interest:

Computer Graphics, Numerical Linear Algebra,
Theory of Least Squares

Professional Experience:

Research Assistant, Florida State University.

Computer Programmer and Analyst,
St. Johns River Water Management District

Memberships:

Society for Industrial and Applied Mathematics

Association for Computing Machinery, SIGNUM

Mathematical Association of America

Pi Mu Epsilon