Global Astrometric Solutions with Sparse Matrix Techniques

Richard L. Branham, Jr.

Instituto Argentino de Nivología y Glaciología (IANIGLA), C.C. 330, 5500 Mendoza, Argentina

Abstract. Modern astrometric techniques lead to large, linear systems solved by the precepts of least-squares. These systems are usually sparse, and one should take advantage of the sparsity to facilitate their solution. As long as the matrix **A** of the equations of condition possesses the weak Hall property, characteristic of linear systems derived from astrometric reductions, it is possible to find a sparse Cholesky factor. Before the equations of condition are accumulated, by use of the fast Givens transformation, a symbolic factorization of **A** using Tewarson's length of intersection technique determines the ordering of the columns of **A** that result in low fill-in. The non-null elements are stored in a sparse, dynamic data structure by use of dynamic hashing. Numerical experimentation shows that this competes well with alternatives such as nested dissection, and large, but sparse, linear systems with several thousand unknowns can be solved in a reasonable amount of time, even on personal computers.

1. Introduction

Modern astrometric techniques lead to large, linear systems solved by the precepts of least-squares. Examples are global astrolabe reductions, plate overlap techniques, and radio astrometric reductions. These systems are usually sparse, and one should take advantage of the sparsity to facilitate their solution. In fact, a standard technique for symmetric, positive definite matrices, which arise when the equations of condition are accumulated, and known as "nested dissection" (George and Liu, 1981), originated over a century ago to solve geodetic problems (Helmert, 1880). Let **A**, of size $m \times n$, where m represents the number of equations and n the number of unknowns, be the matrix of the equations of condition and **b** the m-vector of the right-hand-side. If one were to use least-squares one could form the matrix $\mathbf{A}^T \cdot \mathbf{A}$ of the normal equations and then decompose the normal equations into $\mathbf{S}^T \cdot \mathbf{S}$, where the Cholesky factor **S** is upper triangular. If the normal equations are sparse then processing by use of, for example, nested dissection produces a sparse Cholesky factor **S**. If one uses orthogonal transformations in lieu of normal equations to reduce **A** to the upper triangular matrix **R**, **R** and **S** are identical to within roundoff or chopping error and exhibit the same sparsity structure.

Unfortunately, although **A** may be sparse $\mathbf{A}^T \cdot \mathbf{A}$ will not in general conserve sparsity. If the non-null elements of **A** occur at random locations and **A** is

$100d$ per cent dense ($0 \leq d \leq 1$) a simple probability argument shows that $\mathbf{A}^T \cdot \mathbf{A}$ fills to a density of $mn(1 - d^2)^m$. Unless $d \approx 0$ the fill-in can approach 100% even for low d. Because the Cholesky factor **S** is unique and identical with **R**, this argument remains germane even if one does not *explicitly* form the normal equations but rather uses orthogonal transformations. Fortunately, with astrometric data the unknowns tend to occur in groups within a row of **A**, for example the station coordinates for radio astrometric reductions, and also with unknowns common to all of the rows, for example the position of a radiosource observed by all, or nearly all, of the stations. Thus, the matrix **A** contains some columns close to 100% dense and other, sparse columns with a far from random sparsity structure. Such a structure embodies what is referred to in sparse matrix terminology as the "weak Hall property" (Björck, 1996) and assures that $\mathbf{A}^T \cdot \mathbf{A}$, although more dense than **A**, nevertheless remains sparse. One should therefore take advantage of the sparsity of $\mathbf{A}^T \cdot \mathbf{A}$ to find a sparse Cholesky factor.

As an example I generated two 30,000×2000 matrices, the first with elements in positions fulfilling the weak Hall property, the other with elements in purely random locations, the strong Hall property. Both matrices were 0.85% dense. The first matrix lead to normal equations that were 8.7% dense and justify the search for a sparse Cholesky factor, whereas the normal equations for the second matrix were 65% dense and scarcely merit the computational labor needed to calculate a sparse Cholesky factor that in fact will not be sparse.

2. A Sparse Cholesky Factor

Although $\mathbf{A}^T \cdot \mathbf{A}$ may be sparse, **S** will not be unless the columns of **A** are permuted in such a way as to minimize the fill-in of the Cholesky factor. There are many ways to permute the columns of **A**. Nested dissection (George and Liu, 1981) was originally developed for just such problems as those that fulfill the conditions of the geodetic problem. To understand nested dissection one must delve into graph theory. Tewarson (1973) offers an alternative, "length of intersection." Although originally designed to reduce the bandwidth of band matrices, I have found that it works as well, usually better, only infrequently worse, than nested dissection: on test matrices with sizes from $n = 150$ up to $n = 2000$, length of intersection generated from 6% to 28% less fill-in than nested dissection.

Tewarson should be consulted for the reasons *why* length of intersection works, which can be understood without recourse to graph theory. Its implementation on the computer is straightforward. Because **A** is a sparse matrix it becomes necessary to store only its non-null elements A_{ij} along with their i and j positions. Assume that these are read from a disk file in the order i, j, A_{ij}. To find the permutations of $\mathbf{A}^T \cdot \mathbf{A}$ that lead to a sparse Cholesky factor it is unnecessary to work with a full matrix **A**, only with the i and j positions of the non-null elements within **A**. This permits a considerable savings of memory as a data structure such as a bitmap (Branham, 1990) may be used: if an element is present at position i, j set the bit to 1, otherwise to 0. See Branham (1990) for details of how to implement a bitmap. Languages such as C and C++ are particularly efficient for managing bitmaps because they offer a data structure, the bit field, ideal for use with bitmaps.

Once the bitmap of $\mathbf{A}$ is formed, that of $\mathbf{A}^T \cdot \mathbf{A}$ may be calculated easily. Having the bitmap of the latter we can implement length of intersection by calculating the matrix $\mathbf{W} = (\mathbf{A}^T \cdot \mathbf{A})^2$, where $\mathbf{W}$ is no longer a bitmap. That is, although the elements of $\mathbf{A}^T \cdot \mathbf{A}$ are 0 and 1, the elements of $\mathbf{W}$ represent the genuine products calculated from the rows and columns of $\mathbf{A}^T \cdot \mathbf{A}$. Define a vector $\mathbf{V}$ of dimension n and consisting of one's, $\mathbf{V}$=(1 1·1 1), and vectors $\mathbf{e}_i$ of dimension n with all null elements except a unit at position i, $\mathbf{e}_i = (0\ 0 \cdots 1 \cdots 0\ 0)$. Now calculate n vectors

$$\mathbf{v}_i = \mathbf{e}_i \cdot \mathbf{W} \cdot \mathbf{V}. \tag{1}$$

Define an n-vector **ip** of the column permutations of $\mathbf{A}$ initialized to **ip** = (1 2·n). Sort the values of $\mathbf{v}_i$ in increasing order with the corresponding values of **ip** in the same order. Thus, if $\mathbf{v}_1$ were the largest value of the $\mathbf{v}_i$ and $\mathbf{v}_n$ the smallest, **ip** would become (n $2 \cdots 1$). **ip** contains the column permutations of $\mathbf{A}$ that lead to a sparse Cholesky factor. Because Cholesky decomposition is stable, pivoting, which could change the ordering in **ip** and hence the sparsity of $\mathbf{S}$, becomes unnecessary.

3. Dynamic Hashing

The vector **ip** can be calculated by working only with bitmaps, but to calculate the actual solution we need to read in the matrix elements A_{ij} themselves. To avoid having to physically move data elements, these may be processed as $A_{ip(i),ip(j)}$. One may use normal equations or, to take advantage of the lower condition number of $\mathbf{A}$, orthogonal transformations applied directly to the latter without the explicit formation of normal equations. For sparse matrix processing the fast Givens transformations (FGT) possesses the advantage of nearly the same operation count as the Householder transformation, half that of the standard Givens transformation, but, seeing as we process $\mathbf{A}$ one element at a time, we can take advantage of the sparsity of $\mathbf{A}$ to only operate on the matrix's non-null elements, unlike the situation with Householder transformations where at least a part of an entire column of $\mathbf{A}$ must be processed. Nor does the FGT require the calculation of square roots. See Gentleman (1974) for details.

Although the bitmap serves to store the sparsity structure of a matrix, actual processing of the matrix requires a storage scheme for the elements A_{ij} themselves. Storage schemes take many forms. Hashing, a way of rapidly encountering information in a table, could be used but suffers drawbacks. Usually the table is nothing more than a 1-dimensional array whose elements are referred to as "buckets." The information, for our purpose a non-null matrix element indexed by a single parameter $k = j(j-1)/2 + i$, is entered in the table by a transformed key. The original key could be the position of the matrix element in the array. The key is transformed to occupy a smaller range so that the table will not be excessively large. Because of the transformation more than one element can be slated to occupy the same position in the table, called in hashing terminology a "collision." Standard hashing becomes unwieldy for sparse matrix processing because fill-in may result in table overflow if linear probing, find the first available empty bucket and deposit the element there, is used to resolve collisions or in inefficient operation if chained scheduling, use a linked list anchored at the bucket, resolves collisions.

Dynamic hashing, allowing the table to grow or shrink according to the density, avoids this difficulty. For a summary of dynamic hashing see Enbody and Du (1988) or Larson (1988). To briefly summarize dynamic hashing let α be the load factor, defined as the ratio of the number of elements in the matrix to the size of the hash table. For dynamic hashing with chained scheduling the hash table will be an array of pointers to the first element in a linked list of collided matrix elements with the same key. With a load factor of three there will typically be three elements in the list. Thus, we will be able to find a given element with an average of 1.5 searches. As fill-in occurs the load factor will increase. The hash table will be expanded dynamically, one bucket at a time, so that the average load factor remains reasonable. With dynamic hashing the entire table need not be reorganized, only a portion of it. To assure a good distribution of the keys I follow Knuth's (1973) suggestion that the original key should be transformed modulo $4l+3$, where l is a prime number. l should be selected initially as the prime closest to the size of the hash table divided by the load factor.

Let us look at the space requirements. $\mathbf{A}^T \cdot \mathbf{A}$ is symmetric and $\mathbf{S}$ (or equivalently $\mathbf{R}$) upper triangular and may be stored as a vector of $n(n+1)/2$ elements indexed by the mapping function $k = j(j-1)/2 + i$. If double-precision is used the vector needs $8n(n+1)/2$ bytes plus $8n$ bytes for the right-hand-side. For dynamic hashing let d be the initial density, f the fill-in, and take $\alpha = 3$ for the load factor. The right-hand-side needs $8n$ bytes. The hash table itself is an array of pointers. Pointers usually occupy four bytes. We thus need $2(d+f)n(n+1)/3$ bytes for the hash table. Each bucket of the hash table points to a list of matrix elements, each one of which needs sixteen bytes (eight for the element itself, four for k and four for a pointer to the next element in the list), for a total of $8n(n+1)(d+f)$ for the matrix elements. Dynamic hashing requirements are thus $26n(n+1)(d+f)/3 + 8n$. Let F be a figure of merit given by the ratio of the size of the matrix needed by dynamic hashing to that needed by a vector; leave the right-hand-side out of consideration. Then

$$F = 13(d+f)/6 \tag{2}$$

Thus, for dynamic hashing to represent less memory the ratio F should be less than unity and from Eq. (2) $(d+f)$ should be less than 40.15%. Fortunately, the exigencies of the geodetic problem frequently comply with this requirement.

4. Calculating the Solution

Having $\mathbf{S}$, the least-squares solution follows upon solution of the linear system

$$\mathbf{S} \cdot \mathbf{x} = \mathbf{Q}^T \cdot \mathbf{b}, \tag{3}$$

where $\mathbf{Q}^T$ represents the accumulated FGT applied to $\mathbf{b}$. To calculate the covariance matrix one would have to compute $(\mathbf{A}^T \cdot \mathbf{A})^{-1} = \mathbf{S}^{-1} \cdot \mathbf{S}^{-T}$. The inverse of a sparse matrix, unfortunately, is usually not sparse. To avoid catastrophic fill-in one may instead generate the covariance matrix one column at a time by calculating the solutions of n linear systems. We need an auxiliary n-vector $\mathbf{y}$:

$$\begin{aligned}\mathbf{y} &= \mathbf{S}\cdot\mathbf{s}_i;\\ \mathbf{S}^T\cdot\mathbf{y} &= \mathbf{e}_i.\end{aligned} \qquad (4)$$

The n-vectors $\mathbf{s}_i$ are the columns of the covariance matrix.

As an example I solved a test problem of size 30,000×2000 with $\mathbf{A}$ 0.84% dense in 2.5 hours on a 300Mhz Pentium machine with 32 MB of memory using the algorithm outlined here whereas the problem could not be solved at all (the "out of memory" message) when I used a least-squares algorithm.

5. Conclusions

When the requirements of the problem comply with the assumption of $\mathbf{A}'s$ being a sparse matrix fulfilling the weak Hall property, the algorithm presented in this paper uses less memory than usual least-squares algorithms. This means that a solution, along with its covariance matrix, may be possible that would otherwise be impossible or that it may be computed more efficiently. As the density of $\mathbf{A}$ increases, however, the efficiency of the algorithm decreases. The density of the Cholesky factor $\mathbf{S}$ should never be greater than 40.15% and for efficient operation considerably less.

References

Björck, A., 1996, *Numerical Methods for Least Squares Problems*, Philadelphia: SIAM, 162–163.

Branham, Jr., R.L., 1990, *Scientific Data Analysis*, New York: Springer, Sec. 3.3.1.

Enbody, R.J. & Du, H.C., 1988, *ACM Computing Surveys*, **20**, 85.

Gentleman, W.M., 1974, *App. Stat.*, **23**, 448.

George, A. & Liu, J.W., 1981, *Computer Solution of Large Sparse Positive Definite Systems*, Englewood Cliffs, New Jersey: Prentice-Hall.

Helmert, F.R., 1880, *Die Mathematischen und Physikalishen Teorien der höheren Geodäsie*, 1 Teil, Leipzig: Teubner.

Knuth, D., 1973, *The Art of Computer Programming*, Vol. 3, Sorting and Searching, Reading, Mass.: Addison Wesley, Sec. 6.4, 5.2.1.

Larson, P.A., 1988, *Comm. ACM*, **31**, 446.

Tewarson, R.P., 1973, *Sparse Matrices*, New York: Academic, Sec. 2.5, 3.2.