CSCI 694

HIGH PERFORMANCE NUMERICAL COMPUTATION

Sparse Matrices
Direct Methods with Graph Theory

Srilalitha Parimi
00586684
Savitri Devisetty
00556735

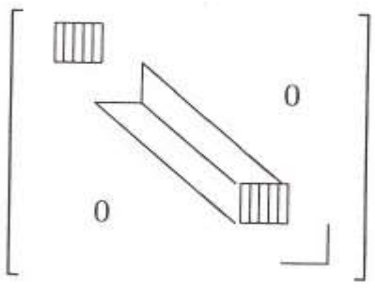
Table of Contents

Introduction to Sparse Matrix 3
Storage of Sparse matrices 4
Data Structures 4
- Static Data Structures are: 4
  1) Compressed Row Storage: 4
  2) Compressed Column Storage : 5
  3) Compressed Diagonal Storage: 5
  4) Jagged Diagonal Storage: 5
  Linked List: 6

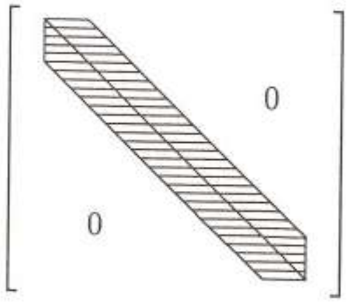
Introduction to Graph Theory 7
Direct Methods for Sparse matrices 8
Gaussian Elimination 8
Reduction to block triangular form 8
  Depth first search algorithm for traversal extension 9
  Symmetric permutations to block triangular form: 10
  Algorithm for Sargent and Westerberg 11
Tarjan’s algorithm 12
Fill in Sparse Matrices: 14
Local Strategies: 14
  Markowitz Reordering Algorithm 15
  Local minimum fill in 15
  Minimal degree Tinney/Walker reordering 16
Desired forms 16
  Cuthill and McKee Ordering 18
  Reverse Cuthill McKee reordering 19
  Refined quotient trees 19
  Hellerman – Rarick reordering 20
Partitioning Methods 21
  Solving equations using perturbation 22
  Branch tearing 23
Applications 23
References 27

Introduction to Sparse Matrix

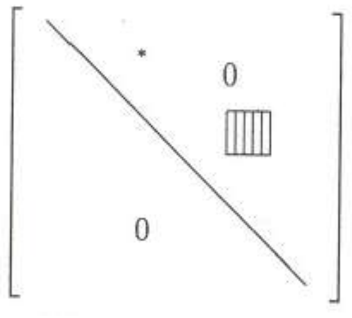
A(n×n) matrix A=[a_{ik}] is said to be a sparse matrix if only a small percentage of all matrix elements a_{ik}, i,k=1,2,3,4.5……,n, is nonzero(in practice less than 10%).



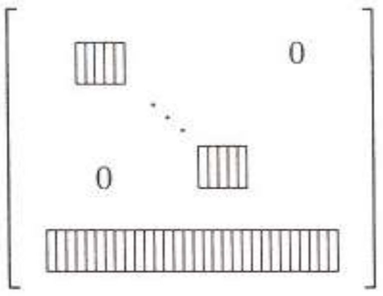
Circuit design



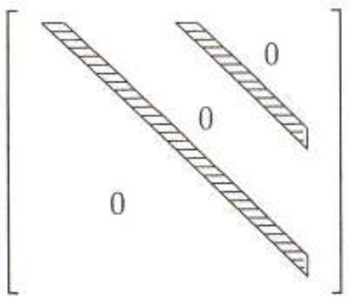
Matrix with constant bandwidth



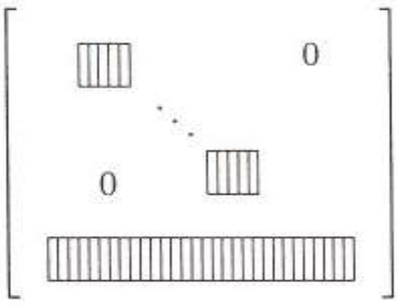
Arbitrary Sparse Matrix



Band Matrix with step



Strip Matrix



Block diagonal matrix with Margin

- The advantages of sparse matrices are size and speed.

For example, a 5000 by 5000 matrix in full storage requires space for 25 million complex numbers even if only 50,000 are nonzero. The same 50,000-nonzero element matrix, in sparse storage, would store 50,000 complex numbers and 50,000 pairs of integer indices.

Storage of Sparse matrices

A sparse matrix can be stored in full-matrix storage mode or a packed storage mode.

- When a sparse matrix is stored in full-matrix storage mode, all its elements, including its zero elements, are stored in an array.
- Packed storage mode is used to store only non zero elements in the matrix.

Advantages for avoiding to store zero elements:

- Memory usage reduction
- Decomposition is faster since you do need to access them

Data Structures

- **Static Data Structures are:**

1) Compressed Row Storage:

The compressed row storage (CRS) format puts the subsequent nonzeros of the matrix rows in contiguous memory locations. Assuming we have a nonsymmetric sparse matrix A, we create three vectors: one for floating point numbers (val) and the other two for integers (col_ind, row_ptr). The val vector stores the values of the nonzero elements of the matrix A as they are traversed in a row-wise fashion. The col_ind vector stores the column indexes of the elements in the val vector. The row_ptr vector stores the locations in the val vector that start a row.

$$A = \begin{bmatrix} 10 & 0 & 0 & 0 & -2 & 0 \\ 3 & 9 & 0 & 0 & 0 & 3 \\ 0 & 7 & 8 & 7 & 0 & 0 \\ 3 & 0 & 8 & 7 & 5 & 0 \\ 0 & 8 & 0 & 9 & 9 & 13 \\ 0 & 4 & 0 & 0 & 2 & -1 \end{bmatrix} .$$

val	10	-2	3	9	3	7	8	7	3 ···9	13	4	2	-1		
col_ind	1	5	1	2	6	2	3	4	1 ···5	6	2	5	6		

row_ptr	1	3	6	9	13	17	20

Instead of storing n2 elements, we need only 2nnz+n+1 storage locations.

2) Compressed Column Storage :

The CCS format is specified by the $\mathbf{3}$ arrays {val, row_ind, col_ptr}, where row_ind stores the row indices of each nonzero, and col_ptr stores the index of the elements in val which start a column of $\boldsymbol{A}$.

col_ptr	1	4	8	10	13	17	20

val	10	3	3	9	7	8	4	8	8 ⋯9	2	3	13	-1		
row_ind	1	2	4	2	3	5	6	3	4 ⋯5	6	2	5	6		

3) Compressed Diagonal Storage:

we can allocate for the matrix $\boldsymbol{A}$ an array val(1:n,-p:q). The declaration with reversed dimensions (-p:q,n) corresponds to the LINPACK band format, which, unlike compressed diagonal storage (CDS), does not allow for an efficiently vectorizable matrix-vector multiplication if p+q is small.

$$A = \begin{bmatrix} 10 & -3 & 0 & 0 & 0 & 0 \\ 3 & 9 & 6 & 0 & 0 & 0 \\ 0 & 7 & 8 & 7 & 0 & 0 \\ 0 & 0 & 8 & 7 & 5 & 0 \\ 0 & 0 & 0 & 9 & 9 & 13 \\ 0 & 0 & 0 & 0 & 2 & -1 \end{bmatrix} .$$

the rows of the val(:,:) array are

val(:,-1)	0	3	7	8	9	2				
val(:, 0)	10	9	8	7	9	-1				
val(:,+1)	-3	6	7	5	13	0				

4) Jagged Diagonal Storage:

The jagged diagonal storage (JDS) format can be useful for the implementation of iterative methods on parallel and vector processors. Like the CDS format, it gives a vector length of essentially the same size as the matrix. It is more space-efficient than CDS at the cost of a gather/scatter operation. All rows are padded with zeros on the right to give them equal length. Corresponding to the array of matrix elements val(:,:), an array of column indices, col_ind(:,:), is also stored.

$$\begin{bmatrix} 10 & -3 & 0 & 1 & 0 & 0 \\ 0 & 9 & 6 & 0 & -2 & 0 \\ 3 & 0 & 8 & 7 & 0 & 0 \\ 0 & 6 & 0 & 7 & 5 & 4 \\ 0 & 0 & 0 & 0 & 9 & 13 \\ 0 & 0 & 0 & 0 & 5 & -1 \end{bmatrix} \longrightarrow \begin{bmatrix} 10 & -3 & 1 & \\ 9 & 6 & -2 & \\ 3 & 8 & 7 & \\ 6 & 7 & 5 & 4 \\ 9 & 13 & & \\ 5 & -1 & & \end{bmatrix}$$

val(:, 1)	10	9	3	6	9	5
val(:, 2)	−3	6	8	7	13	−1
val(:, 3)	1	−2	7	5	0	0
val(:, 4)	0	0	0	4	0	0

col_ind(:, 1)	1	2	1	2	5	5
col_ind(:, 2)	2	3	3	4	6	6
col_ind(:, 3)	4	5	4	5	0	0
col_ind(:, 4)	0	0	0	6	0	0

- **Dynamic Data Structures**

Linked List:

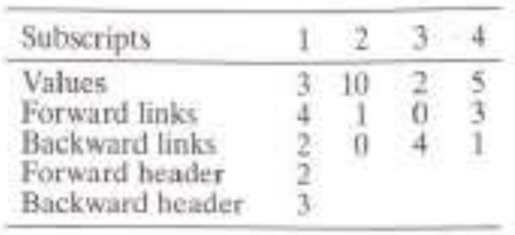
Linked list is the most efficient way to store the sparse matrices. We can easily inserting and deleting the values in the linked lists.

Example for storing linked list:

Subscripts	1	2	3	4
Values	10	3	5	2
Links	2	3	4	0
Header	1			

Table 2.8.1. Linked list for holding (10, 3, 5, 2).

Doubly linked list is more efficient than singly linked list. We can also store forward links and backward links in doubly linked list.



Subscripts	1	2	3	4
Values	3	10	2	5
Forward links	4	1	0	3
Backward links	2	0	4	1
Forward header	2			
Backward header	3			

Table 2.8.6. Doubly-linked list holding (10, 3, 5, 2).

Introduction to Graph Theory

Graph Theory is a visualize tool what is happening in sparse matrix computation. A graph A=G(V,E) is a set of nodes V and set of edges E between edges.
A directed graph or digraph consists of a set of nodes (also called vertices) and directed edges between nodes. Any square sparse matrix pattern has an associated digraph, and any digraph has an associated square sparse matrix pattern. For a given square sparse matrix A, a node is associated with each row. If *aij* is an entry, there is an edge from node *i* to node *j* in the directed graph. The more general representation of the directed graph includes self-loops on nodes corresponding to diagonal entries.
The pattern of a square sparse matrix can be represented by a graph. The use of graph theory mainly as a tool to visualize what is happening in sparse matrix computation.

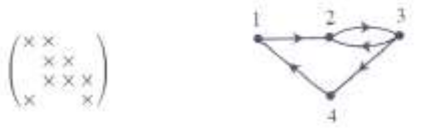


Figure 1.2.1. An unsymmetric matrix and its digraph.

For a symmetric matrix a connection from node i to node j implies that there must also be a connection from node j to node i; therefore the arrows may be dropped, and we obtain an undirected graph or graph.

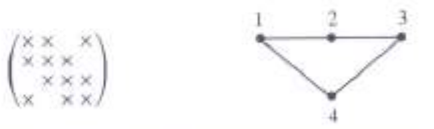


Figure 1.2.2. A symmetric matrix and its graph.

the digraph associated with the matrix A, is not a picture but is a set X of nodes and a set E of edges. An edge is an ordered pair of nodes (Xi,Xj) and is associated with the matrix entry aij.

Direct Methods for Sparse matrices

Direct methods are used to solving the sparse system of linear equations.
Advantages of direct methods are:

- High accuracy
- Easy to package
- Method of choice in many applications
- Not dramatically affected by conditioning
- Reasonably independent of structure

Gaussian Elimination

It is a basic method of solving linear system of equations.
Various methods for solving systems of *n* linear equations

$$\sum_{j=1}^{n} a_{ij}x_j = b_i, \quad i=1, 2,\ldots, n,$$

Or matrix notation,

$$Ax=b$$

The process of eliminating variables from each equation in turn and then solving for the components in reverse order is called **Gaussian elimination.** Before solving linear equations we need to reduce sparse matrix into block triangular form.

$$PAQ \rightarrow LU$$

Permutations P and Q chosen to preserve sparsity and maintain stability

L: Lower triangular (sparse)
U: Upper triangular (sparse)

SOLVE:

$$Ax = b$$

by

$$Ly = Pb$$

then

$$UQx = y$$

Reduction to block triangular form

If we are solving a set of linear equations

$$Ax = b$$

whose matrix has a block triangular form. Our purpose of this is how a given matrix may be permuted to this form, and We find it convenient to consider permutations to the block lower triangular form though with only minor modification the block upper triangular form could be obtained.
A matrix which can be permuted to the form (6.1.2), with N> 1, is said to be reducible. If no block triangular form other than the trivial one (N == 1) can be found, the matrix is called irreducible.
If A is symmetric with nonzero diagonal entries, reducibility means that A may be permuted to block diagonal form.

$$\mathbf{PAQ} = \begin{bmatrix} \mathbf{B}_{11} & & & & \\ \mathbf{B}_{21} & \mathbf{B}_{22} & & & \\ \mathbf{B}_{31} & \mathbf{B}_{32} & \mathbf{B}_{33} & & \\ \cdot & \cdot & \cdot & \ddots & \\ \mathbf{B}_{N1} & \mathbf{B}_{N2} & \mathbf{B}_{N3} & \cdots & \mathbf{B}_{NN} \end{bmatrix}.$$

In many areas produce reducible systems, including chemical process design and economic modeling Because of the speed of the block triangularization algorithms and the savings that result from their use in solving reducible systems of equations, they can be very beneficial.

To reduce sparse matrix to block triangular form, we need to perform two steps:

1) Permuted entries onto the diagonal.
2) Symmetric permutations to block triangular form.

Depth first search algorithm for traversal extension

Our algorithm provides a systematic means of permuting the rows so that the presence of entries in the first k-1 diagonal positions is preserved while an entry is moved to position (k,k). If column k has an entry in row k or beyond, say in row i, a simple interchange of rows i and k suffices.
Suppose the first entry in column k is in row j. If column j has an entry in row k or beyond, say in row p, this means that an interchange between rows p and j will preserve the (j, j) entry and move the entry in column k to row p>= k. where k=6, j=2, p=4, and the pair of interchanges (2,4) and (4,6) suffices.

$$\begin{array}{c|cccccc} & 1 & 2 & 3 & 4 & 5 & 6 \\ \hline 1 & \times & & & & & \\ 2 & & \times & & & & \times \\ 3 & & & \times & & & \\ 4 & & \times & & \times & & \\ 5 & & & & & \times & \\ 6 & & & & \times & & \end{array} \quad \text{becomes} \quad \begin{array}{c|cccccc} & 1 & 2 & 3 & 4 & 5 & 6 \\ \hline 1 & \times & & & & & \\ 4 & & \times & & \times & & \\ 3 & & & \times & & & \\ 6 & & & & \times & & \\ 5 & & & & & \times & \\ 2 & & \times & & & & \times \end{array}$$

Figure 6.3.1. Two row interchanges needed in the transversal algorithm.

In general we seek a sequence of columns c[, c2, ..• , cj with c] = k having entries in rows r1, r2, ... , rj, with r; = Ci+1' i=1,2, ... ,j-1, and rj >= k. The sequence of row interchanges (r1, r2), ... , (rj-1, rj) achieves what we need. Starting with the first entry in column k we take its row number to indicate the next column and continue letting the first off-diagonal entry in each column indicate the subsequent column. In each column we look for an entry in row k or beyond. Always taking the next column rather than trying other rows in the present column is the depth-first part

This sequence has one of the following possible outcomes:

(1) we find a column with an entry in row k or beyond.

(2) We reach a row already considered.

(3) we come to a dead end.

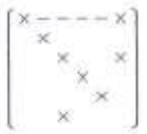


Figure 6.3.2. Encountering a dead end.

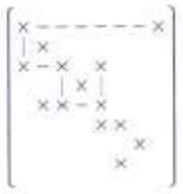


Figure 6.3.3. Reaching a row already considered.

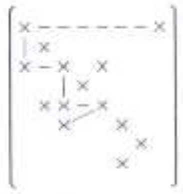


Figure 6.3.4. Returning to a previous column.

In case (1) we have the sequence of columns that we need. In case (2) We take the next entry in the current column. In case (3) we return (backtrack) to the previous column and start again with the next entry there The row interchanges for Figure 6.3.3 are (1,3), (3,5), (5,6), (6,8). The interchanges for Figure 6.3.4 are (1,3), (3,6), (6,8).

Symmetric permutations to block triangular form:

In symmetric permutations, the diagonal entries play no role so we have no need for self-loops associated with diagonal entries. Applying a symmetric permutation to the matrix causes no change in the associated digraph except for the relabeling of its nodes.

If we cannot find a closed path through all the nodes of the digraph, then we must be able to divide the digraph into two parts which are such that there is no path from the first part to the second. Renumbering the first group of nodes 1,2, ... ,k and the second group k+l, ... ,n will produce a corresponding (permuted) matrix in block lower triangular form. where there is no connection from nodes (1,2) to nodes (3,4,5). The same process may now be applied to each resulting block until no further subdivision is possible. The resulting sets of nodes corresponding to diagonal blocks are called strong components. The digraph of Figure 6.6.1 contains just two strong components, which correspond to the two irreducible diagonal blocks.

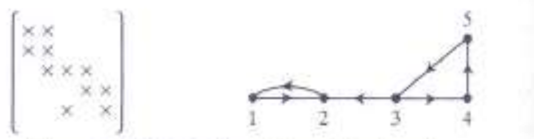


Figure 6.6.1. A 5 × 5 matrix and its digraph.

Algorithm for Sargent and Westerberg

if A is a symmetric permutation of a triangular matrix, there must be a node in its digraph from which no path leaves. This node should be ordered first in the relabeled digraph eliminating this node and all edges pointing into it leaves a remaining sub graph which again has a node from which no paths leave. Continuing in this way, we eventually permute the matrix to lower triangular form. We may start anywhere in the digraph and trace a path until we encounter a node from which no paths leave. We number the node at the end of the path first and eliminate it and all edges pointing to it from the digraph, then continue from the previous node until once again we reach a node with no path leaving it. In this way the triangular form is identified and no edge is inspected more than once.

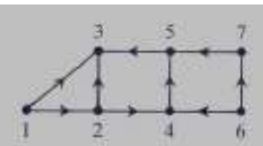
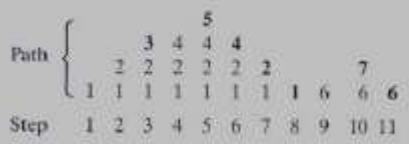


Figure 6.7.1. A digraph corresponding to a triangular matrix.



Path					**5**						
			3	4	4	**4**					
		2	2	2	2	2	**2**			**7**	
	1	1	1	1	1	1	1	**1**	6	6	**6**
Step	1	2	3	4	5	6	7	8	9	10	11

Figure 6.7.2. The sequence of paths used for the Figure 6.7.1 case, where nodes selected for ordering are shown in bold.

It may be verified (see Figure 6.7.3) that the relabeled digraph 3→ 1, 5→2, 4→ 3, 2→ 4, 1→5, 7→ 6, 6→7 corresponds to a triangular matrix.

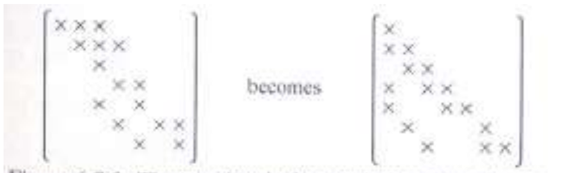


Figure 6.7.3. The matrices before and after renumbering.

Composite node is any group of nodes through which a closed path (cycle) has been found. Starting from any node, a path is followed through the digraph until

(1) a closed path is found
(2) a node or composite node is encountered with no edges leaving it.
In case (1), all the nodes on the closed path must belong to the same strong component and the digraph is modified by collapsing all nodes on the closed path into a single composite node. Edges within a composite node are ignored and edges entering or leaving any node of the composite node are regarded as entering or leaving the composite node. The path is now continued from the composite node.

Thus the blocks of the required form are obtained successively. This generalization of the triangularization algorithm shares the property that each edge of the original digraph is inspected at most once.

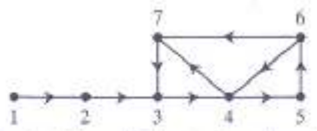


Figure 6.7.4. A digraph illustrating the algorithm of Sargent and Westerberg.

The difficulty with this approach is that there may be large overheads associated with the relabeling in the node collapsing step. A simple scheme such as labeling each composite node with the lowest label of its
constituent nodes can result in O(n2) relabeling. successive composite nodes are (4,5),(3,4,5,6), (2,3,4,5,6,7), (1,2,3,4,5,6,7,8); in general, such a digraph with n nodes will involve 2+4+6+ ... +n = n2/4+n/2 relabellings.

Tarjan’s algorithm

Same idea of previous algorithm but stack is used to record the current paths and all the closed paths.

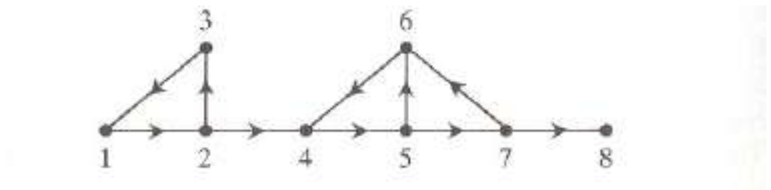
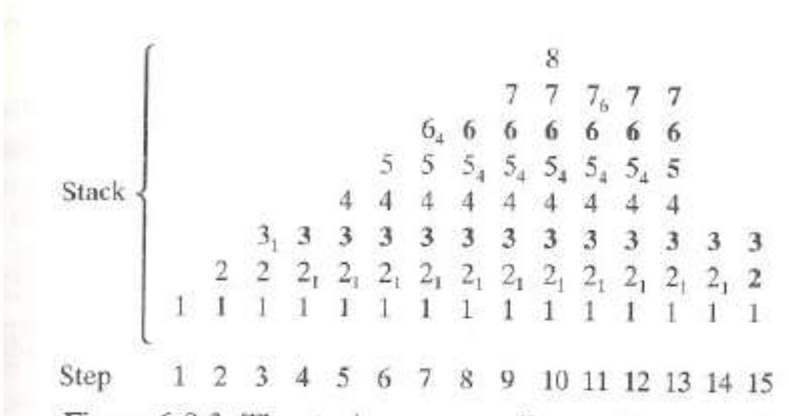


Figure 6.8.2. An example with two nontrivial strong components.



										8					
									7	7	7_6	7	7		
							6_4	6	6	6	6	6	6		
						5	5	5_4	5_4	5_4	5_4	5_4	5		
Stack					4	4	4	4	4	4	4	4	4		
			3_1	3	3	3	3	3	3	3	3	3	3	3	3
		2	2	2_1	2_1	2_1	2_1	2_1	2_1	2_1	2_1	2_1	2_1	2_1	2
	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1
Step	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15

Figure 6.8.3. The stack corresponding to Figure 6.8.2.

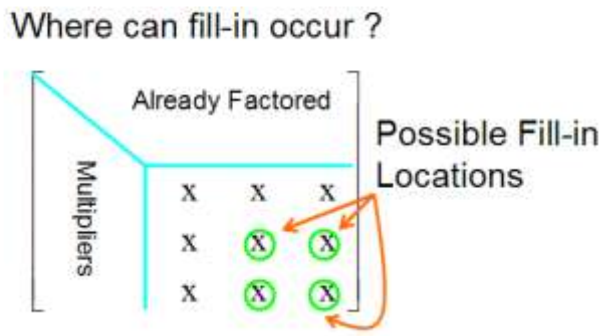
(1) The edge points to a node not on the stack, in which case this new node is added to the top of the stack and given a link that points to itself.
(2) The edge points to a node lower on the stack than the node linked from the current node, in which case the link is reset to point to this lower node.
(3) The edge points to a node higher on the stack than the node linked from the current node, in which case no action is needed.
(4) There are no unsearched edges from the current node and the link from the current node points below it. In this case the node is left on the stack but removed from the path. The link for the node before it on the path is reset to the lesser of its old value and that of the current node.
(5) There are no unsearched edges from the current node and the link from the current node does not point below it. In this case the current node and all those above it on the stack constitute a strong
component and so are removed from the stack.

- It takes O(n) costs associated with initialization and accessing the rows.

- The overall cost is O(n)+O(t) for a matrix of order n with t entries.

Fill in Sparse Matrices:

While performing Gaussian elimination, in each step there are row operations performed. During the process, the zero elements of the matrix could become non zero. This creeping in of non-zero elements in the sparse matrix is called fill in. This is shown in the example below.



To exploit the beauty of sparse matrices, the sparsity of the matrices should be preserved while solving for the system of linear equations. To preserve the sparsity, reordering of the rows and columns of the matrix is done so that the fill in is minimum.
There are many methods and algorithms used to reduce the fill in the sparse matrix. These algorithms are categorized into 2. the first is local strategies and the second is desirable forms. The type of reordering used for a particular sparse matrix, depends on the type or the structure of the matrix (band, variable band, arbitrary, etc.,)
All these algorithms follow the same basic high level steps, which are summarized below.

1. Determine the structure of the matrix.
2. Reorder rows and columns for efficiency.
3. Store numerical values in matrix and right hand side.
4. Factor the matrix.
5. Solve the system (may be repeated for multiple right hand sides).
6. Unpermute the solution.

Local Strategies:

the local strategies aim at reducing the fill in locally, which is at each step of Gaussian elimination, the fill in is reduced by reordering the rows and the columns. It is not necessary that the reduction of fill in at each would result in an overall reduction in the fill in. some of the strategies are good but none of the solutions is the best and not the most optimal.

The first algorithm considered here is Markowitz Reordering Algorithm

Markowitz Reordering Algorithm

This ordering strategy has proven o be extremely successful for general purpose use. Suppose Gaussian elimination has proceeded through first k stages. The Markowitz criterion is to select the entry a_{ij} from the (n-k)x(n-k) sub matrix that is not too small numerically.
Markowitz product $(r_i - 1) * (c_j - 1)$
For i to n

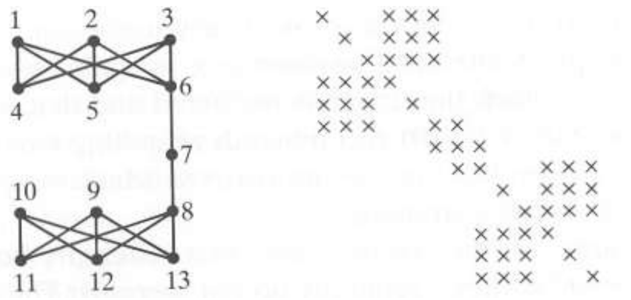
- Find diagonal j>=i with min Markowitz Product
- Swap rows j!=i and columns j!=i
- Factor the new row i and determine fill-ins

Markowitz has 5% more fill ins than the others but is faster. It is a Greedy Algorithm (but close to optimal).

Local minimum fill in

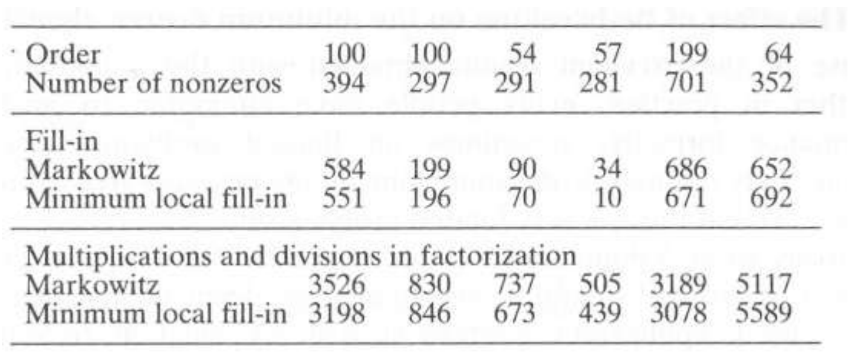
this local strategy also reduces the fill in at every stage, whether or not it reduces the fill in globally.

- In the kth stage of Gaussian elimination, select a pivot which is non zero that introduces least fill in at this stage
- Doesn't always produce minimum fill in globally



Local minimum strategy applied to this graph and the corresponding matrix

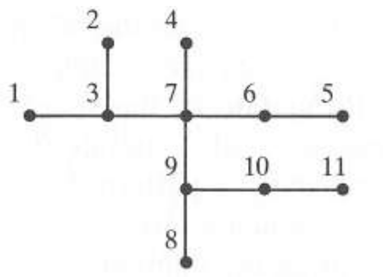
The Markowitz and minimum fill in are compared in the table below.



Order	100	100	54	57	199	64
Number of nonzeros	394	297	291	281	701	352
Fill-in						
Markowitz	584	199	90	34	686	652
Minimum local fill-in	551	196	70	10	671	692
Multiplications and divisions in factorization						
Markowitz	3526	830	737	505	3189	5117
Minimum local fill-in	3198	846	673	439	3078	5589

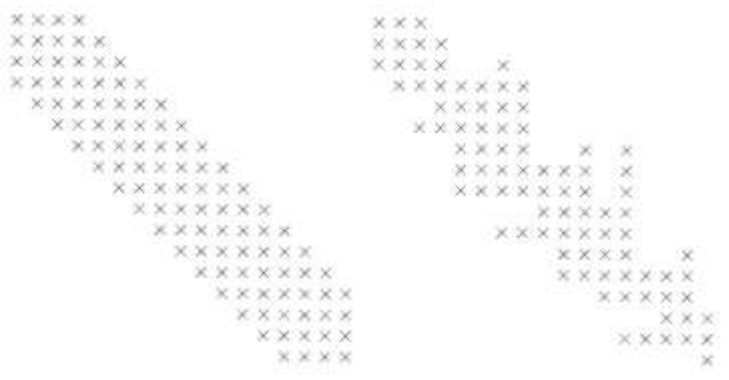
Minimal degree Tinney/Walker reordering

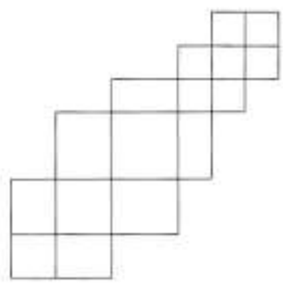
it is a special case of Markowitz for symmetric matrices. This strategy corresponds to choosing a node for the next elimination which has least edges connected to it. Least fills in for tree graphs. Diagonal entry of the minimum degree will always Markowitz count equal to the minimum for any diagonal or off diagonal entry. This property is maintained by choosing pivots from the diagonal, since symmetry is preserved. A tree graph is shown below.

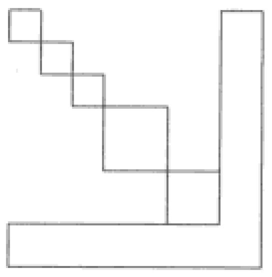


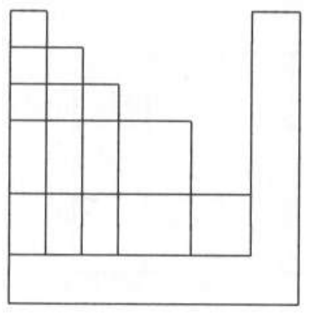
Desired forms

There are strategies to reduce fill in sparse matrices while Gaussian elimination , by forming the matrix into some desired forms, such that even on operations of Gaussian elimination, the desired form of the matrix is maintained, that is there is minimum fill in. some of the most desired forms are shown below.


Band and variable band matrices


Block tridiagonal matrix

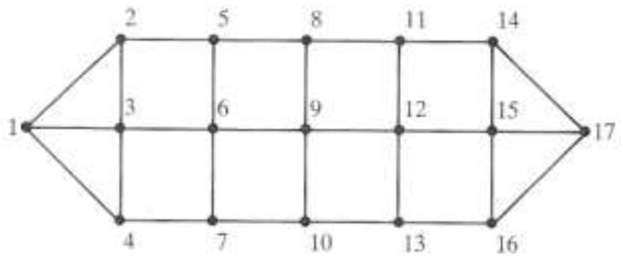

Doubly bordered block tridiagonal matrix


Bordered block tridiagonal matrix

Some of the strategies used to bring the sparse matrix to one of the desired forms are discussed below.

Cuthill and McKee Ordering

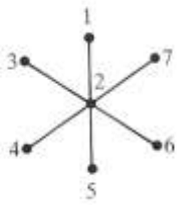
below is the graph for a sparse matrix.



The algorithm is as below.

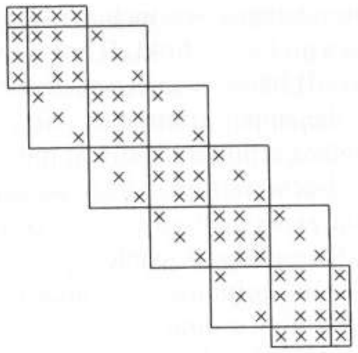
- Divide the nodes into level sets Si with S1 consisting of a single node
- S2 consists of all neighbors of this node in S1
- S3 consists of all neighbors of nodes in S2 that are not in S1 or S2

An example is shown below



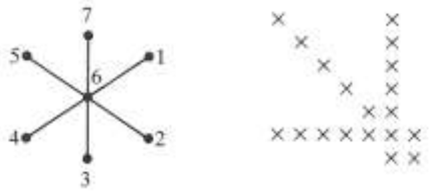
$$\begin{array}{ccccccc}
\times & \times & & & & & \\
\times & \times & \times & \times & \times & \times & \times \\
 & \times & \times & 0 & 0 & 0 & 0 \\
 & \times & 0 & \times & 0 & 0 & 0 \\
 & \times & 0 & 0 & \times & 0 & 0 \\
 & \times & 0 & 0 & 0 & \times & 0 \\
 & \times & 0 & 0 & 0 & 0 & \times
\end{array}$$

Below is the matrix form of Cuthill and McKee reordering



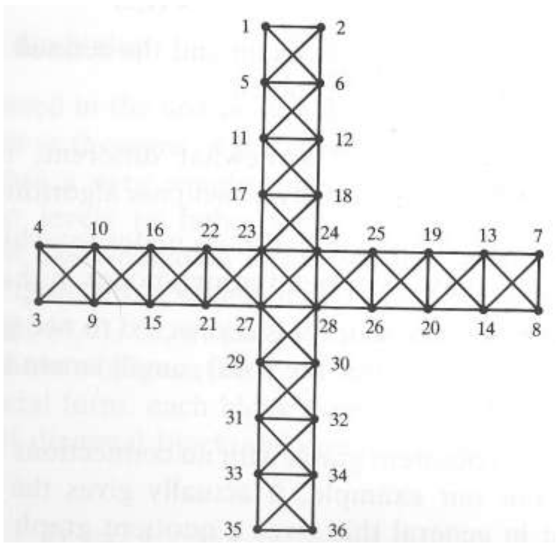
Reverse Cuthill McKee reordering

The reverse of Cuthill and McKee was proposed and said that it is better than the earlier as, on reversing the order; all these zeros move outside the form and will not need storage as shown in the figure below.

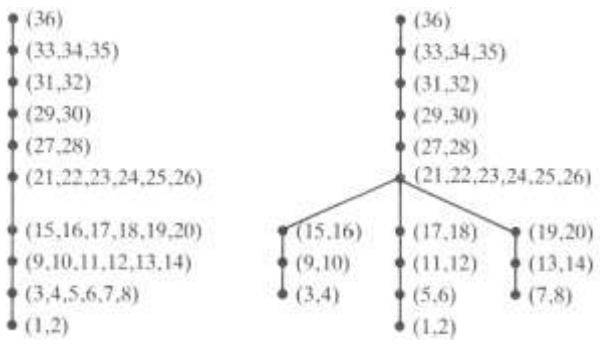


Reverse Cuthill McKee ordering
This form of ordering is better than Cuthill McKee ordering.

Refined quotient trees



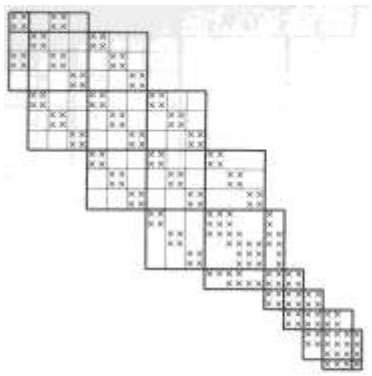
Plus shaped problem, ordered by Reverse Cuthill-McKee



Level set chain Refined quotient tree

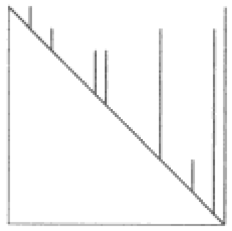
Algorithm to build refined quotient tree:

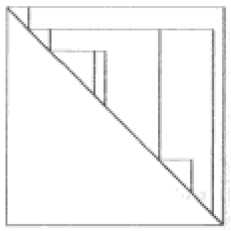
- In each level set, group together those nodes for which there is a path from any one to any other one through nodes in the level sets
- Whenever a group at one level is connected to two groups at the next higher level, amalgamate the two groups at the higher level


Block matrix corresponding to the tree

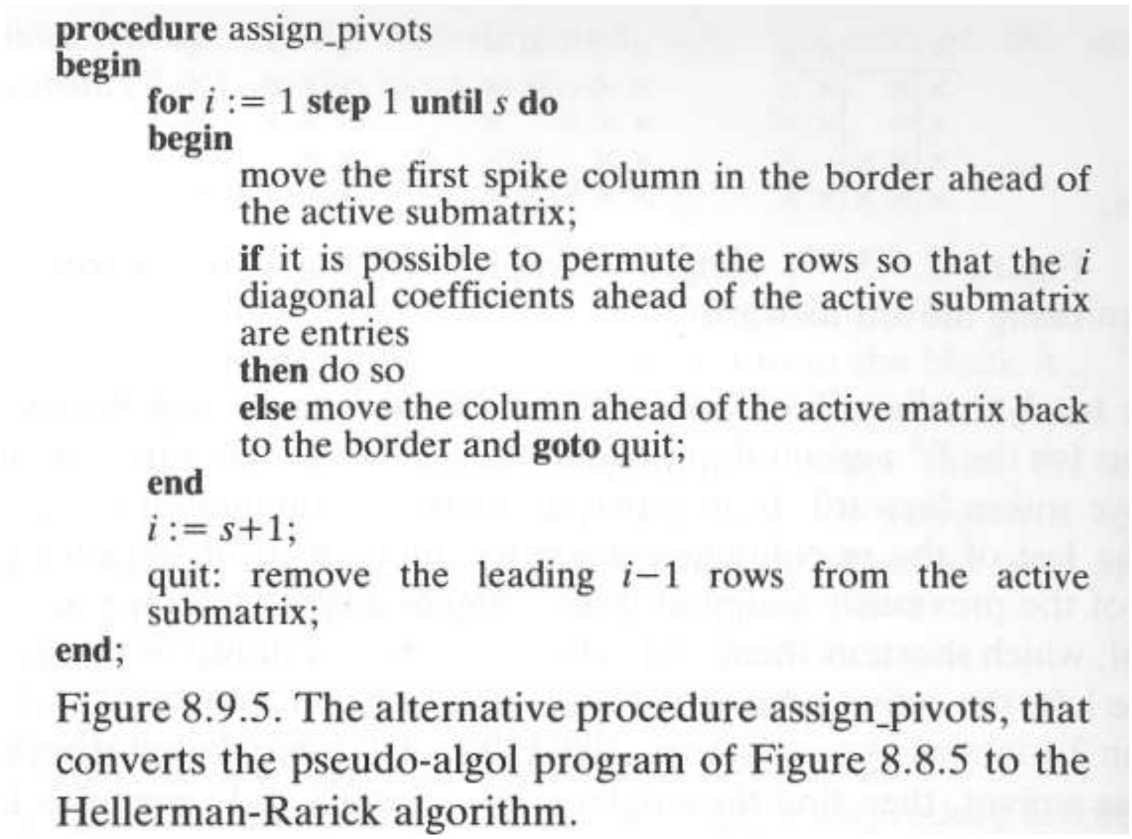
Hellerman – Rarick reordering

One of the desired forms for the sparse matrices is the spiked matrix which produces very little fill in. here the spike lies above the diagonal. Any pair of spikes has the property that the set of rows for the first is either contained in or disjoint from the set of rows for the second. The figure below shows spiked matrices.


Spiked matrix


Nested bordered block triangular form of the spiked matrix

The algorithm to assign pivots is given below.

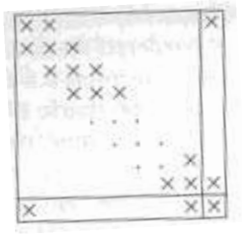


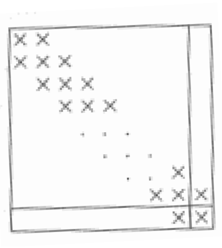
```
procedure assign_pivots
begin
        for $i$ := 1 step 1 until $s$ do
        begin
                move the first spike column in the border ahead of
                the active submatrix;
                if it is possible to permute the rows so that the $i$
                diagonal coefficients ahead of the active submatrix
                are entries
                then do so
                else move the column ahead of the active matrix back
                to the border and goto quit;
        end
        $i := s+1$;
        quit: remove the leading $i-1$ rows from the active
        submatrix;
end;
```

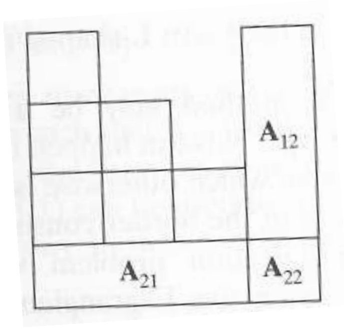
Figure 8.9.5. The alternative procedure assign_pivots, that converts the pseudo-algol program of Figure 8.8.5 to the Hellerman-Rarick algorithm.

Partitioning Methods

Till now, we have seen methods to reduce fill in the normal sized matrices. There are many systems like power systems, where large sparse matrices are encountered and we need to solve such matrices. One way to solve them is to partition them into small matrices and then solve.
Some of the partitioning methods are discussed below.
The matrix is divided into four blocks and then portioned in the following ways.

Partitioning using $\begin{pmatrix} \mathbf{L}_{11} & \\ \mathbf{L}_{21} & \mathbf{L}_{22} \end{pmatrix} \begin{pmatrix} \mathbf{U}_{11} & \mathbf{U}_{12} \\ & \mathbf{U}_{22} \end{pmatrix}$ would result in 

Partitioning using $\begin{pmatrix} \mathbf{A}_{11} & \\ \mathbf{A}_{21} & \bar{\mathbf{A}}_{22} \end{pmatrix} \begin{pmatrix} \mathbf{I} & \bar{\mathbf{U}}_{12} \\ & \mathbf{I} \end{pmatrix}$ will result in 

Partitioning using $\begin{pmatrix} \mathbf{I} & \\ \ddot{\mathbf{L}}_{21} & \mathbf{I} \end{pmatrix} \begin{pmatrix} \mathbf{A}_{11} & \mathbf{A}_{12} \\ & \ddot{\mathbf{A}}_{22} \end{pmatrix}$ will result in 

Partitioning into these block triangular matrices will help in solving the equations more easily.

Solving equations using perturbation

Using perturbation will help solve some of the equations easily. Let’s take an example. The original matrix is

$$\mathbf{A} = \begin{bmatrix} 2 & 1 & 1 & 1 & 1 & 1 \\ 1 & 2 & 1 & 1 & 1 & 1 \\ 1 & 1 & 2 & 1 & 1 & 1 \\ 1 & 1 & 1 & 2 & 1 & 1 \\ 1 & 1 & 1 & 1 & 2 & 1 \\ 1 & 1 & 1 & 1 & 1 & 2 \end{bmatrix} .$$

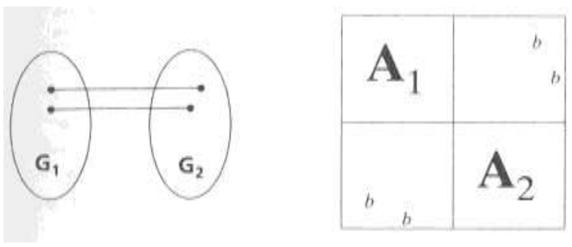
Choose $\Delta\mathbf{A} = \begin{bmatrix} 1 \\ 1 \\ 1 \\ 1 \\ 1 \\ 1 \end{bmatrix} (1)\ (1\ 1\ 1\ 1\ 1\ 1)$ such that

$$\mathbf{A} - \Delta\mathbf{A} = \begin{bmatrix} 1 & & & & & \\ & 1 & & & & \\ & & 1 & & & \\ & & & 1 & & \\ & & & & 1 & \\ & & & & & 1 \end{bmatrix}$$

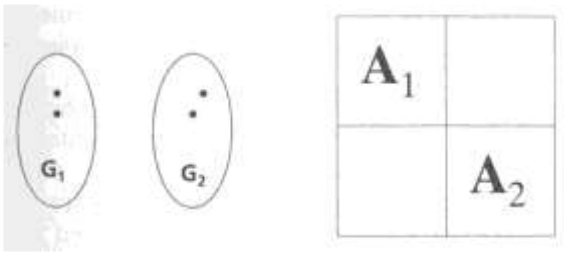
Then solving the modified matrix $\begin{bmatrix} 1 & & & & & & 1 \\ & 1 & & & & & 1 \\ & & 1 & & & & 1 \\ & & & 1 & & & 1 \\ & & & & 1 & & 1 \\ & & & & & 1 & 1 \\ 1 & 1 & 1 & 1 & 1 & 1 & -1 \end{bmatrix} \begin{bmatrix} \\ \\ \\ \mathbf{x} \\ \\ \\ \mathbf{y} \end{bmatrix} = \begin{bmatrix} \\ \\ \\ \mathbf{b} \\ \\ \\ \mathbf{0} \end{bmatrix}$ is much easier.

Branch tearing

If removing a small number of braches from a graph of the matrix yields a graph that is disconnected into parts, then it is natural to consider the problem resulting after their removal. It will be easier to solve because each part may be treated independently. This technique is called branch tearing. The figures below show how a normal graph and a disconnected graph look like.



Connected graphs



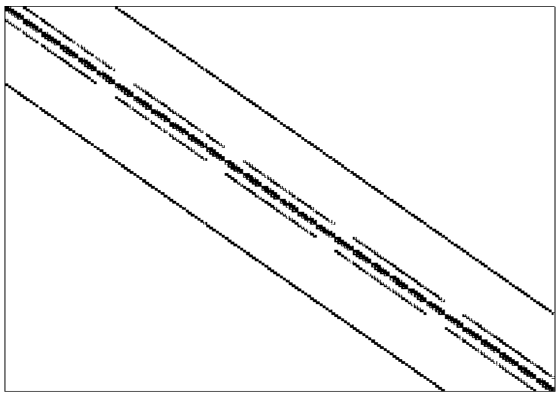
Torn apart problem

Applications

Sparse matrices are used widely almost in every field. Some of them are

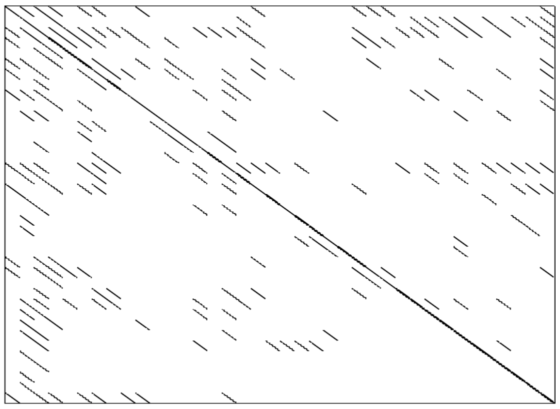
- Differential equations
- Linear Programming … Simplex
- Optimization/Nonlinear Equations
- Eigen system Solution
- Two Point Boundary Value Problems
- Least Squares Calculations

Thermal Simulation



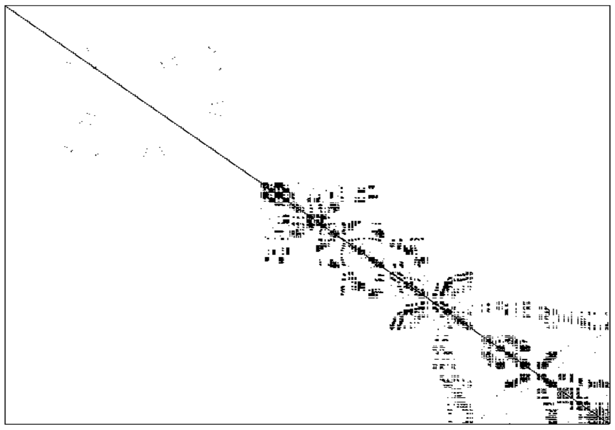
Thermal Simulation; SHERMAN2

Weather Matrix

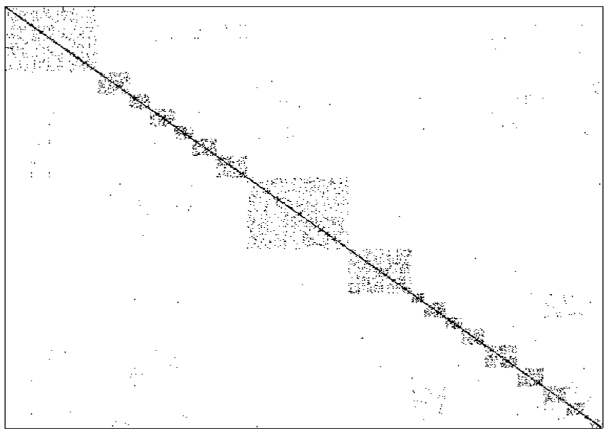


Weather Matrix; FS 760 3

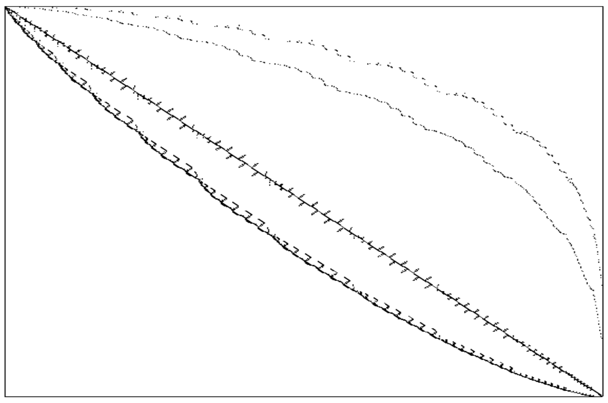
Dynamic Calculation in Structures


Dynamic Calculation in Structures; BCSSTM13

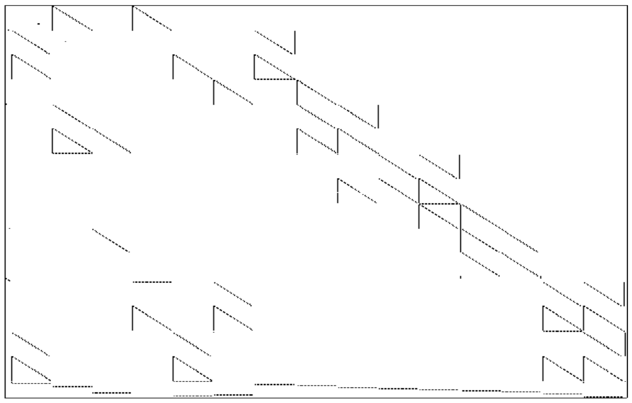
Power Systems


Power Systems; BCSPWR07

Simulation Of Computing Systems

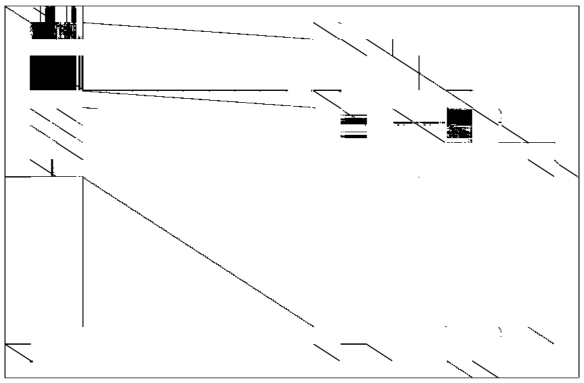

Simulation of Computing Systems; GRE 1107

Chemical Engineering



Chemical Engineering; WEST0381

Economic Modeling



Economic Modelling; ORANI678

References

[1] I.S. Duff, A.M Erisman and J.K. Reid, Direct Methods for Sparse Matrices.
[2] ftp://ftp.numerical.rl.ac.uk/pub/talks/isd.oxford.16X01.ppt.
[3] crd.lbl.gov/~xiaoye/SuperLU/SIAM_cse03.ppt
[4] ocw.mit.edu/NR/rdonlyres/Electrical-Engineering-and-Computer-Science/6-336JFall2003/81D484F6-94C3-4FB2-9838.../**lec4**.pdf