Sparsity

1. Introduction

We saw in previous notes that the very common problem, to solve for the $n\times 1$ vector $\underline{x}$ in

$$\underline{A}\underline{x} = \underline{b} \tag{1}$$

when n is very large, is done without inverting the $n\times n$ matrix $\underline{A}$, using LU decomposition.

Even though LU decomposition is much faster than matrix inversion, it is typical for many applications that this computation requires a significant percentage of the overall computation time. In addition, very large systems can tax the memory of some computers. Both of these issues are significant for the Newton-Raphson (NR) solution to the power flow problem.

The implication is that we would like to find ways to increase efficiency of LU decomposition.

One way is through recognition of a very interesting attribute of the Jacobian matrix. Let's take a look at the Jacobian expressions, copied from notes on Power Flow, but with four of the equations modified to replace P_j and Q_j with their full expressions.

➔ $$J_{jk}^{P\theta} = \frac{\partial P_j(\underline{x})}{\partial \theta_k} = |V_j||V_k|\left(G_{jk}\sin(\theta_j-\theta_k) - B_{jk}\cos(\theta_j-\theta_k)\right) \quad \text{(T7.47)}$$

$$J_{jj}^{P\theta} = \frac{\partial P_j(\underline{x})}{\partial \theta_j} = -B_{jj}|V_j|^2 - \sum_{k=1}^{N}|V_j||V_k|\left(G_{jk}\sin(\theta_j-\theta_k) - B_{jk}\cos(\theta_j-\theta_k)\right) \text{(T7.48)}$$

➔ $$J_{jk}^{Q\theta} = \frac{\partial Q_j(\underline{x})}{\partial \theta_k} = -|V_j||V_k|\left(G_{jk}\cos(\theta_j-\theta_k) + B_{jk}\sin(\theta_j-\theta_k)\right) \quad \text{(T7.49)}$$

$$J_{jj}^{Q\theta} = \frac{\partial Q_j(\underline{x})}{\partial \theta_k} = -G_{jj}|V_j|^2 + \sum_{k=1}^{N}|V_j||V_k|\left(G_{jk}\cos(\theta_j-\theta_k) + B_{jk}\sin(\theta_j-\theta_k)\right) \text{(T7.50)}$$

➔ $$J_{jk}^{PV} = \frac{\partial P_j(\underline{x})}{\partial |V_k|} = |V_j|\left(G_{jk}\cos(\theta_j-\theta_k) + B_{jk}\sin(\theta_j-\theta_k)\right) \quad \text{(T7.51)}$$

$$J_{jj}^{PV} = \frac{\partial P_j(\underline{x})}{\partial |V_j|} = G_{jj}|V_j| + \sum_{k=1}^{N}|V_k|\left(G_{jk}\cos(\theta_j-\theta_k) + B_{jk}\sin(\theta_j-\theta_k)\right) \quad \text{(T7.52)}$$

➔ $$J_{jk}^{QV} = \frac{\partial Q_j(\underline{x})}{\partial |V_k|} = |V_j|\left(G_{jk}\sin(\theta_j-\theta_k) - B_{jk}\cos(\theta_j-\theta_k)\right) \quad \text{(T7.53)}$$

$$J_{jj}^{QV} = \frac{\partial Q_j(\underline{x})}{\partial |V_j|} = B_{jj}|V_j| - \sum_{k=1}^{N}|V_k|\left(G_{jk}\sin(\theta_j-\theta_k) - B_{jk}\cos(\theta_j-\theta_k)\right) \quad \text{(T7.54)}$$

Observe that the equations marked with a dark arrow on the left are expressions for off-diagonal elements in the corresponding submatrix. Let's answer three questions about these particular elements:

Question 1: What is the numerical value of these elements for $G_{jk}=B_{jk}=0$?

Question 2: What causes $G_{jk}=B_{jk}=0$?

Question 3: What percentage of Jacobian off-diagonal elements have $G_{jk}=B_{jk}=0$?

Answer 1: These elements are zero!

Answer 2: There is no branch connecting buses j & k.

Answer 3: Typically, over 99%. Let's see why….

Consider a power system having N buses and L branches. **Let's just consider the Y-bus for now**.

It is the case that for most power network models, L is about 3 times N, i.e., $L \approx 3N$.

In the Y-bus, we know that every diagonal is *filled* (contains a non-zero number). This gives N non-zero elements.

We also know there are 2 off-diagonal elements per branch that are filled. This gives 2L non-0 elements.

Therefore the total number of non-zero elements is $T=N+2L$. But $L \approx 3N$, so $T \approx N+2(3N)=7N$.

The total number of elements is N^2. It is interesting to look at the ratio of filled elements to total elements, which is $7N/N^2=7/N$. Consider, for example:

N=500	➔7/500=1.4% filled.
N=2000	➔7/2000=0.35% filled.
N=10000	➔7/10000=0.075% filled.
N=50000	➔7/50000=0.0145% filled.

It can be expected that the P-θ submatrix of the Jacobian will have approximately the same percentage of fills (non-zero elements), since it will have the exact same structure as the Y-bus (but loss of one row/column) because of Answer 1 above.

(Answer 1 indicates that $J_{jk}^{P\theta}$ is zero when G_{jk} and B_{jk} are zero which occurs only when the corresponding element in the Y-bus is zero which occurs when there is no connection between buses j and k).

The same analysis holds for the Q-V submatrix, since it is square.

The P-V and Q-θ submatrices are not square and so we may see a slightly different ratio than 7/N. Rather than re-develop the ratio, we will just assume that the percentage of fills in these two submatrices will be about the same. It is clear that the percentage of fills in the power flow Jacobian will be small.

We refer to matrices that have a very small number of fills, or alternatively, matrices that have a very large number of zeros, as being *sparse*.

2. Storage techniques

A very important rule in sparse matrix programming is: DO NOT STORE THE ZEROS!!!!

There are at least two classes of sparsity programming methods: the entry-row-column storage method and the chained data structure method [1].

2.1 Entry-row-column storage method

In this method, three arrays are used: STO[I] contains the value of the I_{th} non-zero element, and IR[I] and IC[I] contains the row and column, respectively, of the I_{th} non-zero element. The following example illustrates.

$$\begin{bmatrix} 1 & 0 & 0 & 1 \\ 4 & 3 & 0 & 0 \\ 0 & 0 & 0 & 2 \\ 0 & 0 & 1 & 0 \end{bmatrix}$$

I	STO[I]	IR[I]	IC[I]
1	1	1	1
2	1	1	4
3	4	2	1
4	3	2	2
5	2	3	4
6	1	4	3

Observe the entries are in "row-order," i.e., they are ordered in the arrays I=1,… in the order they appear in the matrix starting in row 1, then row 2, etc. (we could also place the entries in "column order."). It is important to have some kind of order, in contrast to random storage, because it gives a basis to search for a particular matrix element.

For example, if we wanted an element in position (j,k), we could do a "golden search" (choose an element I and identify whether it is below or above element (j,k). If above, choose I/2. If below, choose 2I. Then repeat.

If the elements are stored randomly, as indicated below, then every time we wanted an element (j,k), we would have to scan the arrays until we found it. This would take significant "search time."

$$\begin{bmatrix} 1 & 0 & 0 & 1 \\ 4 & 3 & 0 & 0 \\ 0 & 0 & 0 & 2 \\ 0 & 0 & 1 & 0 \end{bmatrix}$$

I	STO[I]	IR[I]	IC[I]
1	1	1	4
2	1	4	3
3	4	2	1
4	3	2	2
5	1	1	1
6	2	3	4

It is interesting to see what the storage requirement of this scheme is like.
Recall that for power systems, we typically get about 7N non-zero elements. For every non-zero element, we must store 3 numbers (STO, IC, and IR). This

creates the need for making 3×7N=21N stores. So total number of stores is 21N, which means our storage requirement, as a percentage of the number of matrix elements (giving us a sense of what we need to store relative to what we would store if we just kept the entire matrix in memory) is $21N/N^2=21/N$. This is illustrated below.

Size	Percent of matrix filled	Percent of stores required
N=500	➔7/500=1.4% filled.	4.2%
N=2000	➔7/2000=0.35% filled.	1.05%
N=10000	➔7/10000=0.075% filled.	0.21%
N=50000	➔7/50000=0.0145% filled.	0.042%

What this says, for example (for the first case of 500 buses), is that whereas only 1.4% of the matrix size will have non-zero elements, our storage scheme will require a storage space requirement of only 4.2% of the matrix size. This is much better than storing the entire matrix!

There is one problem with the entry-row-column storage method. What if we need to insert a non-zero element into a position that was previously zero? For example, consider changing the data structures for the original matrix, shown below on the left, so that the data structures reflect the one shown on the right.

$$\begin{bmatrix} 1 & 0 & 0 & 1 \\ 4 & 3 & 0 & 0 \\ 0 & 0 & \textcircled{0} & 2 \\ 0 & 0 & 1 & 0 \end{bmatrix} \Rightarrow \begin{bmatrix} 1 & 0 & 0 & 1 \\ 4 & 3 & 0 & 0 \\ 0 & 0 & \textcircled{5} & 2 \\ 0 & 0 & 1 & 0 \end{bmatrix}$$

I	STO[I]	IR[I]	IC[I]
1	1	1	1
2	1	1	4
3	4	2	1
4	3	2	2
5	2	3	4
6	1	4	3

I	STO[I]	IR[I]	IC[I]
1	1	1	1
2	1	1	4
3	4	2	1
4	3	2	2
5	5	3	3
6	2	3	4
7	1	4	3

In the modified data structures on the right, all values "below" the inserted non-zero element at I=5 had to be "pushed down" the array structures, i.e.,

- what was STO[5], IR[5], and IC[5] has now become STO[6], IR[6], and IC[6], and
- what was STO[6], IR[6], and IC[6] has now become STO[7], IR[7], and IC[7].

This "pushing down" the array structures for an insertion takes time, and when the matrices are very large, this can take significant time (even for sparse matrices), especially when an element needs to be inserted "high" up the array structures.

We avoid this insertion time by allowing for random storage. However, in this case, we increase our search time, as described on page 6. So with this method, we either incur insertion time or search time. We avoid both problems using a chained data structure, with slightly increased memory requirements.

2.2 Chained data structure method

The so-called chained-data structure method uses four arrays.

- STO[I]: Contains value of Ith non-zero element.
- IC[I]: Contains col number of Ith non-zero element.
- NEXT[I]: Points to the location in STO and IC where the next non-zero element is located for the row. It will be I+1 unless STO[I] is the last non-zero element in the row, in which case it is zero.
- LOC[K]: Points to location in STO and IC where the first non-zero element is stored for row K.

Let's illustrate in what follows. Consider the matrix:

$$\begin{bmatrix} 1 & 0 & 0 & 1 \\ 4 & 3 & 0 & 0 \\ 0 & 0 & 0 & 2 \\ 0 & 0 & 1 & 0 \end{bmatrix}$$

I	STO[I]	IC[I]	NEXT[I]
1	1	1	2
2	1	4	0
3	4	1	4
4	3	2	0
5	2	4	0
6	1	3	0

K	LOC[K]
1	1
2	3
3	5
4	6

NEXT[I] signals you (with a zero) where the current row ends. LOC[K] tells you where in STO and IC row K begins.

A common need is to obtain the element in position (j,k). For example, lets obtain the element in position (2,3).

One does that using the above in the following way:

- LOC[2]=3 indicates the first non-zero element in row 2 is stored as element I=3 in STO[I], IC[I], & NEXT[I].
- IC[3]=1 indicates the column number of the third element in STO, IC, and NEXT is 1.
- 1<3? tests to see if this column (column 1) is before the column of interest (column 3).
- Since answer to 1<3? is "yes," then get NEXT[3]=4. This indicates that this row has another non-zero element, and it is stored in location I=4 of STO[I], IC[I], and NEXT[I].
- IC[4]=2 indicates that the column number of the fourth element in STO, IC, and NEXT is 2.
- 2<3? tests to see if this column (column 2) is before the column of interest (column 3).
- Since answer to 2<3? is "yes," then get NEXT[4]=0. This indicates that there are no more non-zero elements in this row. Therefore, we know that the element in position (2, 3) must be zero.

One can add information to decrease computation, but it will be at the expense of additional memory,

showing a typical tradeoff between computation and memory. However, if the matrix is sparse, the above search on a row is very fast, since typically there will only be a very few non-zero elements in each row.
It is interesting to see what the storage requirement of this scheme is like.

Recall that for power systems, we typically get about 7N non-zero elements.

For every non-zero element, we must store 3 numbers (STO, IC, and NEXT). This creates the need for making 3×7N=21N stores.

Then LOC creates one store for every row so that we will have from here the need for making N stores.

So total number of stores is 21N+N=22N, which means our storage requirement, as a percentage of the number of matrix elements (giving us a sense of what we need to store relative to what we would store if we just kept the entire matrix in memory) is $22N/N^2=22/N$. This is illustrated below.

Size	Percent of matrix filled	Percent of stores required
N=500	➔7/500=1.4% filled.	4.4%
N=2000	➔7/2000=0.35% filled.	1.1%
N=10000	➔7/10000=0.075% filled.	0.22%
N=50000	➔7/50000=0.0145% filled.	0.044%

What this says, for example (for the first case of 500 buses) is that whereas only 1.4% of the matrix will have non-zero elements, our storage scheme will require a storage space requirement of only 4.4% of the matrix size. This is much better than storing the entire matrix!

3. Optimal ordering

Recall the algorithm for LU decomposition.
The algorithm is as follows:

1. Perform Gaussian elimination on $\underline{A}$. Let i=1. In each repetition below, row i is the *pivot row* and a_{ii} is the *pivot*.
    a. $L_{ji}=a_{ji}$ for j=i,…,n.
    b. Divide row i by a_{ii}.
    c. If [i=n, go to 2] else [go to d].
    d. Eliminate all a_{ji}, j=i+1,…,n. This means to make all elements directly beneath the pivot equal to 0 by adding an appropriate multiple of the pivot row to each row beneath the pivot.
    e. i=i+1, go to a.
2. The matrix U is what remains.

Observe that Step 1d consists of multiplying row i, the pivot row, by an appropriate constant and then adding it to row j. The constant is always $-a_{ji}$, so that element (j,i) is annihilated (changed to zero). We call this operation of row multiplication and addition a "row operation."

General speed-up strategy: Minimize the number of row operations.

Fact: Given pivot row i, row operations are only necessary if $a_{ji} \neq 0$.

<u>Speed-up approach #1</u>:
Test each a_{ji}. If 0, go to next row without performing row operation.

This is effective, but very obvious. Can we do better? Let's make two definitions:

<u>Lower Triangle (LT):</u> The portion of the matrix below the diagonal, illustrated in Fig. 1.

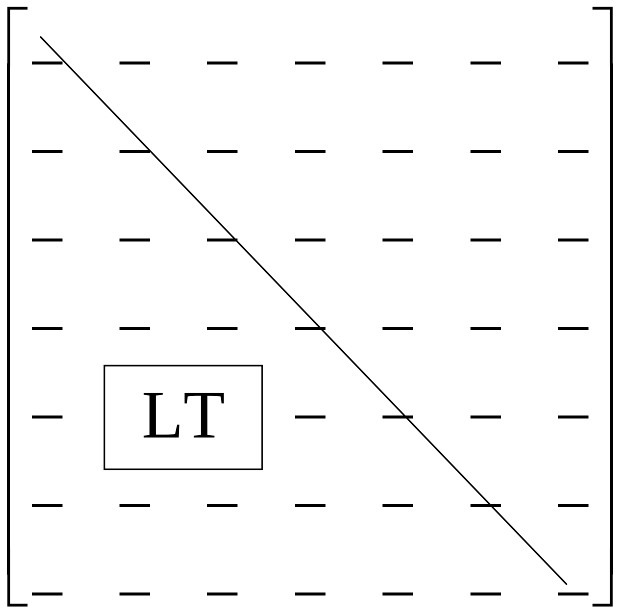

Fig. 1

Remaining Lower Triangular Element (RLTE): This is an element in the LT but to the right of the ith column when row i is the pivot row, as illustrated in Fig. 2. An RLTE can be denoted a_{kl}, where $k,l > i$.

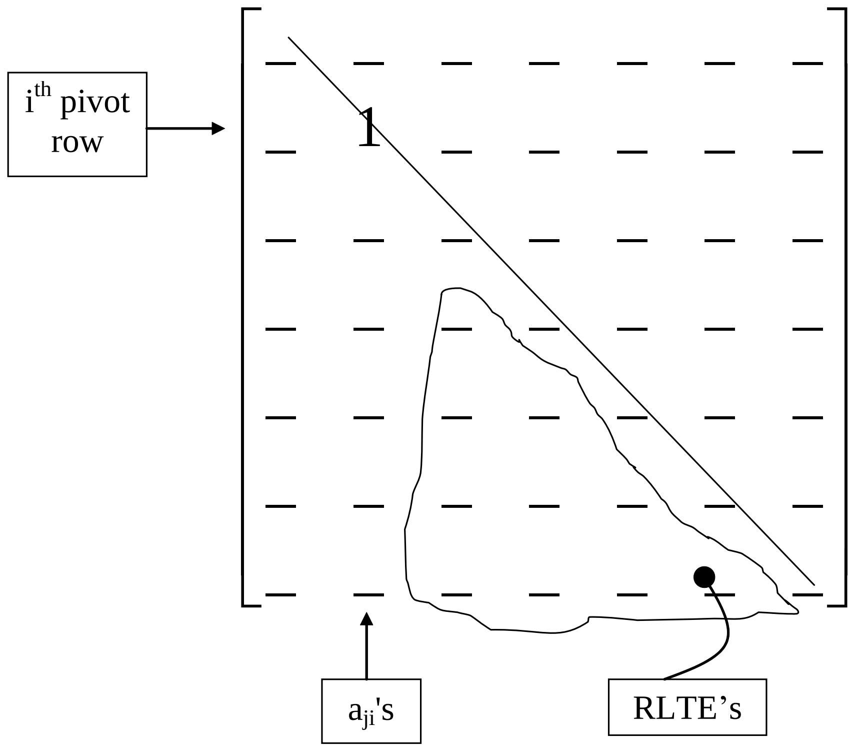

Fig. 2

Observe that

- The RLTE's are the future a_{ji}'s. To take advantage of Speedup approach #1, we want as many of these elements as possible to be zero.
- If RLTE a_{kl} is initially zero (before the Gaussian elimination process is started), it could become non-zero during a row operation. We call such elements "fill-ups". This will add a future row operation. An illustration of such a case is below.

$$3x_1 + 3x_2 + 6x_3 + 9x_4 = 1$$
$$x_1 + 3x_2 + 4x_3 + x_4 = 3$$
$$x_1 + 2x_2 + 5x_3 + 6x_4 = 2$$
$$x_1 \qquad\qquad - x_3 + 4x_4 = 1$$

$$\begin{bmatrix} 3 & 3 & 6 & 9 \\ 1 & 3 & 4 & 1 \\ 1 & 2 & 5 & 6 \\ 1 & \enclose{circle}{0} & -1 & 4 \end{bmatrix}$$

Divide first row by 3 and then add multiples of it to remaining rows so that first element in remaining rows gets zeroed.

$$\begin{bmatrix} 1 & 1 & 2 & 3 \\ 0 & 2 & 2 & -2 \\ 0 & 1 & 3 & 3 \\ 0 & \enclose{circle}{-1} & -3 & 1 \end{bmatrix}$$

Observe that the element in row 4, col 2 (circled) was a zero in the original matrix but became a “fill” (and therefore a “fill-up”). We just created an extra row operation for ourselves!

How could we have avoided this situation?

What if we could have started so that that original zero was in the first column, as shown below?

$$3x_2 + 3x_1 + 6x_3 + 9x_4 = 1$$

$$3x_2 + x_1 + 4x_3 + x_4 = 3$$

$$2x_2 + x_1 + 5x_3 + 6x_4 = 2$$

$$x_1 - x_3 + 4x_4 = 1$$

$$\begin{bmatrix} 3 & 3 & 6 & 9 \\ 3 & 1 & 4 & 1 \\ 2 & 1 & 5 & 6 \\ 0 & 1 & -1 & 4 \end{bmatrix}$$

Observe that we just exchanged the first two columns, which is equivalent to interchanging the order of variables x_1 and x_2.

Now divide first row by 3 and then add multiples of it to remaining rows so that first element in remaining rows gets zeroed.

Notice that we only have to perform two row operations in this case because the last element (row 4) is already zero! So we preserved our elimination of a row operation caused by a zero element!

$$\begin{bmatrix} 1 & 1 & 2 & 3 \\ 0 & -2 & -2 & -8 \\ 0 & -1 & 1 & 0 \\ 0 & 1 & -1 & 4 \end{bmatrix}$$

So what is the general approach to take here?

<u>Speed-up approach #2</u>:

➔Minimize row operations.

➔Minimize row operations by minimizing fill-ups.

➔Minimize fill-ups by pushing zero elements in LT to left-hand side of matrix. This will effectively move them away from positions where a row operation can affect them.

We will attempt to accomplish this by ordering the buses in certain intelligent ways. There are several ways of doing so. Here is the first way.

<u>Optimal Ordering Scheme #1 (OOS1)</u>:

The nodes of a network are ordered in such a way that a lower numbered node has less or equal number of *actual* adjacent branches than any higher numbered node. (An "adjacent branch" is an interconnection with another bus).

We will illustrate the power of this method. But to do so, we will need another concept.

Symbolic factorization: In symbolic factorization, we identify only the extent to which the Gaussian Elimination procedure produces fill-ups, but we do not actually compute the numbers (and therefore no floating point operations!).

Given pivot row i, we assume that "we never get lucky," in that non-zero-elements *always* result in position (j,k), j,k>i, from row operations having

- non-zero element in position (i,k) of pivot row i and/or
- non-zero element in position (j,k) before the pivot operation.

It is possible that if position (i,k) of pivot row i *and* position (j,k) are both non-zero, that they could sum to zero and therefore produce a zero element in position (j,k) after the row operation, but we will assume in our symbolic factorization procedure that "being lucky" in this sense cannot happen.

Example: Consider the following power system.

1. Identify fills and total number of row operations if the power system is numbered as shown.
2. Identify fills and total number of row operations if the power system is numbered using OOS1.

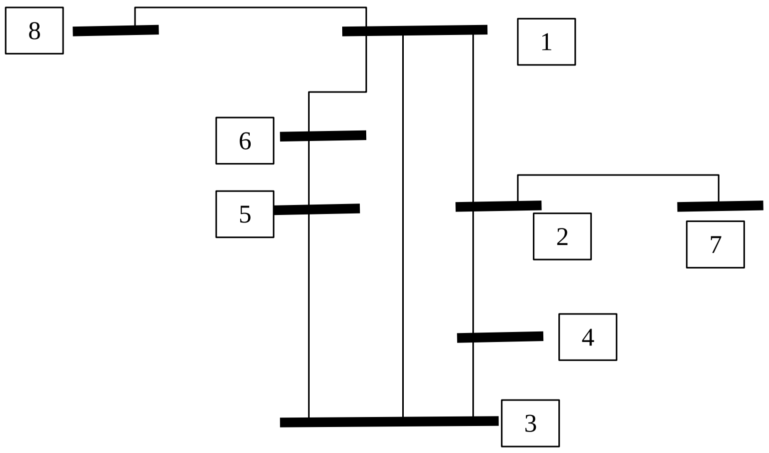

Fig. 3

Table 1: Fills

	1	2	3	4	5	6	7	8
1	X	X	X			X		X
2	X	X		X			X	
3	X		X	X	X			
4		X	X	X				
5			X		X	X		
6	X				X	X		
7		X					X	
8	X							X

Now perform the symbolic factorization and count the number of row operations and fills for each pivot. We place an “Fk” for each fill-up produced by a row operation based on pivot row k.

Starting with pivot row 1, we see it requires 4 row operations since there are 4 non-0 elements beneath position (1,1). Performing symbolic factorization, we see that pivot row 1 produces 12 fill-ups.

Table 2: Fills & Fill-ups produced by pivot row 1

	1	2	3	4	5	6	7	8
1	X	X	X			X		X
2	X	X	F1	X		F1	X	F1
3	X	F1	X	X	X	F1		F1
4		X	X	X				
5			X		X	X		
6	X	F1	F1		X	X		F1
7		X					X	
8	X	F1	F1			F1		X

Moving to pivot row 2, we see that it requires 5 row operations since there are 5 non-0 elements beneath position (2,2). Performing the symbolic factorization, we see that pivot row 2 will produce 12 fill-ups.

Table 3: Fills and Fill-ups produced by pivot row 2

	1	2	3	4	5	6	7	8
1	X	X	X			X		X
2	X	X	F1	X		F1	X	F1
3	X	F1	X	X	X	F1	F2	F1
4		X	X	X		F2	F2	F2
5			X		X	X		
6	X	F1	F1	F2	X	X	F2	F1
7		X	F2	F2		F2	X	F2
8	X	F1	F1	F2		F1	F2	X

Continuing in this manner, we show the complete symbolic factorization in the table below.

Table 4: Fills and Fill-ups produced by all pivot rows

	1	2	3	4	5	6	7	8
1	X	X	X			X		X
2	X	X	F1	X		F1	X	F1
3	X	F1	X	X	X	F1	F2	F1
4		X	X	X	F3	F2	F2	F2
5			X	F3	X	X	F3	F3
6	X	F1	F1	F2	X	X	F2	F1
7		X	F2	F2	F3	F2	X	F2
8	X	F1	F1	F2	F3	F1	F2	X

The total number of row operations is summarized in Table 5.

Table 5: Summary for Random Numbering

Pivot	Row ops	Fill-ups
1	4	12
2	5	12
3	5	6
4	4	0
5	3	0
6	2	0
7	1	0
TOTAL	24	

Now renumber according to OOS1, shown in Fig. 4, and re-perform the symbolic factorization.

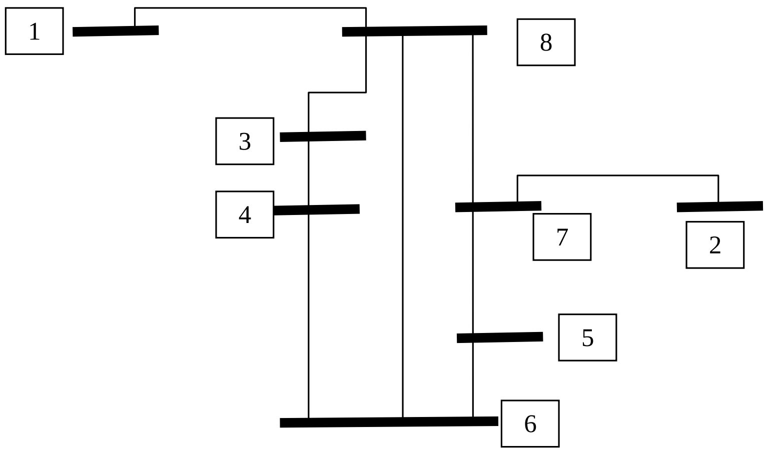

Fig. 4

Table 6: Fills and Fill-ups for OOS1

	1	2	3	4	5	6	7	8
1	X							X
2		X					X	
3			X	X				X
4			X	X		X		F3
5					X	X	X	
6				X	X	X	F5	X
7		X			X	F5	X	X
8	X		X	F3		X	X	X

The total number of row operations is summarized in Table 7.

Table 7: Summary for OOS1

Pivot	Row ops	Fill-ups
1	1	0
2	1	0
3	2	2
4	2	0
5	2	1
6	2	0
7	1	0
TOTAL	11	

In comparing Table 5 to Table 7, one notes that the OOS1 requires only 11 row operations whereas random numbering requires 24 row operations. OOS1 provides us with a significant computational benefit!

======================================

There is an improvement that can be performed which is based on the following observation:

> If the original matrix is symmetric, then fill-ups are produced symmetrically, that is, if a fill-up is produced in location (j,k), then a fill-up will be produced in location (k,j). Therefore, we may view fill-ups to represent new branches (keeping in mind that these new branches are fictitious).

One can observe the above from Table 6 (repeated below), where we see that, if (3,8) fills-up (4,8), then we know:

We know (4,3) must be a fill, otherwise the row operation that filled (4,8) would not have been necessary

- (3,8) is a fill ➔(by symmetry) (8,3) is a fill
- (4,3) is a fill ➔(by symmetry) (3,4) is a fill
- (8,4) is 0 (because (4,8) just got filled so it must have been 0 previously, then by symmetry (8,4) is 0)

If (8,3) is a fill, then we will have to eliminate it using pivot row 3. When we do this, (3,4) will fill-up (8,4).

	1	2	3	4	5	6	7	8
1	X							X
2		X					X	
3			X	X				X
4			X	X		X		F3
5					X	X	X	
6				X	X	X	F5	X
7		X			X	F5	X	X
8	X		X	F3		X	X	X

This leads us to optimal ordering scheme #2.

Optimal Ordering Scheme #2 (OOS2):

The nodes of a network are ordered in such a way that a lower numbered node has less or equal number of *actual or fictitious* adjacent branches than any higher numbered node.

Compare to OOS1:

Optimal Ordering Scheme #1 (OOS1):

The nodes of a network are ordered in such a way that a lower numbered node has less or equal number of *actual* adjacent branches than any higher numbered node.

The main difference between these schemes is that OOS1 does one ordering at the beginning based only on number of actual branches for each node, whereas OOS2 also does a check for re-ordering at the beginning of each elimination.
➔Let’s see how it works for our previous example.

Repeating Table 6 below, which shows fill-ups for OOS1, we see that row 4 has a total of 4 non-zero elements whereas row 5 has only 3 non-zero elements. Thus, we conclude that node 4 is connected to three other buses whereas node 5 is connected to only two other buses. So let’s exchange these two rows, giving the matrix of Table 8.

Table 7: Fills and Fill-ups for OOS1

	1	2	3	4	5	6	7	8
1	X							X
2		X					X	
3			X	X				X
4			X	X		X		F3
5					X	X	X	
6				X	X	X	F5	X
7		X			X	F5	X	X
8	X		X	F3		X	X	X

Table 8 shows the re-ordering and subsequent symbolic factorization.

Table 8: Fills and Fill-ups for OOS2

	1	2	3	5	4	6	7	8
1	X							X
2		X					X	
3			X		X			X
5				X		X	X	
4			X		X	X		F3
6				X	X	X	F5	X
7		X		X		F5	X	X
8	X		X		F3	X	X	X

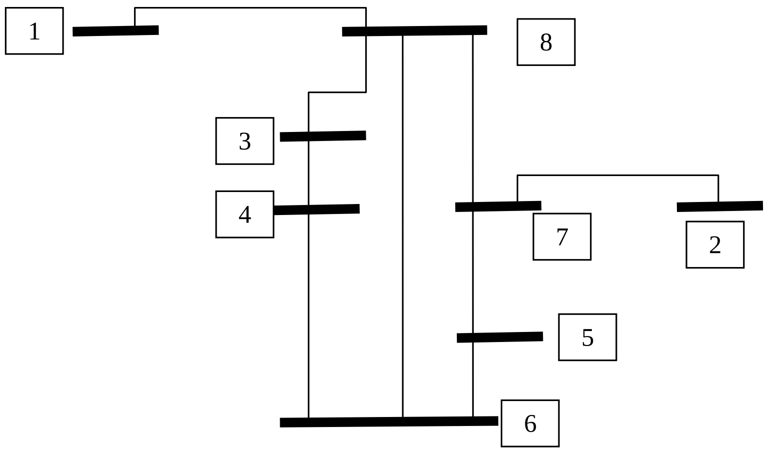

Table 7: Summary for OOS2

Pivot	Row ops	Fill-ups
1	1	0
2	1	0
3	2	2
5	2	2
4	2	0
6	2	0
7	1	0
TOTAL	11	

In this case, the row operations were the same, but for larger systems, OOS2 outperforms OOS1.

There is a third approach, OOS3, but it has been shown to be more complex without much speed improvement.

This work comes from W. Tinney's famous paper [2] on sparsity. The work has been heavily adopted in many disciplines, e.g., see the VLSI text [3]. The work has also been significantly expanded since then, for example, see [4].

Also, you should be aware of the multi-frontal method, as explained in the paper by Khaitan and McCalley (posted on webpage).

---

[1] G. Heydt, "Computer analysis methods for power systems," McMillian, 1986,
[2] W. Tinney and J. Walker, "Direct Solutions of sparse Network equations by Optimally Ordered triangular Factorization," Proceedings of the IEEE, Vol, 55, No. 11, Nov., 1967.
[3] Vlach and Singhal, "Computer Methods for Circuit Analysis and Design," 2nd edition, 1994.
[4] S. Khaitan, "On-line cascading event tracking and avoidance decision support tool," PhD dissertation, Iowa State University, 2008.