ECEN 615
Methods of Electric Power Systems Analysis

Lecture 9: Sparse Systems, Advanced Power Flow

Prof. Tom Overbye
Dept. of Electrical and Computer Engineering
Texas A&M University
overbye@tamu.edu

TEXAS A&M UNIVERSITY

Announcements

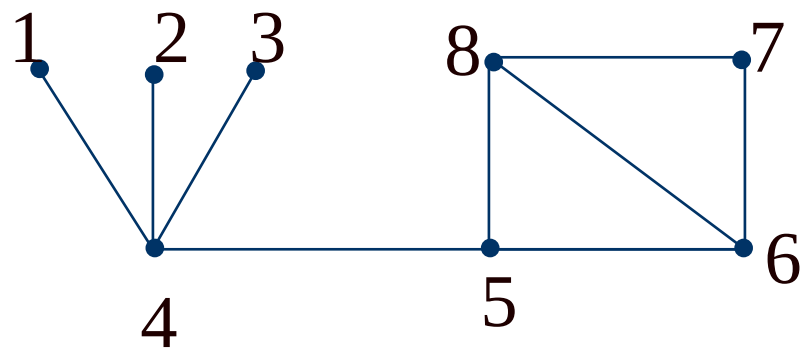
- Homework 3 should be done before the first exam but need not be turned in
- Start reading Chapter 7 (the term reliability is now often used instead of security)
- First exam is in class on Thursday Oct 1
  - Distance learning students do not need to take the exam during the class period
  - Closed book, notes. One 8.5 by 11 inch notesheet and calculators allowed
  - Last's years exam is available in Canvas

Tinney Scheme 2

- The Tinney Scheme 2 usually combines adding the fills with the ordering in order to update the valence on-the-fly as the fills are added
- As before the nodes are chosen based on their valence, but now the valence is the actual valence they have with the added lines (fills)
  - This is also known as the Minimum Degree Algorithm (MDA)
  - Ties are again broken using the lowest node number
- This method is quite effective for power systems, and is highly recommended; however it is certainly not guaranteed to result in the fewest fills (i.e. not optimal)

Tinney Scheme 2 Example

- Consider the previous network:



- Nodes 1,2,3 are chosen as before. But once these nodes are eliminated the valence of 4 is 1, so it is chosen next. Then 5 (with a new valence of 2 tied with 7), followed by 6 (new valence of 2), 7 then 8.

Coding Tinney 2

- The following slides show how to code Tinney 2 for an n by n sparse matrix **A**
- First we setup linked lists grouping all the nodes by their original valence
- vcHead is a pointer vector [0..mvValence]
  - If a node has no connections its incidence is 0
  - Theoretically mvValence should be n-1, but in practice a much smaller number can be used, putting nodes with valence values above this into the vcHead[mvValence] is

Coding Tinney 2, cont.

- Setup a boolean vectors chosenNode[1..n] to indicate which nodes are chosen and BSWR[1..n] as a sparse working row; initialize both to all false
- Setup an integer vector rowPerm[1..n] to hold the permuted rows; initialize to all zeros
- For i := 1 to n Do Begin
  - Choose node from valence data structure with the lowest current valence; let this be node k
    - Go through vcHead from lastchosen level (last chosen level may need to be reduced by one during the following elimination process;
  - Set rowPerm[i] = k; set chosenNode[k] = true

Coding Tinney 2, cont.

- Modify sparse matrix **A** to add fills between all of k's adjacent nodes provided
  1. a branch doesn't already exist
  2. both nodes have not already been chosen (their chosenNode entries are false)
  - These fills are added by going through each element in row k; for each element set the BSWR elements to true for the incident nodes; add fills if a connection does not already exist (this requires adding two new elements to **A**)
- Again go through row k updating the valence data structure for those nodes that have not yet been chosen
  - These values can either increase or go down by one (because of the elimination of node k)

Coding Tinney 2, cont.

- This continues through all the nodes; free all vectors except for rowPerm
- At this point in the algorithm the rowPerm vector contains the new ordering and matrix **A** has been modified so that all the fills have been added
  - The order of the rows in **A** has not been changed, and its columns are no longer sorted

Coding Tinney 2, cont

- Sort the rows of **A** to match the order in rowPerm
  - Surprising sorting **A** is of computational order equal to the number of elements in **A**
    - Go through **A** putting its elements into column linked lists; these columns will be ordered by row
    - Then through the columns linked lists in reverse order given by rowPerm
      - That is For i := n downto 1 Do Begin
        p1 := TSparmatLL(colHead[rowPerm[i]).Head;
        ….
- That's it – the matrix **A** is now readying for factoring
- Pivoting may be required, but usually isn't needed in the power flow

Some Example Values for Tinney 2

Number of buses	Nonzeros before fills	Fills	Total nonzeros	Percent nonzeros
37	63	72	135	9.86%
118	478	168	646	4.64%
18,190	64,948	31,478	96,426	0.029%
62,605	228,513	201,546	430,059	0.011%

Tinney Scheme 3

- "Number the rows so that at each step of the process the next row to be operated upon is the one that will introduce the fewest new nonzero terms."
- "If more than one row meets this criterion, select any one. This involves a trial simulation of every feasible alternative of the elimination process at each step. Input information is the same as for scheme 2)."
- Tinney 3 takes more computation and in general does not give fewer fills than the quicker Tinney 2
- Tinney got into the NAE in 1998

These are direct quotes from the Tinney-Walker 1967 IEEE Proceedings Paper

Sparse Forward Substitution with a Permutation Vector

Pass in **b** in bvector

```
For i := 1 to n Do Begin
   k = rowPerm[i];  // this is the only change, except using k
   p1 := rowHead[k];  // the row needs to be ordered correctly!
   While p1 <> rowDiag[k] Do Begin
      bvector[k] = bvector[k] – p1.value*bvector[p1.col];
      p1 := p1.next;
    End;
  End;
```

Sparse Backward Substitution with Permutation Vector

Pass in **b** in bvector

```
For i := n downto 1 Do Begin
   k = rowPerm[i];
   p1 := rowDiag[k].next;
   While p1 <> nil Do Begin
      bvector[k] = bvector[k] – p1.value*bvector[p1.col];
      p1 := p1.next;
    End;
    bvector[k] := bvector[k]/rowDiag[k].value;
  End;
```

- Note, numeric problems such as matrix singularity are indicated with rowDiag[k].value being zero!

Sparse Vector Methods

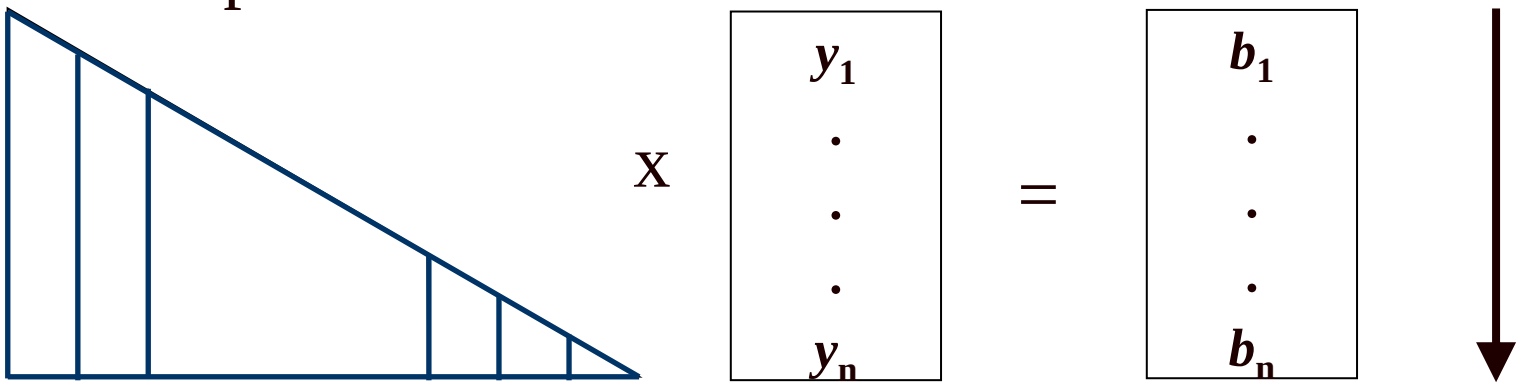
- Sparse vector methods are useful for cases in solving **Ax=b** in which
  - **A** is sparse
  - **b** is sparse
  - only certain elements of **x** are needed
- In these right circumstances sparse vector methods can result in extremely fast solutions!
- A common example is to find selected elements of the inverse of **A**, such as diagonal elements.

Sparse Vector Methods

- Often times multiple solutions with varying **b** values are required
  - **A** only needs to be factored once, with its factored form used many times
- Key reference is

W.F. Tinney, V. Brandwajn, and S.M. Chan, "Sparse Vector Methods", *IEEE Transactions on Power Apparatus and Systems*, vol. PAS-104, no. 2, February 1985, pp. 295-300

Sparse Vector Methods Introduced

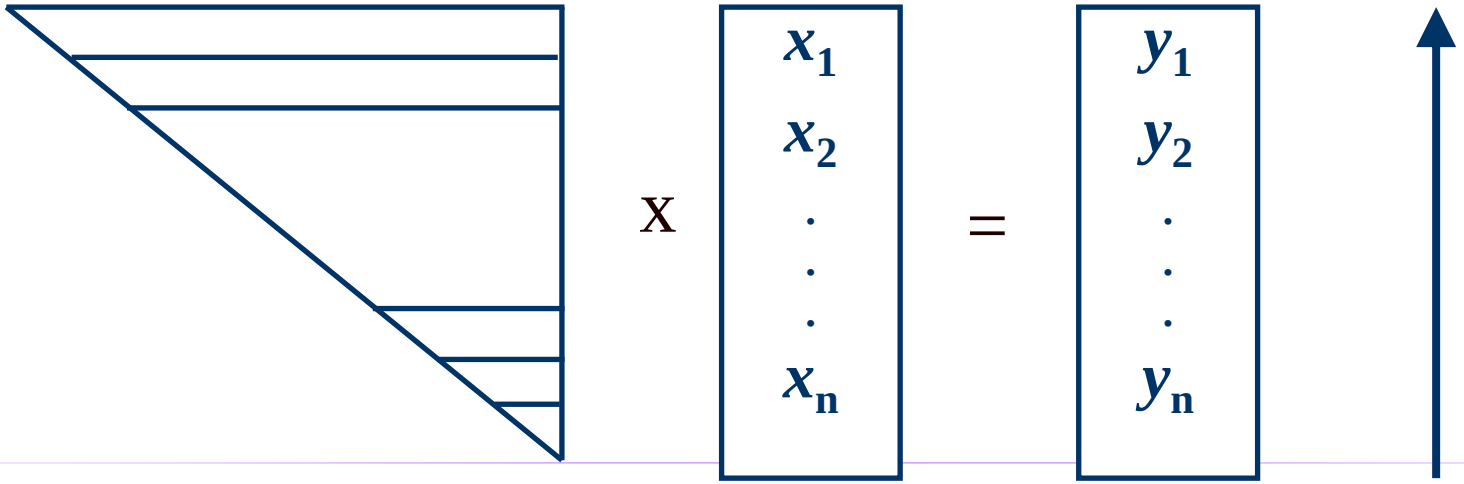
- Assume we are solving **Ax = b** with **A** factored so we solve **LUx = b** by first doing the forward substitution to solve **Ly = b** and then the backward substitution to solve **Ux = y**
- A key insight: In the solution of **Ly = b** if **b** is sparse then only certain columns of **L** are required, and **y** is often sparse
- 

Fast Forward Substitution

- If **b** is sparse, then the fast forward (FF) substitution takes advantage of the fact that we only need certain columns of **L**
- We define {FF} as the set of columns of L needed for the solution of **Ly** = **b**; this is equal to the nonzero elements of **y**
- In general the solution of **Ux = y** will NOT result in **x** being a sparse vector
- However, oftentimes only certain elements of **x** are desired
  - E.g., the sensitivity of the flows on certain lines to a change in generation at a single bus; or a diagonal of $\mathbf{A}^{-1}$

Fast Backward Substitution

- In the case in which only certain elements of **x** are desired, then we only need to use certain rows in **U** below the desired elements of **x**; define these columns as {FB}
- This is known as a fast backward substitution (FB), which is used to replace the standard backward substitution



Factorization Paths

- We observe that
  - {FF} depends on the sparsity structures of **L** and **b**
  - {FB} depends on the sparsity structures of **U** and **x**
- The idea of the factorization path provides a systematic way to construct these sets
- A factorization path is an ordered set of nodes associated with the structure of the matrix
- For FF the factorization path provides an ordered list of the columns of **L**
- For FB the factorization path provides an ordered list of the rows of **U**

Factorization Path

- The factorization path is traversed in the forward direction for FF and in the reverse direction for FB
  - Factorization paths should be built using doubly linked lists
- A singleton vector is a vector with just one nonzero element. If this value is equal to one then it is a unit vector as well.

Factorization Path, cont.

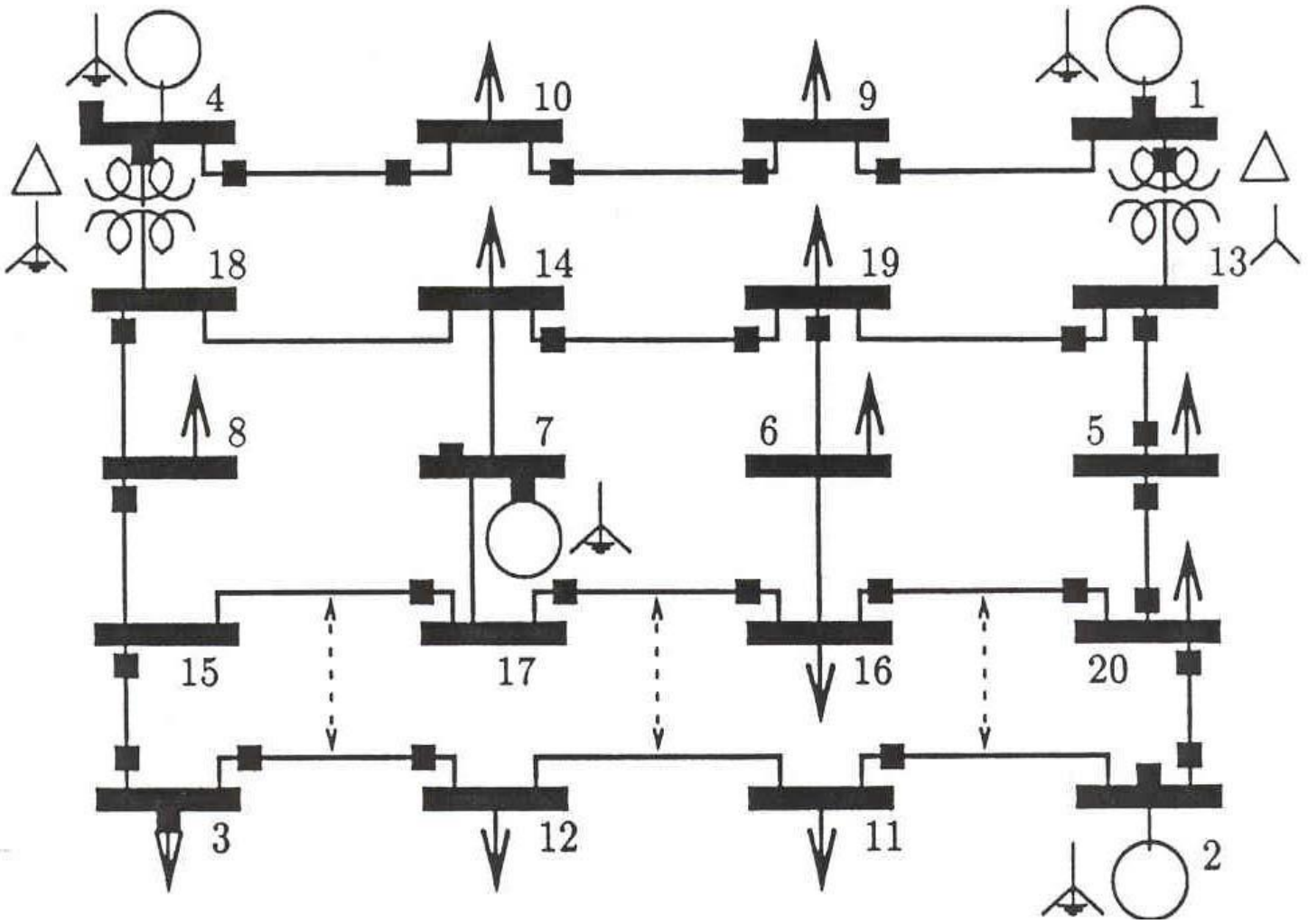
- With a sparse matrix structure ordered based upon the permutation vector order the path for a singleton with a now zero at position arow is build using the following code:

```
p1:= rowDiag[arow];
  While p1 <> nil Do Begin
    AddToPath(p1.col);   // Setup a doubly linked list!
    p1 := rowDiag[p1.col].next;
  End;
```

Path Table and Path Graph

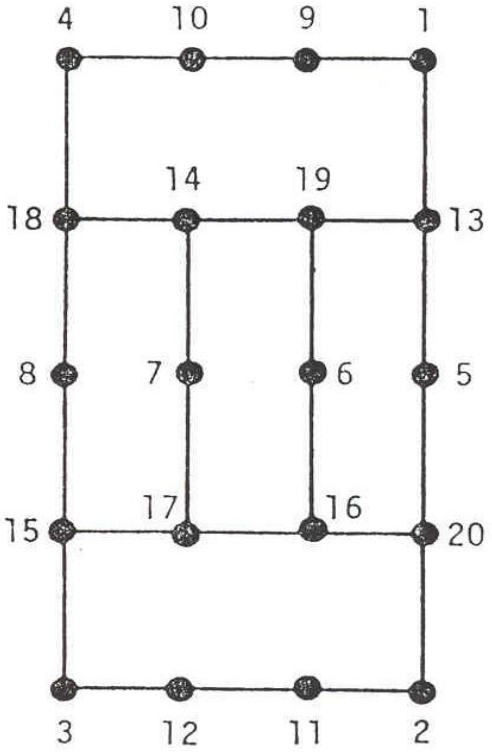
- The factorization path table is a vector that tells the next element in the factorization path for each row in the matrix
- The factorization path graph shows a pictorial view of the path table

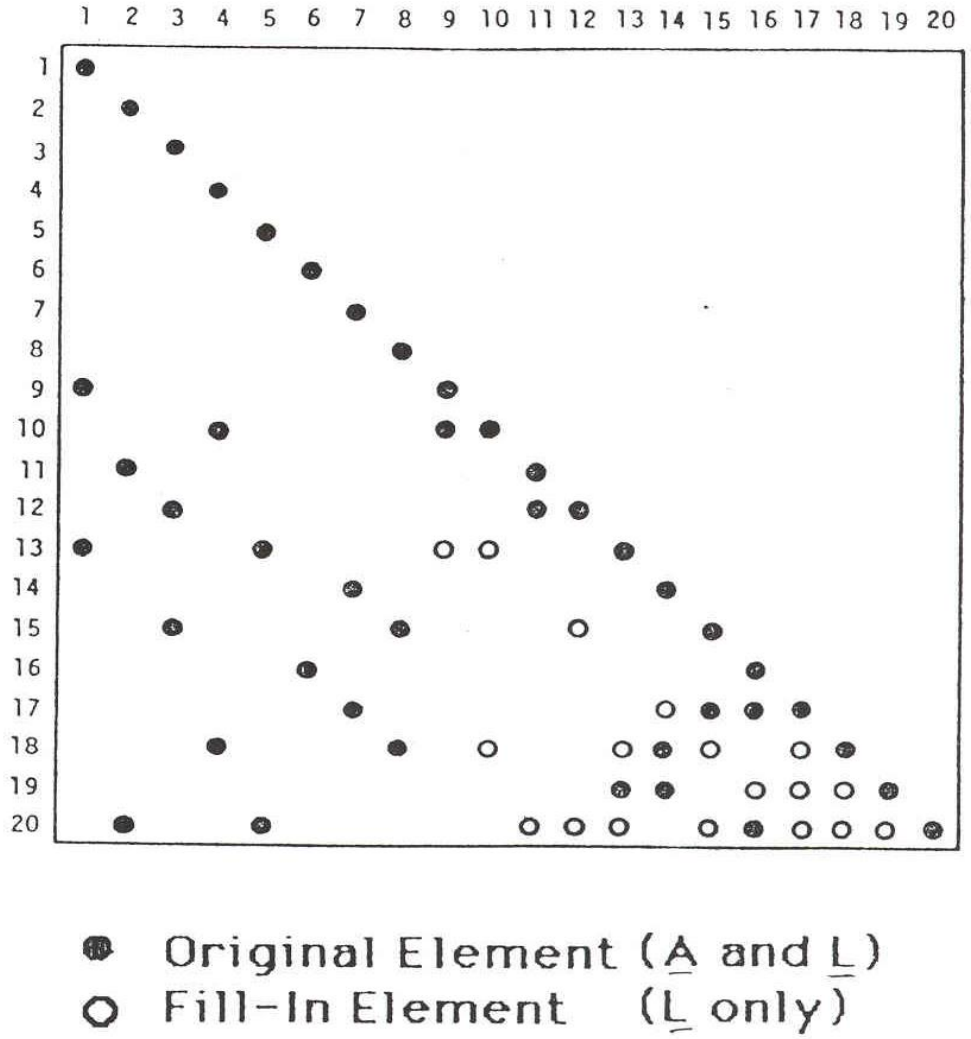
20 Bus Example



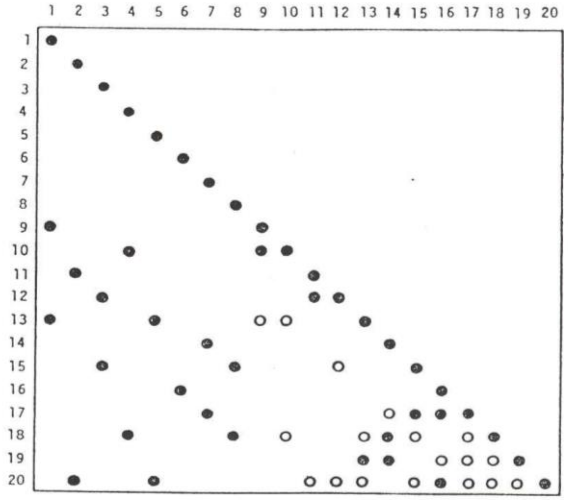
20 Bus Example

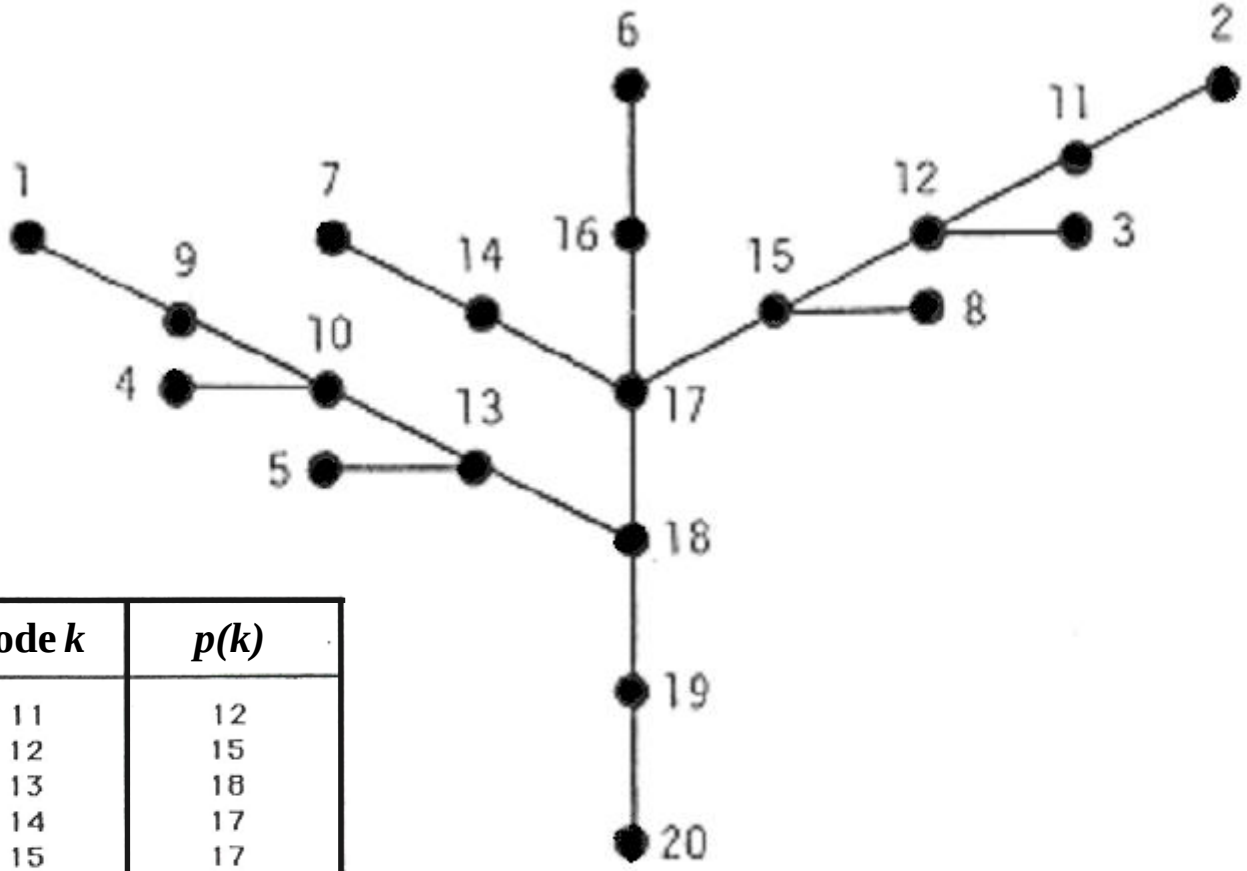
Only showing **L**





20 Bus Example





node *k*	*p(k)*	node *k*	*p(k)*
1	9	11	12
2	11	12	15
3	12	13	18
4	10	14	17
5	13	15	17
6	16	16	17
7	14	17	18
8	15	18	19
9	10	19	20
10	13	20	0

20 Bus Example

- Suppose we wish to evaluate a sparse vector with the nonzero elements for components 2, 6, 7, and 12
- From the path table or path graph, we obtain the following factorization paths (f.p.)

2 → *f.p.* {2, 11, 12, 15, 17, 18, 19, 20}

6 → *f.p.* {6, 16, 17, 18, 19, 20}

7 → *f.p.* {7, 14, 17, 18, 19, 20}

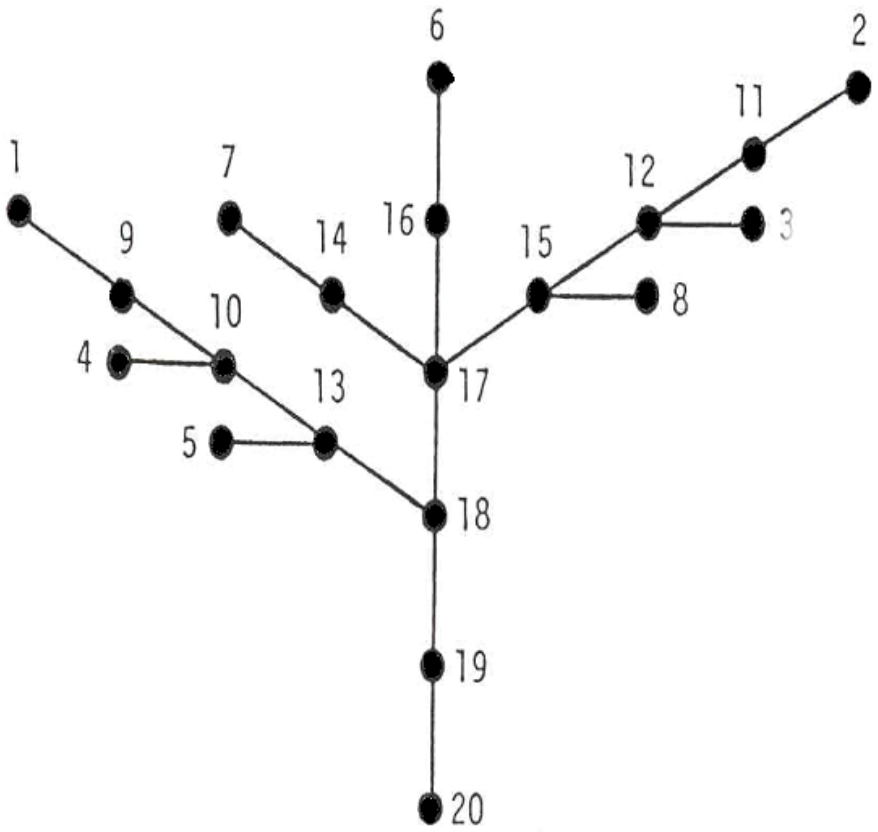
12 → *f.p. already contained in that for node* 2

- This gives the following path elements

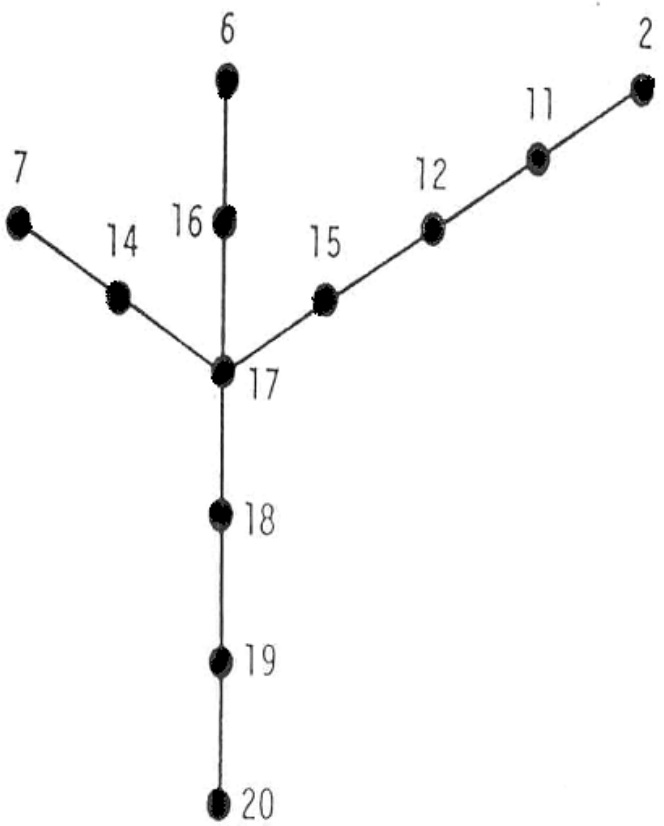
{7, 14, 6, 16, 2, 11, 12, 15, 17, 18, 19, 20}

20 Bus Example

Full path



Desired subset

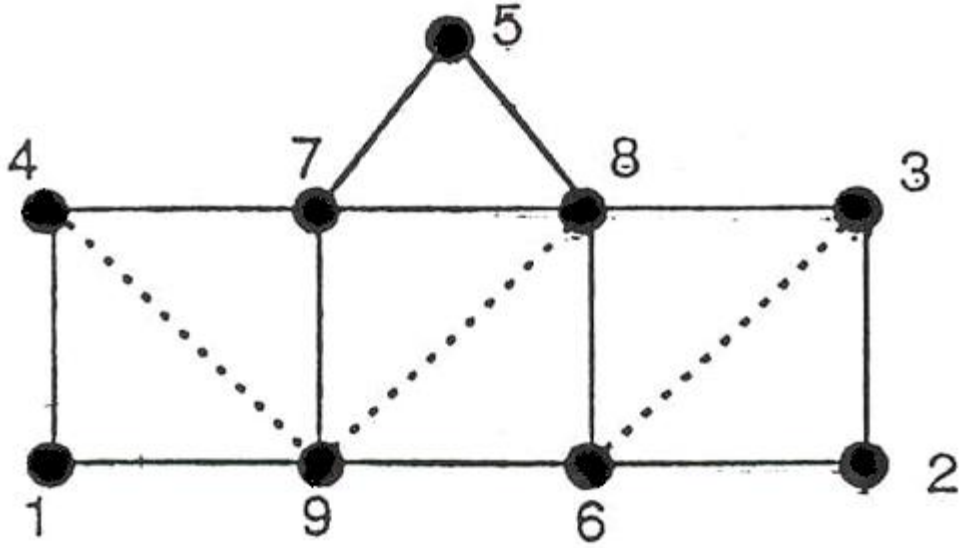


Remarks

- Since various permutations may be used to order a particular path subgroup, a precedence rule is introduced to make the ordering valid
- This involves some sorting operation; for the *FF*, the order value of a node cannot be computed until the order values of all lower numbered nodes are defined
- The order of processing branches above a junction node is arbitrary; for each branch, however, the precedence rule in force applies
- We can paraphrase this statement as: perform first everything above the junction point using the precedence ordering in each branch

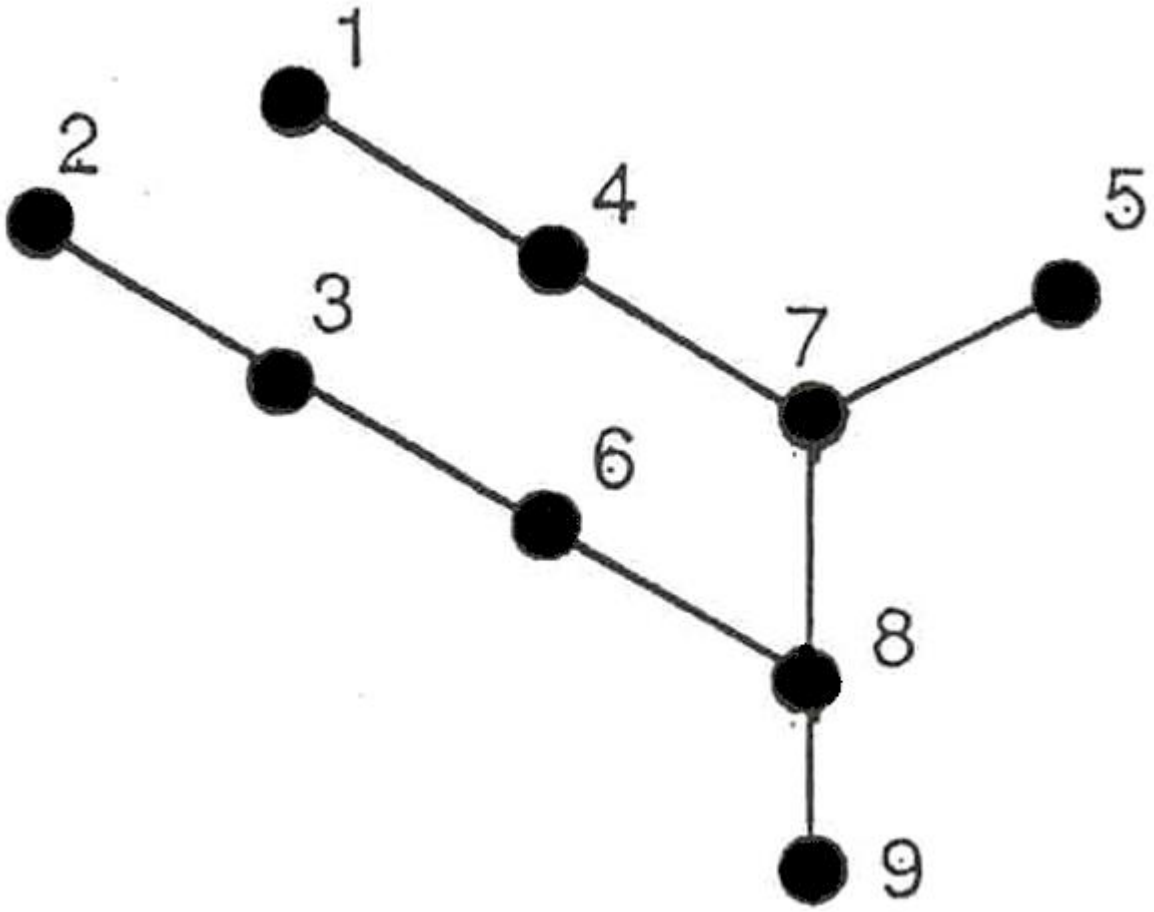
Nine Bus Example

- We next consider the example of the 9-bus network shown below



- For the given ordering, the sparsity structure leads to the following path graph and the table

Nine Bus Example

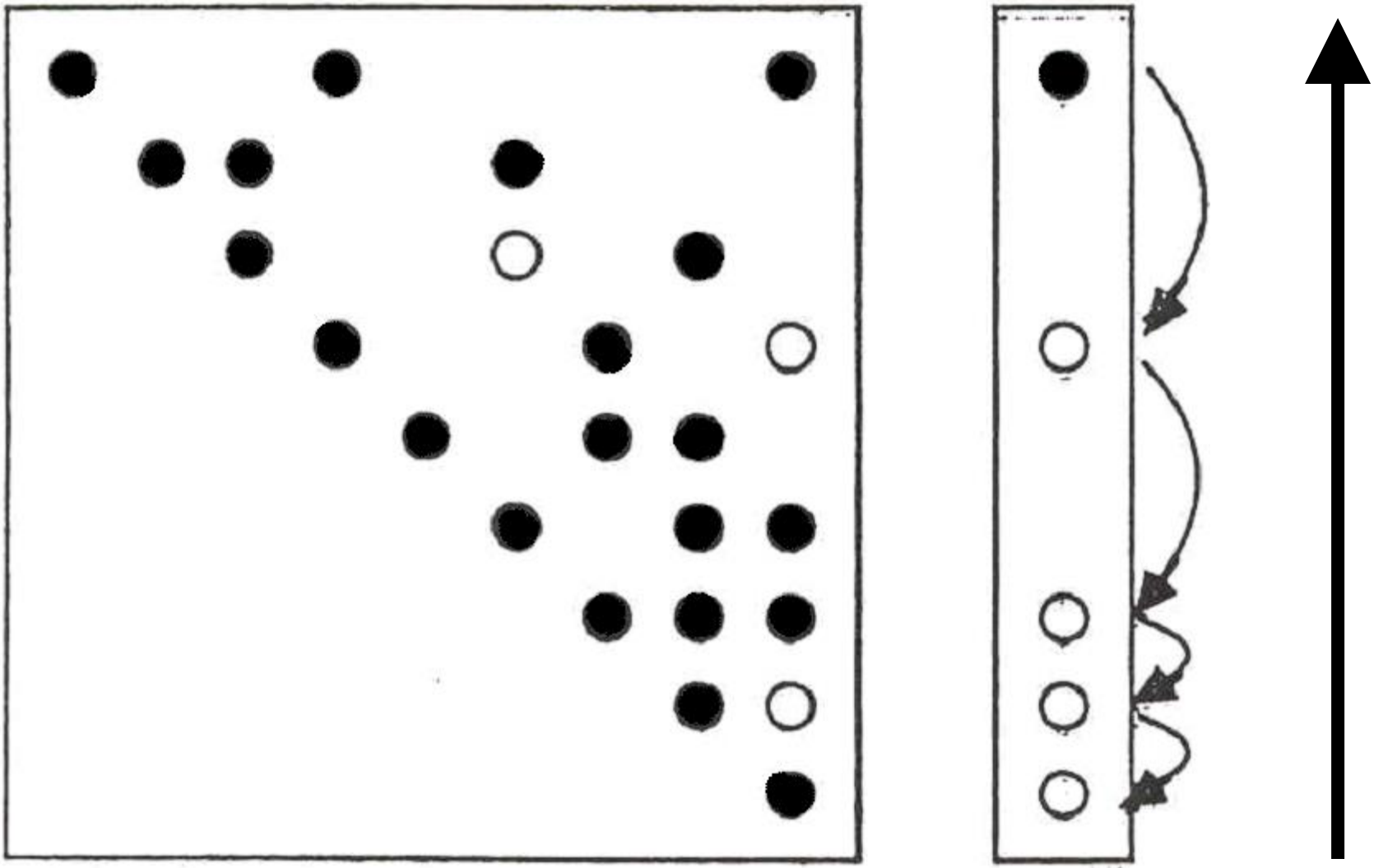


k	*p(k)*
1	4
2	3
3	6
4	7
5	7
6	8
7	8
8	9
9	0

Nine Bus Example

- Suppose next we are interested in the value determination of only component, node 1
  - That is, calculating a diagonal of the inverse of the original matrix
- *FF* involves going down the path from 1-4-7-8-9, and the *FB* requires coming back up, 9-8-7-4-1
- This example makes evident the savings in operations we may realize from the effective use of a sparse vector scheme

Nine Bus Example



Example Application

- In ongoing geomagnetic disturbance modeling work we need to determine the sensitivity of the results to the assumed substation grounding resistance
  - Since the induced voltages are quasi-dc, the network is modeled by setting up the conductance matrix $\mathbf{G} = \mathbf{R}^{-1}$
  - Initial work focused on calculating the driving point impedance values, which required knowing diagonal elements of **R**, which were easily calculated with sparse vector methods
  - But R_{ii} depends on the assumed grounding values are nearby substations, so we need to determine this impact as well; so we'd like small blocks of the inverse of **R**, which will require using the unions of the factorization paths to get some R_{ij}

Ordering for Shorter Paths

- The paper 1990 IEEE Transactions on Power Systems paper "Partitioned Sparse A^{-1} Methods" (by Alvarado, Yu and Betancourt) they introduce ordering methods for decreasing the length of the factorization paths
- Factorization paths also indicate the degree to which parallel processing could be used in solving **Ax** = **b** by **LU** factorization
  - Operations in the various paths could be performed in parallel

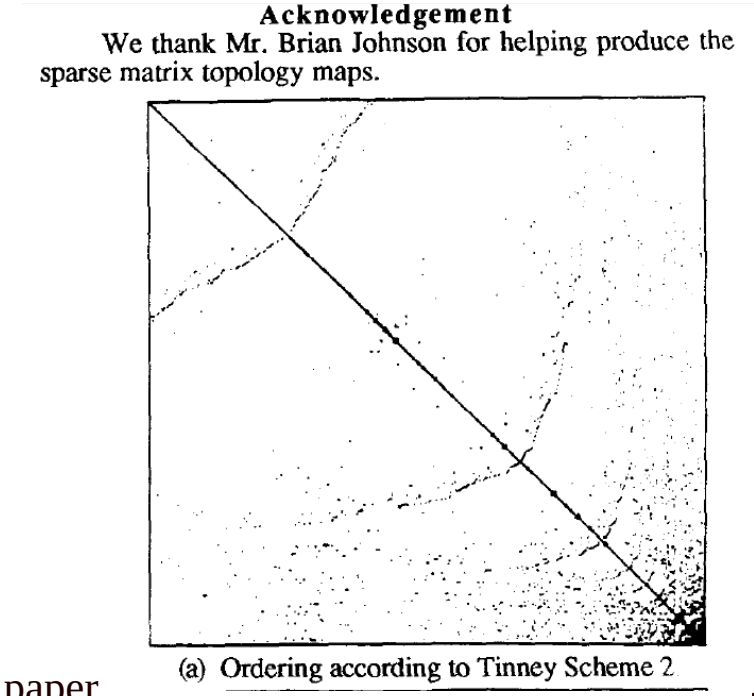


Image from Alvarado 1990 paper

Computation with Complex and Blocked Matrices

- In the previous analysis we have implicitly assumed that the values involved were real numbers (stored as singles or doubles in memory)
- Nothing in the previous analysis prevents using other data structures for analysis
  - Complex numbers would be needed if factoring the bus admittance matrix ($\mathbf{Y}_{bus}$); this is directly supported in some programming languages and can be easily added to others; all values are complex numbers
  - Two by two block matrices are common for power flow Jacobian factorization; for this we use 2 by 2 blocks in the matrices and 2 by 1 blocks in the vectors

2 by 2 Block Matrix Computation

- By treating our data structures as two by two blocks, we reduce the computation required to add fills substantially
  - Half the number of rows, and four times fewer elements
- Overall computation is reduced somewhat since we have four times fewer elements, but we do have more computation per element

2 by 2 Block Matrix Example

- In the backward substitution we had

```
For i := n downto 1 Do Begin
   k = rowPerm[i];
   p1 := rowDiag[k].next;
   While p1 <> nil Do Begin
      bvector[k] = bvector[k] – p1.value*bvector[p1.col];
      p1 := p1.next;
    End;
    bvector[k] := bvector[k]/rowDiag[k].value;
  End;
```

2 by 2 Block Matrix Example

- We replace the scalar bvector entries by objects with fields .r and .i (for the real and imaginary parts) and we replace the p1.value field with four fields .ul, .ur, .ll and .lr corresponding to the upper left, upper right, lower left and lower right values.
- The first multiply goes from

```
bvector[k] = bvector[k] – p1.value*bvector[p1.col]
```

to

$$\begin{bmatrix} \text{bvector[k].r} \\ \text{bvector[k].i} \end{bmatrix} = \begin{bmatrix} \text{bvector[k].r} \\ \text{bvector[k].i} \end{bmatrix} - \begin{bmatrix} \text{p1.ul} & \text{p1.ur} \\ \text{p1.ll} & \text{p1.lr} \end{bmatrix} \times \begin{bmatrix} \text{bvector[p1.col].r} \\ \text{bvector[p1.col].i} \end{bmatrix}$$

2 by 2 Block Matrix Example

- The second numeric calculation changes from
  bvector[k] := bvector[k]/rowDiag[k].value
- To

$$\begin{bmatrix} \text{bvector[k].r} \\ \text{bvector[k].i} \end{bmatrix} = \begin{bmatrix} \text{bvector[k].r} \\ \text{bvector[k].i} \end{bmatrix} - \begin{bmatrix} \text{rowDiag[k].ul} & \text{rowDiag[k].ur} \\ \text{rowDiag[k].ll} & \text{rowDiag[k].lr} \end{bmatrix}^{-1} \times \begin{bmatrix} \text{bvector[p1.col].r} \\ \text{bvector[p1.col].i} \end{bmatrix}$$

- Which can be coded by directly doing the inverse as

$$\begin{bmatrix} \text{bvector[k].r} \\ \text{bvector[k].i} \end{bmatrix} = \begin{bmatrix} \text{bvector[k].r} \\ \text{bvector[k].i} \end{bmatrix} - \frac{1}{\det} \begin{bmatrix} \text{rowDiag[k].lr} & \text{-rowDiag[k].ur} \\ \text{-rowDiag[k].ll} & \text{rowDiag[k].ul} \end{bmatrix} \times \begin{bmatrix} \text{bvector[p1.col].r} \\ \text{bvector[p1.col].i} \end{bmatrix}$$

with

$$\det = \text{rowDiag[k].ul} \times \text{rowDiag[k].lr} - \text{rowDiag[k].ll} \times \text{rowDiag[k].ur}$$

Sparse Matrix and Vector Method Summary

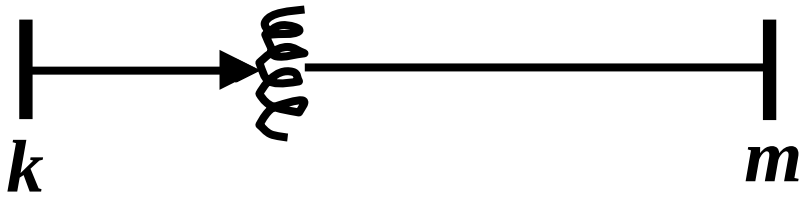
- Previous slides have presented sparse matrix and sparse vector methods commonly used in power system and some circuit analysis applications
- These methods are widely used, and have the ability to substantially speed up power system computations
- They will be applied as necessary throughout the remainder of the course
- We'll now move on to sensitivity analysis with a quick introduction of contingency analysis

Modeling Transformers with Off-Nominal Taps and Phase Shifts

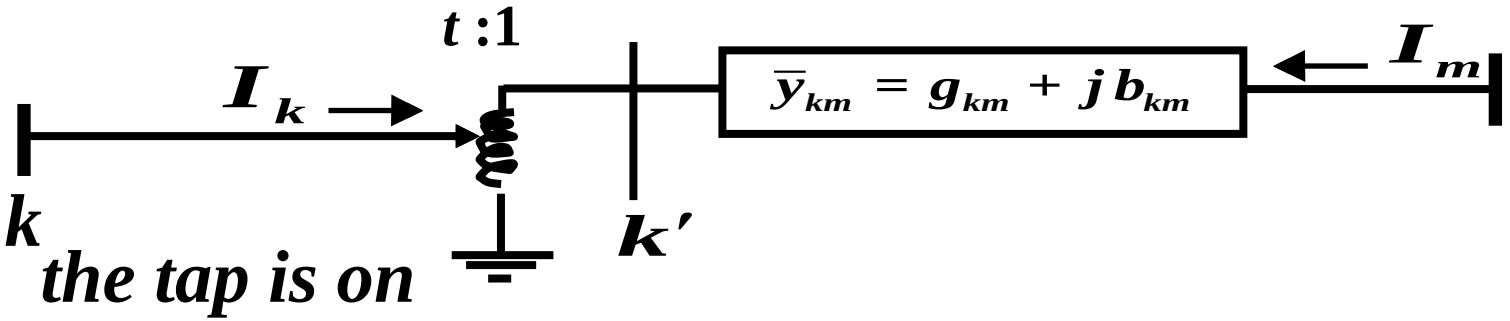
- If transformers have a turns ratio that matches the ratio of the per unit voltages than transformers are modeled in a manner similar to transmission lines.
- However it is common for transformers to have a variable tap ratio; this is known as an "off-nominal" tap ratio
  - The off-nominal tap is t, initially we'll consider it a real number
  - We'll cover phase shifters shortly in which t is complex

Transformer Representation

- The one–line diagram of a branch with a variable tap transformer



- The network representation of a branch with off–nominal turns ratio transformer is



Transformer Nodal Equations

- From the network representation

$$\bar{I}_m = \bar{I}_{k'} = \bar{y}_{km}\left(\bar{E}_m - \bar{E}_{k'}\right) = \bar{y}_{km}\left(\bar{E}_m - \frac{\bar{E}_k}{t}\right)$$

$$= \left(\bar{y}_{km}\right)\bar{E}_m + \left(-\frac{\bar{y}_{km}}{t}\right)\bar{E}_k$$

- Also

$$\bar{I}_k = -\frac{1}{t}\bar{I}_{k'} = \left(-\frac{\bar{y}_{km}}{t}\right)\bar{E}_m + \left(\frac{\bar{y}_{km}}{t^2}\right)\bar{E}_k$$

Transformer Nodal Equations

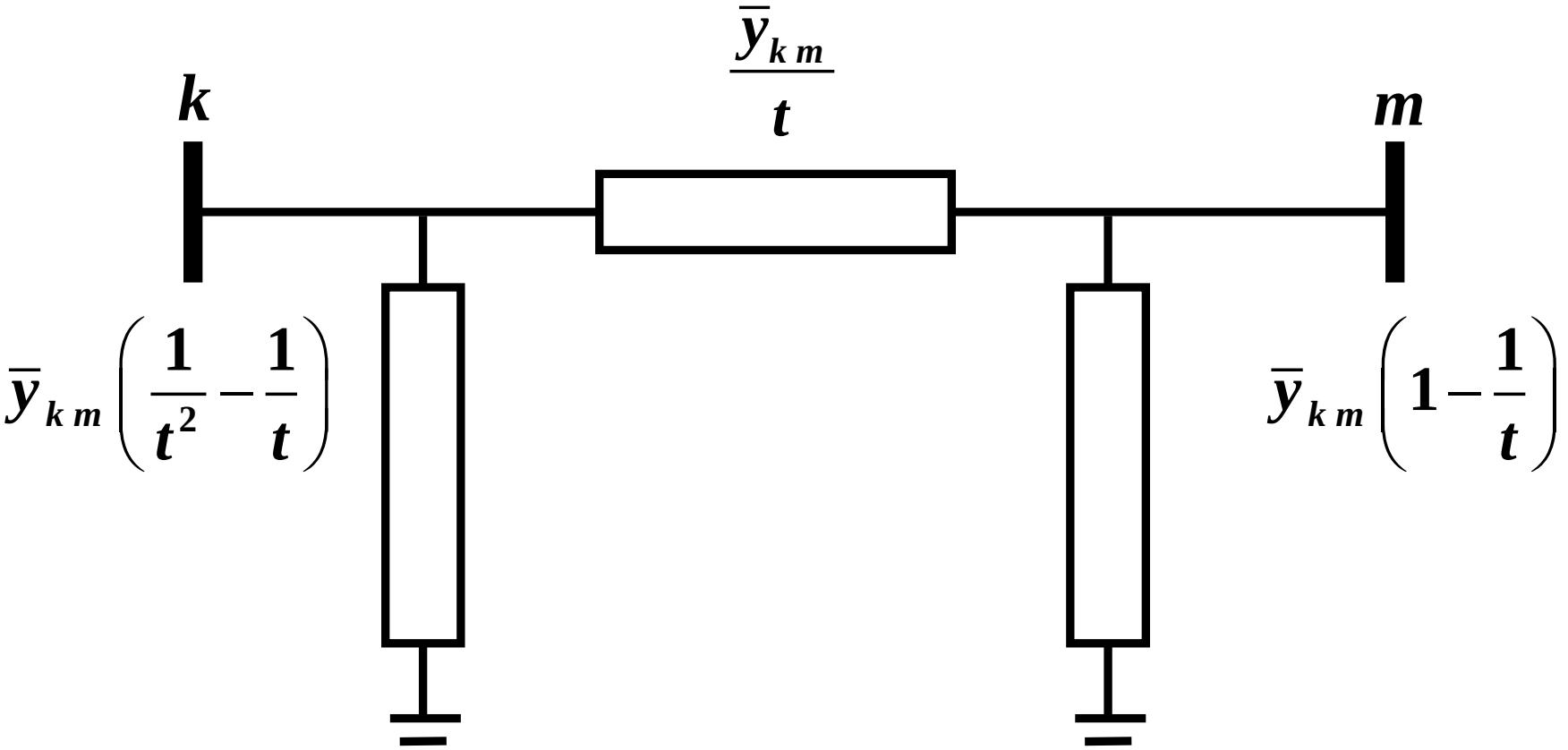
- We may rewrite these two equations as

$$\begin{bmatrix} \bar{I}_k \\ \\ \bar{I}_m \end{bmatrix} = \begin{bmatrix} \frac{\bar{y}_{km}}{t^2} & -\frac{\bar{y}_{km}}{t} \\ \\ -\frac{\bar{y}_{km}}{t} & \bar{y}_{km} \end{bmatrix} \begin{bmatrix} \bar{E}_k \\ \\ \bar{E}_m \end{bmatrix}$$

$\mathbf{Y}_{\text{bus}}$ is still symmetric here (though this will change with phase shifters)

This approach was first presented in F.L. Alvarado, "Formation of Y-Node using the Primitive Y-Node Concept," IEEE Trans. Power App. and Syst., December 1982

The π-Equivalent Circuit for a Transformer Branch



Variable Tap Voltage Control

- A transformer with a variable tap, i.e., the variable t is not constant, may be used to control the voltage at either the bus on the side of the tap or at the bus on the side away from the tap
- This constitutes an example of single criterion control since we adjust a single control variable (i.e., the transformer tap t) to achieve a specified criterion: the maintenance of a constant voltage at a designated bus
- Names for this type of control are on-load tap changer (LTC) transformer or tap changing under load (TCUL)
- Usually on low side; there may also be taps on high side that can be adjusted when it is de-energized

Variable Tap Voltage Control

- An LTC is a discrete control, often with 32 incremental steps of 0.625% each, giving an automatic range of ± 10%
- It follows from the $\pi-$equivalent model for the transformer that the transfer admittance between the buses of the transformer branch and the contribution to the self admittance at the bus away from the tap explicitly depend on t
- However, the tap changes in discrete steps; there is also a built in time delay in how fast they respond
- Voltage regulators are devices with a unity nominal ratio, and then a similar tap range

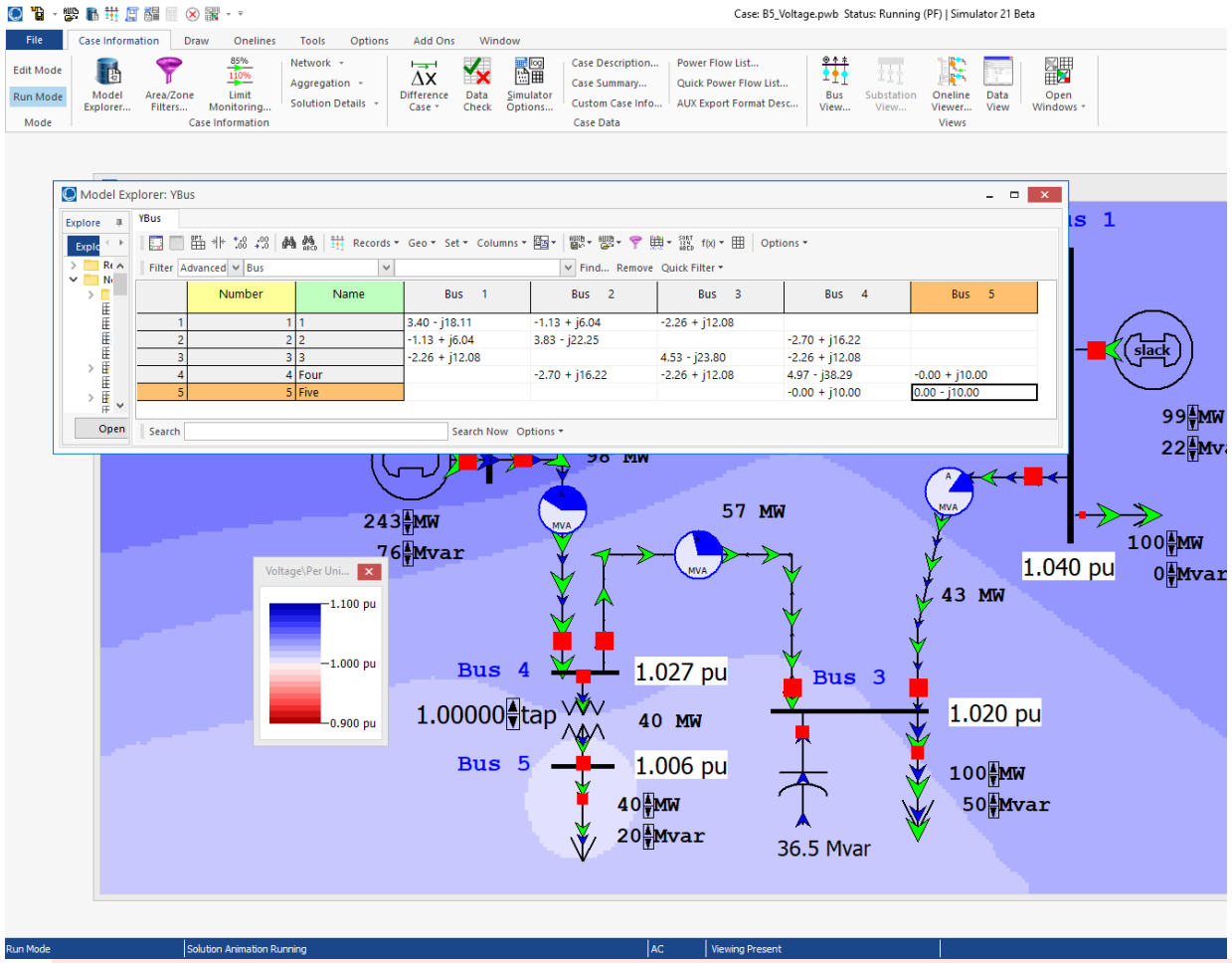
Ameren Champaign (IL) Test Facility Voltage Regulators

These are connected on the low side of a 69/12.4 kV transformer; each phase can be regulated separately

Variable Tap Voltage Control in the Power Flow

- LTCs (or voltage regulators) can be directly included in the power flow equations by modifying the $\mathbf{Y}_{\mathbf{bus}}$ entries; that is by scaling the terms by 1, 1/t or $1/t^2$ as appropriate
- If t is fixed then there is no change in the number of equations
- If t is variable, such as to enforce a voltage equality, then it can be included either by adding an additional equation and variable (t) directly, or by doing an "outer loop" calculation in which t is varied outside of the NR solution
  - The outer loop is used in PowerWorld because of limit issues

Five Bus PowerWorld Example



PowerWorld Case: **B5_Voltage**

With an impedance of j0.1 pu between buses 4 and 5, the y node primitive with t=1.0 is

$$\begin{bmatrix} -j10 & j10 \\ j10 & -j10 \end{bmatrix}$$

If t=1.1 then it is

$$\begin{bmatrix} -j10 & j9.09 \\ j9.09 & -j8.26 \end{bmatrix}$$