ECEN 615
Methods of Electric Power Systems Analysis

Lecture 10: Gaussian Elimination, Sparse Systems

Prof. Tom Overbye

Dept. of Electrical and Computer Engineering

Texas A&M University

overbye@tamu.edu

Special Guest Lecture: TA Iyke Idehen

Announcements

- Read Chapter 6
- Homework 2 is due today

Linear System Solution: Introduction

- A problem that occurs in many is fields is the solution of linear systems **Ax = b** where **A** is an n by n matrix with elements a_{ij}, and **x** and **b** are n-vectors with elements x_i and b_i respectively
- In power systems we are particularly interested in systems when n is relatively large and **A** is sparse
  - How large is large is changing
- A matrix is sparse if a large percentage of its elements have zero values
- Goal is to understand the computational issues (including complexity) associated with the solution of these systems

Introduction, cont.

- Sparse matrices arise in many areas, and can have domain specific structures
  - Symmetric matrices
  - Structurally symmetric matrices
  - Tridiagnonal matrices
  - Banded matrices
- A good (and free) book on sparse matrices is available at www-users.cs.umn.edu/~saad/IterMethBook_2ndEd.pdf
- ECEN 615 is focused on problems in the electric power domain; it is not a general sparse matrix course
  - Much of the early sparse matrix work was done in power!

Gaussian Elimination

- The best known and most widely used method for solving linear systems of algebraic equations is attributed to Gauss
- Gaussian elimination avoids having to explicitly determine the inverse of **A**, which is O(n^3)
- Gaussian elimination can be readily applied to sparse matrices
- Gaussian elimination leverages the fact that scaling a linear equation does not change its solution, nor does adding on linear equation to another

$$2x_1 + 4x_2 = 10 \rightarrow x_1 + 2x_2 = 5$$

Gaussian Elimination, cont.

- Gaussian elimination is the elementary procedure in which we use the first equation to eliminate the first variable from the last n-1 equations, then we use the new second equation to eliminate the second variable from the last n-2 equations, and so on
- After performing n-1 such eliminations we end up with a triangular system which is easily solved in a backward direction

Example 1

- We need to solve for **x** in the system

$$\begin{bmatrix} 2 & 3 & -1 & 0 \\ -6 & -5 & 0 & 2 \\ 2 & -5 & 6 & -6 \\ 4 & 2 & 2 & -3 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \\ x_4 \end{bmatrix} = \begin{bmatrix} 20 \\ -45 \\ -3 \\ 30 \end{bmatrix}$$

- The three elimination steps are given on the next slides; for simplicity, we have appended the r.h.s. vector to the matrix
- First step is set the diagonal element of row 1 to 1 (i.e., normalize it)

Example 1, cont.

- Eliminate x_1 by subtracting row 1 from all the rows below it

$$\begin{array}{l} \text{multiply row } 1 \text{ by } \frac{1}{2} \\ \text{multiply row } 1 \text{ by } 6 \\ \text{and add to row } 2 \\ \text{multiply row } 1 \text{ by} -2 \\ \text{and add to row } 3 \\ \text{multiply row } 1 \text{ by} -4 \\ \text{and add to row } 4 \end{array} \quad \left[\begin{array}{cccc|c} 1 & \frac{3}{2} & -\frac{1}{2} & 0 & 10 \\ 0 & 4 & -3 & \frac{1}{2} & 15 \\ 0 & -8 & -7 & -6 & -23 \\ 0 & -4 & 7 & -3 & -10 \end{array}\right]$$

Example 1, cont.

- Eliminate x_2 by subtracting row 2 from all the rows below it

$$
\begin{array}{l}
\text{multiply row } 2 \text{ by } \frac{1}{4} \\
\text{multiply row } 2 \text{ by } 8 \\
\text{and add to row } 3 \\
\text{multiply row } 2 \text{ by } 4 \\
\text{and add to row } 4
\end{array}
\quad
\left[\begin{array}{cccc|c}
1 & \frac{3}{2} & -\frac{1}{2} & 0 & 10 \\
0 & 1 & -\frac{3}{4} & \frac{1}{2} & \frac{15}{4} \\
0 & 0 & 1 & -2 & 7 \\
0 & 0 & 1 & -1 & 5
\end{array}\right]
$$

Example 1, cont.

- Elimination of x_3 from row 3 and 4

$$\begin{array}{r}
\\
\\
\textbf{subtract row 3} \\
\textbf{from row 4}
\end{array}
\left[\begin{array}{cccc|c}
1 & \frac{3}{2} & -\frac{1}{2} & 0 & 10 \\
0 & 1 & -\frac{3}{4} & \frac{1}{2} & \frac{15}{4} \\
0 & 0 & 1 & -2 & 7 \\
0 & 0 & 0 & 1 & -2
\end{array}\right]$$

Example 1, cont.

- Then, we solve for x by "going backwards", i.e., using back substitution:

$$x_4 = -2$$

$$x_3 - 2x_4 = 7 \Rightarrow x_3 = 3$$

$$x_2 - \frac{3}{4}x_3 + \frac{1}{2}x_4 = \frac{15}{4} \Rightarrow x_2 = 7$$

$$x_1 + \frac{3}{2}x_2 - \frac{1}{2}x_3 = 10 \Rightarrow x_1 = 1$$

Triangular Decomposition

- In this example, we have:
  - triangularized the original matrix by Gaussian elimination using column elimination
  - then, we used back substitution to solve the triangularized system
- The following slides present a general scheme for triangular decomposition by Gaussian elimination
- The assumption is that **A** is a nonsingular matrix (hence its inverse exists)
- Gaussian elimination also requires the diagonals to be nonzero; this can be achieved through ordering
- If **b** is zero then we have a trivial solution **x** = **0**

Triangular Decomposition

- We form the matrix $\mathbf{A}_a$ using **A** and **b** with

$$\mathbf{A}_a = \left[\mathbf{A} \vdots \mathbf{b}\right] = \begin{bmatrix} a_{11} & a_{12} & \cdots & a_{1n} & b_1 \\ a_{21} & a_{22} & \cdots & a_{2n} & b_2 \\ a_{31} & a_{32} & \cdots & a_{3n} & b_3 \\ \vdots & \vdots & \ddots & \vdots & \vdots \\ a_{n1} & a_{n2} & \cdots & a_{nn} & b_n \end{bmatrix}$$

and show the steps of the triangularization scheme

Triangular Decomposition, Step 1

- Step 1: normalize the first equation

$$a_{1j}^{(1)} = \frac{a_{1j}}{a_{11}} \qquad j = 2, \ldots, n$$

$$b_1^{(1)} = \frac{b_1}{a_{11}}$$

Triangular Decomposition, Step 2

- Step 2: a) eliminate x_1 from row 2:

$$a_{2j}^{(1)} = a_{2j} - a_{21} a_{1j}^{(1)}, \quad j = 2, \ldots, n$$

$$b_2^{(1)} = b_2 - a_{21} b_1^{(1)}$$

- Step 2: b) normalize the second equation

$$a_{2j}^{(2)} = \frac{a_{2j}^{(1)}}{a_{22}^{(1)}}, \quad j = 3, \ldots, n$$

$$b_2^{(2)} = \frac{b_2^{(1)}}{a_{22}^{(1)}}$$

Triangular Decomposition, Step 2

and we end up at the end of step 2 with

$$\begin{bmatrix} 1 & a_{12}^{(1)} & a_{13}^{(1)} & \cdots & a_{1n}^{(1)} & b_{1}^{(1)} \\ 0 & 1 & a_{23}^{(2)} & \cdots & a_{2n}^{(2)} & b_{2}^{(2)} \\ a_{31} & a_{32} & a_{33} & \cdots & a_{3n} & b_{3} \\ \vdots & \vdots & \vdots & \ddots & \vdots & \vdots \\ a_{n1} & a_{n2} & a_{n3} & \cdots & a_{nn} & b_{n} \end{bmatrix}$$

Triangular Decomposition, Step 3

- Step 3: a) eliminate x_1 and x_2 from row 3:

$$a_{3j}^{(1)} = a_{3j} - a_{31}\, a_{1j}^{(1)} \qquad j = 2, \dots, n$$

$$b_3^{(1)} = b_3 - a_{31}\, b_1^{(1)}$$

$$a_{3j}^{(2)} = a_{3j}^{(1)} - a_{32}^{(1)}\, a_{2j}^{(2)} \qquad j = 3, \dots, n$$

$$b_3^{(2)} = b_3^{(1)} - a_{32}^{(1)}\, b_2^{(2)}$$

- Step 3: b) normalize the third equation:

$$a_{3j}^{(3)} = \frac{a_{3j}^{(2)}}{a_{33}^{(2)}} \qquad j = 4, \dots, n$$

$$b_3^{(3)} = \frac{b_3^{(2)}}{a_{33}^{(2)}}$$

Triangular Decomposition, Step 3

and we have the system at the end of step 3

$$\begin{bmatrix} 1 & a_{12}^{(1)} & a_{13}^{(1)} & a_{14}^{(1)} & \cdots & a_{1n}^{(1)} & b_{1}^{(1)} \\ 0 & 1 & a_{21}^{(2)} & a_{24}^{(2)} & \cdots & a_{2n}^{(2)} & b_{2}^{(2)} \\ 0 & 0 & 1 & a_{34}^{(3)} & \cdots & a_{3n}^{(3)} & b_{3}^{(3)} \\ a_{41} & a_{42} & a_{43} & a_{44} & \cdots & a_{4n} & b_{4} \\ \vdots & \vdots & \vdots & \vdots & \ddots & \vdots & \vdots \\ a_{n1} & a_{n2} & a_{n3} & a_{n4} & \cdots & a_{nn} & b_{n} \end{bmatrix}$$

Triangular Decomposition, Step k

- In general, we have for step k:

  a) eliminate $x_1, x_2, \ldots, x_{k-1}$ from row k:

$$a_{kj}^{(m)} = a_{kj}^{(m-1)} - a_{km}^{(m-1)} a_{mj}^{(m)} \qquad j = m+1, \ldots, n$$

$$b_{k}^{(m)} = b_{k}^{(m-1)} - a_{km}^{(m-1)} b_{m}^{(m)} \qquad m = 1, 2, \ldots, k-1$$

  b) normalize the kth equation:

$$a_{kj}^{(k)} = \frac{a_{kj}^{(k-1)}}{a_{kk}^{(k-1)}} \qquad j = k+1, \ldots, n$$

$$b_{k}^{(k)} = \frac{b_{k}^{(k-1)}}{a_{kk}^{(k-1)}}$$

Triangular Decomposition: Upper Triangular Matrix

- and proceed in this manner until we obtain the upper triangular matrix (the nth derived system):

$$\begin{bmatrix} 1 & a_{12}^{(1)} & a_{13}^{(1)} & a_{14}^{(1)} & \cdots & a_{1\mathrm{n}}^{(1)} & b_{1}^{(1)} \\ 0 & 1 & a_{21}^{(2)} & a_{24}^{(2)} & \cdots & a_{2n}^{(2)} & b_{2}^{(2)} \\ 0 & 0 & 1 & a_{34}^{(3)} & \cdots & a_{3n}^{(3)} & b_{3}^{(3)} \\ 0 & 0 & 0 & 1 & \cdots & a_{4n}^{(4)} & b_{4}^{(4)} \\ \vdots & \vdots & \vdots & \vdots & \ddots & \vdots & \vdots \\ 0 & 0 & 0 & 0 & \cdots & 1 & b_{n}^{(n)} \end{bmatrix}$$

Triangular Decomposition

- Note that in the scheme presented, unlike in the first example, we triangularly decompose the system by eliminating row-wise rather than column-wise
  - In successive rows we eliminate (reduce to *0*) each element to the left of the diagonal rather than those below the diagonal
  - Either could be used, but row-wise operations will work better for power system sparse matrices

Solving for X

- To compute x we perform back substitution

$$x_n = b_n^{(n)}$$

$$x_{n-1} = b_{n-1}^{(n-1)} - a_{n-1,n}^{(n-1)} x_n$$

$$\vdots$$

$$x_k = b_k^{(k)} - \sum_{j=k+1}^{n} a_{kj}^{(k)} x_j \qquad k = n-1,\ n-2,\ \ldots,\ 1$$

Upper Triangular Matrix

- The triangular decomposition scheme applied to the matrix **A** results in the upper triangular matrix **U** with the elements

$$u_{ij} = \begin{cases} 1 & i = j \\ a_{ij}^{(i)} & j > i \\ 0 & j < i \end{cases}$$

- The following theorem is important in the development of the sparse computational scheme

LU Decomposition Theorem

- Any nonsingular matrix **A** has the following factorization:

$$\mathbf{A} = \mathbf{LU}$$

where **U** could be the upper triangular matrix previously developed (with 1's on its diagonals) and **L** is a lower triangular matrix defined by

$$\ell_{ij} = \begin{cases} a_{ij}^{(j-1)} & j \le i \\ 0 & j > i \end{cases}$$

LU Decomposition Application

- As a result of this theorem we can rewrite

  Ax = LUx = b

  Define **y = Ux**

  Then **Ly = b**

- Can also be set so **U** has non unity diagonals
- Once **A** has been factored, we can solve for **x** by first solving for **y**, a process known as forward substitution, then solving for **x** in a process known as back substitution
- In the previous example we can think of **L** as a record of the forward operations preformed on **b**.

LDU Decomposition

- In the previous case we required that the diagonals of **U** be unity, while there was no such restriction on the diagonals of **L**
- An alternative decomposition is

$$\mathbf{A} = \tilde{\mathbf{L}}\mathbf{D}\mathbf{U}$$

$$\text{with } \mathbf{L} = \tilde{\mathbf{L}}\mathbf{D}$$

where **D** is a diagonal matrix, and the lower triangular matrix is modified to require unity for the diagonals

Symmetric Matrix Factorization

- The LDU formulation is quite useful for the case of a symmetric matrix

$$\mathbf{A} = \mathbf{A}^T$$

$$\mathbf{A} = \tilde{\mathbf{L}}\mathbf{D}\mathbf{U} = \mathbf{U}^T\mathbf{D}\tilde{\mathbf{L}}^T = \mathbf{A}^T$$

$$\mathbf{U} = \tilde{\mathbf{L}}^T$$

$$\mathbf{A} = \mathbf{U}^T\mathbf{D}\mathbf{U}$$

- Hence only the upper triangular elements and the diagonal elements need to be stored, reducing storage by almost a factor of 2

Symmetric Matrix Factorization

- There are also some computational benefits from factoring symmetric matrices. However, since symmetric matrices are not common in power applications, we will not consider them in-depth
- However, topologically symmetric sparse matrices are quite common, so those will be our main focus

Pivoting

- An immediate problem that can occur with Gaussian elimination is the issue of zeros on the diagonal; for example

$$\mathbf{A} = \begin{bmatrix} 0 & 1 \\ 2 & 3 \end{bmatrix}$$

- This problem can be solved by a process known as “pivoting,” which involves the interchange of either both rows and columns (full pivoting) or just the rows (partial pivoting)
  - Partial pivoting is much easier to implement, and actually can be shown to work quite well

Pivoting, cont.

- In the previous example the (partial) pivot would just be to interchange the two rows

$$\tilde{\mathbf{A}} = \begin{bmatrix} 2 & 3 \\ 0 & 1 \end{bmatrix}$$

  obviously we need to keep track of the interchanged rows!
- Partial pivoting can be helpful in improving numerical stability even when the diagonals are not zero
  - When factoring row k interchange rows so the new diagonal is the largest element in column k for rows j >= k

LU Algorithm Without Pivoting Processing by row

- We will use the more common approach of having ones on the diagonals of **L**. Also in the common, diagonally dominant power system problems pivoting is not needed Below algorithm is in row form (useful with sparsity!)

```
For i := 2 to n Do Begin  // This is the row being processed
  For j := 1 to i-1 Do Begin  // Rows subtracted from row i
    A[i,j] = A[i,j]/A[j,j]  // This is the scaling
    For k := j+1 to n Do Begin  // Go through each column in i
      A[i,k] = A[i,k] - A[i,j]*A[j,k]
    End;
  End;
End;
```

LU Example

- Starting matrix

$$\mathbf{A} = \begin{bmatrix} 20 & -12 & -5 \\ -5 & 12 & -6 \\ -4 & -3 & 8 \end{bmatrix}$$

- First row is unchanged; start with i=2
- Result with i=2, j=1; done with row 2

$$\mathbf{A} = \begin{bmatrix} 20 & -12 & -5 \\ -0.25 & 9 & -7.25 \\ -4 & -3 & 8 \end{bmatrix}$$

LU Example, cont.

- Result with i=3, j=1;

$$\mathbf{A} = \begin{bmatrix} 20 & -12 & -5 \\ -0.25 & 9 & -7.25 \\ -0.2 & -5.4 & 7 \end{bmatrix}$$

- Result with i=3, j=2; done with row 3; done!

$$\mathbf{A} = \begin{bmatrix} 20 & -12 & -5 \\ -0.25 & 9 & -7.25 \\ -0.2 & -0.6 & 2.65 \end{bmatrix}$$

LU Example, cont.

- Original matrix is used to hold **L** and **U**

$$\mathbf{A} = \begin{bmatrix} 20 & -12 & -5 \\ -0.25 & 9 & -7.25 \\ -0.2 & -0.6 & 2.65 \end{bmatrix} = \mathbf{LU}$$

$$\mathbf{L} = \begin{bmatrix} 1 & 0 & 0 \\ -0.25 & 1 & 0 \\ -0.2 & -0.6 & 1 \end{bmatrix}$$

$$\mathbf{U} = \begin{bmatrix} 20 & -12 & -5 \\ 0 & 9 & -7.25 \\ 0 & 0 & 2.65 \end{bmatrix}$$

With this approach
the original **A** matrix
has been replaced
by the factored values!

Forward Substitution

Forward substitution solves **b = Ly** with values in **b** being over written (replaced by the **y** values)

```
For i := 2 to n Do Begin  // This is the row being processed
  For j := 1 to i-1 Do Begin
    b[i] = b[i] - A[i,j]*b[j]    // This is just using the L matrix
  End;
End;
```

Forward Substitution Example

$$\text{Let } \mathbf{b} = \begin{bmatrix} 10 \\ 20 \\ 30 \end{bmatrix}$$

$$\text{From before } \mathbf{L} = \begin{bmatrix} 1 & 0 & 0 \\ -0.25 & 1 & 0 \\ -0.2 & -0.6 & 1 \end{bmatrix}$$

$$y[1] = 10$$

$$y[2] = 20 - (-0.25) * 10 = 22.5$$

$$y[3] = 30 - (-0.2) * 10 - (-0.6) * 22.5 = 45.5$$

Backward Substitution

- Backward substitution solves $\mathbf{y} = \mathbf{U}\mathbf{x}$ (with values of **y** contained in the **b** vector as a result of the forward substitution)

```
For i := n to 1 Do Begin  // This is the row being processed
  For j := i+1 to n Do Begin
    b[i] = b[i] - A[i,j]*b[j]    // This is just using the U matrix
  End;
  b[i] = b[i]/A[i,i]    // The A[i,i] values are <> 0 if it is nonsingular
End
```

Backward Substitution Example

$$\text{Let } \mathbf{y} = \begin{bmatrix} 10 \\ 22.5 \\ 45.5 \end{bmatrix}$$

$$\text{From before } \mathbf{U} = \begin{bmatrix} 20 & -12 & -5 \\ 0 & 9 & -7.25 \\ 0 & 0 & 2.65 \end{bmatrix}$$

$$x[3] = (1/2.65) * 45.5 = 17.17$$

$$x[2] = (1/9) * \left(22.5 - (-7.25) * 17.17\right) = 16.33$$

$$x[1] = (1/20) * \left(10 - (-5) * 17.17 - (-12) * 16.33\right) = 14.59$$

Computational Complexity

- Computational complexity indicates how the number of numerical operations scales with the size of the problem
- Computational complexity is expressed using the “Big O” notation; assume a problem of size n
  - Adding the number of elements in a vector is O(n)
  - Adding two n by n full matrices is O(n^2)
  - Multiplying two n by n full matrices is O(n^3)
  - Inverting an n by n full matrix, or doing Gaussian elimination is O(n^3)
  - Solving the traveling salesman problem by brute-force search is O(n!)

Computational Complexity

- Knowing the computational complexity of a problem can help to determine whether it can be solved (at least using a particular method)
  - Scaling factors do not affect the computation complexity
    - an algorithm that takes $n^3/2$ operations has the same computational complexity of one the takes $n^3/10$ operations (though obviously the second one is faster!)
- With $O(n^3)$ factoring a full matrix becomes computationally intractable quickly!
  - A 100 by 100 matrix takes a million operations (give or take)
  - A 1000 by 1000 matrix takes a billion operations
  - A 10,000 by 10,000 matrix takes a trillion operations!

Sparse Systems

- The material presented so far applies to any arbitrary linear system
- The next step is to see what happens when we apply triangular factorization to a sparse matrix
- For a sparse system, only nonzero elements need to be stored in the computer since no arithmetic operations are performed on the 0's
- The triangularization scheme is adapted to solve sparse systems in such a way as to preserve the sparsity as much as possible

Sparse Matrix History

- A nice overview of sparse matrix history is by Iain Duff at http://www.siam.org/meetings/la09/talks/duff.pdf
- Sparse matrices developed simultaneously in several different disciplines in the early 1960’s with power systems definitely one of the key players (Bill Tinney from BPA)
- Different disciplines claim credit since they didn’t necessarily know what was going on in the others

Sparse Matrix History

- In power systems a key N. Sato, W.F. Tinney, "Techniques for Exploiting the Sparsity of the Network Admittance Matrix," Power App. and Syst., pp 944-950, December 1963
  - In the paper they are proposing solving systems with up to 1000 buses (nodes) in 32K of memory!
  - You'll also note that in the discussion by El-Abiad, Watson, and Stagg they mention the creation of standard test systems with between 30 and 229 buses (this surely included the now famous 118 bus system)
  - The BPA authors talk "power flow" and the discussors talk "load flow."
- Tinney and Walker present a much more detailed approach in their 1967 IEEE Proceedings paper titled "Direct Solutions of Sparse Network Equations by Optimally Order Triangular Factorization"

Sparse Matrix Computational Order

- The computational order of factoring a sparse matrix, or doing a forward/backward substitution depends on the matrix structure
  - Full matrix is O(n^3)
  - A diagonal matrix is O(n); that is, just invert each element
- For power system problems the classic paper is F. L. Alvarado, "Computational complexity in power systems," *IEEE Transactions on Power Apparatus and Systems*, ,May/June 1976
  - O($n^{1.4}$) for factoring, O($n^{1.2}$) for forward/backward
  - For a 100,000 by 100,000 matrix changes computation for factoring from 1 quadrillion to 10 million!

Inverse of a Sparse Matrix

- The inverse of a sparse matrix is NOT in general a sparse matrix
- We never (or at least very, very, very seldom) explicitly invert a sparse matrix
  - Individual columns of the inverse of a sparse matrix can be obtained by solving $\mathbf{x} = \mathbf{A}^{-1}\mathbf{b}$ with **b** set to all zeros except for a single nonzero in the position of the desired column
  - If a few desired elements of $\mathbf{A}^{-1}$ are desired (such as the diagonal values) they can usually be computed quite efficiently using sparse vector methods (a topic we'll be considering soon)
- We can't invert a singular matrix (with sparse or not)

Computer Architecture Impacts

- With modern computers the processor speed is many times faster than the time it takes to access data in main memory
  - Some instructions can be processed in parallel
- Caches are used to provide quicker access to more commonly used data
  - Caches are smaller than main memory
  - Different cache levels are used with the quicker caches, like L1, have faster speeds but smaller sizes; L1 might be 64K, whereas the slower L2 might be 1M
- Data structures can have a significant impact on sparse matrix computation

ECEN 615 Sparsity Limitations

- Sparse matrices arise in many areas, and can have domain specific structures
  - Symmetric matrices
  - Structurally symmetric matrices
  - Tridiagnonal matrices
  - Banded matrices
- ECEN 615 is focused on problems that arise in the electric power; it is not a general sparse matrix course

Full Matrix versus Sparse Matrix Storage

- Full matrices are easily stored in arrays with just one variable needed to store each value since the value's row and column are implicitly available from its matrix position
- With sparse matrices two or three elements are needed to store each value
  - The zero values are not explicitly stored
  - The value itself, its row number and its column number
  - Storage can be reduced by storing all the elements in a particular row or column together
- Because large matrices are often quite sparse, the total storage is still substantially reduced

Sparse Matrix Usage Can Determine the Optimal Storage

- How a sparse matrix is used can determine the best storage scheme to use
  - Row versus column access; does structure change
- Is the matrix essentially used only once? That is, its structure and values are assumed new each time used
- Is the matrix structure constant, with its values changed
  - This would be common in the N-R power flow, in which the structure doesn't change each iteration, but its values do
- Is the matrix structure and values constant, with just the **b** vector in **Ax=b** changing
  - Quite common in transient stability solutions

Numerical Precision

- Required numerical precision determines type of variables used to represent numbers
  - Specified as number of bytes, and whether signed or not
- For Integers
  - One byte is either 0 to 255 or -128 to 127
  - Two bytes is either smallint (-32,768 to 32,767) or word (0 to 65,536)
  - Four bytes is either Integer (-2,147,483,648 to 2,147,483,647) or Cardinal (0 to 4,294,967,295)
    - This is usually sufficient for power system row/column numbers
  - Eight bytes (Int64) if four bytes is not enough

Numerical Precision, cont.

- For floating point values using choice is between four bytes (single precision) or eight bytes (double precision); extended precision has ten bytes
  - Single precision allows for 6 to 7 significant digits
  - Double precision allows for 15 to 17 significant digits
  - Extended allows for about 18 significant digits
  - More bytes requires more storage
  - Computational impacts depend on the underlying device; on PCs there isn’t much impact; GPUs can be 3 to 8 times slower for double precision
- For most power problems double precision is best

General Sparse Matrix Storage

- A general approach for storing a sparse matrix would be using three vectors, each dimensioned to number of elements
  - AA: Stores the values, usually in power system analysis as double precision values (8 bytes)
  - JR: Stores the row number; for power problems usually as an integer (4 bytes)
  - JC: Stores the column number, again as an integer
- If unsorted then both row and column are needed
- New elements could easily be added, but costly to delete
- Unordered approach doesn't make for good computation since elements used next computationally aren't necessarily nearby
- Usually ordered, either by row or column

Sparse Storage Example

- Assume

$$\mathbf{A} = \begin{bmatrix} 5 & 0 & 0 & -4 \\ 0 & 4 & 0 & -3 \\ 0 & 0 & 3 & -2 \\ -4 & -3 & -2 & 10 \end{bmatrix}$$

- Then

$$\mathbf{AA} = \begin{bmatrix} 5 & -4 & 4 & -3 & 3 & -2 & -4 & -3 & -2 & 10 \end{bmatrix}$$
$$\mathbf{JR} = \begin{bmatrix} 1 & 1 & 2 & 2 & 3 & 3 & 4 & 4 & 4 & 4 \end{bmatrix}$$
$$\mathbf{JC} = \begin{bmatrix} 1 & 1 & 2 & 2 & 3 & 3 & 1 & 2 & 3 & 4 \end{bmatrix}$$

Note, this example is a symmetric matrix, but the technique is general

Compressed Sparse Row Storage

- If elements are ordered (as was case for previous example) storage can be further reduced by noting we do not need to continually store each row number
- A common method for storing sparse matrices is known as the Compressed Sparse Row (CSR) format
  - Values are stored row by row
  - Has three vector arrays:
    - AA: Stores the values as before
    - JA: Stores the column index (done by JC in previous example)
    - IA: Stores the pointer to the index of the beginning of each row

CSR Format Example

- Assume as before

$$\mathbf{A} = \begin{bmatrix} 5 & 0 & 0 & -4 \\ 0 & 4 & 0 & -3 \\ 0 & 0 & 3 & -2 \\ -4 & -3 & -2 & 10 \end{bmatrix}$$

- Then

$$\mathbf{AA} = \begin{bmatrix} 5 & -4 & 4 & -3 & 3 & -2 & -4 & -3 & -2 & 10 \end{bmatrix}$$

$$\mathbf{JA} = \begin{bmatrix} 1 & 1 & 2 & 2 & 3 & 3 & 1 & 2 & 3 & 4 \end{bmatrix}$$

$$\mathbf{IA} = \begin{bmatrix} 1 & 3 & 5 & 7 \end{bmatrix}$$

CSR Comments

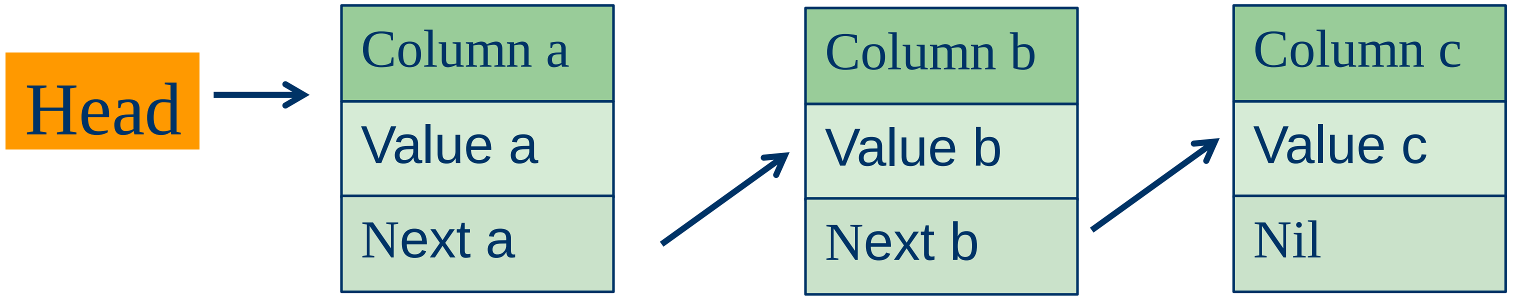
- The CSR format reduces the storage requirements by taking advantage of needing only one element per row
- The CSR format has good advantages for computation when using cache since (as we shall see) during matrix operations we are often sequentially going through the vectors
- An alternative approach is Compressed Sparse Column (CSC), which identical, except storing the values by column
- It is difficult to add values.
- We'll mostly use the linked list approach here, which makes matrix manipulation simpler

Linked Lists: Classes and Objects

- In explaining the linked list approach it is helpful to use the concepts from object oriented programming (OOP) of classes and objects
  - Approach can also be used in non-OOP programming
- OOP can be thought of as a collection of objects interacting with each other
- Objects are instances of classes.
- Classes define the object fields and actions (methods)
- We'll define a class called sparse matrix element, with fields of value, column and next; each sparse matrix element is then an object of this class

Linked Lists

- A linked list is just a group of objects that represent a sequence
  - We'll used linked lists to represent a row or column of a sparse matrix
- Each linked list has a head pointer that points to the first object in the list
  - For our sparse matrices the head pointer will be a vector of the rows or columns



Sparse Matrix Storage with Linked Lists by Rows

- If we have an n by n matrix, setup a class called TSparseElement with fields column, value and next
- Setup an n-dimensional head pointer vector that points to the first element in each row
- Each nonzero corresponds to an object of class (type) TSparseElement
- We do not need to store the row number since it is given by the object’s row
- For power system sparse matrices, which have nonzero diagonals, we also have a header pointer vector that points to the diagonal objects

Linked Lists, cont.

- Linked lists can be singly linked, which means they just go in one direction (to the next element), or doubly linked, pointing to both the previous and next elements
  - Mostly we'll just need singularly linked lists
- With linked lists it is quite easy to add new elements to the list. This can be done in sorted order just by going down the list until the desired point is reached, then changing the next pointer for the previous element to the new element, and for the new element to the next element (for a singly linked list)

On Board Example

- Draw the data structures for the matrix

$$\mathbf{A} = \begin{bmatrix} 5 & 0 & 0 & -4 \\ 0 & 4 & 0 & -3 \\ 0 & 0 & 3 & -2 \\ -4 & -3 & -2 & 10 \end{bmatrix}$$