ECE 4960

Spring 2017

# Lecture 7

## Linear Algebra: Sparse Matrices

**Edwin C. Kan**
School of Electrical and Computer Engineering
Cornell University

## Large Number of Degree of Freedoms and Independent Variables

- Dow Jones index: $10^7$ variables of companies, funds and bills
  - Interdepencies among companies and relation to forecast: nonlinear
- Weather simulation: 100km × 100km × 10km on 10m grids: $10^{11}$ variables
  - Navier Stokes fluidic equations: nonlinear
- Microprocessor: Data bus (64 bits) on ALU internal nodes ($10^7$)
  - Transistor I-V relations: nonlinear

## Vectors and Matrices

- Most mathematical and physical problems, after linearization, can be translated into problems involving vectors and matrices.
- Mapping data bus (64 bits) on ALU internal nodes ($10^7$)
  - Matrix $A$ will have a dimension [$10^7$, 64]
  - Modeling the RC branches connected the $10^7$ ALU internal nodes by Kerchhoff's laws: Matrix $A$ will have a dimension [$10^7$, $10^7$] that transforms the $10^7$ voltage node vector at $t = t_0$ to the $10^7$ voltage node vector at $t = t_0 + \Delta t$.
- Matrix of rank $n$ will need $O(n^2)$ storage and $O(n^3)$ operations in the solution of $Ax = b$.
- Problems become untreatable for present computers: $A$ of [$10^7$, $10^7$] matrix with every element in double precision: $10^{15}$ bytes (1,000TB) to store every value.

## Sparse Matrices

- Most matrices of practical problems are “sparse”, i.e., most of their elements have zero values.
- For the ALU circuits, if every node has on average 10 elements connected to it, the matrix storage is just *$10n$* instead of $n^2$.
- If the problem is derived from finite-element or finite-difference of the partial differential equations, 3D formulation has an average of 7 – 21 elements per node (depending on the number of equations that need to be solved self consistently).
- Handling sparse matrices efficiently in terms of memory usage and computation is the heart of many computing problems:
  - video games,
  - stock market analysis
  - engineering simulation

# Sparse Matrix Resources

- **BLAS** (basic linear algebra subprograms): Legacy Fortran and C functions that are highly optimized for CPU with limited cache space. Although the library had been created and maintained from the 1960's, it is still the workhorse for many scientific simulation softwares. See the web page at http://www.netlib.org/blas/.
- **Sparselib++**: Production of the National Institute of Standards and Technology. See the web page at: http://math.nist.gov/sparselib++/.
- **Trilinos**: A more recent effort with all modern language support. Also very good software engineering principles. Trilinos is an international collaboration. In US, it is mostly maintained by the Sandia National Lab. See the web page at: https://trilinos.org/.

- **Matlab**??? Not free…

# Sparse Matrix Representation

- Although OOP should prescribe “methods” or “application procedural interface (API)” instead of “data structure, most important consideration of sparse matrices is the efficient memory handling: understanding data structure is *necessary* (though hidden behind API).
- For the most basic operation of $Ax = b$,
  - $A$ is of dimension $[m, n]$
  - $x$ of dimension $m$
  - $b$ of dimension $n$
- we will only need sparse representation for $A$. Unless $A$ is severely degenerate (i.e., $\mathrm{rank}(A) \ll \min(m, n)$), $b$ will not be sparse even though $x$ is sparse.
- Assume matrices with equal row and column ranks $n$, unless otherwise mentioned.

# Coordinate Sparse Matrix Storage

$$A = \begin{pmatrix} 1 & 2 & 0 & 0 & 3 \\ 4 & 5 & 6 & 0 & 0 \\ 0 & 7 & 8 & 0 & 9 \\ 0 & 0 & 0 & 10 & 0 \\ 11 & 0 & 0 & 0 & 12 \end{pmatrix}$$

**Coordinate storage of sparse matrix A**

| | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| value( ) | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 |
| rowInd( ) | 0 | 0 | 0 | 1 | 1 | 1 | 2 | 2 | 2 | 3 | 4 | 4 |
| colInd( ) | 0 | 1 | 4 | 0 | 1 | 2 | 1 | 2 | 4 | 4 | 0 | 4 |

- Here rank($A$) = $n$ = 5, dim($A$) = [5, 5]
- 12 non-zero elements within the 25 matrix elements.
- Row index of 0…4 and column index of 0…4 in C/C++.

## Compressed Row or Column Storage

$$A = \begin{pmatrix} 1 & 2 & 0 & 0 & 3 \\ 4 & 5 & 6 & 0 & 0 \\ 0 & 7 & 8 & 0 & 9 \\ 0 & 0 & 0 & 10 & 0 \\ 11 & 0 & 0 & 0 & 12 \end{pmatrix}$$

**Compressed row storage of sparse matrix A**

| | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| value( ) | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 |
| rowPtr( ) | 0 | 3 | 6 | 9 | 10 | 12 | | | | | | |
| colInd( ) | 0 | 1 | 4 | 0 | 1 | 2 | 1 | 2 | 4 | 4 | 0 | 4 |

**Compressed column storage of sparse matrix A**

| | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| value( ) | 1 | 4 | 11 | 2 | 5 | 7 | 6 | 8 | 10 | 3 | 9 | 12 |
| rowInd( ) | 0 | 1 | 4 | 0 | 1 | 2 | 1 | 2 | 3 | 0 | 2 | 4 |
| colPtr( ) | 0 | 3 | 6 | 8 | 9 | 12 | | | | | | |

# Coordinate vs. Compressed

- Coordinate storage:
  - No assumption of element order: easy add and delete.
- Compressed storage:
  - Slightly smaller storage space
  - Assume sorted element orders in row or column
  - Easy trasversing in multiplication
  - Overhead in add and delete
- Many practical problems like circuit simulation have a fixed matrix topology derived from the circuit topology (aka, the netlist).
  - Non-zero elements in the matrix have fixed row and column position, or they are "**symbolic** non-zero elements"
  - The nonzero element has various different floating-point values during the computation: "**numerical** non-zero elements"

# Common Sparse Matrix Utility Functions

```
matrix*   createMatrix(n);   // intend to create a sparse matrix of rank n,
                             // but without reserving all storage.
                             // Memory scheme varies.

void      addElement(*matrix, rowInd, colInd, value);
void      deleteElement(*matrix, rowInd, colInd);

double    retrieveElement(*matrix, rowInd, colInd);

integer   rankMatrix(*matrix);                // return 5 for above
integer   countElementMatrix(*matrix);        // return 12 for above
```

# Hacker Practice

- Create the sparse matrix *A* in the row compressed format of Table 2 with three vectors: `value[12]; rowPtr[6]; colInd[12]`.
- Implement the function retrieveElement( ).
- Use the 5 by 5 printout to verify your program

$$A = \begin{pmatrix} 1 & 2 & 0 & 0 & 3 \\ 4 & 5 & 6 & 0 & 0 \\ 0 & 7 & 8 & 0 & 9 \\ 0 & 0 & 0 & 10 & 0 \\ 11 & 0 & 0 & 0 & 12 \end{pmatrix}$$

**Compressed row storage of sparse matrix A**

| | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| value( ) | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 |
| rowPtr( ) | 0 | 3 | 6 | 9 | 10 | 12 | | | | | | |
| colInd( ) | 0 | 1 | 4 | 0 | 1 | 2 | 1 | 2 | 4 | 4 | 0 | 4 |