Sparse Matrices: Storage Schemes I

- Sparse matrices: most elements are zero
- They are common in engineering applications
- Without storing zeros, we can handle very large matrices
- An example

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

Sparse Matrices: Storage Schemes II

- Storage schemes:

  There are different ways to store sparse matrices
- Coordinate format

`a`	1	3	6	4	7	10	2	5	8	11
`arow_ind`	1	2	3	2	3	4	1	2	3	4
`acol_ind`	1	1	1	2	3	3	4	4	4	4

- Indices may not be well ordered
- Is it easy to do operations? $A + B$, Ax
- $A + B$: if (i, j) are not ordered, difficult
- $y = Ax$:

Sparse Matrices: Storage Schemes III

```
for l = 1:nnz
   i = arow_ind(l)
   j = acol_ind(l)
   y(i) = y(i) + a(l)*x(j)
end
```

- `nnz`: usually used to represent the number of nonzeros
- x: vector in dense format
- In general we directly store a vector without using sparse format
- Access one column

Sparse Matrices: Storage Schemes IV

```
for l = 1:nnz
   if acol_ind(l) == i
      x(arow_ind(l)) = a(l)
   end
end
```

Cost: $O(nnz)$

When do we need to access a column? An example is to solve $Lx = b$

$$\begin{bmatrix} l_{11} & & & \\ l_{21} & l_{22} & & \\ \vdots & & \ddots & \\ l_{n1} & l_{n2} & & l_{nn} \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ x_n \end{bmatrix} = \begin{bmatrix} b_1 \\ b_2 \\ \vdots \\ b_n \end{bmatrix}$$

Sparse Matrices: Storage Schemes V

$$\begin{bmatrix} b_2 \\ \vdots \\ b_n \end{bmatrix} - x_1 \begin{bmatrix} l_{21} \\ \vdots \\ l_{n1} \end{bmatrix}$$

- A format that has the easy access of one column: Compressed column format

```
a         1 3 6 4 7 10 2 5 8 11
arow_ind  1 2 3 2 3 4  1 2 3  4
acol_ptr  1 4 5 7 11
```

- *j*th column:

  from `a(acol_ptr(j))` to `a(acol_ptr(j+1)-1)`

  Example: 3rd column

Sparse Matrices: Storage Schemes VI

```
acol_ptr(3) = 5
acol_prr(4) = 7
a(5) = 7
a(6) = 10
```

- `nnz = acol_ptr(n+1) - 1`

  `acol_ptr` contains $n+1$ elements
- $C = A + B$

  ```
  for j = 1:n
      get A's jth column
      get B's jth column
      do a vector addition
  end
  ```
- C is still with column format

Sparse Matrices: Storage Schemes VII

- $y = Ax = A_{:,1}x_1 + \cdots + A_{:,n}x_n$

```
for j = 1:n
   for l = acol_ptr(j):acol_ptr(j+1)-1
      y(arow_ind(l)) = y(arow_ind(l)) +
           a(l)*x(j)
   end
end
```

- Row indices of the same column may not be sorted

```
a        6 3 1 4 7 10 2 5 8 11
arow_ind 3 2 1 2 3 4  1 2 3  4
acol_ptr 1 4 5 7 11
```

- $C = AB$ is similar
- Access one column is easy

Sparse Matrices: Storage Schemes VIII

- Access one row is very difficult
- Compressed row format

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

`a`	1	2	3	4	5	6	7	8	10	11
`acol_ind`	1	4	1	2	4	1	3	4	3	4
`arow_ptr`	1	3	6	9	11					

Sparse Matrices: Storage Schemes IX

- An issue is that some languages start arrays with 0 but some with 1.
- In a C implementation we have

```
a        1 3 6 4 7 10 2 5 8 11
arow_ind 0 1 2 1 2 3  0 1 2  3
acol_ptr 0 3 4 6 10
```

- There are many variations of sparse structures.
- It's difficult to have standard sparse libraries as different formats are suitable for different matrices

Sparse Matrix and Factorization I

- This is a more advanced topic
- Factorization generates fill-ins

  fill-ins: new nonzero positions
- Consider the following Matlab program

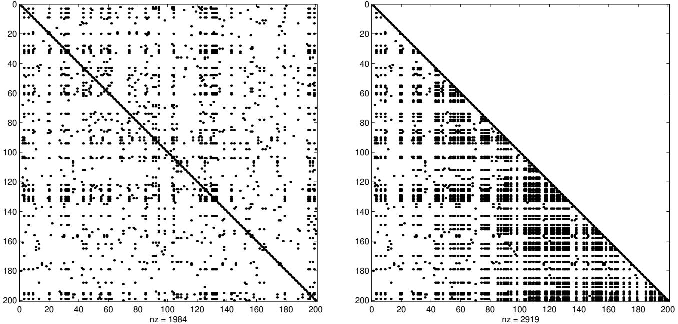
  ```
  A = sprandsym(200, 0.05, 0.01, 1) ;
  L = chol(A)' ;
  spy(A) ;
  print -deps A
  spy(L) ;
  print -deps L
  ```
- 0.05: density

  0.01: 1/(condition number)

Sparse Matrix and Factorization II

1: type of matrix, 1 gives a matrix with 1/(condition number) exactly 0.01

- spy: draw the sparsity pattern



(a) A (b) L

Sparse Matrix and Factorization III

- Clearly L is denser

Permutation and Reordering I

$$A = \begin{bmatrix} 3 & 2 & 1 & 2 \\ 2 & 4 & 0 & 0 \\ 1 & 0 & 5 & 0 \\ 2 & 0 & 0 & 6 \end{bmatrix},$$

$$\text{chol}(A) = \begin{bmatrix} 1.7321 & 0 & 0 & 0 \\ 1.1547 & 1.6330 & 0 & 0 \\ 0.5774 & -0.4082 & 2.1213 & 0 \\ 1.1547 & -0.8165 & -0.4714 & 1.9437 \end{bmatrix}$$

Permutation and Reordering II

$$P = \begin{bmatrix} 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 \\ 0 & 1 & 0 & 0 \\ 1 & 0 & 0 & 0 \end{bmatrix}, AP^T = \begin{bmatrix} 2 & 1 & 2 & 3 \\ 0 & 0 & 4 & 2 \\ 0 & 5 & 0 & 1 \\ 6 & 0 & 0 & 2 \end{bmatrix}$$

Permutation and Reordering III

$$
\begin{aligned}
& PAP^T \\
= & \begin{bmatrix} 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 \\ 0 & 1 & 0 & 0 \\ 1 & 0 & 0 & 0 \end{bmatrix} \begin{bmatrix} 2 & 1 & 2 & 3 \\ 0 & 0 & 4 & 2 \\ 0 & 5 & 0 & 1 \\ 6 & 0 & 0 & 2 \end{bmatrix} \\
= & \begin{bmatrix} 6 & 0 & 0 & 2 \\ 0 & 5 & 0 & 1 \\ 0 & 0 & 4 & 2 \\ 2 & 1 & 2 & 3 \end{bmatrix}
\end{aligned}
$$

Permutation and Reordering IV

$$\operatorname{chol}(PAP^T) = \begin{bmatrix} 2.4495 & 0 & 0 & 0 \\ 0 & 2.2361 & 0 & 0 \\ 0 & 0 & 2.0000 & 0 \\ 0.8165 & 0.4472 & 1.0000 & 1.0646 \end{bmatrix}$$

- $\operatorname{chol}(PAP^T)$ is sparser

  $$Ax = b$$
  $$(PAP^T)Px = Pb$$

  Get Px first and then x
- There are different ways of permutations

Permutation and Reordering V

- For example, MATLAB provides methods such as
    - Column Count Reordering
    - Reverse Cuthill-McKee Reordering
    - Minimum Degree Reordering
    - Nested Dissection Permutation
- Finding the ordering with the least entries in the factorization ⇒ minimum fill-in problem
- This is a difficult problem
- However, minimum fill-in may not be the best: we need to consider the numerical stability, implementation efforts, etc

Permutation and Reordering VI

- Subsequently we will discuss iterative methods, which do not have this issue of fill-ins