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

  \[
  \begin{align*}
  a & = 1 \ 3 \ 6 \ 4 \ 7 \ 10 \ 2 \ 5 \ 8 \ 11 \\
  arow\_ind & = 1 \ 2 \ 3 \ 2 \ 3 \ 4 \ 1 \ 2 \ 3 \ 4 \\
  acol\_ind & = 1 \ 1 \ 1 \ 2 \ 3 \ 3 \ 4 \ 4 \ 4 \ 4
  \end{align*}
  \]

- Indices may not be well ordered
- Is it easy to do operations? \( A + B, Ax \)
  - \( A + B \): if \((i, j)\) are not ordered, difficult
  - \( y = Ax \):
for \( l = 1 : \text{nnz} \)
\[
\begin{align*}
    i &= \text{arow\_ind}(l) \\
    j &= \text{acol\_ind}(l) \\
    y(i) &= y(i) + a(l) \times x(j)
\end{align*}
\]
end

- \text{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
for \( l = 1:nnz \)
\[
\text{if acol\_ind}(l) == i
\]
\[
x(\text{arow\_ind}(l)) = a(l)
\]
\end{verbatim}
\end{align*}

Cost: \( O(nnz) \)

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

\[
\begin{bmatrix}
  l_{11} &   &   \\
  l_{21} & l_{22} &   \\
    &   & \ddots \\
  l_{n1} & l_{n2} & \cdots & l_{nn}
\end{bmatrix}
\begin{bmatrix}
  x_1 \\
  x_2 \\
  \vdots \\
  x_n
\end{bmatrix}
= \begin{bmatrix}
  b_1 \\
  b_2 \\
  \vdots \\
  b_n
\end{bmatrix}
\]
A format that has the easy access of one column: Compressed column format

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

- 3rd column:
  from \( a(\text{acol_ptr}(j)) \) to \( a(\text{acol_ptr}(j+1)-1) \)

Example: 3rd column
Sparse Matrices: Storage Schemes VI

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

- $\text{nnz} = \text{acol_ptr}(n+1) - 1$
  - $\text{acol_ptr}$ contains $n + 1$ elements
- $C = A + B$
  - for $j = 1:n$
    - get A’s $j$th column
    - get B’s $j$th column
    - do a vector addition
  - end
- $C$ is still with column format
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

\begin{array}{c}
    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
\end{array}

\begin{itemize}
    \item C = AB is similar
    \item Access one column is easy
\end{itemize}
Access one row is very difficult.

Compressed row format

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

\[ \begin{align*}
a & = 1 2 3 4 5 6 7 8 10 11 \\
apcol_{\text{ind}} & = 1 4 1 2 4 1 3 4 3 4 \\
arow_{\text{ptr}} & = 1 3 6 9 11
\end{align*} \]
An issue is that some languages start arrays with 0 but some with 1.

In a C implementation we have:

\begin{verbatim}
  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
\end{verbatim}

There are many variations of sparse structures.

It’s difficult to have standard sparse libraries as different formats are suitable for different matrices.
This is a more advanced topic

Factorization generates fill-ins

fill-ins: new nonzero positions

Consider the following Matlab program

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

- spy: draw the sparsity pattern
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

\[ P = \begin{bmatrix} 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 \\ 0 & 1 & 0 & 0 \\ 1 & 0 & 0 & 0 \end{bmatrix}, \quad AP^T = \begin{bmatrix} 2 & 1 & 2 & 3 \\ 0 & 0 & 4 & 2 \\ 0 & 5 & 0 & 1 \\ 6 & 0 & 0 & 2 \end{bmatrix} \]
Permutation and Reordering III

\[
PAP^T = \begin{bmatrix} 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 \\ 0 & 1 & 0 & 0 \\ 1 & 0 & 0 & 0 \end{bmatrix} \begin{bmatrix} 2 & 1 & 2 & 3 \\ 0 & 0 & 4 & 2 \\ 0 & 5 & 0 & 1 \\ 6 & 0 & 0 & 2 \end{bmatrix} = \begin{bmatrix} 6 & 0 & 0 & 2 \\ 0 & 5 & 0 & 1 \\ 0 & 0 & 4 & 2 \\ 2 & 1 & 2 & 3 \end{bmatrix}
\]
Permutation and Reordering IV

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

- \text{chol}(PAP^T) \text{ 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 \(\Rightarrow\) 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
Subsequently we will discuss iterative methods, which do not have this issue of fill-ins.