WORKING WITH SPARSE MATRICES

- *See Chap. 3 of text*
- *Data structures for storing sparse matrices*
- *How these are implemented*
- *Basic operation: Sparse matrix by dense vector product*

Data Structures for sparse matrices

➤ The use of a proper data structures is critical to achieving good performance.

✍1 Generate a symmetric sparse matrix A in matlab and time the operations of accessing (only) all entries by columns and then by rows. Observations?

➤ Many data structures; sometimes unnecessary variants.

➤ These variants are more useful in the context of iterative methods

➤ Basic linear algebra kernels (e.g., matrix-vector products) depend on data structures.

Some Common Data Structures (from SPARSKIT)

DNS	Dense	**ELL**	Ellpack-Itpack
BND	Linpack Banded	**DIA**	Diagonal
COO	Coordinate	**BSR**	Block Sparse Row
CSR	Compressed Sparse Row	**SSK**	Symmetric Skyline
CSC	Compressed Sparse Column	**NSK**	Nonsymmetric Skyline
MSR	Modified CSR	**JAD**	Jagged Diagonal

- Most common (and important): CSR (/ CSC), COO

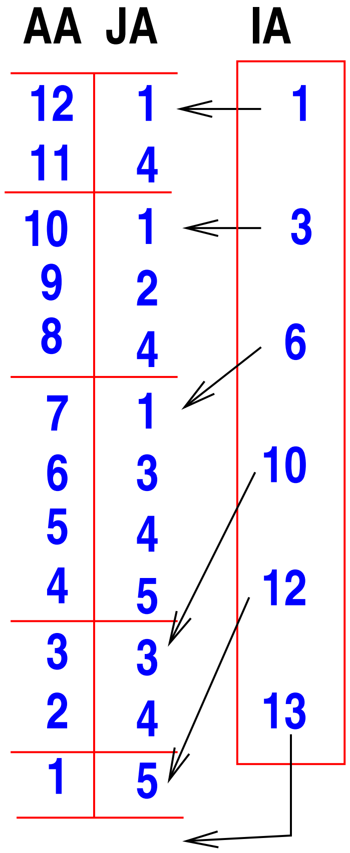
The coordinate format (COO)

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

AA	JR	JC
12.	5	5
9.	3	5
7.	3	3
5.	2	4
1.	1	1
2.	1	4
11.	4	4
3.	2	1
6.	3	1
4.	2	2
8.	3	4
10.	4	3

- Simplest data structure -
- Often used as 'entry' format in packages
- Variant used in matlab and matrix market
- Also known as 'triplet format'
- Note: order of entries is arbitrary [in matlab: sorted by columns]

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



- IA(j) points to beginning or row j in arrays AA, JA
- Related formats: Compressed Sparse Column format, Modified Sparse Row format (MSR).
- Used mainly in Fortran & portable codes – what about C?

CSR (CSC) format - C-style

* CSR: Collection of pointers of rows & array of row lengths

```
typedef struct SpaFmt {
/*---------------------------------------------
| C-style CSR format - used internally
| for all matrices in CSR/CSC format
|---------------------------------------------*/
  int n;            /* size of matrix              */
  int *nzcount;     /* length of each row          */
  int **ja;         /* to store column indices     */
  double **ma;      /* to store nonzero entries    */
} SparMat;
```

`aa[i][*]` == entries of i-th row (col.);

`ja[i][*]` == col. (row) indices,

`nzcount[i]` == number of nonzero elmts in row (col.) `i`

Data structure used in Csparse [T. Davis' SuiteSparse code]

```
typedef struct cs_sparse
{/* matrix in compressed-column or triplet form */
 int nzmax ; /* maximum number of entries */
 int m ;     /* number of rows */
 int n ;     /* number of columns */
 int *p ;    /* column pointers (size n+1) or
                col indices (size nzmax) */
 int *i ;    /* row indices, size nzmax */
 double *x ; /* numerical values, size nzmax */
 int nz ;    /* # of entries in triplet matrix,
                -1 for compressed-col */
} cs ;
```

- Can be used for CSR, CSC, and COO (triplet) storage
- Easy to use from Fortran

The Diagonal (DIA) format

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

$$DA = \boxed{\begin{matrix} * & 1. & 2. \\ 3. & 4. & 5. \\ 6. & 7. & 8. \\ 9. & 10. & * \\ 11 & 12. & * \end{matrix}} \qquad IOFF = \boxed{\begin{matrix} -1 & 0 & 2 \end{matrix}}$$

The Ellpack-Itpack format

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

$$\text{AC} = \boxed{\begin{array}{ccc} 1. & 2. & 0. \\ 3. & 4. & 5. \\ 6. & 7. & 8. \\ 9. & 10. & 0. \\ 11 & 12. & 0. \end{array}} \qquad \text{JC} = \boxed{\begin{array}{ccc} 1 & 3 & 1 \\ 1 & 2 & 4 \\ 2 & 3 & 5 \\ 3 & 4 & 4 \\ 4 & 5 & 5 \end{array}}$$

Block matrices

$$A = \left(\begin{array}{cc|cc|cc} \mathbf{1.} & \mathbf{2.} & \mathbf{0.} & \mathbf{0.} & \mathbf{3.} & \mathbf{4.} \\ \mathbf{5.} & \mathbf{6.} & \mathbf{0.} & \mathbf{0.} & \mathbf{7.} & \mathbf{8.} \\ \hline \mathbf{0.} & \mathbf{0.} & \mathbf{9.} & \mathbf{10.} & \mathbf{11.} & \mathbf{12.} \\ \mathbf{0.} & \mathbf{0.} & \mathbf{13.} & \mathbf{14.} & \mathbf{15.} & \mathbf{16.} \\ \hline \mathbf{17.} & \mathbf{18.} & \mathbf{0.} & \mathbf{0.} & \mathbf{20.} & \mathbf{21.} \\ \mathbf{22.} & \mathbf{23.} & \mathbf{0.} & \mathbf{0.} & \mathbf{24.} & \mathbf{25.} \end{array}\right)$$

$$\text{AA} = \begin{array}{|cc|cc|cc|} \hline 1. & 3. & 9. & 11. & 17. & 20. \\ 5. & 7. & 15. & 13. & 22. & 24. \\ 2. & 4. & 10. & 12. & 18. & 21. \\ 6. & 8. & 14. & 16. & 23. & 25. \\ \hline \end{array}$$

$$\text{JA} = \boxed{1 \;\; 5 \;\; 3 \;\; 5 \;\; 1 \;\; 5}$$

$$\text{IA} = \boxed{1 \;\; 3 \;\; 5 \;\; 7}$$

- Columns of AA hold 2 x 2 blocks. JA(k) = col. index of (1,1) entries of k-th block. FORTRAN: declare as AA(2,2,6)

- Can also store the blocks row-wise in AA.

$$AA = \begin{array}{|cccc|} \hline 1. & 5. & 2. & 6. \\ 3. & 7. & 4. & 8. \\ \hline 9. & 15. & 10. & 14. \\ 11. & 13. & 12. & 16. \\ \hline 17. & 22. & 18. & 23. \\ 20. & 24. & 21. & 25. \\ \hline \end{array} \qquad \begin{array}{l} JA = \boxed{1\ \ 5\ \ 3\ \ 5\ \ 1\ \ 5} \\ IA = \boxed{1\ \ 3\ \ 5\ \ 7} \end{array}$$

- In other words AA is simply transposed

✍2 What are the advantages and disadvantages of each scheme?

- Block formats are important in many applications..

- Also valuable: block structure with variable block size.

Sparse matrices – data structure in C

- Recall:

```
typedef struct SpaFmt {
/*--------------------------------------------
| C-style CSR format - used internally
| for all matrices in CSR format
|---------------------------------------------*/
  int n;
  int *nzcount;  /* length of each row */
  int **ja;      /*  to store column indices  */
  double **ma;   /*  to store nonzero entries */
} CsMat, *csptr;
```

- Can store rows of a matrix (CSR) – or its columns (CSC)
- Q: How to perform the operation $y = A * x$ in each case?

Matvec – row version

```
void matvec( csptr mata, double *x, double *y )
{
    int i, k, *ki;
    double *kr;
    for (i=0; i<mata->n; i++) {
        y[i] = 0.0;
        kr = mata->ma[i];
        ki = mata->ja[i];
        for (k=0; k<mata->nzcount[i]; k++)
            y[i] += kr[k] * x[ki[k]];
    }
}
```

➤ Uses sparse dot products (sparse SDOTS)

✍3 Operation count

Matvec – Column version

```
void matvecC( csptr mata, double *x, double *y )
{
  int n = mata->n, i, k, *ki;
  double *kr;
  for (i=0; i<n; i++)
    y[i] = 0.0;
  for (i=0; i<n; i++) {
    kr = mata->ma[i];
    ki = mata->ja[i];
    for (k=0; k<mata->nzcount[i]; k++)
      y[ki[k]] += kr[k] * x[i];
  }
}
```

➤ Uses sparse vector combinations (sparse SAXPY)

✍4 Operation count

➤ Using the CS data structure from Suite-Sparse:

```
int cs_gaxpy (cs *A, double *x, double *y) {
 int p, j, n, *Ap, *Ai;
 n = A->n; Ap = A-> p; Ai = A->i; Ax = A->x;
 for (j=0; j<n; j++) {
   for (p=Ap[j]; p<Ap[j+1];p++)
     y[Ai[p]] += Ax[p]*x[j];
 }
return(1)
}
```

Matvec – row version - FORTRAN

```
      subroutine amux (n, x, y, a, ja, ia)
      real*8  x(*), y(*), a(*), t
      integer n, ja(*), ia(*), i, k
c----- row loop
      do 100 i = 1,n
c----- inner product of row i with vector x
         t = 0.0d0
         do 99 k=ia(i), ia(i+1)-1
            t = t + a(k)*x(ja(k))
  99     continue
         y(i) = t
 100  continue
      return
      end
```

Matvec – column version - FORTRAN

```
      subroutine atmux (n, x, y, a, ja, ia)
      real*8 x(*), y(*), a(*)
      integer n, ia(*), ja(*)
      integer i, k
c----- set y to zero
      do 1 i=1,n
         y(i) = 0.0
 1    continue
c----- column loop
      do 100 i = 1,n
c----- sparse saxpy
         do 99 k=ia(i), ia(i+1)-1
            y(ja(k)) = y(ja(k)) + x(i)*a(k)
  99     continue
 100  continue
c
      return
      end
```

Sparse matrices in matlab

✍5 Generate a tridiagonal matrix T

✍6 Convert T to sparse format

✍7 See how you can generate this sparse matrix directly using `sparse`

✍8 See how you can use `spconvert` to achieve the same result

✍9 What can you observe about the way the triplets of a sparse matrix are ordered?

✍10 Important for performance: `spalloc`. See the difference between

`A = sparse(m,n)` and `A = spalloc(m,n,nzmax)`

Sparse matrices in Python

➤ Must load *scipy.sparse* : For example, *import sipy.sparse as sp*

➤ Difference with matlab: Must explicitly specify a matrix format. [→ a matrix is not just 'sparse' it is *csr_sparse* for example]

➤ Formats available:

- *bsr_matrix* - Block Sparse Row matrix
- *coo_matrix* - A sparse matrix in COOrdinate format
- *csc_matrix* - Compressed Sparse Column matrix
- *csr_matrix* - Compressed Sparse Row matrix
- *dia_matrix* - Sparse matrix with DIAgonal storage
- *dok_matrix* - Dictionary Of Keys based sparse matrix
- *lil_matrix* - Row-based list of lists sparse matrix

- Can get info on a matrix by accessing fields like: *S.nnz* or *S.shape*
- Also for CSR: *S.data*, *S.indices*, and *S.indptr*, give the arrays *AA*, *JA*, *IA*, seen earlier (vals., col., ptr)
- Convert to another format: e.g., *S.tocsc()*
- Equivalent of `full` on matlab: *A = S.todense()*
- Inverse operation: *A = sp.csr_matrix(S)*

✍11 Create a 6×6 tridiagonal matrix (dense mode) and convert it to a *csr* matrix.

✍12 Try to understand the *dok*, *lil* formats.