Lecture 10

Sparse Matrices, Iterative Methods

David Semeraro

University of Illinois at Urbana-Champaign

September 26, 2013

An application

Latent semantic analysis (LSA) analyzes two-mode data. Looks at relationships between documents and terms.

- natural language processing
- information retrieval
- information filtering
- textual machine learning

Document-term matrix: Document1(D1) = ”I love numerical analysis”
Document1(D2) = ”I do not love numerical analysis, but I love linear algebra.”

	I	love	numerical	linear	algebra
D1	1	1	1	0	0
D2	1	2	1	1	1

An application

	I	love	numerical	linear	algebra
D1	1	1	1	0	0
D2	1	2	1	1	1

One method for weights: Term Count Model
Variation: Term Frequency-Inverse Document Frequency; weight the entries inversely, highlighting infrequent terms

Let X be the matrix of occurrences (or the inverse).

$$X = \begin{bmatrix} x_{1,1} & \cdots & x_{1,n} \\ \vdots & \ddots & \vdots \\ x_{m,1} & \cdots & x_{m,n} \end{bmatrix}$$

Now each row will be a vector relating a term to all documents. Each column will be a vector relating a document to all terms.

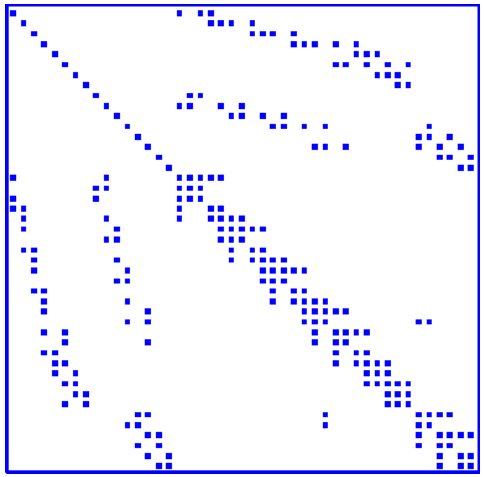
An application

$$X = \begin{bmatrix} x_{1,1} & \cdots & x_{1,n} \\ \vdots & \ddots & \vdots \\ x_{m,1} & \cdots & x_{m,n} \end{bmatrix}$$

- X has many zeros
- a dot product of the rows gives the correlation between terms over the documents
- XX^T gives a cumulative view of the correlation
- same with $X^T X$
- singular value decompositions, eigenvalue analysis, etc give other information

Sparse Matrices

ack: Y. Saad



- Vague definition: matrix with few nonzero entries
- For all practical purposes: an $m \times n$ matrix is sparse if it has $\mathcal{O}(\min(m, n))$ nonzero entries.
- This means roughly a constant number of nonzero entries per row and column

Sparse Matrices

ack: Y. Saad

- Other definitions use a slow growth of nonzero entries with respect to n or m.
- Wilkinson's Definition: "..matrices that allow special techniques to take advantage of the large number of zero elements." (J. Wilkinson)"
- A few applications which lead to sparse matrices: Structural Engineering, Computational Fluid Dynamics, Reservoir simulation, Electrical Networks, optimization, data analysis, information retrieval (LSI), circuit simulation, device simulation, . . .

Sparse Matrices: The Goal

- To perform standard matrix computations economically i.e., without storing the zeros of the matrix.
- For typical Finite Element /Finite difference matrices, number of nonzero elements is $\mathcal{O}(n)$.

Example

To add two square dense matrices of size n requires $\mathcal{O}(n^2)$ operations. To add two sparse matrices A and B requires $\mathcal{O}(nnz(A) + nnz(B))$ where $nnz(X) =$ number of nonzero elements of a matrix X.

remark

A^{-1} is usually dense, but L and U in the LU factorization may be reasonably sparse (if a good technique is used).

Goal

- Principle goal: *solve*

$$Ax = b$$

  where $A \in \mathbb{R}^{n \times n}$, $x, b \in \mathbb{R}^n$
- Assumption: A is very sparse
- General approach: iteratively improve the solution
- Given x_0, ultimate "correction" is

$$x_1 = x_0 + e_0$$

  where $e_0 = x - x_0$, thus $Ae_0 = Ax - Ax_0$,
- or

$$x_1 = x_0 + A^{-1} r_0$$

  where $r_0 = b - Ax_0$

Goal

- Principle difficulty: how do we "approximate" $A^{-1}r$ or reformulate the iteration?
- One simple idea:

$$x_1 = x_0 + \alpha r_0$$

- operation is inexpensive if r_0 is inexpensive
- requires very fast sparse mat-vec (matrix-vector multiply) Ax_0

Sparse Matrices

- So how do we store A?
- Fast mat-vec is certainly important; also ask
  - what type of access (rows, cols, diag, etc)?
  - dynamic allocation?
  - transpose needed?
  - inherent structure?
- Unlike dense methods, not a lot of standards for iterative
  - dense BLAS have been long accepted
  - sparse BLAS still iterating
- Even data structures for dense storage not as obvious
- Sparse operations have low operation/memory reference ratio

Sparse Matrix Qualification

Matrix Market attempts to classify the sparse matrix.
First Qualification (type of values and number of values):

identifier	description
Real	All entries are float
Complex	All entries are a pair of float
Integer	All entries are int
Pattern	Matrix is a pattern. Actual entries are omitted
Parallel	*Parallel structure is identified*

Sparse Matrix Qualification

Second Qualification (interpreting values):

identifier	description
General	A has no symmetry, no symmetry is utilized, or A is not square
Symmetric	$a_{ij} = a_{ji}$; only entries on the diagonal and below(or above) are stored.
Skew-Symmetric	$a_{ij} = -a_{ji}$; only entries below (or above) the diagonal ($= 0$) are stored.
Hermitian	$a_{ij} = \bar{a}_{ji}$; only entries on the diagonal and below (or above) are stored.

see “The Matrix Market Exchange Formats: Initial Design” by Boisvert, Pozo, Remington

Popular Storage Structures

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	**BSR**	Nonsymmetric Skyline
MSR	Modified CSR	**JAD**	Jagged Diagonal
LIL	Linked List		

note: CSR = CRS, CCS = CSC, SSK = SKS in some references

DNS

$$A = \begin{bmatrix} 1.0 & 2.0 & 3.0 \\ 4.0 & 5.0 & 6.0 \\ 7.0 & 8.0 & 9.0 \end{bmatrix}$$

$$AA = \begin{bmatrix} 3 & 3 & 1.0 & 2.0 & 3.0 & 4.0 & 5.0 & 6.0 & 7.0 & 8.0 & 9.0 \end{bmatrix}$$

- simple
- row-wise
- easy blocked formats

COO

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

$$\begin{array}{lcl} AA & = & [\ 12.0\ \ 9.0\ \ 7.0\ \ 5.0\ \ 1.0\ \ 2.0\ \ 11.0\ \ 3.0\ \ 6.0\ \ 4.0\ \ 8.0\ \ 10.0\ ] \\ JR & = & [\ 5\ \ 3\ \ 3\ \ 2\ \ 1\ \ 1\ \ 4\ \ 2\ \ 3\ \ 2\ \ 3\ \ 4\ ] \\ JC & = & [\ 5\ \ 5\ \ 3\ \ 4\ \ 1\ \ 4\ \ 4\ \ 1\ \ 1\ \ 2\ \ 4\ \ 3\ ] \end{array}$$

- simple, often used for entry

Question: Do you need this much storage?

CSR

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

$$\begin{array}{lllllllllllll} AA & = & [& 1.0 & 2.0 & 3.0 & 4.0 & 5.0 & 6.0 & 7.0 & 8.0 & 9.0 & 10.0 & 11.0 & 12.0 &] \\ JA & = & [& 1 & 4 & 1 & 2 & 4 & 1 & 3 & 4 & 5 & 3 & 4 & 5 &] \\ IA & = & [& 1 & 3 & 6 & 10 & 12 & 13 &] & & & & & & \end{array}$$

- Length of AA and JA is nnz; length of IA is $n+1$
- $IA(j)$ gives the index (offset) to the beginning of row j in AA and JA (one origin due to Fortran)
- no structure, fast row access, slow column access (why?)
- related: CSC, MSR

MSR

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

$$\begin{array}{lllllllllllll} AA = & [1.0 & 4.0 & 7.0 & 11.0 & 12.0 & * & 2.0 & 3.0 & 5.0 & 6.0 & 8.0 & 9.0 & 10.0] \\ JA = & [7 & 8 & 10 & 13 & 14 & 14 & 4 & 1 & 4 & 1 & 4 & 5 & 3] \end{array}$$

- places importance on diagonal (often nonzero and accessed frequently)
- first n entries are the diag
- $n+1$ is empty
- rest of AA are the nondiagonal entries
- first $n+1$ entries in JA give the index (offset) of the beginning of the row (the IA of CSR is in this JA)
- rest of JA are the columns indices

DIA

or CDS

$$A = \begin{bmatrix} 1 & 0 & 2 & 0 & 0 \\ 3 & 4 & 0 & 5 & 0 \\ 0 & 6 & 7 & 0 & 8 \\ 0 & 0 & 9 & 10 & 0 \\ 0 & 0 & 0 & 11 & 12 \end{bmatrix} \quad DIAG = \begin{bmatrix} * & 1.0 & 2.0 \\ 3.0 & 4.0 & 5.0 \\ 6.0 & 7.0 & 8.0 \\ 9.0 & 10.0 & * \\ 11.0 & 12.0 & * \end{bmatrix} \quad IOFF = \begin{bmatrix} -1 & 0 & 2 \end{bmatrix}$$

- need to know the offset structure
- some entries will always be empty

ELL

$$A = \begin{bmatrix} 1 & 0 & 2 & 0 & 0 \\ 3 & 4 & 0 & 5 & 0 \\ 0 & 6 & 7 & 0 & 8 \\ 0 & 0 & 9 & 10 & 0 \\ 0 & 0 & 0 & 11 & 12 \end{bmatrix} \quad COEF = \begin{bmatrix} 1.0 & 2.0 & 0.0 \\ 3.0 & 4.0 & 5.0 \\ 6.0 & 7.0 & 8.0 \\ 9.0 & 10.0 & 0.0 \\ 11.0 & 12.0 & 0.0 \end{bmatrix} \quad JCOEF = \begin{bmatrix} 1 & 3 & 1 \\ 1 & 2 & 4 \\ 2 & 3 & 5 \\ 3 & 4 & 4 \\ 4 & 5 & 5 \end{bmatrix}$$

- Form columns from first non-zero in each row, repeat.
- used more on vector machines (what? why?)
- assumes low number of nnz per row (=number of columns in $COEFF$ and $JCOEFF$)

JDS

- like DIA (and CDS),but more space-efficient
- costs more to gather and scatter

Take

$$A = \begin{bmatrix} 10 & -3 & 0 & 1 & 0 & 0 \\ 0 & 9 & 6 & 0 & -2 & 0 \\ 3 & 0 & 8 & 7 & 0 & 0 \\ 0 & 6 & 0 & 7 & 5 & 4 \\ 0 & 0 & 0 & 0 & 9 & 13 \\ 0 & 0 & 0 & 0 & 5 & -1 \end{bmatrix}$$

And shift:

$$A \rightarrow \begin{bmatrix} 10 & -3 & 1 & \\ 9 & 6 & -2 & \\ 3 & 8 & 7 & \\ 6 & 7 & 5 & 4 \\ 9 & 13 & & \\ 5 & -1 & & \end{bmatrix}$$

$$A \rightarrow \begin{bmatrix} 10 & -3 & 1 & \\ 9 & 6 & -2 & \\ 3 & 8 & 7 & \\ 6 & 7 & 5 & 4 \\ 9 & 13 & & \\ 5 & -1 & & \end{bmatrix}$$

Now store the columns and column indices:

$$VAL = \begin{bmatrix} 10 & 9 & 3 & 6 & 9 & 5 \\ -3 & 6 & 8 & 7 & 13 & -1 \\ 1 & -2 & 7 & 5 & 0 & 0 \\ 0 & 0 & 0 & 4 & 0 & 0 \end{bmatrix} \quad COL = \begin{bmatrix} 1 & 2 & 1 & 2 & 5 & 5 \\ 2 & 3 & 3 & 4 & 6 & 6 \\ 4 & 5 & 4 & 5 & 0 & 0 \\ 0 & 0 & 0 & 6 & 0 & 0 \end{bmatrix}$$

JDS

- now reorder in terms of largest to smallest nnz in each row
- in JDS, the number of jagged diagonals is the number of nnz in the first row of VAL. This is the max nnz in any row.
- $PERM$: permutation array to reorder rows
- $JDIAG$: jagged diags in order
- COL: column indices
- $JDPTR$: points to the beginning of each diagonal
- advantage for mat-mat (see “Krylov subspace methods on supercomputers” by Saad)
- this is actually ITPACK or Purdue storage

$$
\begin{array}{ll}
JDIAG = & [6 \quad 9 \quad 3 \quad 10 \quad 9 \quad 5; \quad 7 \quad 6 \quad 8 \quad -3 \quad 13 \quad -1; \quad 5 \quad -2 \quad 7 \quad 1; \quad 4] \\
COL = & [2 \quad 2 \quad 1 \quad 1 \quad 5 \quad 5; \quad 4 \quad 3 \quad 3 \quad 2 \quad 6 \quad 6; \quad 5 \quad 5 \quad 4 \quad 4; \quad 6] \\
PERM = & [4 \quad 2 \quad 3 \quad 1 \quad 5 \quad 6] \\
JDPTR = & [1 \quad 7 \quad 13 \quad 17]
\end{array}
$$

Blocked

$$A = \begin{bmatrix} 1.0 & 2.0 & 0.0 & 0.0 & 3.0 & 4.0 \\ 5.0 & 6.0 & 0.0 & 0.0 & 7.0 & 8.0 \\ 0.0 & 0.0 & 9.0 & 10.0 & 11.0 & 12.0 \\ 0.0 & 0.0 & 13.0 & 14.0 & 15.0 & 16.0 \\ 17.0 & 18.0 & 0.0 & 0.0 & 20.0 & 21.0 \\ 22.0 & 23.0 & 0.0 & 0.0 & 24.0 & 25.0 \end{bmatrix}$$

$$AA = \begin{bmatrix} 1.0 & 3.0 & 9.0 & 11.0 & 17.0 & 20.0 \\ 5.0 & 7.0 & 15.0 & 13.0 & 22.0 & 24.0 \\ 2.0 & 4.0 & 10.0 & 12.0 & 18.0 & 21.0 \\ 6.0 & 8.0 & 14.0 & 16.0 & 23.0 & 25.0 \end{bmatrix}$$

$$JA = \begin{bmatrix} 1 & 5 & 3 & 5 & 5 & 1 & 5 \end{bmatrix}$$

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

Blocked

- each column of AA is a 2×2 block
- $JA(k) =$ column index of $(1,1)$ entries of the kth block
- declared as $AA(2,2,6)$
- blocks arise in *many* apps
- variant: variable block size

Blocked

Also row-wise

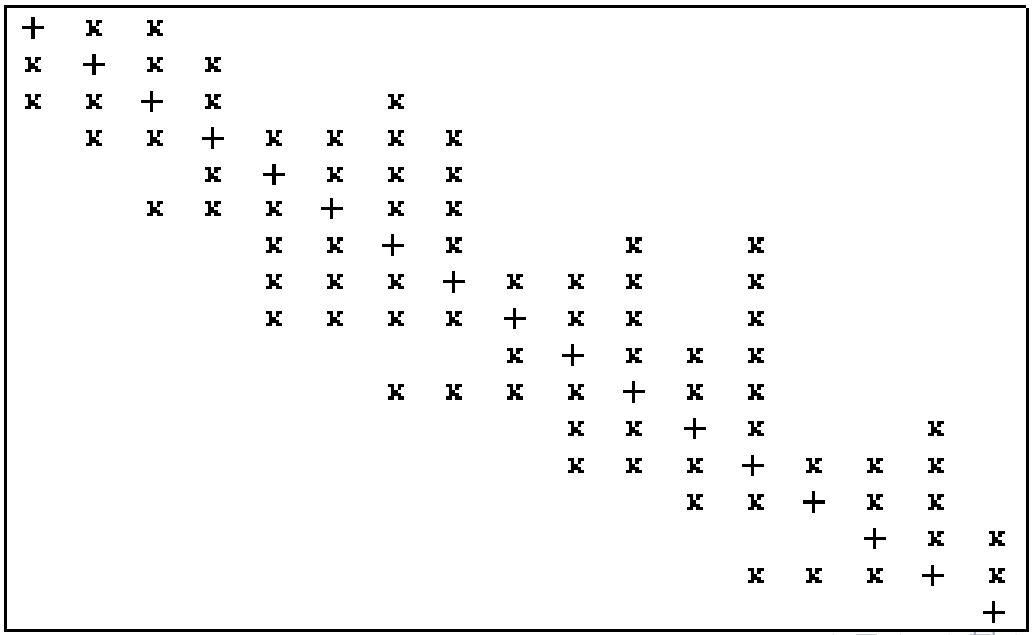
$$AA = \begin{bmatrix} 1.0 & 5.0 & 2.0 & 6.0 \\ 3.0 & 7.0 & 4.0 & 8.0 \\ 9.0 & 15.0 & 10.0 & 14.0 \\ 11.0 & 13.0 & 12.0 & 16.0 \\ 17.0 & 22.0 & 18.0 & 23.0 \\ 20.0 & 24.0 & 21.0 & 25.0 \end{bmatrix}$$

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

- each row of AA is a 2×2 block (can be a drawback)
- JA, IA same, $AA(6, 2, 2)$
- if elements of blocks are accessed at the same time: rows are better (C)
- if elements of similar positions in different blocks are accessed at the same time: columns are better (C)

SSK, NSK

- for “skyline” matrices (variable band, see “Direct methods for sparse matrices” by Duff, Erisman, Reid)
- can be used for diagonal block matrices
- skyline structure is preserved in basic GE
- for symmetric: Place all the rows (in order) into VAL
- IA points to the beginning of each row
- JA implicit
- for nonsymmetric: store L in SSK format, then U in column-wise SSK see “SPARSEKIT” by Saad



LIL

- Similar to CSR, but rather than a flat AA vector, each row is a linked list of elements
- first element of each row is accessed by $ROOT$
- each element in AA has a corresponding $NEXT$ entry
- -1 indicates the end of a row
- column lookup take $\mathcal{O}(nnz)$; one semi-costly fix: store a columnwise index in the same way as rows.
- very good element insertion time, but more memory

try it...

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

- CSR
- CSC
- COO
- LIL

Example

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

i	IA	JA	AA
1	2	2	1
2	3	4	2
3	4	5	5
4	2	3	2
5	5	6	4
6	1	1	7
7	5	5	6
8	3	2	2

COO

i	IA	JA	AA
1	1	1	7
2	2	2	1
3	4	3	2
4	6	2	2
5	7	4	2
6	9	5	5
7	-	5	6
8	-	6	4

CSR

i	IA	JA	AA	$NEXT$
1	4	3	2	-1
2	3	2	2	5
3	2	2	1	1
4	8	1	7	-1
5	6	4	2	-1
6	-	5	6	7
7	-	6	4	-1
8	-	5	5	-1

LIL

Sparse Matrix-Vector Multiply

$z = Ax$, $A_{m \times n}$, $x_{n \times 1}$, $z_{m \times 1}$

```
1 input A, x
2 z = 0
3 for i = 1 to m
4    for col = A(i,:)
5       z(i) = z(i) + A(i,col)x(col)
6    end
7 end
```

- difference between CSR and LIL is computing line 4
- CSR: rows are contiguous...(next slide)
- LIL: follow rows through linked list

Sparse Matrix-Vector Multiply

CSR

$z = Ax, A_{m \times n}, x_{n \times 1}, z_{m \times 1}$

```
1 DO I=1, m
2   Z(I)=0
3   K1 = IA(I)
4   K2 = IA(I+1)-1
5   DO J=K1, K2
6     z(I) = z(I) + A(J)*x(JA(J))
7   ENDDO
8 ENDDO
```

- $\mathcal{O}(nnz)$
- marches down the rows
- very cheap

Sparse Matrix-Vector Multiply

CSR and Data Streams

$z = Ax$, $A_{m \times n}$, $x_{n \times 1}$, $z_{m \times 1}$

```
1  DO I=1, M/2
2    z(I)=0
3    z(I+M/2)=0
4    K1 = IA(I)
5    K2 = IA(I+1)-1
6    K3 = IA(I+M/2)
7    K4 = IA(I+M/2+1)-1
8    DO J=0,MIN(K2-K1,K4-K3)
9      z(I) = z(I) + A(K1+J)*x(JA(K1+J))
10     z(I+M/2) = z(I+M/2) + A(K3+J)*x(JA(K3+J))
11   ENDDO
12   ! ... finish up
13 ENDDO
```

- IA structure allows two data streams

Sparse Matrix-Matrix Multiply

- ways to optimize ("SMPP", Douglas, Bank)

$Z = AB$, $A_{m \times n}$, $B_{n \times p}$, $z_{m \times p}$

```
1 for i = 1 to m
2   for j = 1 to n
3     Z(i,j) = dot(A(i,:), B(:,j))
4   end
5 end
6 return Z
```

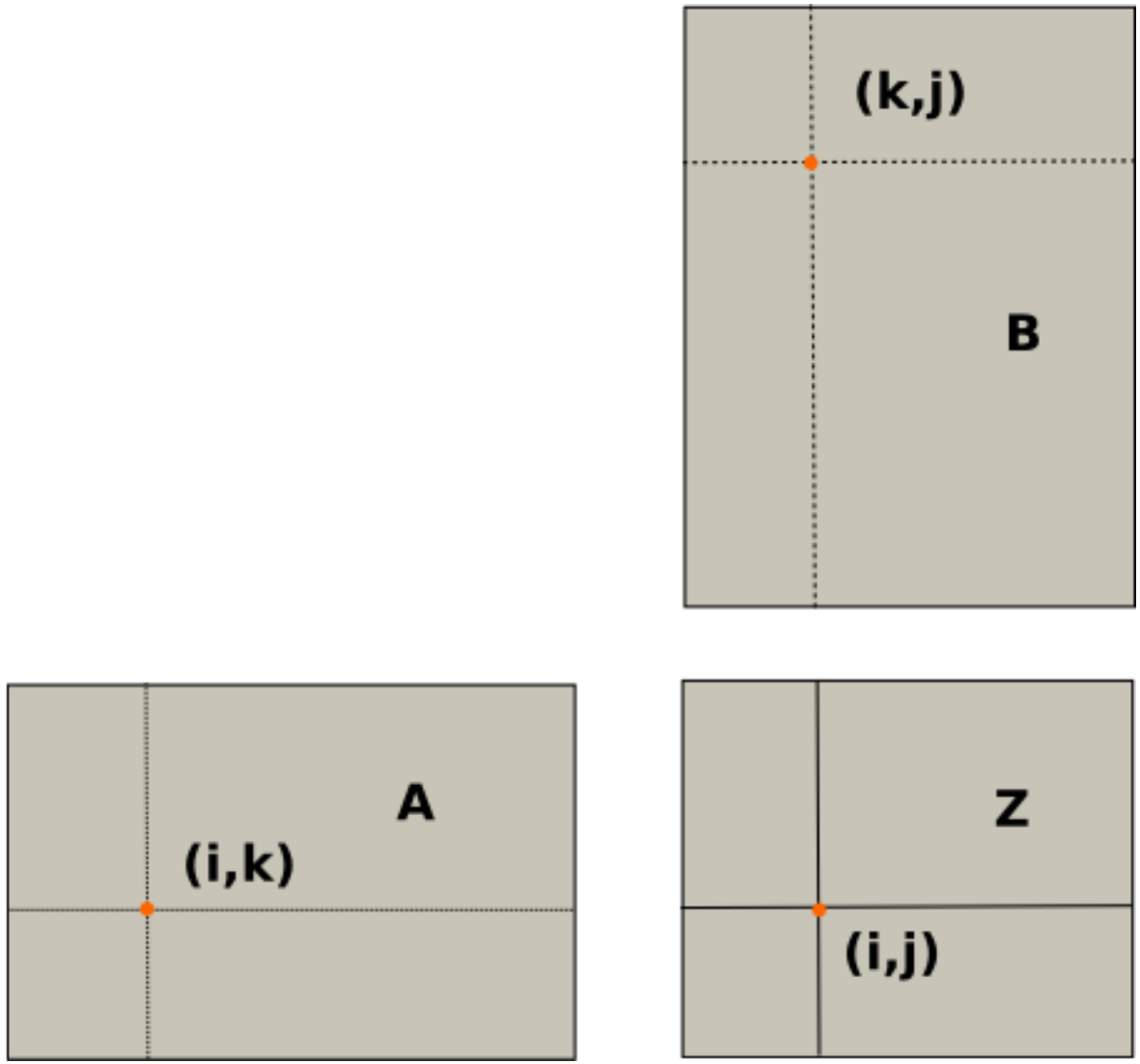
- obvious problem: column selection of B is expensive for CSR
- not-so-obvious problem: Z is sparse(!!), but the algorithm doesn't account for this.

Sparse Matrix-Matrix Multiply

$Z = AB$, $A_{m \times n}$, $B_{n \times p}$, $z_{m \times p}$

```
1 Z=0
2 for i = 1 to m
3   for colA = A(i,:)
4     for colB = A(colA,:)
5       Z(i,colB) += A(i,colA) · B(colA,colB)
6     end
7   end
8 end
9 return Z
```

- only marches down rows
- only computes nonzero entries in Z (aside from fortuitous subtractions)
- line 5 will do and insert into Z. Two options:
  1. precompute sparsity of Z in CSR
  2. use LIL for Z


(k,j)
B
A
(i,k)
Z
(i,j)

Some Python

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

i	IA	JA	AA
1	2	2	1
2	3	4	2
3	4	5	5
4	2	3	2
5	5	6	4
6	1	1	7
7	5	5	6
8	3	2	2

COO

```
1 from scipy import sparse
2 from numpy import array
3 IA=array([1,2,3,1,4,0,4,2])
4 JA=array([1,3,4,2,5,0,4,1])
5 V=array([1,2,5,2,4,7,6,2])
6
7 A=sparse.coo_matrix((V,(IA,JA)),shape=(5,6))
```

Some Python

From COO to CSC:

```
from scipy import sparse
from numpy import array
import pprint
IA=array([1,2,3,1,4,0,4,2])
JA=array([1,3,4,2,5,0,4,1])
V=array([1,2,5,2,4,7,6,2])

A=sparse.coo_matrix((V,(IA,JA)),shape=(5,6)).tocsr()
```

Nonzeros:

```
print(A.nnz)
```

To full and view:

```
B=A.todense()
pprint.pprint(B)
```

Simple Matrix Iterations

- Solve

$$Ax = b$$

- Assumption: A is very sparse
- Let $A = N + M$, then

$$\begin{aligned} Ax &= b \\ (N+M)x &= b \\ Nx &= b - Mx \end{aligned}$$

- Make this into an iteration:

$$\begin{aligned} Nx_k &= b - Mx_{k-1} \\ x_k &= N^{-1}(b - Mx_{k-1}) \end{aligned}$$

- Careful choice of N and M can give effective methods
- More powerful iterative methods exist