TDB – NLA

Numerical Linear Algebra

Self reading: Sparse matrices, factorization

Maya Neytcheva, TDB, Feb-March 2021

Plan of the lecture:

- Sparse matrices and why are those a topic of special interest?
- Handling sparse matrices. Sparse data formats (FYI)
- Solution methods for sparse matrices
  - Direct methods
    - Fill-ins and can we get rid of them?
    - Reordering strategies
    - Sparse Cholesky factorization
    - Sparse QR, SVD
- Examples

Large matrices

What has been and is considered as large through the years $N(t)$

1970	200
1975	1000
1980	10000
1985	100000
1990	250000
1995	500000
2000	2000000
since 2005	500000000

What is a sparse matrix?

$$A(N \times N), \quad nnz(A) = kN, \quad 2 \leq k \leq logN$$

Sparse matrix storage schemes

Sparse matrix storage schemes

There are more than 20 different sparse storage schemes...

Sparse matrix storage schemes

Coordinate scheme:

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

V :	1	2	3	4	5	6
I :	1	1	2	2	3	4
J :	2	3	1	2	3	1

Advantages and disadvantages

Sparse matrix storage schemes

Diagonal-wise storage scheme:

$$A = \begin{bmatrix} a_{11} & a_{12} & 0 & a_{14} & 0 & 0 \\ a_{21} & a_{22} & a_{23} & 0 & a_{25} & 0 \\ 0 & a_{32} & a_{33} & a_{34} & 0 & a_{36} \\ 0 & 0 & a_{43} & a_{44} & a_{45} & 0 \\ 0 & 0 & 0 & a_{54} & a_{55} & a_{56} \\ 0 & 0 & 0 & 0 & a_{65} & a_{66} \end{bmatrix} \qquad V = \begin{bmatrix} 0 & a_{11} & a_{12} & a_{14} \\ a_{21} & a_{22} & a_{23} & a_{25} \\ a_{32} & a_{33} & a_{34} & a_{36} \\ a_{43} & a_{44} & a_{45} & 0 \\ a_{54} & a_{55} & a_{56} & 0 \\ a_{65} & a_{66} & 0 & 0 \end{bmatrix}$$

OF : -1 0 1 3

Sparse matrix storage schemes

Sparse compressed schemes: $A = \begin{bmatrix} 0 & 1 & 2 & 0 \\ 3 & 4 & 0 & 0 \\ 0 & 0 & 5 & 0 \\ 6 & 0 & 0 & 0 \end{bmatrix}$

V :	1	2	3	4	5	6
C :	2	3	1	2	3	1
R :	1	3	5	6	7	

(a) CSR

V :	3	6	1	4	2	5
R :	2	4	1	2	3	3
C :	1	3	5	7	7	

(b) CSC

Sparse matrix storage schemes

Jagged diagonal storage: The Jagged Diagonal Storage format can be useful for the implementation of iterative methods on parallel and vector processors. Like the Compressed Diagonal format, it gives a vector length essentially of the size of the matrix. It is more space-efficient than CDS at the cost of a gather/scatter operation.

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

col_ind(:,1)	1	2	1	2	5	5
col_ind(:,1)	2	3	3	4	6	6
col_ind(:,1)	4	5	4	5	0	0
col_ind(:,1)	0	0	0	6	0	0

Jagged diagonals, cont.

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

vals	6	9	3	10	9	5;	7	6	8	-3	13	-1;	5	-2	7	1;	4;
cols	2	2	1	1	5	5;	4	3	3	2	6	6;	5	5	4	4;	6;

perm	4	2	3	1	5	6

jd_ptr	1	7	13	17

LU factorization for sparse matrices

Direct methods: $A = LU,\ LU\mathbf{x} = \mathbf{b},\ L\mathbf{y} = \mathbf{b},\ U\mathbf{x} = \mathbf{y}$

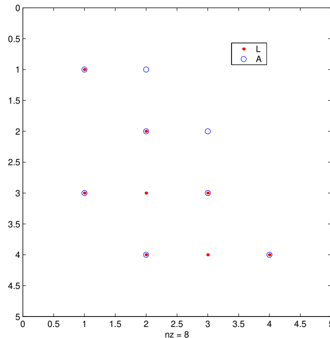
Triangular factorization for the case of *sparse* matrices.
Note: In general, during factorization we have to do pivoting in order to assure numerical stability.
The computational complexity of a direct solution algorithm is as follows.

Type of matrix A	Factor	LU solve	Memory
general dense	$2/3n^3$	$O(n^2)$	$n(n+1)$
symmetric dense	$1/3n^3$	$O(n^2)$	$1/2n(n+1)$
band matrix $(2q+1)$	$O(q^2n)$	$O(qn)$	$n(2q+1)$

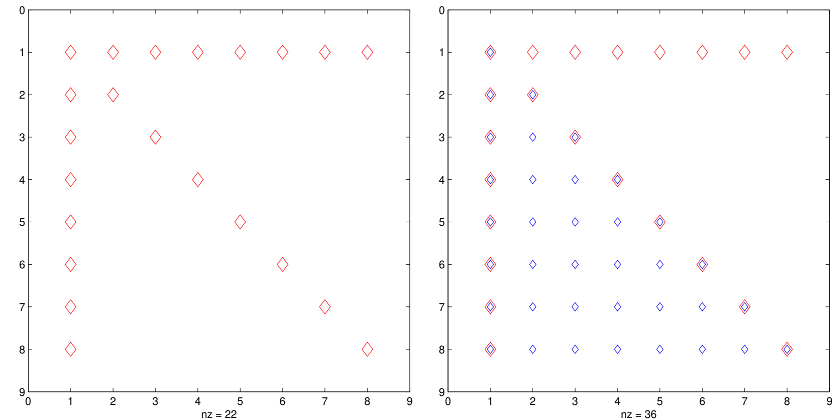
The reason to consider particularly factorizations of sparse matrices

is the effect of *fill-in*, namely, obtaining nonzero entries in the LU factors in positions where $A_{i,j}$ is zero.

$$a_{i,j}^{(k+1)} \longleftarrow a_{i,j}^{(k)} + \frac{a_{i,k}^{(k)} a_{k,j}^{(k)}}{a_{k,k}^{(k)}}$$



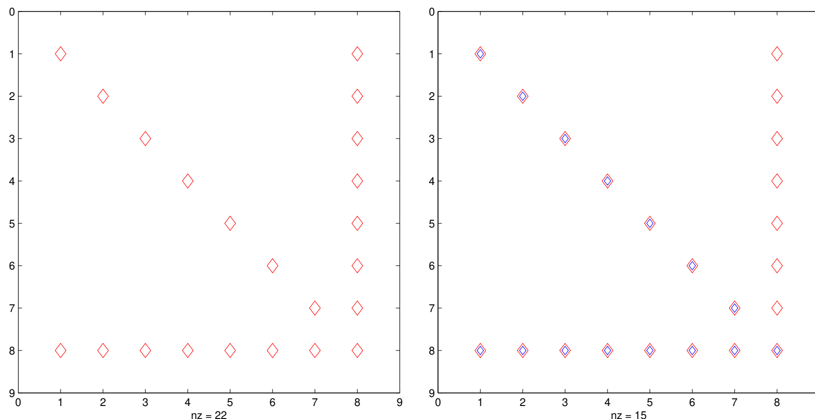
Effect on sparsity structure on factorization:



(c) Arrow matrix (d) The structure of the L-factor

The arrow matrix structure - the L and U factors are full.

Effect on sparsity structure on factorization



(e) Arrow matrix permuted (f) The structure of the L-factor

We can permute the matrix A first and then factorize!

We pose now the question to $\cdots$

find permutation matrices P and Q, such that when we factorize $\widetilde{A} = Q^T A P^T$, the fill-in in the so-obtained L and U factors will be minimal.

The solution algorithm takes the form:

(1) Factorize $Q^T A P^T = LU$
(2) Solve $PL\mathbf{z} = \mathbf{b}$ and $UQ\mathbf{x} = \mathbf{z}$.

How to construct P and Q in general?

The strive to achieve complexity $O(n) + O(nnz(A))$ entails very complicated sparse codes.
Some important aspects when implementing the direct solution techniques for sparse matrices in practice:

- sparse data structures and manipulations with those;
- computer platform related issues, such as handling of indirect addressing; lack of locality;
  difficulties with cache-based computers and parallel platforms;

  short inner-most loops.

Straightforward implementation of *matvec* in CRS

```
% SMULV:      x <-- A * y ; A is a sparse matrix
%                x(l1) <-- A(l1,l2) * y(l2)
% Sizes:         AR(l1+1), AC(lA), AV(lA)

      function x= SMULV(AR, AC, AV, y, lA, l1, l2)

      for i=1:l1
         x(i) = 0;
         iF = AR(i);
         iL = AR(i+1) - 1;
         if iL>= iF,
            for j=iF:iL
            X(i) = X(i) + AV(j) * y(AC(j))
   end
```

Extra difficulties come from the fact that···

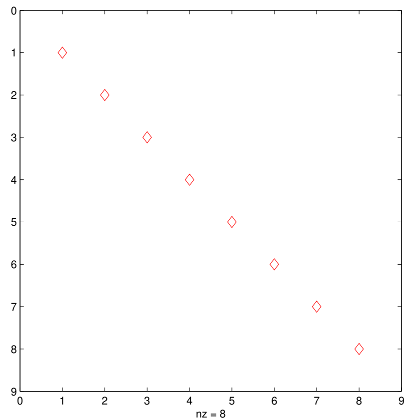
we have to choose a pivot element and its proper choice may contradict to the strive to minimize fill-in.

We are most often dealing with *'Given-the-matrix'* case

I.e., the only source of information is the matrix itself and we will try to reorder the entries so that the resulting structure will limit the possible fill-in.

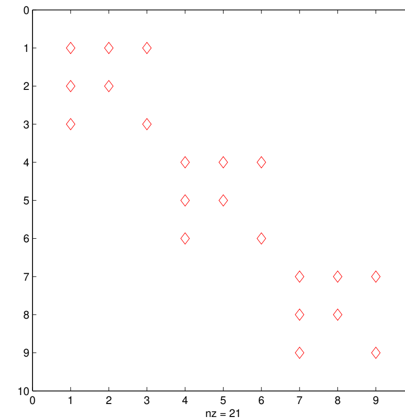
What is the matrix structure to aim at?

Given-the-matrix strategy

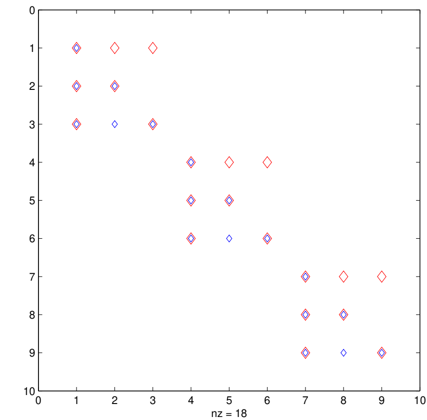


(g) Diagonal matrix

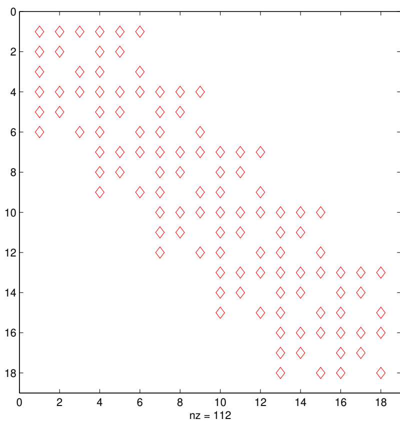
- diagonal
- block-diagonal
- block-tridiagonal
- arrow matrix
- band matrix
- block-triangular



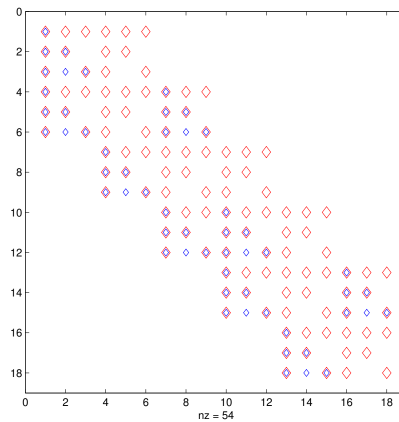
(h) block-diagonal matrix



(i) The structure of the L-factor



(j) Block-tridiagonal matrix

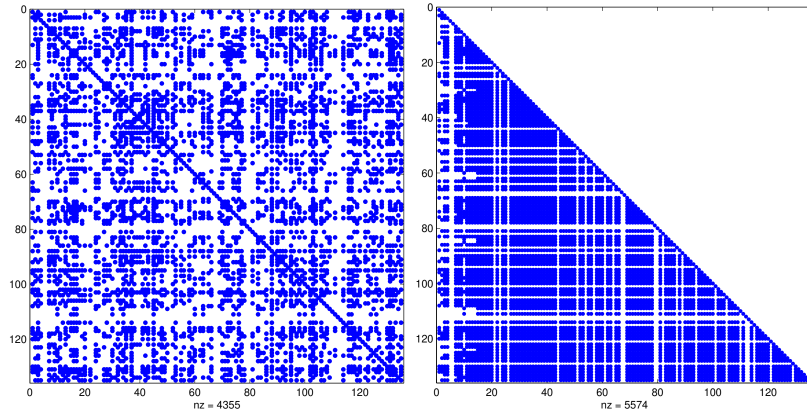


(k) The structure of the L-factor

Consider the case of symmetric matrices ($P = Q$) and three popular methods based on manipulations on the graph representation of the matrix.

- (generalized) reverse Cuthill-McKee algorithm (1969);
- nested dissection method (1973);
- minimum degree ordering (George and Liu, 1981) and variants.

A matrix from somewhere



Generalized Reverse Cuthill-McKee (RCM)

Aim: minimize the envelope (in other words a band of variable width) of the permuted matrix.

1. *Initialization*. Choose a starting (root) vertex r and set $v_1 = r$.
2. *Main loop*. For $i = 1, \ldots, n$ find all non-numbered neighbours of v_i and number them in the increasing order of their degrees.
3. *Reverse order*. The reverse Cuthill-McKee ordering is $w_1, \ldots, w_n$, where $w_i = v_{n+1-i}$.

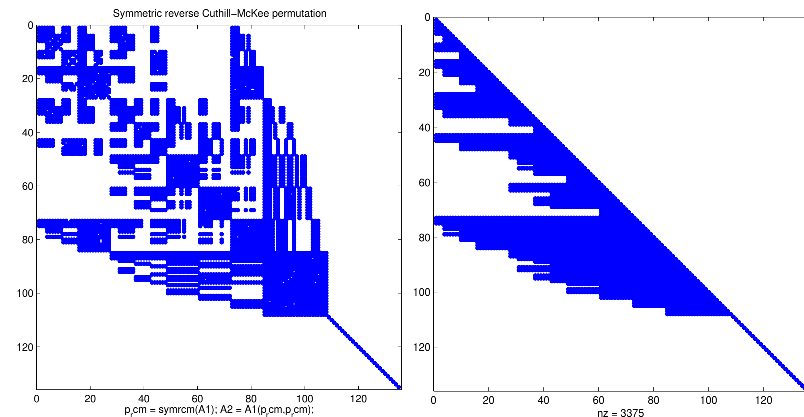
Generalized Reverse Cuthill-McKee (RCM)

One can see that GenRCM tends to number first the vertices adjoint to the already ordered ones, i.e., it gathers matrix entries along the main diagonal.

The choice of a root vertex is of a special interest.

The complexity of the algorithm is bounded from above by $O(m\ nnz(A))$, where m is a maximum degree of vertices, $nnz(A)$ - number of nonzero entries of matrix A.

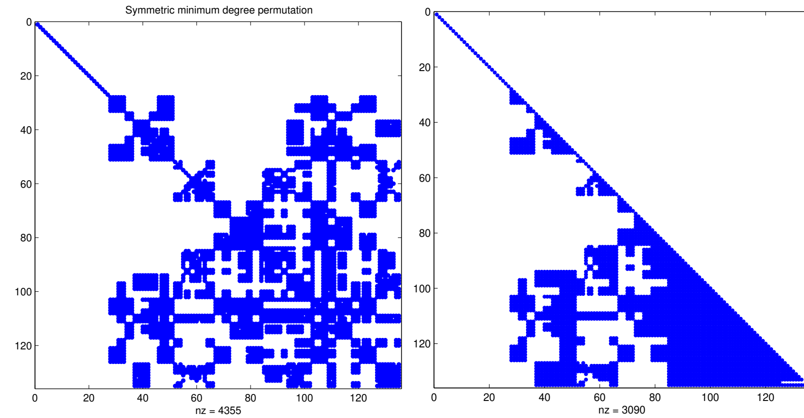
Generalized Reverse Cuthill-McKee (RCM)



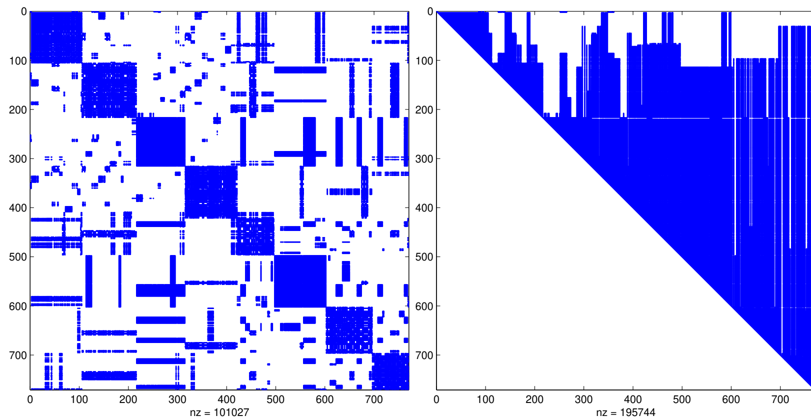
The Quotient Minimum Degree (QMD)

Aims to minimize a local fill-in taking a vertex of minimum degree at each elimination step. The straightforward implementation of the algorithm is time consuming since the degree of numerous vertices adjoint to the eliminated one must be recomputed at each step. Many important modifications have been made in order to improve the performance of the MD algorithm and this research remains still active .
In many references the MD algorithm is recommended as a general purpose fill-reducing reordering scheme. Its wide acceptance is largely due to its effectiveness in reducing fill and its efficient implementation.

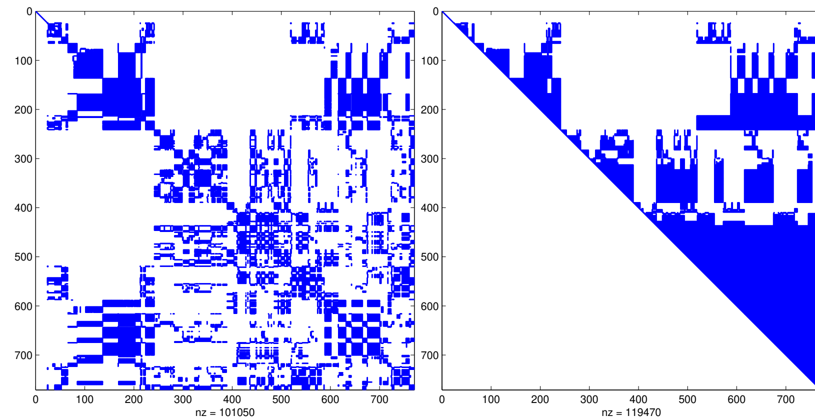
The Quotient Minimum Degree (QMD)



IBD (Identity By Descent) matrix



IBD matrix: MMD



The Nested Dissection algorithm

A recursive algorithm which on each step finds a separator of each connected graph component. A separator is a subset of vertices whose removal subdivides the graph into two or more components. Several strategies how to determine a separator in a graph are known. Numbering the vertices of the separator last results in the following structure of the permuted matrix with prescribed zero blocks in positions $(2,1)$ and $(1,2)$

$$\begin{pmatrix} A_{11} & 0 & A_{13} \\ 0 & A_{22} & A_{23} \\ A_{31} & A_{32} & A_{33} \end{pmatrix}.$$

The Nested Dissection algorithm

Under the assumption that subdivided components are of equal size the algorithm requires no more than $\boxed{\log_2 n}$ steps to terminate.

ND is optimal (up to a constant factor) for some class of model 2D problems originating from discretized PDEs. The Cholesky factor contains $O(m^2 log_2 m)$ nonzero entries. This is the _best low order bounds_ derived for direct elimination methods.

When we are dealing with *'Given-the-problem'* case

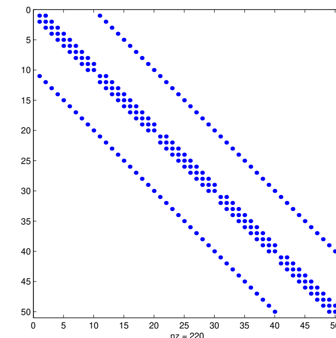
I.e., we know more - the mesh, the discretization method, the element matrices (the discretization stencils).

What can we do then?

In the PDE world and not only...

41	42	43	44	45	46	47	48	49	50
31	32	33	34	35	36	37	38	39	40
21	22	23	24	25	26	27	28	29	30
11	12	13	14	15	16	17	18	19	20
1	2	3	4	5	6	7	8	9	10

(l) Column-wise ordering

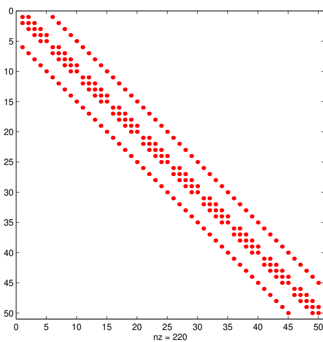


(m) The structure of the matrix A

In the PDE world and not only...

5	10	15	20	25	30	35	40	45	50
4	9	14	19	24	29	34	39	44	49
3	8	13	18	23	28	33	38	43	48
2	7	12	17	22	27	32	37	42	47
1	6	11	16	21	26	31	36	41	46

(n) Column-wise ordering

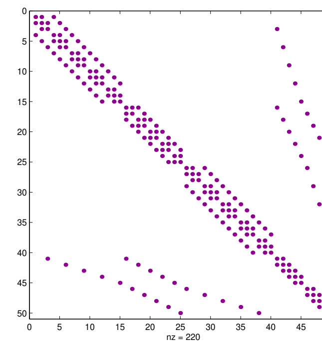


(o) The structure of the matrix A

In the PDE world and not only...

13	14	15	***45***	24	25	***50***	38	39	40
10	11	12	Ω1 ***44***	22	23	Ω2 ***49***	35	36	Ω3 37
7	8	9	***43***	20	21	***48***	32	33	34
4	5	6	***42***	18	19	***47***	39	30	31
1	2	3	***41***	16	17	***46***	26	27	28

(p) Column-wise ordering

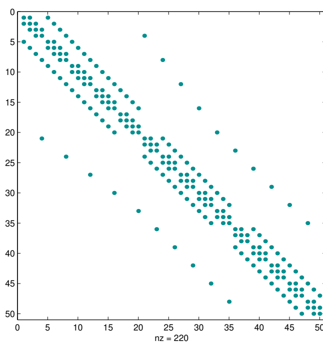


(q) The structure of the matrix A

In the PDE world and not only...

17	18	19	20	33	34	35	48	49	50
13	14	15 Ω1	16	30	31 Ω2	32	45	46 Ω3	47
9	10	11	12	27	28	29	42	43	44
5	6	7	8	24	25	26	39	40	41
1	2	3	4	21	22	23	36	37	38

(r) Column-wise ordering



(s) The structure of the matrix A

Summary:

- There is no one good buy.
- The best code in any situation will depend on
  - the solution environment;
  - the computing platform;
  - the structure of the matrix.

An appetizer to iterative methods

```
n=10000;
tic,x=A\b;toc
Elapsed time is 1.881219 seconds.
tic,x=AF\b;toc
Elapsed time is 12.504630 seconds.

n=50000;
R=sprand(n,n,1/n);I=speye(n);b=rand(n,1);A=10*I+0.5*(R+R');
tic,x=A\b;toc
Elapsed time is TOO MANY seconds.


tic,[x,flag,relres,iter,resvec]=pcg(A,b,1e-6,1000);toc
Elapsed time is 0.015673 seconds.
iter   = 5
relres = 4.67 e-07
```