Computational Methods in Statistics with Applications
Sparse matrices

Maya Neytcheva

SeSe, March 2016

Plan of the lecture:

- Sparse matrices - who are those?
- Where do sparse matrices occur in Statistical applications?
- Why are sparse matrices a topic of special interest?
- Handling sparse matrices. Sparse data formats
- 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?

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

Where do sparse matrices arise?

acoustic scattering	demography	network flow
air traffic control	economics	oceanography
astrophysics	electrical eng.	petroleum eng.
biochemical	electric nets	reactor modelling
chemical eng.	climate/pollution studies	statistics
chemical kinetics	fluid flow	structural eng
circuit physics	laser optics	survey data
computer simulations	linear programming	signal processing

Sparse matrices in Statistical applications

One application: Quantitative trait loci (QTL)
Inheritance of **quantitative traits** or polygenic inheritance refers to the inheritance of a phenotypic characteristic that varies in degree and can be attributed to the interactions between two or more genes and their environment.
Though not necessarily genes themselves, quantitative trait loci (QTLs) are stretches of DNA that are closely linked to the genes that underlie the trait in question. QTLs can be molecularly identified, for example, with PCR (Polimerase Chain reaction) or AFLP (Amplified Fragment Length Polymorphism) to help map regions of the genome that contain genes involved in specifying a quantitative trait.

This can be an early step in identifying and sequencing these genes.

Sparse matrices in Statistical appl.: QTL

Description: Given an IBD matrix A (Identity By Descent) - symmetric positive semidefinite.
Needed: to minimize some log-likelihood matrix L, which boils down to a **nonlinear** solution method with approximated Hessian (HI) and gradient (GL) of L as follows:

$$HI = \begin{pmatrix} y^T PAPAPy & y^T PAPPy \\ y^T PAPPy & y^T PPPy \end{pmatrix} \quad GL = -\begin{pmatrix} tr(AP) - y^T PAPy \\ tr(P) - y^T PPy \end{pmatrix}$$

where $V = \sigma_1 A + \sigma_2 I$ and
$P = V^{-1} - V^{-1}X(X^T V^{-1} X)^{-1} X^T V^{-1}$.
Task: find the blocks in GL and HI and execute the nonlinear solver for a sequence of IBD matrices A.

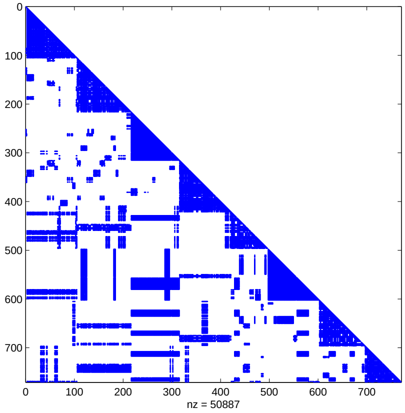
IBD ...

$$A - \mathrm{spsd} \implies A = A^T, eig(A) \geq 0$$

$$V = \sigma_1 A + \sigma_2 I \implies V = V^T, eig(V) = \sigma_1 eig(A) + \sigma_2, V^{-1} exists.$$

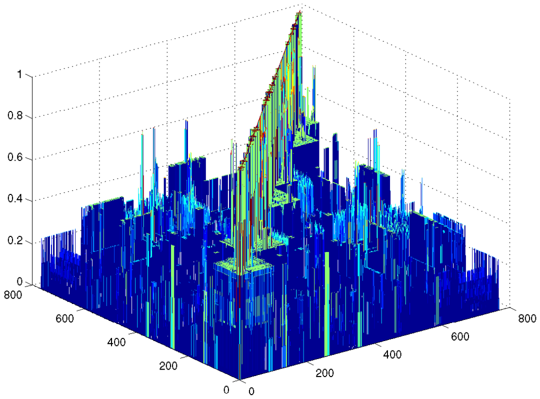
$$P = V^{-1} - V^{-1}X(X^T V^{-1} X)^{-1} X^T V^{-1} \implies P = P^T$$

Sparse matrices in Statistical appl.: QTL



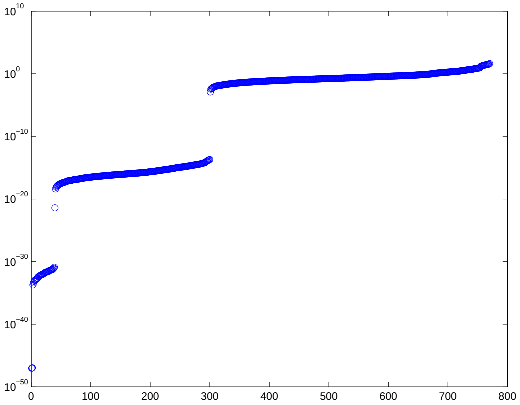
A portrait of one IBD matrix (spy(A))

Sparse matrices in Statistical appl.: QTL



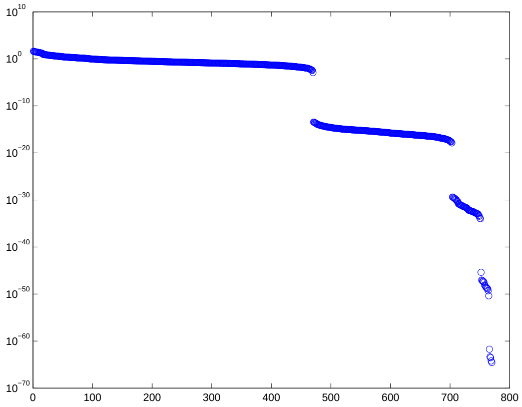
Another portrait of the same IBD matrix (`mesh(A)`)

Sparse matrices in Statistical appl.: QTL



The eigenvalues of the IBD matrix (`plot(eig(A),'o')`)

Sparse matrices in Statistical appl.: QTL



The singular values of the IBD matrix (`plot(svd(A),'o')`)

Major computational tasks in Statistical applications

- LS: solving the normal equation, Cholesky factorization
  $min||A\mathbf{x} - \mathbf{b}||$, $A^TA = LL^T$, $LL^T\mathbf{x} = A^T\mathbf{b}$
- LS: $A = QR$, $\mathbf{x} = R^{-1}Q_1^T\mathbf{b}$
- LS: $A = U\Sigma V^T$
- PCA (principal component analysis): SVD
- PCR (principal component regression): truncated SVD
- $A\mathbf{x} = \lambda\mathbf{x}$
- $trace(A^{-1})$

Major computational tasks in Statistical applications

Thanks to software (Matlab, $\mathbb{R}$) and powerful computers on our desk, we do not care that much about sparse-dense etc...
until we face large enough problems or we have to repeat a computational task 100, 1000, 10000 times.

The lecture today concerns storage book-keeping and programming aspects which will help to

- do the computations faster
- save computer memory

Before discussing sparse matrices...

we are going to look first at dense matrices...
because these are easier.

Dense matrix storage schemes

Given a dense matrix $A(m,n)$.
Two main possibilities to store dense matrices:
row-wise and *column-wise*.

$$A = \begin{bmatrix} a_{11} & a_{12} & \cdots & a_{1n} \\ a_{21} & a_{22} & \cdots & a_{2n} \\ \vdots & \vdots & \vdots & \vdots \\ a_{m1} & a_{m2} & \cdots & a_{mn} \end{bmatrix}$$

First row

$$DRW: \quad a_{11}\ a_{12}\ \cdots\ a_{1n} \,\big|\, a_{21}\ a_{22}\ \cdots\ a_{2n} \,\big|\, a_{m1}\ a_{m2}\ \cdots\ a_{mn} \,\big|$$

First column

$$DCW: \quad a_{11}\ a_{21}\ \cdots\ a_{m1} \,\big|\, a_{12}\ a_{22}\ \cdots\ a_{m2} \,\big|\, a_{1n}\ a_{2n}\ \cdots\ a_{mn} \,\big|$$

Dense matrix storage schemes and computations

What difference does the storage scheme make with respect to computations?

Dense matrix storage schemes and computations

What difference does the storage scheme make with respect to computations?

Matrix-vector multiplications: $\mathbf{y} = A\mathbf{x}, A(m, n)$

$$y_i = \sum_{j=1}^{n} a_{ij} x_j, \quad i = 1, \dots, m.$$

Dense matrix storage schemes and computations

What difference does the storage scheme make with respect to computations?

Matrix-vector multiplications: $\mathbf{y} = A\mathbf{x}, A(m, n)$

$$y_i = \sum_{j=1}^{n} a_{ij} x_j, \quad i = 1, \dots, m.$$

A: stored row-wise (inner product scheme)

```
for i=1:m
    y(i) = 0
    for j = 1:n
        y(i) = y(i) + A(i,j) * x(j)
    end
end
```

Dense matrix storage schemes and computations

What difference does the storage scheme make with respect to computations?

Matrix-vector multiplications: $\mathbf{y} = A\mathbf{x}, A(m, n)$

$$y_i = \sum_{j=1}^{n} a_{ij} x_j, \quad i = 1, \ldots, m.$$

```
  for i=1:m
     y(i) = 0
     y(i) = y(i) + A(i,:)  * x(:)
end
```

Dense matrix storage schemes and computations

Matrix-vector multiplications: $\mathbf{y} = A\mathbf{x}$

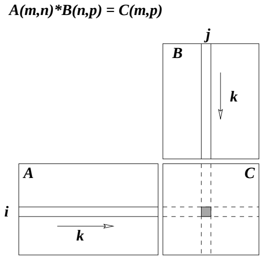
A: stored column-wise (outer product scheme)

```
y = 0
for j=1:n
    for i = 1:m
        y(i) = y(i) + A(i,j) * x(j)
    end
end
```

```
$\mathbf{y} = 0$
for $j = 1 : n$
    $\mathbf{y} = \mathbf{y} + x(j) * A(:, j)$
end
```

(vector operation)

Dense matrix-matrix multiply (ijk)

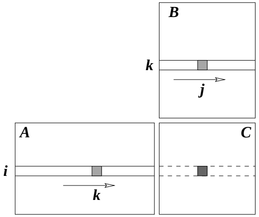


```
C = 0
for  i=1:m
   for  j=1:p
      for  k=1:n
         C(i,j) = C(i,j) + A(i,k)*B(k,j)
      end
   end
end
```

Scalar-product type of computation

Dense matrix-matrix multiply (ijk)

*A(m,n)*B(n,p) = C(m,p)*

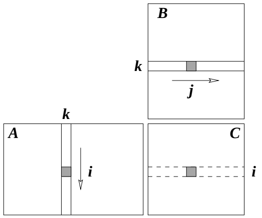


```
C = 0
for  i=1:m
   for  k=1:n
      for  j=1:p
         C(i,j) = C(i,j) + A(i,k)*B(k,j)
      end
   end
end
```

Outer-product type of computation ('ikj' - row-wise and 'jki' - column-wise)

Dense matrix-matrix multiply (ijk)

*A(m,n)*B(n,p) = C(m,p)*



```
C = 0
for  k=1:n
   for  i=1:m
      for  j=1:p
         C(i,j) = C(i,j) + A(i,k)*B(k,j)
      end
    end
end
```

accumulated update form

Dense matrix storage schemes and computations

Bottom line:

- The storage scheme of a dense matrix affects the order how the matrix entries are accessed in the computer memory. This may have a significant effect on the performance of an algorithm since the memory accesses are much slower than arithmetic operations
- One storage scheme is better for some operations and not so preferable for other operations (A^T).

Sparse matrix storage schemes

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

Sparse matrix storage schemes

Coordinate scheme:

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

$$\begin{array}{lcccccc} V: & 1 & 2 & 3 & 4 & 5 & 6 \\ I: & 1 & 1 & 2 & 2 & 3 & 4 \\ J: & 2 & 3 & 1 & 2 & 3 & 1 \end{array}$$

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

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

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

The process of triangular factorization (Gaussian elimination) for the case of *sparse* matrices.

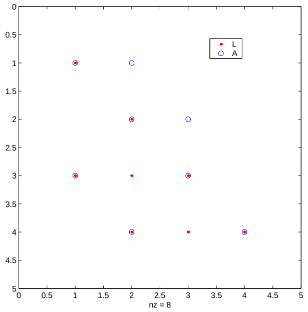
Note: In general, during factorization we have to do pivoting in order to assure numerical stability.

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 $(2q+1)$	$O(q^2n)$	$O(qn)$	$n(2q+1)$

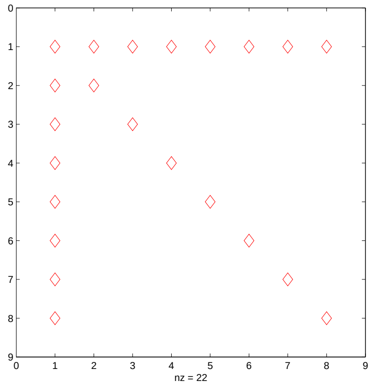
The reason to consider particularly factorizations of sparse matrices

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

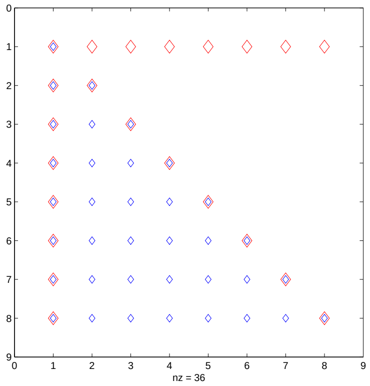


$$a_{i,j}^{(k+1)} \longleftarrow a_{i,j}^{(k)} + \frac{a_{i,k}^{(k)}\ a_{k,j}^{(k)}}{a_{k,k}^{(k)}}$$

Effect on sparsity structure on factorization:



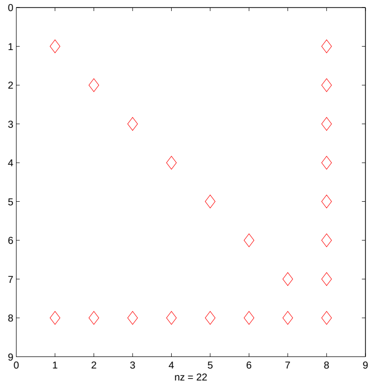
(c) Arrow matrix



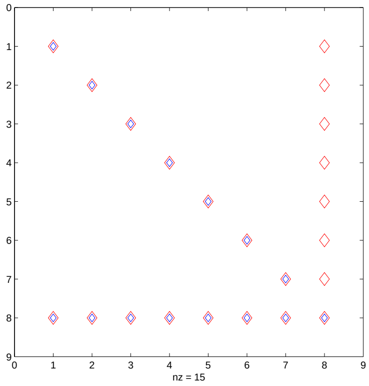
(d) The structure of the L-factor

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.

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.

```
n=500;
R=sprand(n,n,5/n); I=speye(n); b=rand(n,1); A=I+R; AF=full(A);
tic,x=A\b;toc
Elapsed time is 0.006472 seconds.
tic,x=AF\b;toc
Elapsed time is 0.036819 seconds.

n=5000;
tic,x=A\b;toc
Elapsed time is 0.336134 seconds.
tic,x=AF\b;toc
Elapsed time is 1.666255 seconds.

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.
```

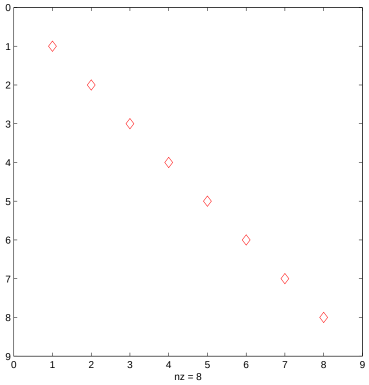
```
tic,[x,flag,relres,iter,resvec]=pcg(A,b,1e-6,1000);toc
Elapsed time is 0.015673 seconds.
iter   = 5
relres = 4.67 e-07
```

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.

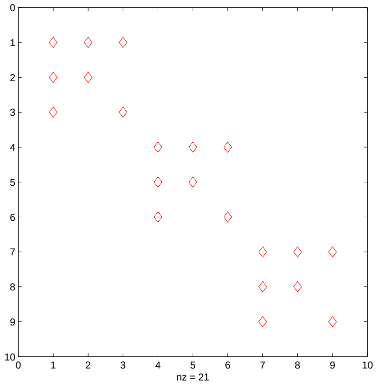
What is the matrix structure to aim at?

Given-the-matrix strategy

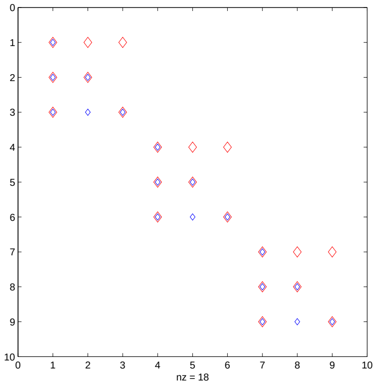


(g) Diagonal matrix

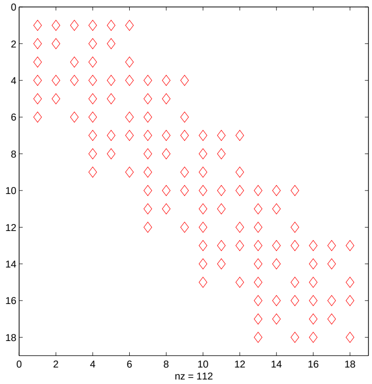
- diagonal
- block-diagonal
- block-tridiagonal
- arrow matrix
- band matrix
- block-triangular



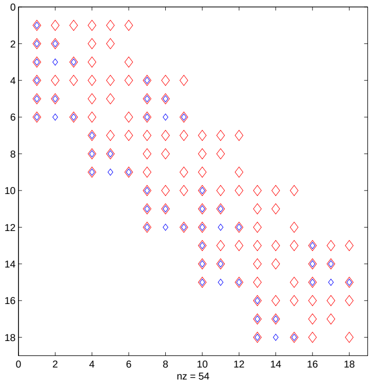
(h) block-diagonal matrix



(i) The structure of the L-factor



(j) Block-tridiagonal matrix

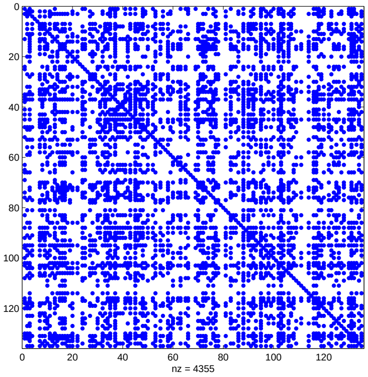


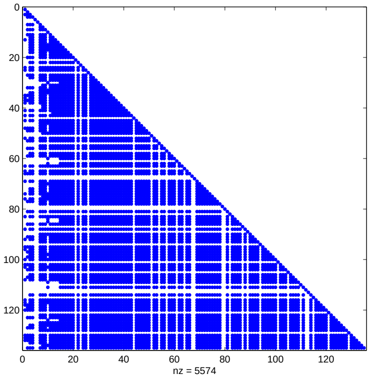
(k) The structure of the L-factor

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 $i = 1, ..., n$ 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 $w_1, ..., w_n$, where $w_i = v_{n+1-i}$.

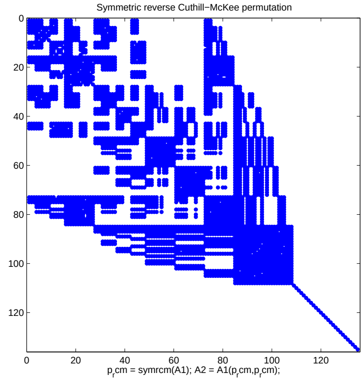
Generalized Reverse Cuthill-McKee (RCM)

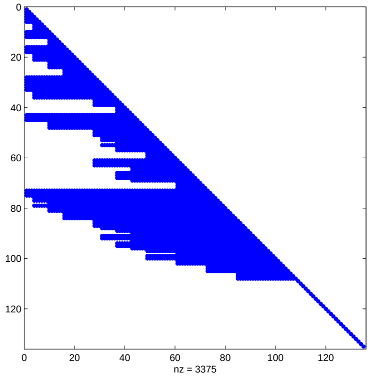
One can see that GenRCM tends to number first the vertices adjoint to the already ordered ones, i.e., it gathers matrix entries along the main diagonal.

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)



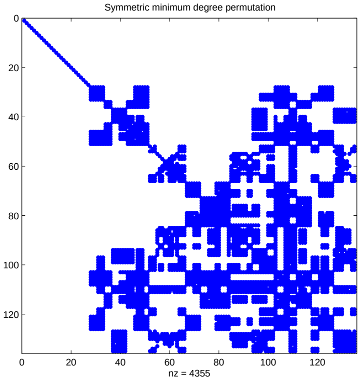


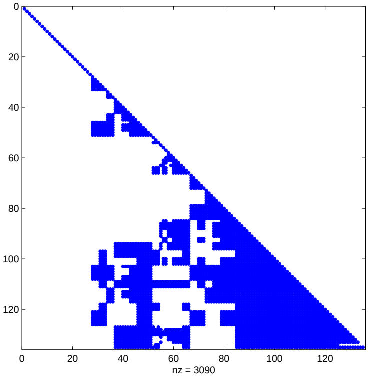
The Quotient Minimum Degree (QMD)

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 .

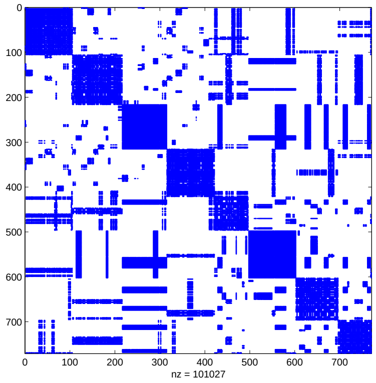
In many references the MD algorithm is recommended as a general purpose fill-reducing reordering scheme. Its wide acceptance is largely due to its effectiveness in reducing fill and its efficient implementation.

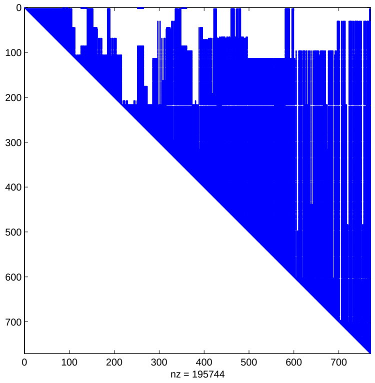
The Quotient Minimum Degree (QMD)



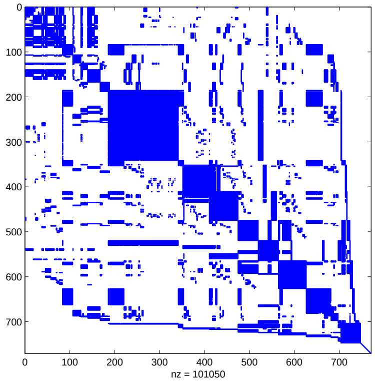


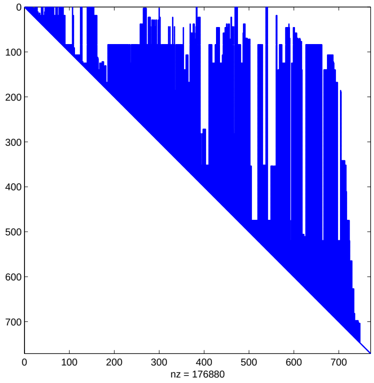
IBD matrix: X20



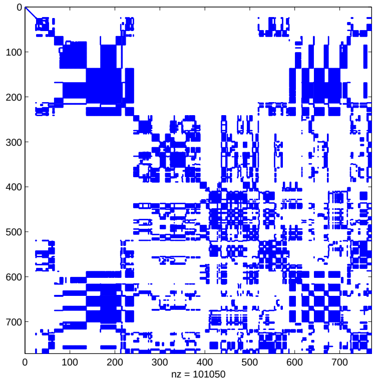


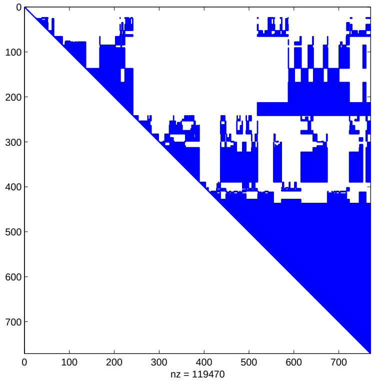
IBD matrix: X20: RCM





IBD matrix: X20: 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 $(1,2)$

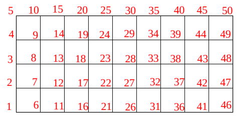
$$\begin{pmatrix} A_{11} & 0 & A_{13} \\ 0 & A_{22} & A_{23} \\ A_{31} & A_{32} & A_{33} \end{pmatrix}.$$

The Nested Dissection algorithm

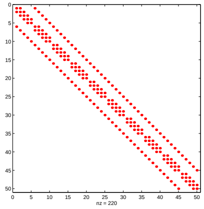
Under the assumption that subdivided components are of equal size the algorithm requires no more than $\boxed{\log_2 n}$ steps to terminate.

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.

In the PDE world and not only...



(l) Column-wise ordering



(m) The structure of the matrix A

Major computational tasks in Statistical applications

- Cholesky factorization $\surd$
  $min||A\mathbf{x} - \mathbf{b}||$, $A^TA = LL^T$, $LL^T\mathbf{x} = A^T\mathbf{b}$
- LS: $A = QR$, $\mathbf{x} = R^{-1}Q_1^T\mathbf{b}$
- LS: $A = U\Sigma V^T$
- PCA (principal component analysis): SVD
- $A\mathbf{x} = \lambda\mathbf{x}$
- $trace(A^{-1})$

Sparse QR

$$A = LL^T$$
$$A = QR, \text{ then } A^TA = R^TQ^TQR = R^TR!$$

Thus, if we know R, we have factorized A^TA!
But: if A is sparse, we want that R is as sparse as possible.
This is achieved by performing symbolic factorization of A^TA.

Sparse SVD, `Matlab`

The Matlab SVD implementation follows that of `LINPACK`, which is for general dense matrices.
To find some of the singular values (largest or smallest) of a large sparse matrix, one can use `svds`.
`svds(A,k)` uses `eigs` to find the k largest magnitude eigenvalues and corresponding eigenvectors of

$$B = \begin{bmatrix} 0 & A \\ A^T & 0 \end{bmatrix}$$

.

Sparse SVD, `Matlab`

Demo:

```
load 20.dat                        S=svds(AS,5);
A=X20+tril(X20,1)';                  S=svds(AS,5,0.51);
S =svd(A);          0.27 s             S=svds(AS,5,0.01);
AS=sparse(A);
SS=svds(AS,10);     0.09 s
```

Sparse matrices in Statistical applications

- Genetic mapping of quantitative traits
- Sparse inverse of covariance matrix for QTL effects with incomplete marker data
  R. Mark Thallman, Kathryn Hanford, Stephen Kachman, L. Dale Van Vleck
  Statistical Applications in Genetics and Molecular Biology, 2005
- Sparse spatial autoregression
- Pace and Barry (1997), "Kriging with Large Data Sets," Communications in Statistics, Simulation and Compassion
  The authors discuss sparse krigging in Communication in Statistics, Simulation and Computation. Using published estimates on a spherical variogram they solve the estimates 432 times as fast as using more conventional solution techniques.

Sparse matrices in Statistical applications

- Barry and Pace (1999), "Monte Carlo Estimates of the Log Determinant of Large Sparse Matrices," Linear Algebra and its Applications
  The authors devise a means of estimating the log-determinant of large, sparse matrices. Estimation of the log-determinant of the variance-covariance matrix (or its inverse) allows maximum likelihood estimation of large-scale spatial statistical problems. Most importantly, the article shows a way of providing confidence intervals for the estimate and show these work via a coverage study. To illustrate the potential of the estimator, we estimated the log-determinant of a 1,000,000 by 1,000,000 matrix (on a Pentium 133 MHz machine). The estimator has a simple form and its performance depends only upon the degree of sparsity and not its pattern. Source code and executable code for it resides in SpaceStatPack.

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.

Some practical issues

$\mathbb{R}$ packages for working with sparse matrices:

- spam
- SparseM
- Matrix

ℝ - Matlab: sparse matrix exchange

```
library(MASS)
> M=matrix(1:12,3,4)
> M
     [,1] [,2] [,3] [,4]
[1,]    1    4    7   10
[2,]    2    5    8   11
[3,]    3    6    9   12
write.matrix(M,file="M.dat",sep=" ")
```

```
>> load M.dat
>> M
     1     4     7    10
     2     5     8    11
     3     6     9    12
```

Matlab-$\mathbb{R}$: sparse matrix exchange

```
>> N=M/2
N =
    0.5000    2.0000    3.5000    5.0000
    1.0000    2.5000    4.0000    5.5000
    1.5000    3.0000    4.5000    6.0000
>> save N.dat -ASCII N
```

```
> N0=read.csv("N.dat",sep="",header=F)
> is.matrix(N0)
[1] FALSE
> N=as.matrix(N0)
> N
      V1  V2  V3  V4
[1,] 0.5 2.0 3.5 5.0
[2,] 1.0 2.5 4.0 5.5
[3,] 1.5 3.0 4.5 6.0
```

Matlab-$\mathbb{R}$: sparse matrix exchange

Source file: `20.dat`
Read in Matlab:

```
>> load 20.dat
>> size(X20)
ans =    770   770

>> X20sp=sparse(X20);
>> whos
  Name          Size                     Bytes  Class     Attributes
  X20         770x770                  4743200  double
  X20sp       770x770                   613728  double    sparse
```

Matlab-ℝ : sparse matrix exchange

Source file: `20.dat`
Read in ℝ :

```
> library("MASS")
> X20=read.delim("20.dat")
> dim(X20)
[1] 769   1
> AL=read.csv("20.dat",sep="",header=F)
> dim(AL)
[1] 770 770
> is.matrix(AL)
[1] FALSE
> AL=as.matrix(AL)
> AU=t(AL)
> A=AL+AU
> S=svd(A)
> plot(S$d)
```

Matlab-ℝ : sparse matrix exchange

Export the triangular factor from ℝ to Matlab:

```
> is.numeric(U)
[1] TRUE
> write.table(U,file = "Fname", quote = FALSE, sep = "")
```

Experience with ℝ - spam

```
library(spam)
> N0=read.csv("A20.dat",sep="",header=F)
> is.matrix(N0)
[1] FALSE
> A=as.matrix(N0)
> dim(A)
[1] 770 770
> Asp=as.spam(A)
> is.spam(Asp)
[1] TRUE
> D=diag.spam(x=10, 770, 770)
> B=Asp+D
> system.time(U=chol(B))
> b <- rnorm(770)
> q=solve(B,b)
> Q=chol(B)
> spam.options( imagesize=1000)
> display(B)
```

Data format of the 'spam' package

The 'old Yale sparse format":
a (sparse) matrix is stored with four elements (vectors), which are:
– (1) the nonzero values row by row, – (2) the ordered column indices of nonzero values, – (3) the position in the previous two vectors corresponding to new rows, given as pointers, – (4) the column dimension of the matrix.

ℝ : SparseM

```
library(SparseM)
A <- rnorm(10*10)
A[abs(A) < 0.7] <- 0
B <- matrix(A,10,10)
B
B.csr <- as.matrix.csr(B)
B.csr
```

ℝ : SparseM

```
as.matrix.csr(B)
An object of class "matrix.csr"
Slot "ra":
 [1] -2.4102317  2.5728546  0.9818316 -0.9926660  1.3441660 -1.3187796
 [7]  0.7838835  0.9944858  1.7196010  0.8811503  2.0851579  1.6376506
[13]  0.8536892  1.3447788 -2.5888581 -1.6815039  1.1941216 -1.8795499
[19] -1.7724763 -2.2327467  0.9569822 -0.8778493  0.8796285 -1.0345438
[25] -0.7602273  2.5978370 -1.7801954 -0.8674773 -0.7682890 -0.9491965
[31]  0.8423863 -0.8469827 -0.7912244 -1.8190693 -1.0183608 -0.9646922
[37]  1.3862937 -2.3116806  1.1205653 -2.0512307 -1.4883147  1.3637320
[43] -1.5820963 -0.9229476  0.8861217  1.4164354  0.9828918  1.1553508
[49] -1.0872901 -0.9350768 -0.7850330

Slot "ja":
 [1]  1  4  5  7  8  9  3  4  5  7  8  9  1  3  5  6  7  8  3  4  6 10  2  3  7
[26] 10  3  5  7  9 10  4  5  6  7  8  1  3  4  6  7  9 10  5  6  8  9  4  5  7
[51] 10

Slot "ia":
 [1]  1  7 13 19 23 27 32 37 44 48 52

Slot "dimension":
[1] 10 10
```

ℝ : Sparse SVD, needs

http://stats.stackexchange.com/questions/41259/svd-on-a-65-million-by-3-4-million-sparse-mat

Q: What is the best way compute SVD on a very large positive matrix (65M x 3.4M) where data is extremely sparse? sparse: less than 0.1% of the matrix is non zero.
Should be computed in a reasonable time: 3,4 days
It would be great to have a Haskell, Python, C# etc. library which implements it. I am not using mathlab or R but if necessary I can go with R.

A: Try 'Matrix' and 'irlba'
Package 'irlba', July 2, 2014; Jim Baglama and Lothar Reichel.
'Fast partial SVD by implicitly-restarted Lanczos bidiagonalization'

Augmented Implicitly Restarted Lanczos Bidiagonalization Methods, J. Baglama and L. Reichel, *SIAM J. Sci. Comput.* 2005

Some references:

P. R. Amestoy, T. A. Davis and I. S. Duff. An approximate minimum degree ordering algorithm. *SIAM J. Matr. Anal. Appl.*, 17, 886-905, 1996.
C. Ashcraft and J.W.H. Liu. Robust ordering of sparse matrices using multisection. *SIAM J. Matrix Anal. Appl.*, 19, 816-832, 1998.
E. Cuthill, J. McKee. Reducing the bandwidth of sparse symmetric matrices. *Proc. 24th Nat. Conf. Assoc. Comput. Mach.*, 157-172, 1969.
J.W.H. Liu, A. H. Sherman. Comparative analysis of the Cuthill-McKee and the reverse Cuthill-Mckee ordering algorithms for sparse matrices. *SIAM J. Numer. Anal.*, 13, 198-213, 1975.
J. Dongarra, I. Duff, Sorensen and H. van der Vorst, *Numerical Linear Algebra for High Performance Computers*, SIAM Press.
I. Duff, Direct methods, Technical report TR/PA/98/28, July 29, 1998, CERFACS.
H.W. Berry and A. Sameh (1988), Multiprocessor schemes for solving block tridiagonal linear systems, *The International Journal of Supercomputer Applications*, 12, 37-57.

Some references:

I. S. Duff, A. M. Ersiman and J. K. Reid, *Direct Methods for Sparse Matrices*, Oxford University Press, 1986. Reprinted 1989.
I. S. Duff, R. G. Grimes and J. G. Lewis. Sparse matrix test problems. *ACM Trans. Math. Software*, 15, 1-14, 1989.
J.A. George and J.W.H. Liu. *Computer solution of large sparse positive definite systems*. Prentice-Hall, Englewood Cliffs, New Jersey, 1981.
K.A. Gallivan, R.J. Plemmons, and A.H. Sameh (1990), Parallel algorithms for dense linear algebra computations, *SIAM Review*, 32, 54-135.
J. George and J.W.H. Liu. The evolution of the minimum degree ordering algorithm. *SIAM Rev.*, 31, 1-19, 1989.
F.-C. Lin and K.-L. Chung (1990), A cost-optimal parallel tridiagonal system solver, *Parallel Computing*, 15, 189-199.
E. Rothberg and S.C. Eisenstat. Node selection strategies for bottom-up sparse matrix ordering. *SIAM J. Matr. Anal. Appl.*, 19, 682-695, 1998.
H. van der Vorst and K. Dekker, Vectorization of linear recurrence relations, *SIAM Sci. Stat. Comp.*, 10 (1989), 27–35.