Sparse Matrices in Numerical Mathematics

Miroslav Tůma

Faculty of Mathematics and Physics

Charles University

`mirektuma@karlin.mff.cuni.cz`

Praha, December 15, 2025

Outline

1. Introduction
2. Introductory notation and terminology
3. Factorizations
4. Graphs and sparse matrices
5. Sparse matrices and data structures
6. Symbolic factorizations
7. Symbolic Cholesky factorization
8. Cholesky factorization
9. Sparse LU factorization
10. Stability, ill-conditioning, indefiniteness
11. Blocks in the input matrix
12. Sparse Least Squares and QR factorization
13. Reorderings
14. Algebraic preconditioning
15. Incomplete factorizations
16. Sparse approximate inverses

Introductory notes

Origin

- Originally created to support online lectures of NMNV533.

Preliminaries

- Basic notions of (numerical) linear algebra and programming (software construction): matrices, vectors
- Some knowledge of Cholesky and LU decompositions assumed.
- Basic understanding of algebraic iterative (Krylov space) and direct (dense) solvers (elimination/factorization/solve) is useful. (A lot of these is repeated)

Introductory notes

Limitations

- Prevailably only purely algebraic techniques considered. Such techniques often serve as building blocks for some more complex approaches.
- Some important techniques are only mentioned. Those include
  - Multigrid/multilevel preconditioners,
  - Domain decomposition,
  - Row projection techniques.
- Only preconditioning of real systems considered here, although extension to complex field is typically straightforward.

Introductory notes: resources / history

- The main text resource is:
  Jennifer Scott and Miroslav Tůma: Algorithms for sparse linear systems, Birkhäuser- Springer, 2023, open access.
- Traditional material was the course text in Czech (nowadays outdated, not supported); see the web page of the course, or ask me.

Introductory notes: resources and history of the course

- A few other resources:

  Davis, T. A. (2006). Direct Methods for Sparse Linear Systems. Fundamentals of Algorithms. SIAM, Philadelphia, PA.

  Davis, T. A., Rajamanickam, S., & Sid-Lakhdar, W.M. (2016). A survey of direct methods for sparse linear systems. Acta Numer., 25, 383-566.

  Duff, I. S., Erisman, A.M., & Reid, J. K. (2017). Direct Methods for Sparse Matrices (Second ed.). Oxford University Press, Oxford.

  George, A. & Liu, J. W. H. (1981). Computer Solution of Large Sparse Positive Definite Systems. Prentice Hall, Englewood Cliffs, NJ.

  Saad, Y. (2003b). Iterative Methods for Sparse Linear Systems (Second ed.). SIAM, Philadelphia, PA.

Motivation

- Most of our activities around solving the systems of linear algebraic equations

$$\mathbf{Ax} = \mathbf{b}$$

- There are two principially different classes of the solution methods:
  - Direct methods
  - Iterative methods

Motivation

Direct methods

- Direct methods: Transform A using a finite sequence of elementary transformations
- First step is a decomposition (factorization)
- Standard factorizations:
    - Cholesky factorization $A \to LL^T$ if A is symmetric and positive definite
    - LU factorization if $A \to LU$ is factorizable
- Second step uses the factorization to solve the system (substitutions, formally applying factor inverses)
- An example: $Ax = b$ to be solved, $A = LU$, $y = L^{-1}b$, $x = U^{-1}y$
- Similarly with other factorizations

Motivation

Direct methods: further comments

- Factorizations are the backbone of direct methods.
- Most of the work is in the factorization. And the approaches to get them are more complex than the substitution steps.
- Solving systems with triangular matrices like L, U is generally much cheaper and more straightforward that using A.
- More reasons for different factorizations or their variations:
  - Different matrix types
  - Hardware (computer model) and software properties/capabilities
  - For example: sequential versus concurrent processors, multicore, GPU etc. decide about relative complexity of the two steps.
  - But the latter may influence substitution steps strongly as well.
- Factorizations: in principal $=$ Gaussian elimination. Modern (decompositional) form based a lot on the work of Householder (end of 1950's): from pointwise to matrix/vectorwise descriptions

Motivation

Direct methods

- Designed to be robust, designed to solve the systems of equations.
- If properly implemented, can be used as block-box solvers for computing solutions with (often) predictable accuracy.
- But, they can be expensive, requiring large amounts of memory, which increases with the size of A.
- A lot of effort should be used to make them compatible with the steadily developing hardware/software.

Motivation

Iterative methods

- Compute a sequence of approximations

$$x^{(0)}, x^{(1)}, x^{(2)}, \dots$$

  that (hopefully) converge to the solution x of the linear system.

- Various approaches:
  - Stationary iterative methods like

$$x^{(k+1)} = (D - L)^{-1}(b + Ux^{(k)}$$

  - Here we have $A = D - L - U$ for D diagonal, L strictly lower triangular and U strictly upper triangular (no factorization)
  - Typically more efficient Krylov subspace methods based on projections, search space and constraints space ($x^{(k+1)} \in \mathcal{S}_{k+1}$, $r^{(k+1)} = b - Ax^{(k+1)} \perp \mathcal{C}_{k+1}$)
  - (Some) convergence theory for both classes of methods

Motivation

Iterative methods

- Designed to approximate (not solve)
- This may be an advantage (if only an approximate solution is needed: can be terminated as soon as the required accuracy is achieved) or a disadvantage (matrix properties may prohibit achieving the required accuracy)
- Another disadvantage: iteration counts may strongly depend on the initial guess $x^{(0)}$, A and b
- Potential advantage: A can be used only indirectly, through matrix-vector products → memory requirements are limited to a (small) number of vectors of length the size of A
- That is, A does not need to be available explicitly.

Motivation

Direct versus iterative methods

- To have direct methods faster, more memory efficient, using specific computer architecture, the solution can be less accurate due to possible relaxations.
- Making solution accurate: use an auxiliary iterative method.
- Simple auxiliary iterative method (iterative refinement (IR)):

Algorithm (IR of the solution x of $Ax = b$)

```
Solve $Ax^{(0)} = b$                         ▷ $x^{(0)}$ is the initial computed solution
for $k = 0, 1, \ldots$ do
    Compute $r^{(k)} = b - Ax^{(k)}$          ▷ Residual on iteration $k$
    Solve $A\,\delta x^{(k)} = r^{(k)}$       ▷ Solve correction equation
    $x^{(k+1)} = x^{(k)} + \delta x^{(k)}$
end for
```

Motivation

Direct versus iterative methods

- From the iterative side: pure iterative methods may not converge fast. Or may have low final attainable accuracy.
- Therefore, they should be accompanied by a problem transformation based on a direct method called preconditioner. Often representing a relaxed factorization.
- Such transformation can be expressed in matrix form, for example as :

$$MAx = Mb$$

- M approximating A^{-1} can be then applied to vector quantities of the iterative method.

Motivation

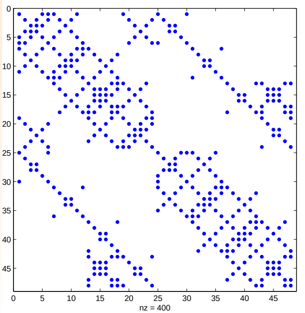
Direct versus iterative methods

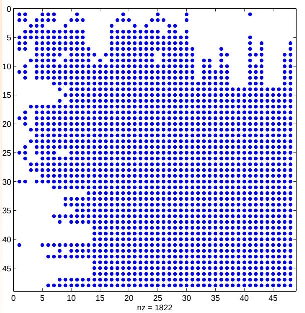
- In practice, many variations, M can represent inverses of approximate factors, or directly an approximation to A^{-1} (approximate inverse preconditioners). Many variations.
- Consequently, practical boundaries between direct and iterative methods are more and more fuzzy.

Motivation

What else?

- For example: matrices and resulting factorizations may look like as follows (showing only nonzeros):





Motivation

What else?

- For example: matrices and resulting factorizations may look like as follows (showing only nonzeros):

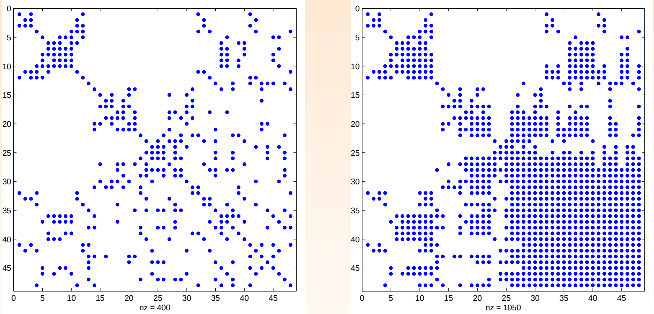


Figure: *The locations of the nonzero entries in a symmetrically permuted A from above (left) and in $\bar{L} + \bar{L}^T$ (right), where $\bar{L}$ is the Cholesky factor of the permuted matrix.*

Motivation

What else?

- For example: matrices and resulting factorizations may look like as follows (showing only nonzeros):

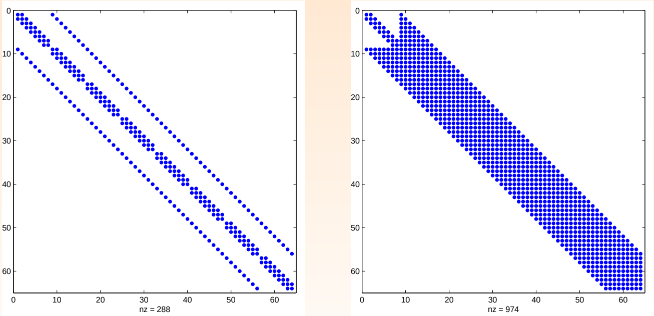


Figure: *The locations of the nonzero entries in another symmetrically permuted A (left) and in $\bar{L} + \bar{L}^T$ (right), where $\bar{L}$ is the Cholesky factor of the permuted matrix.*

Motivation

Where is problem with direct methods?

- Structure of zeros and nonzeros in A called sparsity needs to be exploited
- Sparse factorizations $A = LL^T$, LU (exact up to the floating-point model) needed

Where is problem with iterative methods?

- They should be transformed (preconditioned)
- By using as M (approximate/incomplete) factorizations $A = LL^T$, LU needed like
    - incomplete decompositions ($A \approx LL^T$, LU etc.)
    - incomplete inverse decompositions ($A^{-1} \approx ZZ^T$, WZ^T etc.)
- Or specific (PDE-based, model-based) approaches may be used.

Outline

1. Introduction
2. Introductory notation and terminology
3. Factorizations
4. Graphs and sparse matrices
5. Sparse matrices and data structures
6. Symbolic factorizations
7. Symbolic Cholesky factorization
8. Cholesky factorization
9. Sparse LU factorization
10. Stability, ill-conditioning, indefiniteness
11. Blocks in the input matrix
12. Sparse Least Squares and QR factorization
13. Reorderings
14. Algebraic preconditioning
15. Incomplete factorizations
16. Sparse approximate inverses

Basic Terminology

Matrices, vectors

- Square matrix $A \in \mathbb{R}^{n \times n}$
- Sometimes rectangular $A \in \mathbb{R}^{m \times n}$
- Vectors denoted by small letters as $b \in \mathbb{R}^n$, $x \in \mathbb{R}^n$ etc. (sparse or dense)
- Dimension notations n, m used throughout. We also assume

$$A = (a_{ij}), \;\; 1 \leq i, j \leq n.$$

- Often using Matlab-like notation: nonzero (set a priori), $A_{:,j}$, $A_{i,:}$, $A_{i:j,k:l}$, A_{*j}, A_{i*}.

Basic Terminology

Matrices - special shapes

- A is diagonal if for all $i \neq j$, $a_{ij} = 0$;
- A is lower triangular if for all $i < j$, $a_{ij} = 0$
- A is upper triangular if for all $i > j$, $a_{ij} = 0$.
- A is unit (lower, upper) triangular if it is triangular and all the entries on the diagonal are equal to one.
- Using also strictly (lower, upper) triangular shapes that have zero diagonal entries.
- A is structurally symmetric if for all i and j for which a_{ij} is nonzero the entry a_{ji} is also nonzero.
- A is symmetric if

$$a_{ij} = a_{ji}, \ \text{ for all } \ i, j.$$

Basic Terminology

Matrices - special shapes

- Otherwise, A is nonsymmetric.
- The symmetry index $s(A)$ of A measures level of matrix symmetry: the number of nonzeros a_{ij}, $i \neq j$, for which a_{ji} is also nonzero divided by the total number of off-diagonal nonzeros. Small values of $s(A)$: A is far from symmetric.
- This motivates our terminology sometimes used later: weakly or strongly nonsymmetric matrices.
- Such distinctions may imply different algorithms for solving our problems.

Basic Terminology

Our problem

- Solving systems of linear algebraic equations

$$Ax = b, \tag{1}$$

- $A \in \mathbb{R}^{n \times n}, 1 \le i \le n,$ is nonsingular
- $b \in \mathbb{R}^n$, $x \in \mathbb{R}^n$
- In most of the text we are interested in direct methods. Within them, factorization of A is the crucial operation

Basic Terminology: special matrix classes

Matrix classes (by numerical properties)

- A is symmetric positive definite (SPD) if it is symmetric and satisfies
$$v^T A v > 0 \text{ for all nonzero } v \in \mathbb{R}^n.$$
- Sometimes mentioned redT symmetric positive semidefinite (SPSD) matrices (not regular)
- Other symmetric matrices mentioned here are symmetric indefinite (if regular).
- Remind: we deal the real case only.

Basic Terminology: saddle-point matrices

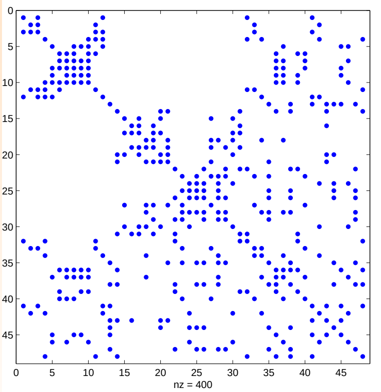
Matrix classes (by numerical properties)

- Symmetric and (typically) indefinite saddle point matrices have the form
$$A = \begin{pmatrix} G & R^T \\ R & B \end{pmatrix},$$
where $G \in \mathbb{R}^{n_1 \times n_1}$, $B \in \mathbb{R}^{n_2 \times n_2}$, $R \in \mathbb{R}^{n_2 \times n_1}$ with $n_1 + n_2 = n$, G is a SPD matrix and B is a symmetric positive semidefinite matrix (that is $v^T B v \geq 0$ for all nonzero $v \in \mathbb{R}^{n_2}$). In some applications, $B = 0$.
- We do not focus specifically on this matrix class. It can be very important to distinguish such matrices in applications and process them specifically.

Basic Terminology: sparsity

Sparsity

- A is a sparse matrix if many of its entries are zero.



Basic Terminology: sparsity

Sparsity: more formally

- Attempts attempts to formalize matrix sparsity more precisely like:

Definition

Matrix $A \in \mathbb{R}^{m \times n}$ is said to be sparse if it has $O(\min\{m, n\})$ entries. Another possibility: if A has row counts bounded by $r_{max} << n$ and/or column counts bounded by $c_{max} << m$.

Definition

Matrix $A \in \mathbb{R}^{m \times n}$ is said to be sparse if its number of nonzero entries is $O(n^{1+\gamma})$ for some $\gamma < 1$.

Definition

(pragmatic, application-based definition: J.H. Wilkinson) Matrix $A \in \mathbb{R}^{m \times n}$ is said to be sparse if the fact that a part of its entries is equal to zero can be (algorithmically) exploited.

Basic Terminology: sparsity

Sparsity: patterns

- The sparsity pattern $\mathcal{S}\{A\}$ of A is the set of nonzeros, that is,

$$\mathcal{S}\{A\} = \{(i,j)\,|\,a_{ij} \neq 0,\, 1 \leq i,j \leq n\}.$$

- Related: sparse vectors The sparsity pattern of $v \in \mathbb{R}^n$ is given by

$$\mathcal{S}\{v\} = \{i\,|\,v_i \neq 0\},$$

  and $|\mathcal{S}\{v\}|$ is the length of v.
- If $\mathcal{S}\{A\}$ is symmetric then A is structurally symmetric.
- The number of nonzeros in A: denoted here by $nz(A)$ (or $|\mathcal{S}\{A\}|$).
- A is structurally (or symbolically) singular if there are no values of the $nz(A)$ entries of A whose row and column indices belong to $\mathcal{S}\{A\}$ for which A is nonsingular.

Basic Terminology: sparsity

Rough comparison of dense and sparse (dimension, storage, time for decomposition using simple algorithms)

Dense matrix

dim	space	dec time (s)
3000	4.5M	5.72
4000	8M	14.1
5000	12.5M	27.5
6000	18M	47.8

Sparse matrix

dim	space	dec time (s)
10000	40k	0.02
90000	0.36M	0.5
1M	4M	16.6
2M	8M	49.8

Clearly, considering matrix as sparse promises huge advantages.

Basic Terminology

Factorizability

- Matrix A is factorizable (or strongly regular) if its principal leading minors (the determinants of its principal leading submatrices) are nonzero: its LU factorization without row/column interchanges does not break down.
- SPD matrices are factorizable (means something a bit different).
- For more general A we have

Theorem

If A is nonsingular then the rows of A can be permuted so that the permuted matrix is factorizable. The row permutations do not need to be known in advance. They can be constructed on-the-fly as the factorization proceeds.

- Sometimes, even more complex permutations are used.
- Note: factorizability here relates to the exact arithmetic

Basic Terminology: factorizations

Factorizations we will deal with:

- For symmetric positive definite A, the (square-root) Cholesky factorization $A = LL^T$, where L is a lower triangular matrix with positive diagonal entries.
  - Rewritten as $A = \widehat{L}D\widehat{L}^T$, where $\widehat{L}$ is a unit lower triangular matrix and D is a diagonal matrix with positive diagonal entries: square root-free Cholesky (LDLT) factorization.
- For nonsymmetric A, the LU factorization $A = LU$, where L is a unit lower triangular matrix and U is an upper triangular matrix. Gaussian elimination is one process to put a matrix into LU form.
  - Rewritten as $A = LD\widehat{U}$, where $\widehat{U}$ is a unit upper triangular matrix and D is a diagonal matrix. This is called the LDU factorization.

Basic Terminology: blocks

Blocks: small ones

- Symmetric block structure of A:
- $$A = (A_{ib,\,jb}), \;\; A_{ib,\,jb} \in \mathbb{R}^{n_i \times n_j}, \;\; 1 \le ib, jb \le nb,$$

  that is,

  $$A = \begin{pmatrix} A_{1,1} & A_{1,2} & \cdots & A_{1,nb} \\ A_{2,1} & A_{2,2} & \cdots & A_{2,nb} \\ \vdots & \vdots & \ddots & \vdots \\ A_{nb,1} & A_{nb,2} & \cdots & A_{nb,nb} \end{pmatrix}.$$
- Assuming nonsingular square blocks $A_{jb,\,jb}$ on the diagonal.
- Special cases: A is block diagonal if $A_{ib,\,jb} = 0$ for all $ib \ne jb$, A is block lower triangular if $A_{1:jb-1,\,jb} = 0$, $2 \le jb \le nb$, block upper triangular if $A_{jb+1:nb,\,jb} = 0$, $1 \le jb \le nb - 1$.
- Most useful; for us: small dense blocks

Basic Terminology: blocks and reducibility

Reducibility: large blocks

Definition

Matrix $A \in \mathbf{R}^{n\times n}$ is reducible, if there is a permutation matrix P such that

$$P^T A P = \begin{pmatrix} A_{11} & A_{12} \\ & A_{22} \end{pmatrix},$$

where A_{11} and A_{22} are square nontrivial matrices (of dimension at least 1). If A is not reducible, it is called irreducible. Matrices of dimension 1 are considered to be irreducible.

- A symmetric reducible matrix is block diagonal.
- These blocks can be (for our problems) sparse or dense

Basic Terminology: blocks and reducibility

Blocks and direct methods

- Blocks are extremely important from the point of view efficiency (fast block operations), standardization (use of efficient libraries)
- Solving systems with symmetric reducible matrices reduces to solving independent systems (since they are block diagonal as stated above)
- Solving systems with nonsymmetric reducible matrices can use block substitution.

$$\begin{pmatrix} A_{11} & A_{12} \\ & A_{22} \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \end{pmatrix} = \begin{pmatrix} b_1 \\ b_2 \end{pmatrix} \rightarrow x_2 = A_{22}^{-1} b_2, x_1 = A_{11}^{-1}(b_1 - A_{12}x_2)$$

Basic Terminology: blocks and reducibility

Blocks and direct methods

- For small blocks: all our factorizations can be (and are in production codes) formulated blockwise: entries → submatrices
- For large blocks: as we have seen above: typically another algorithmic level
- We will show ways to find some small blocks
- Large blocks sometimes follow from applications, but there are also ways to find them.

Basic Terminology: computational environment

Computational environment

- Basic sequential model: the von Neumann architecture: union of a central processing unit (CPU) and the memory, interconnected via input/output (I/O) mechanisms.

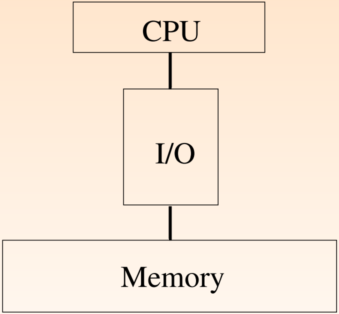


Figure: *A simple uniprocessor von Neumann computer model.*

Basic Terminology: computational environment

Computational environment

- Nowadays, nearly nothing is really sequential
- CPU → a mixture of powerful processors, coprocessors, cores, GPUs, and so on.
- Furthermore, performing arithmetic operations on the processing units is much faster than communication-based operations.
- But, there is a problem: improvements in the speed of the processing units outpace those in other pieces of hardware (memory, communication units).
- Moore's Law about this is an example of an experimentally derived observation of this kind.

Basic Terminology: computational environment

Computational environment

- Important milestones in processor development have been multiple functional units that compute identical numerical operations in parallel and data pipelining (also called vectorization) that enables the efficient processing of vectors and matrices.
- Vectorization often supported by additional tools like instruction pipelining, registers and by memory architectures with multiple layers, including small but fast memories called caches.
- Superscalar processors that enable the overlapping of identical (or different) arithmetic operations during run-time have been a standard component of computers since the 1990s.
- The ever-increasing heterogeneity of processing units and their hardware environment inside computers: expressing the code via units of scheduling and execution called threads.

Basic Terminology: computational environment

Computational environment

- Computer-based limitations:
  - Compute throughput, that is, the number of arithmetic operations that can be performed per cycle.
  - Memory throughput, that is, the number of operands than can be fetched from memory/cache and/or registers each cycle
  - Latency, which is the time from initiating a compute instruction or memory request before it is completed and the result available for use in the next computation.
- Distinguishing in contemporary terminology: algorithms that are compute-bound, memory-bound or latency-bound.
- More ways to hide latency (using blocks, prefetch, threads)

Basic Terminology: computational environment

Computational environment

- Measuring computational intensity: the ratio of the number of operations to the number of operands read from memory.
- Most chips are designed such that dense matrix-matrix multiply, which typically performs k^3 operations on k^2 data can run at full compute throughput,
- whilst matrix-vector multiply performs n^2 operations on n^2 data (ratio 1) and is limited by the memory throughput.

Basic Terminology: computational environment

Computational environment: libraries

- Machine-specific optimized BLAS libraries available for a wide variety of computer architectures.

procedure	*comm*	*ops*	ratio
BLAS 1: AXPY: $y = y + \alpha x$	$3n + 1$	$2n$	$2/3$
BLAS 2: GEMV: $y = Ax$	$n^2 + 2n$	$n(2n - 1)$	2
BLAS 3: GEMM: $C = AB$	$3n^2$	$n^2(2n - 1)$	$n/2$

- The development of basic linear algebra subroutines (BLAS) for performing common linear algebra operations on dense matrices partially motivated by obtaining a high ratio.
- Using Level 3 BLAS in solvers (for both sparse and dense matrices) can improve performance compared to using Level 1 and Level 2 BLAS.
- Other important motivations behind using the BLAS (standardization, portability).

Basic Terminology: finite precision arithmetic

Computational environment: finite precision

- The IEEE standard (1985) expresses real numbers as $a = \pm d_1.d_2 \ldots d_t \times 2^k$, where k is an integer and $d_i \in \{0, 1\}, 1 \leq i \leq t$, with $d_1 = 1$ unless $d_2 = d_3 = \ldots = d_t = 0$.
- $t = 24$ (single precision), $t = 53$ (double precision), exponent k satisfies $-126 \leq k \leq 127$ (single precision) and $-1022 \leq k \leq 1023$ (double precision).

Table: The number of bits in the significand and exponent, unit roundoff u, smallest positive (subnormal) number x^s_{min} , smallest normalized positive number x_{min}, and largest finite number x_{max}.

	t	k	u	x^s_{min}	x_{min}	x_{max}
bfloat16	8	8	3.91×10^{-3}	†	1.18×10^{-38}	3.39×10^{38}
fp16	11	5	4.88×10^{-4}	5.96×10^{-8}	6.10×10^{-5}	6.55×10^{4}
fp32	24	8	5.96×10^{-8}	1.40×10^{-45}	1.18×10^{-38}	3.40×10^{38}
fp64	53	11	1.11×10^{-16}	4.94×10^{-324}	2.22×10^{-308}	1.80×10^{308}

Basic Terminology: finite precision arithmetic

Computational environment: finite precision

- Floating-point (FP) operations:

  $$fl(a\ op\ b) = (a\ op\ b)(1+\delta), \qquad |\delta| \leq \epsilon,$$

  (op is a mathematical operation (such as $=, +, -, \times, /, \sqrt{}$) and $(a\ op\ b)$ is the exact result), ϵ is the machine epsilon.
- $2 \times \epsilon$ is the smallest FP number which when added to the FP number 1.0 gives a result different from 1.0.
- ϵ is $2^{-24} \approx 10^{-7}$ (single precision), $\epsilon = 2^{-53} \approx 10^{-16}$ (double precision) (so far, most often used)
- rounding errors, truncation errors.
- catastrophic errors → numerical instability
- contemporary interest in low precision (AI, exascale computing)

Basic Terminology: bit compatibility

Computational environment: bit compatibility

- Bit compatibility essential for some users because of regulatory requirements (such as within the nuclear or financial industries) or to build trust in their software.
- One critical issue is the way in which N numbers (or, more generally, matrices) are assembled:

  $$sum = \sum_{j=1}^{N} S_j,$$

  where the S_j are computed using one or more processors. The assembly is commutative but, because of the potential rounding of the intermediate results, it is not associative so that the result sum depends on the order in which the S_j are assembled.
- A straightforward approach to achieving bit compatibility is to enforce a defined order in such operations.
- This may adversely limit the scope for parallelism.

Basic Terminology: complexity

Computational complexity

- The computational complexity of a numerical algorithm typically based on estimating asymptotically operation counts / memory usage.

Definition

A real function $f(k)$ of a nonnegative real k satisfies $f = O(g)$ if there exist positive constants c_u and k_0 such that

$$f(k) \leq c_u g(k) \text{ for all } k \geq k_0. \tag{3}$$

We say that $f = \Theta(g)$ if, additionally, there exists a constant $c_l > 0$ such that

$$0 \leq c_l\, g(k) \leq f(k) \leq c_u\, g(k) \text{ for all } k \geq k_0.$$

Basic Terminology: complexity

Computational complexity

- While $O(g)$ bounds f asymptotically from above, $\Theta(g)$ represents an asymptotically tight bound.
- As a simple illustration, consider the quadratic function

  $$f(k) = \alpha * k^2 + \beta * k - \gamma.$$

  Provided $\alpha \neq 0$, $f(k) = \Theta(k^2)$ and the coefficient of the highest asymptotic term is α.
- Distinguishing worst-case complexity and average behaviour average-case complexity.
- Sparse matrix algorithms of complexity $\Theta(n^3)$ are considered to be computationally very expensive.

Basic Terminology: complexity

Computational complexity

- MFLOPs (and unit operation costs) may be misleading at contemporary computer architectures.
- Still terminology $O(.)$ (bounding from above) or $\Theta(.)$ (bounding from both sides) sometimes relevant
- It consists in replacing the bound (bounds) by $constant \times simpler\ function$ (etalon).
- Simpler functions are, e.g., $n^2, n^3, \log n, \ldots$
- In CS: polynomial complexity versus superpolynomial complexity. Our case: even polynomial complexity n^3 may be excessive.
- In CS: decision problems, polynomial reduction, class $\mathcal{NP}$; not discussed here

Basic Terminology: complexity

Inverse Ackermann function

- Very slowly increasing function we mention in some complexity considerations.

$$A(0,j) = j+1 \text{ for } j \geq 0; A(i,0) = A(i-1,1) \text{ for } i > 0$$

$$A(i,j) = A(i-1, A(i,j-1)) \text{ for } i,j > 0$$

$$\alpha(m,n) = \min i \geq 1 \text{ such that } A(i, ceil(m/n)) > \log_2 n$$

- See its implementation in the codes.

Outline

1. Introduction
2. Introductory notation and terminology
3. Factorizations
4. Graphs and sparse matrices
5. Sparse matrices and data structures
6. Symbolic factorizations
7. Symbolic Cholesky factorization
8. Cholesky factorization
9. Sparse LU factorization
10. Stability, ill-conditioning, indefiniteness
11. Blocks in the input matrix
12. Sparse Least Squares and QR factorization
13. Reorderings
14. Algebraic preconditioning
15. Incomplete factorizations
16. Sparse approximate inverses

Factorizations

Introduction to factorizations

- Traditional way of describing factorizations: derived from Gaussian elimination that represents systematic annihilation of the entries in the lower triangular part of A by columns.
- For A factorizable, formally a sequential multiplications by column elimination matrices getting the elimination sequence:

$$A = A^{(1)}, A^{(2)}, \ldots, A^{(n)}$$

  of partially eliminated matrices as follows:

$$A^{(1)} \to A^{(2)} = C_1 A^{(1)} \to A^{(3)} = C_2 C_1 A^{(1)} \to \ldots \to A^{(n)} = C_{n-1} \ldots C_1 A^{(1)}.$$

- The unit lower triangular matrices C_i $(1 \leq i \leq n-1)$ are the column elimination matrices.

Factorizations

Elementwise, assuming $a_{11} = a_{11}^{(1)} \neq 0$, the first step $C_1 A^{(1)} = A^{(2)}$ is

$$\begin{pmatrix} 1 & & & & \\ -a_{21}^{(1)}/a_{11}^{(1)} & 1 & & & \\ -a_{31}^{(1)}/a_{11}^{(1)} & & 1 & & \\ \vdots & & & 1 & \\ -a_{n1}^{(1)}/a_{11}^{(1)} & & & & 1 \end{pmatrix} \begin{pmatrix} a_{11}^{(1)} & a_{12}^{(1)} & \dots & a_{1n}^{(1)} \\ a_{21}^{(1)} & a_{22}^{(1)} & \dots & a_{2n}^{(1)} \\ a_{31}^{(1)} & a_{32}^{(1)} & \dots & a_{3n}^{(1)} \\ \vdots & \vdots & \ddots & \vdots \\ a_{n1}^{(1)} & a_{n2}^{(1)} & \dots & a_{nn}^{(1)} \end{pmatrix} = \begin{pmatrix} a_{11}^{(1)} & a_{12}^{(1)} & \dots & a_{1n}^{(1)} \\ 0 & a_{22}^{(2)} & \dots & a_{2n}^{(2)} \\ 0 & a_{32}^{(2)} & \dots & a_{3n}^{(2)} \\ \vdots & \vdots & \ddots & \vdots \\ 0 & a_{n2}^{(2)} & \dots & a_{nn}^{(2)} \end{pmatrix},$$

Provided $a_{22}^{(2)} \neq 0$, the second step $C_2 A^{(2)} = A^{(3)}$ is

$$\begin{pmatrix} 1 & & & & \\ & 1 & & & \\ & -a_{32}^{(2)}/a_{22}^{(2)} & 1 & & \\ & \vdots & & 1 & \\ & -a_{n2}^{(2)}/a_{22}^{(2)} & & & 1 \end{pmatrix} \begin{pmatrix} a_{11}^{(1)} & a_{12}^{(1)} & \dots & a_{1n}^{(1)} \\ 0 & a_{22}^{(2)} & \dots & a_{2n}^{(2)} \\ 0 & a_{32}^{(2)} & \dots & a_{3n}^{(2)} \\ \vdots & \vdots & \ddots & \vdots \\ 0 & a_{n2}^{(2)} & \dots & a_{nn}^{(2)} \end{pmatrix} = \begin{pmatrix} a_{11}^{(1)} & a_{12}^{(1)} & \dots & \dots & a_{1n}^{(1)} \\ 0 & a_{22}^{(2)} & \dots & \dots & a_{2n}^{(2)} \\ 0 & 0 & a_{33}^{(3)} & \dots & a_{3n}^{(3)} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & a_{n3}^{(3)} & \dots & a_{nn}^{(3)} \end{pmatrix}.$$

Factorizations

Introduction to factorizations

- The k-th partially eliminated matrix is $A^{(k)}$.
- The active entries in $A^{(k)}$: $a_{ij}^{(k)}$, $1 \leq k \leq i, j \leq n$. The submatrix of $A^{(k)}$ with the active entries: active submatrix.

- **The inverse of each C_k is the unit lower triangular matrix obtained by changing the sign of all off-diagonal entries.**
- **The product of unit lower triangular matrices (beware the order) is a unit lower triangular matrix: provided $a_{kk}^{(k)} \neq 0$ ($1 \leq k < n$)**

$$A = A^{(1)} = C_1^{-1} C_2^{-1} \dots C_{n-1}^{-1} A^{(n)} = LU,$$

- **Subdiagonal entries of L are the negative of the subdiagonal entries of the matrix $C_1 + C_2 + \dots + C_{n-1}$.**

Factorizations

For example: diagonal and first two columns of L

$$\begin{pmatrix} 1 & & & & \\ a_{21}^{(1)}/a_{11}^{(1)} & 1 & & & \\ a_{31}^{(1)}/a_{11}^{(1)} & a_{32}^{(2)}/a_{22}^{(2)} & 1 & & \\ \vdots & \vdots & & 1 & \\ a_{n1}^{(1)}/a_{11}^{(1)} & a_{n2}^{(2)}/a_{22}^{(2)} & & & 1 \end{pmatrix}$$

Factorizations

Rewriting LU in matrix/vector form: submatrix LU

- The first step ($k = 1$):

  $$C_1 A = \begin{pmatrix} 1 & \\ -v/a_{11} & I \end{pmatrix} \begin{pmatrix} a_{11} & u^T \\ v & A_{2:n,2:n} \end{pmatrix} = \begin{pmatrix} a_{11} & u^T \\ & A_{2:n,2:n} - vu^T/a_{11} \end{pmatrix},$$

  where

  $$v = (a_{21}, \ldots, a_{n1})^T, \quad (l_{21}, \ldots, l_{n1})^T = v/a_{11}, \quad u^T = (a_{12}, \ldots, a_{1n}).$$

- The $(n-1) \times (n-1)$ active submatrix

  $$S = A_{2:n,2:n} - vu^T/a_{11}$$

  is the Schur complement of A with respect to a_{11}.

- A is factorizable $\Rightarrow$ S is factorizable, and the process can be repeated.

Factorizations

Submatrix LU

- The operations performed at each step k correspond to a sequence of rank-one updates.
- After $k-1$ steps $(1 < k \le n)$, the $(n-k+1) \times (n-k+1)$ Schur complement of A with respect to its $(k-1) \times (k-1)$ principal leading submatrix is given by

$$\begin{aligned} S^{(k)} &= \begin{pmatrix} a_{kk} & \dots & a_{kn} \\ \vdots & \ddots & \vdots \\ a_{nk} & \dots & a_{nn} \end{pmatrix} - \sum_{j=1}^{k-1} \begin{pmatrix} l_{kj} \\ \vdots \\ l_{nj} \end{pmatrix} \begin{pmatrix} u_{jk} & \dots & u_{jn} \end{pmatrix} \\ &= \begin{pmatrix} a_{kk}^{(k)} & \dots & a_{kn}^{(k)} \\ \vdots & \ddots & \vdots \\ a_{nk}^{(k)} & \dots & a_{nn}^{(k)} \end{pmatrix} = A^{(k)}_{k:n,k:n}. \end{aligned} \tag{4}$$

- If A is SPD then the Cholesky and LDLT factorizations are termed right-looking (fan-out) factorizations.

Factorizations

Submatrix LU: example

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

$$\downarrow$$

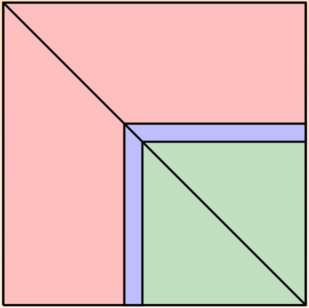
$$\begin{pmatrix} 4 & -1 & -5 & -11 & 6 \\ -0.25 & 3.75 & -3.25 & -3.75 & 4.5 \\ -0.25 & -0.25 & 2.75 & -3.75 & 6.5 \\ 1.5 & 8.5 & 15.5 & 26.5 & -7 \\ 2 & 1 & 9 & 26 & -5 \end{pmatrix} \rightarrow \begin{pmatrix} 4 & -1 & -5 & -11 & 6 \\ -0.25 & 3.75 & -3.25 & -3.75 & 4.5 \\ -0.25 & -0.07 & 2.53 & -4 & 6.8 \\ 1.5 & 2.27 & 22.87 & 35 & -17.2 \\ 2 & 0.27 & 9.87 & 27 & -6.2 \end{pmatrix}$$

$$\downarrow$$

$$\begin{pmatrix} 4 & -1 & -5 & -11 & 6 \\ -0.25 & 3.75 & -3.25 & -3.75 & 4.5 \\ -0.25 & -0.07 & 2.53 & -4 & 6.8 \\ 1.5 & 2.27 & 9.03 & 71.1 & -78.6 \\ 2 & 0.27 & 3.89 & 42.6 & -32.7 \end{pmatrix} \rightarrow \begin{pmatrix} 4 & -1 & -5 & -11 & 6 \\ -0.25 & 3.75 & -3.25 & -3.75 & 4.5 \\ -0.25 & -0.07 & 2.53 & -4 & 6.8 \\ 1.5 & 2.27 & 9.03 & 71.1 & -78.6 \\ 2 & 0.27 & 3.89 & 0.6 & 14.4 \end{pmatrix}$$

Factorizations

Submatrix LU: depiction



- First possibility: form the Schur complement by rows
- Second possibility: form the Schur complement by columns
- But, there are other ways to compute the LU factorization
- The are the same even in finite precision arithmetic!

Factorizations

Submatrix LU: algorithmically

Algorithm (kij LU decomposition (row oriented submatrix dense))

```
Initialise $L = (l_{ij}) = I, U = (u_{ij}) = 0$
for $k = 1 : n - 1$ do
    for $i = k + 1 : n$ do
        $l_{ik} = a_{ik}/a_{kk}$
        for $j = k + 1 : n$ do
            $a_{ij} = a_{ij} - l_{ik} * a_{kj}$
        end for
    end for
    $U_{k,k:n} = A_{k,k:n}$
end for
$u_{nn} = a_{nn}$
```

Factorizations

Submatrix LU: different order to form the Schur complement

Algorithm (kji LU decomposition (column oriented submatrix))

```
Initialise $L = (l_{ij}) = I, U = (u_{ij}) = 0$
for $k = 1 : n - 1$ do
    for $s = k + 1 : n$ do
        $l_{sk} = a_{s,k}/a_{k,k}$
    end for
    for $j = k + 1 : n$ do
        $l_{ik} = a_{ik}/a_{kk}$
        for $i = k + 1 : n$ do
            $a_{ij} = a_{ij} - l_{ik} * a_{kj}$
        end for
    end for
    $U_{k,k:n} = A_{k,k:n}$
end for
$u_{nn} = a_{nn}$
```

Factorizations

Column LU

- As mentioned, the factorization can be computed differently!
- Consider first j columns of A: we must have

$$\begin{pmatrix} A_{1:j-1,1:j-1} & A_{1:j-1,j} \\ A_{j:n,1:j-1} & A_{j:n,j} \end{pmatrix} \to \begin{pmatrix} L_{1:j-1,1:j-1} & \\ L_{j:n,1:j-1} & L_{j:n,j} \end{pmatrix} \begin{pmatrix} U_{1:j-1,1:j-1} & U_{1:j-1,j} \\ & u_{jj} \end{pmatrix}$$

- This implies the relations for new column of U and L:

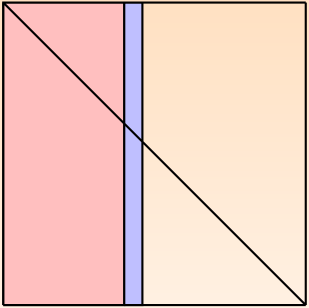
$$U_{1:j-1,j} = L^{-1}_{1:j-1,1:j-1} A_{1:j-1,j}, \quad u_{jj} = a_{jj} - L_{j,1:j-1} U_{1:j-1,j},$$

$$l_{jj} = 1, \; L_{j+1:n,j} = (A_{j+1:n,j} - L_{j+1:n,1:j-1} U_{1:j-1,j})/u_{jj}.$$

- The factors can be computed column by column: $j \to j+1 \to \dots$

Factorizations

Column LU: visualise and emphasize its two phases



- Two different phases of computing the columns of L and U.
  - The strictly upper triangular part of $U_{:j}$ is determined from the solve step
$$L_{1:j-1,1:j-1}U_{1:j-1,j} = A_{1:j-1,j},$$
  - The strictly lower triangular part of column j of L computed as a (scaled) linear combination of column $A_{j+1:n,j}$ of A and previously computed columns of L.

Factorizations

Column LU: algorithmically

Algorithm (Column LU factorization with row interchanges (partial pivoting))

```
Interchange rows of $A$ so that $|a_{11}| = \max\{|a_{i1}| \,|\, 1 \le i \le n\}$
$l_{11} = 1,\ u_{11} = a_{11},\ L_{2:n,1} = A_{2:n,1}/a_{11}$
for $j = 2 : n$ do
    Solve $L_{1:j-1,1:j-1} U_{1:j-1,j} = A_{1:j-1,j}$                ▷ 1st phase
    $z_{1:n-j+1} = A_{j:n,j} - L_{j:n,1:j-1} U_{1:j-1,j}$
    Apply row interchanges to $z$, $A$ and $L$ so that
        $|z_1| = \max\{|z_i| \,|\, 1 \le i \le n-j+1\}$.
    $l_{jj} = 1$, $u_{jj} = z_1$ and $L_{j+1:n,j} = z_{2:n-j+1}/z_1$    ▷ 2nd phase
end for
```

Factorizations

Submatrix LU: example

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

$$\downarrow$$

$$\begin{pmatrix} 4 & -1 & -5 & -11 & 6 \\ -0.25 & 4 & -2 & -1 & 3 \\ -0.25 & 0 & 4 & -1 & 5 \\ 1.5 & 7 & 8 & 10 & 2 \\ 2 & -1 & -1 & 4 & 7 \end{pmatrix} \rightarrow \begin{pmatrix} 4 & -1 & -5 & -11 & 6 \\ -0.25 & 3.75 & -2 & -1 & 3 \\ -0.25 & -0.07 & 4 & -1 & 5 \\ 1.5 & 2.27 & 8 & 10 & 2 \\ 2 & 0.27 & -1 & 4 & 7 \end{pmatrix}$$

- and so on . . .
- Other possibilities? Yes, by rows . . ., but this is equivalent to applying to A^T by columns ☺
- Is there anything else?

Factorizations

Introduction to factorizations: generic scheme of three nested loops

Algorithm (Generic LU factorization)

```
for ———— do
    for ———— do
        for ———— do
            $l_{ik} = a_{ik}^{(k)} a_{kk}^{-1}$
            $a_{ij}^{(k+1)} = a_{ij}^{(k)} - l_{ik} a_{kj}^{(k)}$
        end for
    end for
end for
```

- Both submatrix and column (row) factorizations are covered by this scheme!
- The crucial pointwise operation:

$$a_{ij} = a_{ij} - a_{ik} a_{kk}^{-1} a_{kj} \equiv a_{ij} = a_{ij} - l_{ik} a_{kj}$$

Factorizations

Introduction to factorizations: generic scheme

- Three nested loops: performance differs based on sparsity, computer architecture.
- Some vectorize more, some less, etc.
- Even in finite precision arithmetic: the same L and U since the quantities at each position are modified in the same order
- To identify a variant: order in which the indices are assigned to the loops.
  - kij and kji: submatrix LU factorizations,
  - jik and jki: column factorizations.
  - The remaining ones: row factorizations (column LU factorization applied to A^T.)

Factorizations

Outside the generic scheme

- An alternative is factorization by bordering.
- Set all diagonal entries of L to 1 and assume the first $k-1$ rows of L and first $k-1$ columns of U $(1 < k \leq n)$ have been computed (that is, $L_{1:k-1,1:k-1}$ and $U_{1:k-1,1:k-1}$). At step k, $A_{1:k,1:k}$ satisfies

$$\begin{pmatrix} A_{1:k-1,1:k-1} & A_{1:k-1,k} \\ A_{k,1:k-1} & a_{kk} \end{pmatrix} = \begin{pmatrix} L_{1:k-1,1:k-1} & 0 \\ L_{k,1:k-1} & 1 \end{pmatrix} \begin{pmatrix} U_{1:k-1,1:k-1} & U_{1:k-1,k} \\ 0 & u_{kk} \end{pmatrix}.$$

- Quantities to be computed: in red

Factorizations

Bordering approach: the computation

- That is, the lower triangular part of row k of L and the upper triangular part of column k of U are obtained by solving

$$\begin{aligned} L_{k,1:k-1}U_{1:k-1,1:k-1} &= A_{k,1:k-1}, \\ L_{1:k-1,1:k-1}U_{1:k-1,k} &= A_{1:k-1,k}. \end{aligned}$$

The diagonal entry u_{kk} is then given by

$$u_{kk} = a_{kk} - L_{k,1:k-1}U_{1:k-1,k} \quad (\text{with } \; u_{11} = a_{11}).$$

Factorizations

Switching to A SPD: S is SPD

Lemma

Consider one step of the submatrix factorization of an SPD A. Schur complement of A with respect to (positive) $a_{1,1}$ is positive definite.

Proof.

For $\begin{pmatrix}\alpha & z^T\end{pmatrix}^T$ we have $x^TAx =$

$$\begin{pmatrix}\alpha & z^T\end{pmatrix}\begin{pmatrix}a_{1,1} & a_{1,2:n} \\ a_{2:n,1} & A_{2:n,2:n}\end{pmatrix}\begin{pmatrix}\alpha \\ z\end{pmatrix} =$$

$$\alpha^2 a_{1,1} + \alpha a_{1,2:n} z + \alpha z^T a_{2:n,1} + z^T A_{2:n,2:n} z =$$

$$(\alpha + a_{1,1}^{-1} a_{1,2:n} z)^T a_{1,1} (\alpha + a_{1,1}^{-1} a_{1,2:n} z) + z^T (A_{2:n,2:n} - a_{2:n,1} a_{1,1}^{-1} a_{1,2:n}) z$$

Choosing $z \neq 0$ and setting $\alpha = -a_{1,1}^{-1} a_{1,2:n} z$ we get

$$x^TAx = z^TSz \text{ where } S = A_{2:n,2:n} - a_{2:n,1} a_{1,1}^{-1} a_{1,2:n}.$$

□

Factorizations

Introduction to factorizations: SPD matrices

- A is SPD → setting $U = DL^T$, the LU factorization can be written as
$$A = LDL^T,$$
- This is the square root-free Cholesky factorization.
- Alternatively, expressed as the standard square-root Cholesky factorization
$$A = (LD^{1/2})(LD^{1/2})^T,$$
lower triangular matrix $LD^{1/2}$ has positive diagonal entries.
- Since there is no square-root here, more general case of some indefinite or some SPSD matrices (diagonal entries can be negative, at the end we can have zeros) is covered.
- Often talking about symmetric (LDLT) factorizations only, assuming just factorizability (!)

Factorizations

Introduction to factorizations: SPD matrices

- Two completely different phases of the column construction: implications for Cholesky factorization:
  - If A is symmetric, j-the column of U is the j-th row of L (see the column approach)
  - Consequently, there is no solve phase in the symmetric factorization
  - Similarly, in the submatrix algorithm we do not update strict upper triangle of the Schur complement
  - But if A is symmetric and its factorization is symmetric, L can be computed by the solve step only

Factorizations

Column LU: A symmetric, showing LDLT

Observation

d_{jj} ($1 \leq j \leq n$) of the LDLT factorization of the symmetric A is

$$d_{jj} = u_{jj} = a_{jj} - \sum_{k=1}^{j-1} d_{kk} l_{jk}^2.$$

The L factor is the same as is computed by the column LU factorization and

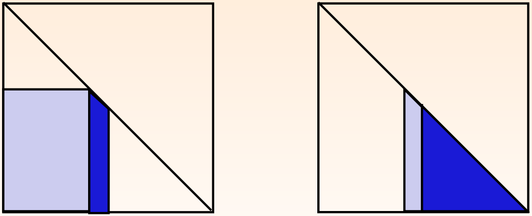
$$d_{jj} L_{j+1:n,j} = A_{j+1:n,j} - \sum_{k=1}^{j-1} L_{j+1:n,k}\, d_{kk}\, l_{jk}.$$

The U factor is equal to DL^T.

Factorizations

Cholesky factorization: summary: three basic ways

- Left-looking schemes (second phase of the column LU)
- Right-looking schemes (submatrix scheme that computes only quantities in L)
- Row scheme based on the first phase (solve) of the column LU (shown algorithmically below)



- LDLT above is the computed by a left-looking scheme

Factorizations

Submatrix (right-looking) Cholesky

$$
\begin{aligned}
A &= \begin{pmatrix} a_{1,1} & a_{1,2} & a_{1,3:n} \\ a_{2,1} & a_{2,2} & a_{2,3:n} \\ a_{3:n,1} & a_{3:n,2} & A_{3:n,3:n} \end{pmatrix} \\
&= \begin{pmatrix} \sqrt{a_{1,1}} & 0 & \\ \frac{a_{2,1}}{\sqrt{a_{1,1}}} & \sqrt{a^{(1)}_{2,2}} & \\ \frac{a_{3:n,1}}{\sqrt{a_{1,1}}} & \frac{a^{(1)}_{3:n,2}}{\sqrt{a^{(1)}_{2,2}}} & I_{n-2} \end{pmatrix} \begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & A^{(2)}_{3:n,3:n} - \frac{a_{3:n,1}a_{1,3:n}}{a_{1,1}} - \frac{a^{(1)}_{3:n,2}a^{(1)}_{2,3:n}}{a^{(1)}_{2,2}} \end{pmatrix} \\
&\quad \begin{pmatrix} \sqrt{a_{1,1}} & \frac{a_{2,1}}{\sqrt{a_{1,1}}} & \frac{a_{1,3:n}}{\sqrt{a_{1,1}}} \\ 0 & \sqrt{a^{(1)}_{2,2}} & \frac{a^{(1)}_{2,3:n}}{\sqrt{a^{(1)}_{2,2}}} \\ & & I_{n-2} \end{pmatrix} \\
&= \begin{pmatrix} l_{1,1} & 0 & 0 \\ l_{2,1} & l_{2,2} & 0 \\ l_{3:n,1} & l_{3:n,2} & I_{n-2} \end{pmatrix} \begin{pmatrix} l_{1,1} & l_{2,1} & l_{1,3:n} \\ 0 & l_{2,2} & l_{2,3:n} \\ 0 & 0 & I_{n-2} \end{pmatrix}
\end{aligned}
$$

Factorizations

Column (left-looking) Cholesky

Algorithm

Column Cholesky factorization: $A \rightarrow$ *square-root factor* $L = (l_{ij})$

1. *for* $j = 1 : n$ *do*
2. *Compute an auxiliary vector* $t_{j:n}$

$$\begin{pmatrix} t_j \\ \vdots \\ t_n \end{pmatrix} = \begin{pmatrix} a_{jj} \\ \vdots \\ a_{nj} \end{pmatrix} - \sum_{\{k \mid l_{jk} \neq 0\}} l_{jk} \begin{pmatrix} l_{jk} \\ \vdots \\ l_{nk} \end{pmatrix} \qquad (5)$$

3. *Get a column of* L *by scaling* $t_{j:n}$

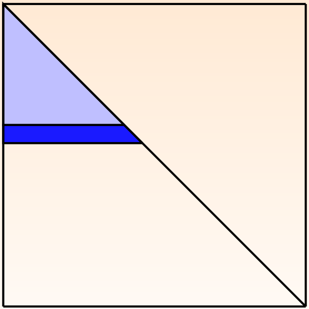
$$\begin{pmatrix} l_{jj} \\ \vdots \\ l_{nj} \end{pmatrix} = \frac{1}{\sqrt{t_j}} \begin{pmatrix} t_j \\ \vdots \\ t_n \end{pmatrix} \qquad (6)$$

4. *end* j

Factorizations

Cholesky factorization: row scheme

- But there is also the row scheme.
- The row scheme is based on the first phase (solve) of the column LU



Factorizations

Row (up-looking) Cholesky

Algorithm

Row Cholesky factorization: : $A \to$ *square-root factor* $L = (l_{ij})$*.*

```
1. for $i = 1 : n$ do
2. Solve the triangular system
   $$L_{1:i-1,1:i-1} \begin{pmatrix} l_{i1} \\ \vdots \\ l_{i,i-1} \end{pmatrix} = \begin{pmatrix} a_{i1} \\ \vdots \\ a_{i,i-1} \end{pmatrix} \qquad (7)$$
3. Compute the diagonal entry $l_{ii} = \sqrt{\left(a_{ii} - \sum_{k=1}^{i-1} l_{ik}^2\right)}$
4. end $i$
```

Factorizations

A not factorizable

- What if A is not factorizable? But regular!!!
- Then there exists a row permutation matrix P such that PA is factorizable (row interchanges).
- Consider the simple 2×2 matrix A and its LU factorization

  $$A = \begin{pmatrix} \delta & 1 \\ 1 & 1 \end{pmatrix} = \begin{pmatrix} 1 & \\ \delta^{-1} & 1 \end{pmatrix} \begin{pmatrix} \delta & 1 \\ & 1 - \delta^{-1} \end{pmatrix}.$$

  If $\delta = 0$ this factorization does not exist and if δ is very small then the entries in the factors involving δ^{-1} are very large.
- Interchanging the rows of A we have

  $$PA = \begin{pmatrix} 1 & 1 \\ \delta & 1 \end{pmatrix} = \begin{pmatrix} 1 & \\ \delta & 1 \end{pmatrix} \begin{pmatrix} 1 & 1 \\ & 1 - \delta \end{pmatrix},$$

  which is valid for all $\delta \neq 1$. But this is a nonsymmetric permutation.
- To keep symmetry: specific ways shown below

Factorizations

Elimination versus factorization: partial summary

- Householder (end of 1950's, beginning of 1960's): expressing Gaussian elimination as a decomposition
- Various reformulations of the same decomposition: different properties in
  - sparse implementations
  - vector processing
  - other parallel implementations, GPUs
- Generic scheme for Cholesky as for LU gives a useful framework. There are also other schemes (bordering, Dongarra-Eisenstat using also submatrices etc.)
- Remind that we can have always a underlying block structure

Factorizations

Factorizations and sparsity

- Factorizations of sparse matrices create new nonzero entries outside $\mathcal{S}\{A\}$ called fill/fill-in/filled entries as in the following arrowhead example

$$\begin{pmatrix} * & * & * & * & * \\ * & * & & & \\ * & & * & & \\ * & & & * & \\ * & & & & * \end{pmatrix} \rightarrow \begin{pmatrix} * & * & * & * & * \\ * & * & f & f & f \\ * & f & * & f & f \\ * & f & f & * & f \\ * & f & f & f & * \end{pmatrix}$$

- Fill-in means more operations, more memory
- This is, of course, very pessimistic example. More typical cases shown before.

Factorizations

Factorizations and sparsity

- Can we expect that some nonzeros become zeros due to cancellation?
- Very rarely. Numerical cancellations in LU factorizations rarely happen. Also difficult to predict.
- We assume non-cancellation: the result of adding, subtracting or multiplying two nonzeros is nonzero again.

Observation

The sparsity structures of the LU factors of A satisfy

$$\mathcal{S}\{A\} \subseteq \mathcal{S}\{L+U\}.$$

- A subtle reason to stick with the non-cancellation assumption: we intend to check only existence of nonzero and not its value in order to use graphs

Outline

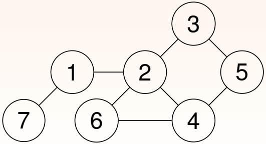
1. Introduction
2. Introductory notation and terminology
3. Factorizations
4. Graphs and sparse matrices
5. Sparse matrices and data structures
6. Symbolic factorizations
7. Symbolic Cholesky factorization
8. Cholesky factorization
9. Sparse LU factorization
10. Stability, ill-conditioning, indefiniteness
11. Blocks in the input matrix
12. Sparse Least Squares and QR factorization
13. Reorderings
14. Algebraic preconditioning
15. Incomplete factorizations
16. Sparse approximate inverses

Graphs and sparse matrices

Sparse matrices and graphs

- To describe the sparse matrix pattern $\mathcal{S}(A)$, graphs can be used
- If A is symmetric, undirected graph model is the right one (not distinguishing between positions (i, j) and (j, i)).

$$\begin{array}{cc}
 & \begin{array}{ccccccc} 1 & 2 & 3 & 4 & 5 & 6 & 7 \end{array} \\
\begin{array}{c} 1 \\ 2 \\ 3 \\ 4 \\ 5 \\ 6 \\ 7 \end{array} &
\left(\begin{array}{ccccccc}
* & * & & & & & * \\
* & * & * & * & & * & \\
 & * & * & & * & & \\
 & * & & * & * & * & \\
 & & * & * & * & & \\
 & * & & * & & * & \\
* & & & & & & *
\end{array}\right)
\end{array}$$

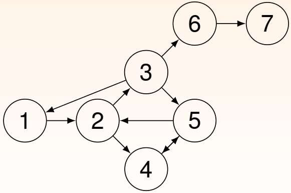


Graphs and sparse matrices

Sparse matrices and graphs

- If A is nonsymmetric, a directed graph model that distinguishes between position (i, j) and (j, i)) can be used.

$$\begin{array}{c|ccccccc}
 & 1 & 2 & 3 & 4 & 5 & 6 & 7 \\
\hline
1 & {\color{red}*} & * & & & & & \\
2 & & {\color{red}*} & * & * & & & \\
3 & * & & {\color{red}*} & & * & * & \\
4 & & & & {\color{red}*} & * & & \\
5 & & * & & * & {\color{red}*} & & \\
6 & & & & & & {\color{red}*} & * \\
7 & & & & & & & {\color{red}*}
\end{array}$$

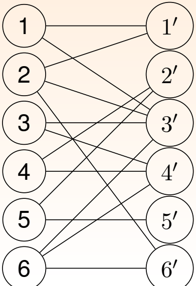


Graphs and sparse matrices

Sparse matrices and graphs

- A more general possibility is to use a bipartite graph that distinguishes between column and row nodes. It can also show sparsity pattern of rectangular matrices.
- This model also nontrivially captures diagonal entries - can be exploited in algorithms
- A simple bipartite graph is an ordered pair of sets (R, C, E) such that $E = \{\{i,j\} | i \in R, j \in C\}$. R is called the row vertex set, C is called the column vertex set and E is called the edge set.

$$\begin{array}{c|cccccc}
 & 1' & 2' & 3' & 4' & 5' & 6' \\
\hline
1 & * & & * & & & \\
2 & * & & * & & & * \\
3 & & & * & * & & \\
4 & & * & & * & & \\
5 & & * & & & * & \\
6 & & & * & * & & *
\end{array}$$



Graphs and sparse matrices

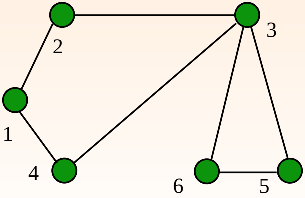
Sparse matrices and graphs: more formally

- A graph $\mathcal{G} = (\mathcal{V}, \mathcal{E})$ is a finite set $\mathcal{V}$ of vertices (or nodes), and a set $\mathcal{E}$ of edges defined as pairs of distinct vertices.
-
- Undirected case: $E \subseteq \binom{V}{2}$, directed case: $E \in V \times V$.
- No distinction between (i, j) and (j, i): edges can be represented by unordered pairs → $\mathcal{G}$ is undirected, otherwise directed (digraph).
- Our bipartite graph was undirected, but it can be also directed.
- Note that we do not indicate diagonal entries by edges (called loops). Formally, our graphs are just simple graphs, without multiple edges. Models can be more complex: bipartite graph can be directed.
- To capture not only $\mathcal{S}(A)$ but also the values, $\mathcal{G}$ can be transformed into a weighted graph using a mapping $\mathcal{E}(A) \to \mathbb{R}$ and/or $\mathcal{V}(A) \to \mathbb{R}$.

Graphs and their matrices

Graph terminology: more formally

- Graph induced by a matrix A (representing $\mathcal{S}(A)$): denoted $\mathcal{G}(A)$
- Labelling (or ordering) of a graph $\mathcal{G} = (\mathcal{V}, \mathcal{E})$ with n vertices: it is a bijection of $\{1, 2, \ldots, n\}$ onto $\mathcal{V}$. The integer i ($1 \le i \le n$) assigned to a vertex in $\mathcal{V}$ is called the label (or simply the number).
- Standard choice of vertex labelling $\mathcal{V} = \{1, \ldots, n\}$: vertices are directly identified by their labels.
- Relabelling of vertices of $\mathcal{G}(A)$ corresponds to a symmetric permutation of the underlying matrix A.
- Another example of a labelled undirected graph:



Simple undirected graph $G = (\{1, 2, 3, 4, 5, 6\}, \{\{1, 2\}, \{2, 3\}, \{1, 4\}, \{3, 4\}, \{3, 6\}, \{3, 5\}, \{5, 6\}\})$

Graphs and their matrices

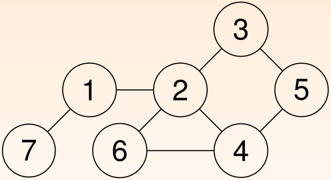
Graph terminology

- $\mathcal{G}_s = (\mathcal{V}_s, \mathcal{E}_s)$ is a subgraph of $\mathcal{G} = (\mathcal{V}, \mathcal{E})$ if and only if $\mathcal{V}_s \subseteq \mathcal{V}$ and $\mathcal{E}_s \subseteq \mathcal{E}$ and $(u_s, v_s) \in \mathcal{E}_s$ implies $u_s, v_s \in \mathcal{V}_s$.
- The subgraph is an induced subgraph if $\mathcal{E}_s$ contains all the edges in $\mathcal{E}$ that have both u and v in $\mathcal{V}_s$.
- Two graphs $\mathcal{G} = (\mathcal{V}, \mathcal{E})$ and $\mathcal{G}_s = (\mathcal{V}_s, \mathcal{E}_s)$ are isomorphic if there is a bijection $g : \mathcal{V} \ \to \mathcal{V}_s$ that preserves adjacency, that is $(u, v) \in \mathcal{E}$ if and only if $(g(u), g(v)) \in \mathcal{E}_s$.
- Undirected graph: two vertices u and v in $\mathcal{V}$ are said to be adjacent (or neighbours) if $e = (u, v) \in \mathcal{E}$; the edge e is incident to the vertex u and to the vertex v. u and v are the endpoints of e.
- We also use the notation $(u \longleftrightarrow v)$ for an edge (or $(u \overset{\mathcal{G}}{\longleftrightarrow} v)$ to emphasise that the edge belongs to the graph $\mathcal{G}$).

Graphs and their matrices

Graph terminology: degree, adjacency

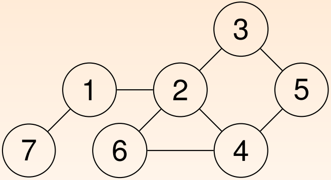
- The degree $deg_{\mathcal{G}}(u)$ of $u \in \mathcal{V}$ is the number of vertices in $\mathcal{V}$ that are adjacent to u, and the adjacency set $adj_{\mathcal{G}}\{u\}$ is the set of these adjacent vertices (thus $|adj_{\mathcal{G}}\{u\}| = deg_{\mathcal{G}}(u)$).
- If $\mathcal{V}_s$ is a subset of the vertices, then the adjacency set $adj_{\mathcal{G}}\{\mathcal{V}_s\}$ is the set of vertices in $\mathcal{V} \setminus \mathcal{V}_s$ that are adjacent to at least one vertex in $\mathcal{V}_s$.



- Degree of vertex 2 is four: $deg_{\mathcal{G}}(2) = 4$, its adjacency set are vertices $1, 3, 4, 6$: $adj_{\mathcal{G}}\{2\} = \{1, 3, 4, 6\}$, $adj_{\mathcal{G}}\{2, 4, 6\} = \{1, 3, 5\}$.

Graph terminology: clique

- A subgraph is a clique when every pair of vertices is adjacent.
- Clique we use just for undirected graphs, apart from similarity of using some other terminology for both directed and undirected graphs.



- Vertices $2, 4, 6$ form a clique: the induced subgraph with vertices $\mathcal{V}_s = \{2, 4, 6\}$ is a clique.

Graphs and their matrices

Transfer between the classes of undirected and directed graphs

- Many pieces of terminology have small variations for both directed and undirected graphs
- Symmetrization: directed → undirected
  - Just considering edges from $V \times V$ as from $\binom{V}{2}$
- Orientation: undirected → directed
  - Not unique. Instead from an edge from $\binom{V}{2}$ we can have one or two edges from $V \times V$.
- Let us remind, part of terminology is shared between the classes of undirected and directed graphs

Graphs and their matrices

Graph terminology: edges, walk, path

- Notation $(u \to v)$ for a directed edge.
- Emphasising the graph to which the edge belongs: $(u \xrightarrow{\mathcal{G}} v)$.
- In a digraph there can be an edge $(u \to v)$ but no edge $(v \to u)$.
- The adjacency set of u can be split into two parts

$$adj^{+}_{\mathcal{G}}\{u\} = \{v \mid (u \to v) \in \mathcal{E}\} \quad \text{and} \quad adj^{-}_{\mathcal{G}}\{u\} = \{v \mid (v \to u) \in \mathcal{E}\}.$$

- A sequence of k edges in an undirected graph $\mathcal{G}$: a walk. (edges distinct: trail, vertices distinct: path.

$$u_0 \longleftrightarrow u_1 \longleftrightarrow \ldots \longleftrightarrow u_{k-1} \longleftrightarrow u_k$$

- k is the length of the walk. A walk of zero length: $k = 0$.
- The walk is closed if $u_0 = u_k$; a closed walk is called a cycle.

Graphs and their matrices

Graph terminology: denoting paths

- If the vertices $\mathcal{V}$ are labelled $1, 2, \ldots, n$ then in the undirected graph $\mathcal{G} = (\mathcal{V}, \mathcal{E})$ a path between a pair of its vertices with labels i and j is denoted by

  $$i \overset{\mathcal{G}}{\Longleftrightarrow} j$$

  or, if it is clear which graph the path is in, by

  $$i \Longleftrightarrow j.$$

  If all intermediate vertices on the path are less than $\min\{i, j\}$ then the path is called a fill-path and is denoted by

  $$i \underset{min}{\overset{\mathcal{G}}{\Longleftrightarrow}} j \quad \text{or} \quad i \underset{min}{\Longleftrightarrow} j.$$

Graphs and their matrices

Graph terminology: denoting paths

- If all intermediate vertices on the path belong to a subset $\mathcal{V}_s$ then the path is denoted by

$$i \xLeftrightarrow[\mathcal{V}_s]{\mathcal{G}} j \quad \text{or} \quad i \xLeftrightarrow[\mathcal{V}_s]{} j.$$

- If $\mathcal{G}$ is a digraph then the sequence: a directed walk.

$$u_0 \longrightarrow u_1 \longrightarrow \ldots \longrightarrow u_{k-1} \longrightarrow u_k$$

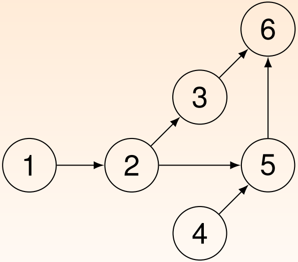
- A (directed) trail is a (directed) walk in which all the edges are distinct and a (directed) path is a (directed) trail in which all the vertices (and therefore also all the edges) are distinct.
- If $\mathcal{G}$ is a digraph, the double-sided arrow symbols are replaced by one-sided ones $\Longrightarrow$ in the direction of the edges. For example,

$$i \xRightarrow{\mathcal{G}} j, \quad i \Longrightarrow j, \quad i \xRightarrow[min]{} j \quad \text{and} \quad i \xRightarrow[\mathcal{V}_s]{} j.$$

Graphs and their matrices

Graph terminology: distances

- Paths can be used to determine distances between pairs of vertices.
- The distance between two vertices is the number of edges in the shortest path connecting them (this is also called the length of the path).



- Distance between 1 and 6 is three.

Graphs and their matrices

Graph terminology: connectedness, trees

- An undirected graph is connected if every pair of vertices is connected by a path.
- A (undirected) connected acyclic graph is called a tree, that is, a tree is an undirected graph in which any two vertices are connected by exactly one path.
- Every tree has at least two vertices of degree 1. Such vertices are called leaf vertices.
- A graph is a forest if it consists of a disjoint union of trees. Connectivity is an equivalence relation and consequently, it provides a partition of $\mathcal{V}$ into disjoint equivalence classes.

Graphs and their matrices

Graph terminology: connectedness, trees

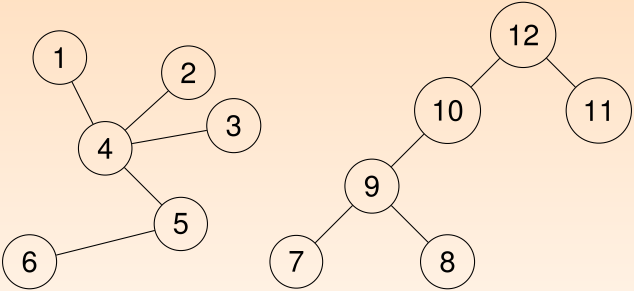


Figure: *A disconnected undirected graph with 12 vertices that is a forest (consisting of two disjoint trees). Vertices 1, 2, 3, 6, 7, 8 and 11 are leaf vertices.*

- In directed trees leaves may be only some vertices of degree 1.

Graphs and their matrices

Graph terminology: connectedness, trees

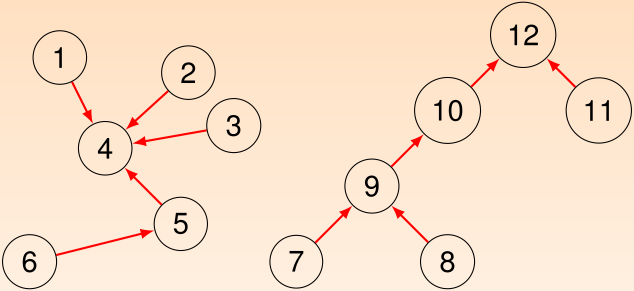


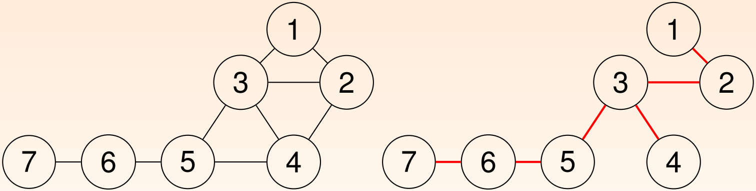
Figure: *An example of a directed graph with 12 vertices that is a directed forest (it consists of two disjoint trees). Vertices* $1, 2, 3, 6, 7, 8, 11$ *considered as leaf vertices.*

- In directed trees leaves may be only some vertices of degree 1 (e.g., those with the outgoing edge).
- Terminology must serve to our goals!

Graphs and their matrices

Graph terminology: connected, strongly connected

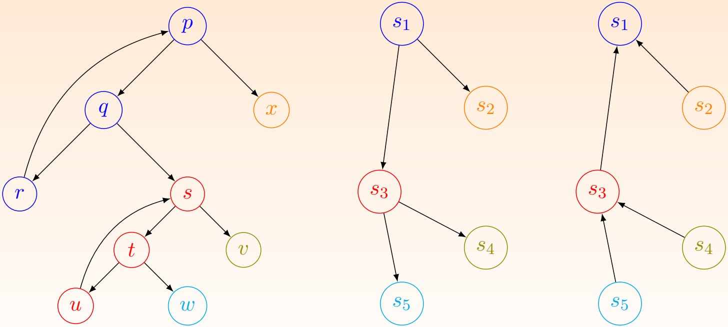
- If $\mathcal{G}$ is connected then a spanning tree of $\mathcal{G}$ is a subgraph of $\mathcal{G}$ that is a tree containing every vertex of $\mathcal{G}$.
- Graph (left) and its spanning tree (right)



Graphs and their matrices

Graph terminology: directed graphs: strong connectedness

- A directed graph $\mathcal{G} = (\mathcal{V}, \mathcal{E})$ is strongly connected if for every pair of vertices $u, v \in \mathcal{V}$ there is a path from u to v and a path from v to u.
- Strongly connected subgraphs: strong components (SCC)
- An example: five SCCs: $\{p, q, r\}$, $\{s, t, u\}$, $\{v\}$, $\{w\}$, and $\{x\}$.



- The graphs with shrinked SCC are called condensations

Graph terminology: directed graphs: strong connectedness

- Strong connectivity is an equivalence relation on $\mathcal{V}$. It induces a partitioning

  $$\mathcal{V} = \mathcal{V}_1 \cup \ldots \cup \mathcal{V}_s$$

  such that each $\mathcal{V}_i$ $(1 \le i \le s)$ is strongly connected and is maximal with this property: no additional vertices from $\mathcal{G}$ can be included in $\mathcal{V}_i$ without breaking its strong connectivity.
- Shrinking the strong components $\mathcal{V}_i$: we get a direct graph without cycles.
- Graphs that do not contain cycles are also called acyclic.

Graphs and their matrices

Graph terminology

- Digraphs with no cycles (directed acyclic graphs, DAGs) are extremely useful to model some applications.
- In a DAG, if there is a path $u \Longrightarrow v$ then u is called an ancestor of v and v is said to be a descendant of u.
- Topological ordering of vertices of a DAG: edges from u to v only for $u < v$.
- A topological ordering of a graph is possible if and only if it is a DAG. Generally non-unique. Any DAG has at least one topological ordering.

Graphs and their matrices

Graph terminology

- DAG with two different topological orderings. Left: vertices 2, 3, 5 and 6 are descendants of 1. Only vertices 5 and 6 are descendants of vertex 4.

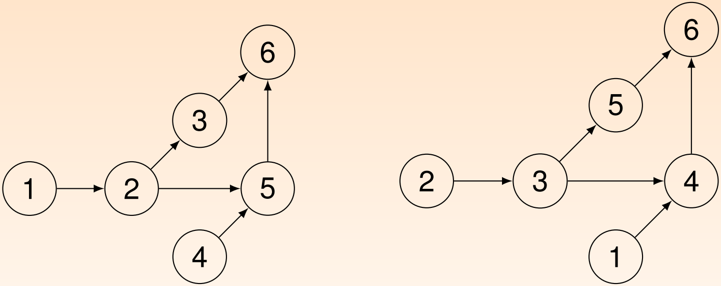


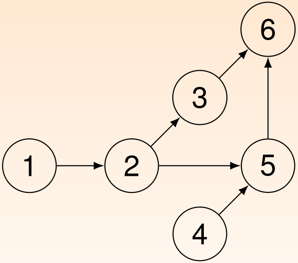
Figure: *An example of a DAG with two different topological orderings.*

- v is the parent of u if the directed edge $(u \to v) \in \mathcal{E}'$; u is a child of v (two or more child vertices are referred to as children). Leaf vertices have no children.

Graphs and their matrices

Graph terminology: reachability

- The vertices u_0 and u_k are connected by the walk and for $k > 0$, u_k is said to be reachable from u_0.
- The set of vertices that are reachable from u_0 is denoted by $\mathcal{R}each(u_0)$.



- $\mathcal{R}each(2) = \{3, 5, 6\}$.
- Reachability may or may not take direction of edges into account: it is a concept for both undirected and directed graphs.

Graphs and their matrices

Graph terminology: reachability through a set

- Given $\mathcal{G}$ a subset $\mathcal{V}_s$ of its vertices: if u and v are two distinct vertices that do not belong to $\mathcal{V}_s$, then v is reachable from u through $\mathcal{V}_s$ if u and v are connected by a path that is either of length 1 or is composed entirely of vertices that belong to $\mathcal{V}_s$ (except for the endpoints u and v).

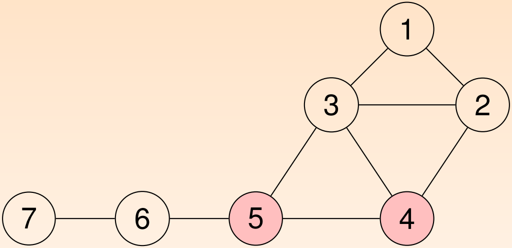
- Given $\mathcal{V}_s$ and $u \notin \mathcal{V}_s$, the reachable set $\mathcal{R}each(u, \mathcal{V}_s)$ of u through $\mathcal{V}_s$ is the set of all vertices that are reachable from u through $\mathcal{V}_s$. Note that if $\mathcal{V}_s$ is empty or u does not belong to $adj_{\mathcal{G}}(\mathcal{V}_s)$ then $\mathcal{R}each(u, \mathcal{V}_s) = adj_{\mathcal{G}}(u)$.

- Reachability can be considered for both undirected or directed or even mixed (directed-undirected) graphs.

- A simple example is given in the following Figure.

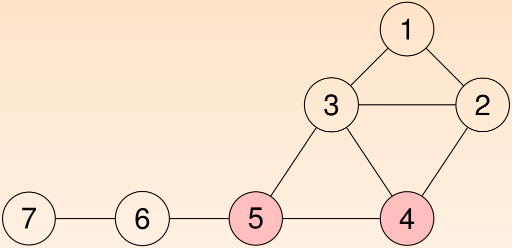
Graphs and their matrices

Graph terminology: reachability through a set



- $\mathcal{V}_s = \{4, 5\}$

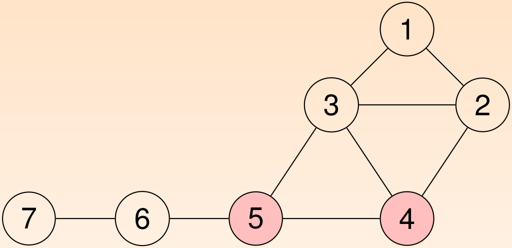
Graph terminology: reachability through a set



- $\mathcal{V}_s = \{4, 5\}$
- $\mathcal{R}each(2, \mathcal{V}_s) = \{1, 3, 6\}$

Graphs and their matrices

Graph terminology: reachability through a set



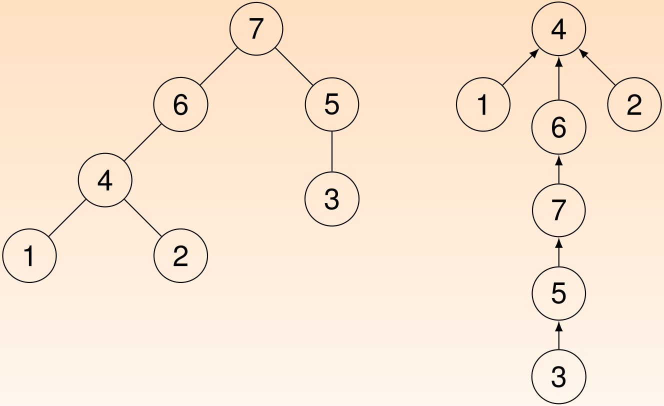
- $\mathcal{V}_s = \{4, 5\}$
- $\mathcal{R}each(2, \mathcal{V}_s) = \{1, 3, 6\}$
- $\mathcal{R}each(6, \mathcal{V}_s) = \{2, 3, 7\}$.

Graphs and their matrices

Graph terminology: rooted trees

- Rooted trees are DAGs defined from an undirected tree by a chosen vertex: root (remind: tree is connected)
- Any undirected tree $\mathcal{T} = (\mathcal{V}, \mathcal{E})$ is converted to a directed rooted tree $\mathcal{T}' = (\mathcal{V}, \mathcal{E}')$ by specifying a root vertex r.
- r can be chosen arbitrarily: any choice gives a directed rooted tree. An edge with endpoints u and v in $\mathcal{E}$ becomes a directed edge $(u \to v)$ in $\mathcal{E}'$ if there is a path from u to r such that the first edge of this path is from u to v.
- v is called the parent of u if the directed edge $(u \to v) \in \mathcal{E}'$; u is said to be a child of v (two or more child vertices are referred to as children). Two vertices in a rooted tree are siblings if they have the same parent. Leaf vertices have no children.
- Given r, this directed path is unique.

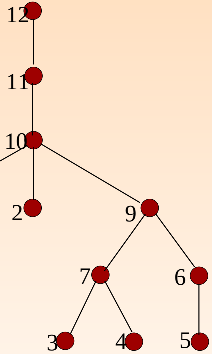
Graph terminology: rooted trees



- Example left: if 7 is a root: get a rooted tree: orientation can be chosen
- . . . but later we will see that we like the orientation upwards ☺
- Example right: 4 is a root, orientation chosen

Graphs and their matrices

Another example of a rooted tree



- The root of this tree is 12, 10 is an ancestor of vertices 2, 8 a 9. These vertices are descendants of 10. Set of ancestors of 10 is $anc(10) = \{10, 11, 12\}$. $parents$ for vertices $1 \ldots 11$: $8, 10, 7, 7, 6, 9, 9, 10, 10, 11, 12,$ null.
- 12 is not considered a leaf

Graphs and their matrices

Graphs can imply matrices by setting numerical values

- Remind: matrices → graphs:

$$\mathcal{E}(A) = \{(i,j) \mid a_{ij} \neq 0,\ i \neq j\}\,.$$

- or for a digraph:

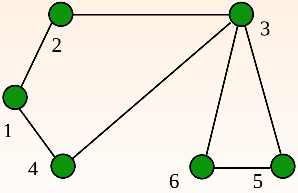
$$\mathcal{E}(A) = \{(i \to j) \mid a_{ij} \neq 0,\ i \neq j\}.$$

- Remind that diagonal entries are typically not included in the model.
- But also the opposite direction graphs → matrices is sometimes useful: adjacency and incidence matrices

Graphs and their matrices

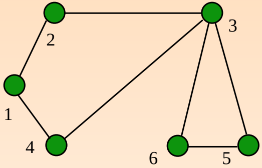
Adjacency and incidence matrices of an undirected graph

- For a simple undirected graph $\mathcal{G} = (V, E)$ with $V = \{1, \ldots, n\}$ the adjacency matrix (vertex by vertex) is the $(0, 1)$ matrix $A_G = (a_{ij})$ $(i, j \in V)$, where a_{ij} is 1 iff $i \leftrightarrow j$ and 0 otherwise.
- Similarly for such graph $\mathcal{G} = (V, E)$, $V = \{1, \ldots, n\}$, $E = \{1, \ldots, m\}$ the incidence matrix (edge by vertex) is the $(0, 1)$ matrix $B_{\mathcal{G}} = (b_{ij})$ $(i \in \{1, \ldots, m\}, j \in \{1, \ldots, n\})$, where $a_{ij} = 1$, if $(i, j) \in E$ and $a_{ij} = 0$ otherwise.



Graphs and their matrices

Adjacency and incidence matrices of an undirected graph



$$\begin{array}{c|cccccc} & 1 & 2 & 3 & 4 & 5 & 6 \\ \hline 1 & & 1 & & 1 & & \\ 2 & 1 & & 1 & & & \\ 3 & & 1 & & 1 & 1 & 1 \\ 4 & 1 & & 1 & & & \\ 5 & & & 1 & & & 1 \\ 6 & & & 1 & & 1 & \end{array}$$

$$\begin{array}{c|cccccc} & 1 & 2 & 3 & 4 & 5 & \\ \hline (1,2) & 1 & 1 & & & & \\ (1,4) & 1 & & & 1 & & \\ (2,3) & & 1 & 1 & & & \\ (3,4) & & & 1 & 1 & & \\ (3,5) & & & 1 & & 1 & \\ (3,6) & & & 1 & & & 1 \\ (5,6) & & & & & 1 & 1 \end{array}$$

- More ways to define matrix values (e.g., setting diagonal values, using -1)

Graphs and their matrices

Graphs of triangular matrices

- A special case is the directed graph associated with a triangular matrix. If L is a lower triangular matrix and U is an upper triangular matrix then the directed graphs $\mathcal{G}(L)$ and $\mathcal{G}(U)$ have edge sets

$$\mathcal{E}(L) = \{(i \to j) \,|\, l_{ij} \neq 0,\ i > j\}$$

$$\mathcal{E}(U) = \{(i \to j) \,|\, u_{ij} \neq 0,\ i < j\}$$

- It is sometimes convenient to use $\mathcal{G}(L^T)$ in which the direction of the edges is reversed

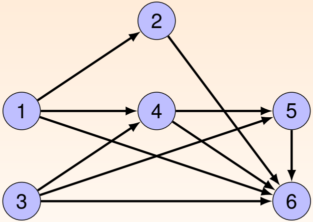
$$\mathcal{E}(L^T) = \{(j \to i) \,|\, l_{ij} \neq 0,\ i > j\}. \tag{8}$$

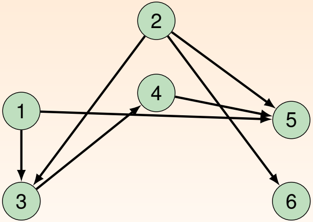
It is straightforward to see that $\mathcal{G}(L)$, $\mathcal{G}(L^T)$ and $\mathcal{G}(U)$ are DAGs.

Graphs and their matrices

LU dags

$$\begin{array}{c|cccccc} & 1 & 2 & 3 & 4 & 5 & 6 \\ \hline 1 & * & & * & & * & \\ 2 & * & * & * & & * & * \\ 3 & & & * & * & & \\ 4 & * & & * & * & * & \\ 5 & & & * & * & * & \\ 6 & * & * & * & * & * & * \end{array}$$





$\mathcal{G}(L^T)$ $\mathcal{G}(U)$

Graphs and their matrices

Permutation matrices

- A permutation matrix: a square matrix with exactly one entry equal to 1 in each row and column, and all remaining entries are zeros. That is, it is a permutation of the identity matrix.
- Premultiplying a matrix by P reorders the rows and postmultiplying by P reorders the columns. P can be represented by an integer-valued permutation vector p, where p_i is the column index of the 1 within the i-th row of P. For example,

$$P = \begin{pmatrix} 0 & 1 & 0 \\ 0 & 0 & 1 \\ 1 & 0 & 0 \end{pmatrix} \text{ and } p = \begin{pmatrix} 2 \\ 3 \\ 1 \end{pmatrix}.$$

Graphs and their matrices

Permutation matrices: labelling

- The graph of a matrix A is unchanged if a symmetric permutation $A' = PAP^T$ is performed, only the labelling of the vertices changes and thus relabelling $\mathcal{G}(A)$ can be used to permute A.
- The digraph of a general matrix A is not invariant under nonsymmetric permutations PAQ, with $Q \neq P^T$. Remind that a topological ordering of a directed graph is a labelling of its vertices such that for every edge $(i \to j)$, vertex i precedes vertex j (i.e., $i < j$).

Factorizations

Simple fill-in results

$$\begin{array}{cccccc} & k & & & j & \\ k & * & & & * & \\ & & * & & & \\ i & * & & * & & \\ & & & & * & \\ & & & & & * \end{array} \rightarrow \begin{array}{cccccc} & k & & & j & \\ k & * & & & * & \\ & & * & & & \\ i & * & & * & f & \\ & & & & * & \\ & & & & & * \end{array}$$

- Fill-in lemma

Lemma

Let $i, j, k \in \{1, \ldots, n\}, k < \min\{i, j\} \leq n$. *Then*

$$a_{ij}^{(k)} \neq 0 \Longleftrightarrow a_{ij}^{(k-1)} \neq 0 \vee (a_{ik}^{(k-1)} \neq 0 \wedge a_{kj}^{(k-1)} \neq 0)$$

Factorizations

Simple fill-in results

Lemma

Let $i, j, k \in \{1, \ldots, n\}, k < \min\{i, j\} \leq n$. *Then*

$$a_{ij}^{(k)} \neq 0 \iff a_{ij}^{(k-1)} \neq 0 \vee (a_{ik}^{(k-1)} \neq 0 \wedge a_{kj}^{(k-1)} \neq 0)$$

Proof.

Trivial for diagonal entries. If $a_{ij}^{(k)} \neq 0$ and also $a_{ij}^{(k-1)} = 0$, then the value at (i, j) was changed in the k-th factorization step. That is, when getting $A^{(k-1)}$ from $A^{(k-1)}$. This implies both $a_{ik}^{(k-1)} \neq 0$ and $a_{kj}^{(k-1)} \neq 0$, since the factorization update is

$$a_{ij}^{(k)} = a_{ij}^{(k-1)} - l_{ik} a_{kj}^{(k-1)} \equiv a_{ij}^{(k-1)} - a_{ik}^{(k-1)} a_{kj}^{(k-1)} / a_{kk}^{(k-1)}.$$

Conversely, the right-hand side of the lemma expression implies $a_{ij}^{(k)}$. □

Factorizations

Describing fill-in during the factorization

- Elimination (factorization) from the structural point of view:
- The elimination sequence

$$A^{(1)} \to A^{(2)} = C_1 A^{(1)} \to A^{(3)} = C_2 C_1 A^{(1)} \to \ldots \to A^{(n)} = C_{n-1} \ldots C_1 A^{(1)}.$$

- Remind the Schur complement

$$\begin{aligned} S^{(k)} &= \begin{pmatrix} a_{kk} & \ldots & a_{kn} \\ \vdots & \ddots & \vdots \\ a_{nk} & \ldots & a_{nn} \end{pmatrix} - \sum_{j=1}^{k-1} \begin{pmatrix} l_{kj} \\ \vdots \\ l_{nj} \end{pmatrix} \begin{pmatrix} u_{jk} & \ldots & u_{jn} \end{pmatrix} \\ &= \begin{pmatrix} a_{kk}^{(k)} & \ldots & a_{kn}^{(k)} \\ \vdots & \ddots & \vdots \\ a_{nk}^{(k)} & \ldots & a_{nn}^{(k)} \end{pmatrix} = A_{k:n,k:n}^{(k)}. \end{aligned} \tag{9}$$

- Factorization changes can be modelled using sequence of Schur complements where the changes are clearly visible

$$S^{(1)} \to S^{(2)} \to S^{(3)} \to \ldots \to S^{(n)} = a_{nn}^{(n-1)}.$$

- The quantities in the factors can be obtained in a different relative order in different algorithms.

Factorizations

Fill-in during the factorization: elimination graphs

- Graphs of the Schur complements are called the elimination graphs: $\mathcal{G}^1 \equiv \mathcal{G}(A), \mathcal{G}^2, \ldots, \mathcal{G}^n$.
- Vertices $\mathcal{V}^k$ of $\mathcal{G}^k = (\mathcal{V}^k, \mathcal{E}^k)$ are the $n-k+1$ uneliminated vertices (correspond to the unused rows and columns $k, \ldots, n$.
- The edge set $\mathcal{E}^k$ contains the edges of $\mathcal{G}(A)$ and fill-in edges among vertices of $\mathcal{V}^k$.
- The process to generate the elimination graphs is called the Parter's rule:

  To obtain the elimination graph $\mathcal{G}^{k+1}$ *from* $\mathcal{G}^k$*, delete vertex* k *and add all edges* $(i \xrightarrow{\mathcal{G}^{k+1}} j)$ *such that* $(i \xrightarrow{\mathcal{G}^k} k)$ *and* $(k \xrightarrow{\mathcal{G}^k} j)$*.*

Factorizations

- An example

$$\begin{array}{cc} & \begin{matrix} 1 & 2 & 3 & 4 & 5 & 6 \end{matrix} \\ \begin{matrix} 1 \\ 2 \\ 3 \\ 4 \\ 5 \\ 6 \end{matrix} & \begin{pmatrix} \ast & & \ast & & & \\ \ast & \ast & & \ast & & \ast \\ \ast & & \ast & & & \ast \\ \ast & \ast & & \ast & \ast & \\ & & & \ast & \ast & \\ & \ast & \ast & & & \ast \end{pmatrix} \end{array} \qquad \begin{array}{cc} & \begin{matrix} 1 & 2 & 3 & 4 & 5 & 6 \end{matrix} \\ \begin{matrix} 1 \\ 2 \\ 3 \\ 4 \\ 5 \\ 6 \end{matrix} & \begin{pmatrix} \ast & & \ast & & & \\ \ast & \ast & \color{red}{f} & \ast & & \ast \\ \ast & & \ast & & & \ast \\ \ast & \ast & \color{red}{f} & \ast & \ast & \\ & & & \ast & \ast & \\ & \ast & \ast & & & \ast \end{pmatrix} \end{array}$$

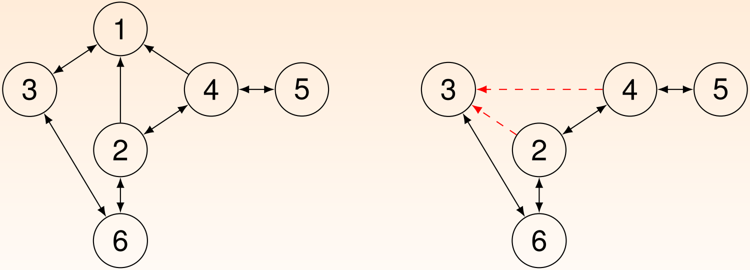


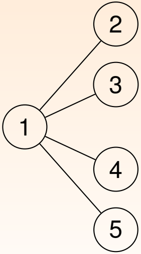
Figure: *The original digraph $\mathcal{G} = \mathcal{G}^1$ (left) and the directed elimination graph $\mathcal{G}^2$ (right) The red dashed lines denote fill edges.*

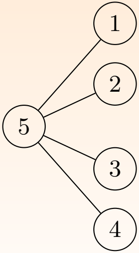
Factorizations

Graphs of arrowhead matrices

- Used here to demonstrate a catastrophic fill-in.

$$\begin{array}{c|ccccc} & 1 & 2 & 3 & 4 & 5 \\ \hline 1 & * & * & * & * & * \\ 2 & * & * & & & \\ 3 & * & & * & & \\ 4 & * & & & * & \\ 5 & * & & & & * \end{array} \qquad \begin{array}{c|ccccc} & 1 & 2 & 3 & 4 & 5 \\ \hline 1 & * & & & & * \\ 2 & & * & & & * \\ 3 & & & * & & * \\ 4 & & & & * & * \\ 5 & * & * & * & * & * \end{array}$$





- The second one is obtained from the first one by relabelling that corresponds to a symmetric permutation.

Factorizations

Parter's rule more formally

- Denoting $\mathcal{G}^k = (\mathcal{V}^k, \mathcal{E}^k)$, $\mathcal{G}^{k+1} = (\mathcal{V}^{k+1}, \mathcal{E}^{k+1})$, the Parter's rule is

$$\mathcal{V}^{k+1} = \mathcal{V}^k \setminus \{k\},\ \mathcal{E}^{k+1} = \mathcal{E}^k \cup \{(i,j)\,|\,i,j \in adj_{\mathcal{G}^k}\{k\}\} \setminus \{(i,k)\,|\,i \in adj_{\mathcal{G}^k}\{k\}\}.$$

- In the symmetric case: no need to consider orientation: To obtain the elimination graph $\mathcal{G}^{k+1}$ from $\mathcal{G}^k$, delete vertex k and add all possible edges between vertices that are adjacent to vertex k in $\mathcal{G}^k$.
- If $\mathcal{S}\{A\}$ is symmetric then Parter's rule says that the adjacency set of vertex k becomes a clique when k is eliminated: Gaussian elimination systematically generates cliques.

Factorizations

- An example where A is assumed to be symmetric, factorized by Cholesky or its extension.

$$\begin{array}{c|cccccc} & 1 & 2 & 3 & 4 & 5 & 6 \\ \hline 1 & * & * & * & * & & \\ 2 & * & * & & * & & * \\ 3 & * & & * & & & * \\ 4 & * & * & & * & * & \\ 5 & & & & * & * & \\ 6 & & * & * & & & * \end{array} \qquad \begin{array}{c|cccccc} & 1 & 2 & 3 & 4 & 5 & 6 \\ \hline 1 & * & * & * & * & & \\ 2 & * & * & f & * & & * \\ 3 & * & f & * & f & & * \\ 4 & * & * & f & * & * & \\ 5 & & & & * & * & \\ 6 & & * & * & & & * \end{array}$$

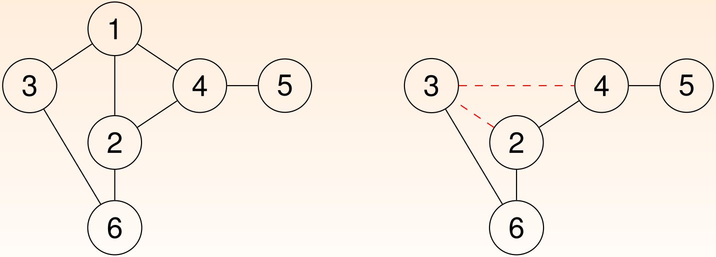


Figure: *The original undirected graph $\mathcal{G} = \mathcal{G}^1$ (left) and the obtained graph $\mathcal{G}^2$ (right). The red dashed lines denote fill edges. The vertices $\{2, 3, 4\}$ become a clique.*

Factorizations

Parter's rule for the final factors

- Remind the non-cancellation assumption: once created fill-in remains ☺
- Note that

$$a_{ij} \neq 0 \Leftrightarrow (i \to j) \in \mathcal{E}.$$

Lemma

Let $i, j, k \in \{1, \ldots, n\}, k < \min\{i, j\} \leq n$. *Denote* $F = L + U$. *Then*

$$(i, j) \in \mathcal{E}(F) \Longleftrightarrow (i, j) \in \mathcal{E}(A) \vee ((i, k) \in \mathcal{E}(F) \wedge (k, j) \in \mathcal{E}(F)),$$

where $\mathcal{G}(F) = (V, \mathcal{E}(F))$

- Note: this was just formulated for a static graph $\mathcal{G}(F)$.

Factorizations

Parter's rule for factors (both symmetric and nonsymmetric cases)

- Nonsymmetric matrix A and its LU factorization:
  $(i \to j)$ is an edge of the digraph $\mathcal{G}(L+U)$ if and only if $(i \to j)$ is an edge of the digraph $\mathcal{G}(A)$ or $(i \to k)$ and $(k \to j)$ are edges of $\mathcal{G}(L+U)$ for some $k < i, j$.
- Symmetric: the repeated application of Parter's rule specifies all the edges in $\mathcal{G}(L+L^T)$ as:
  (i, j) is an edge of $\mathcal{G}(L+L^T)$ if and only if (i, j) is an edge of $\mathcal{G}(A)$ or (i, k) and (k, j) are edges of $\mathcal{G}(L+L^T)$ for some $k < i, j$.

Factorizations

From the Parter's rule for factors to fill-paths

- Parter's rule is only a local rule that uses the dependency on nonzeros obtained in previous steps of the factorization. The following result fully characterizes the nonzero entries in the factors using only paths in $\mathcal{G}(A)$.

Theorem

(a) *Let $\mathcal{S}\{A\}$ be symmetric and $A = LL^T$. Then* $(L + L^T)_{ij} \neq 0$ *if and only if there is a fill-path* $i \overset{\mathcal{G}(A)}{\underset{min}{\Longleftrightarrow}} j$.

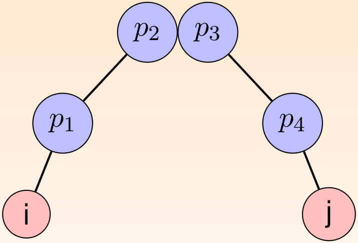
(b) *Let $\mathcal{S}\{A\}$ be nonsymmetric and $A = LU$. Then* $(L + U)_{ij} \neq 0$ *if and only if there is a fill-path* $i \xrightarrow[min]{\mathcal{G}(A)} j$.

The fill-paths may not be unique.

Factorizations

From the Parter's rule for factors to fill-paths

- Not a proof

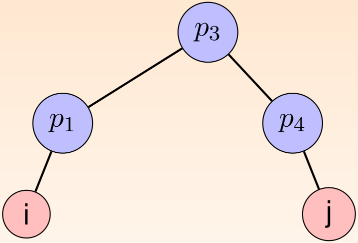


$$p_2 < p_3 < p_1 < p_4 < \min(i, j)$$

Factorizations

From the Parter's rule for factors to fill-paths

- Not a proof

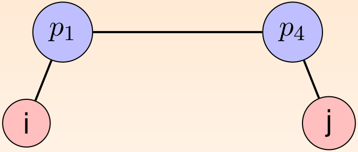


$$p_2 < p_3 < p_1 < p_4 < \min(i, j)$$

Factorizations

From the Parter's rule for factors to fill-paths

- Not a proof

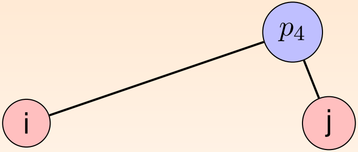


$$p_2 < p_3 < p_1 < p_4 < \min(i, j)$$

Factorizations

From the Parter's rule for factors to fill-paths

- Not a proof

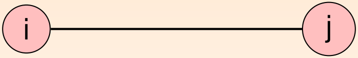


$$p_2 < p_3 < p_1 < p_4 < \min(i, j)$$

Factorizations

From the Parter's rule for factors to fill-paths

- Not a proof



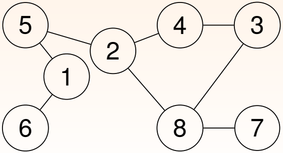
$$p_2 < p_3 < p_1 < p_4 < \min(i, j)$$

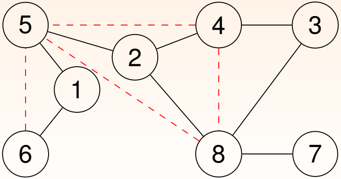
Factorizations

From the Parter's rule for factors to fill-paths

- Symmetric $\mathcal{S}\{A\}$: a filled entry in position $(8,6)$ of L because of a fill-path $8 \xLeftrightarrow[min]{\mathcal{G}(A)} 6$: $8 \longleftrightarrow 2 \longleftrightarrow 5 \longleftrightarrow 1 \longleftrightarrow 6$.

$$\begin{array}{c|cccccccc}
 & 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 \\
\hline
1 & * & & & & * & * & & \\
2 & & * & & * & * & & & * \\
3 & & & * & * & & & & * \\
4 & & * & * & * & & & & \\
5 & * & * & & & * & & & \\
6 & * & & & & & * & & \\
7 & & & & & & & * & * \\
8 & & * & * & & & & * & *
\end{array}
\qquad
\begin{array}{c|cccccccc}
 & 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 \\
\hline
1 & * & & & & * & * & & \\
2 & & * & & * & * & & & * \\
3 & & & * & * & & & & * \\
4 & & * & * & * & f & & & f \\
5 & * & * & & f & * & f & & f \\
6 & * & & & & f & * & & f \\
7 & & & & & & & * & * \\
8 & & * & * & f & f & f & * & *
\end{array}$$





Factorizations

So far, only implicit results to see the fill-in

- Neither the local characterization of filled entries using Parter's rule (fill-in lemma) nor Theorem on paths provide a direct answer as to whether a certain edge belongs to $\mathcal{G}(L+L^T)$ (or $\mathcal{G}(L+U)$) without performing the elimination
- Results presented so far do not tell us immediately whether a given entry of a factor of A is nonzero.
- And there are also practical problems connected to storing the elimination graphs and $\mathcal{G}(L+L^T)$ (or $\mathcal{G}(L+U)$).

Factorizations

Practical problems: storing and using the fill-in

- A clique with m vertices has $m(m-1)/2$ edges. This may be too many! It can be must be represented by storing a list of its vertices, without any reference to edges.
- This leads to implicit storing of the elimination graphs with significant consequences.
- As the elimination process progresses, cliques grow or more than one clique joins to form larger cliques: clique amalgamation.
- Note that in the nonsymmetric case we need to store edges orientation in addition to be happy from cliques.
- We need something more. Such questions should be addressed by a deeper theoretical and algorithmic understanding presented later: we will mention this later defining also another graph model for the elimination.

Outline

1. Introduction
2. Introductory notation and terminology
3. Factorizations
4. Graphs and sparse matrices
5. **Sparse matrices and data structures**
6. Symbolic factorizations
7. Symbolic Cholesky factorization
8. Cholesky factorization
9. Sparse LU factorization
10. Stability, ill-conditioning, indefiniteness
11. Blocks in the input matrix
12. Sparse Least Squares and QR factorization
13. Reorderings
14. Algebraic preconditioning
15. Incomplete factorizations
16. Sparse approximate inverses

Sparse vectors and matrices in a computer

Sparse vector in a computer

Example

Consider the sparse row vector $v \in \mathbb{R}^8$

$$v = \begin{pmatrix} 1. & -2. & 0. & -3. & 0. & 5. & 3. & 0. \end{pmatrix}. \tag{10}$$

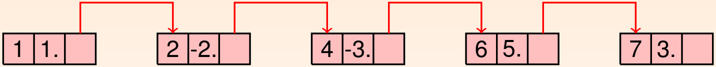
The real array `valV` that stores the nonzero values and corresponding integer array of their indices `indV` are of length $|\mathcal{S}\{v\}| = 5$ and are as follows:

Subscripts	1	2	3	4	5
`valV`	1.	$-2.$	$-3.$	5.	3.
`indV`	1	2	4	6	7

Sparse vectors and matrices in a computer

Sparse vector in a computer: linked lists

- Alternatively, a linked list can be used.
- Linked list - based format: stores matrix rows/columns as items connected by pointers
- Linked lists can be cyclic, one-way, two-way, etc.

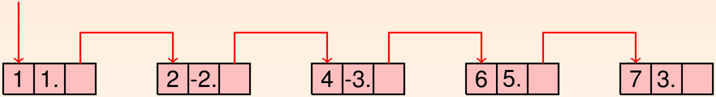


- rows/columns embedded into a larger array: emulated dynamic behavior

Sparse vectors and matrices in a computer

Sparse vector in a computer: linked lists

- Alternatively, a linked list can be used.
- Linked list - based format: stores matrix rows/columns as items connected by pointers
- Linked lists can be cyclic, one-way, two-way, etc.

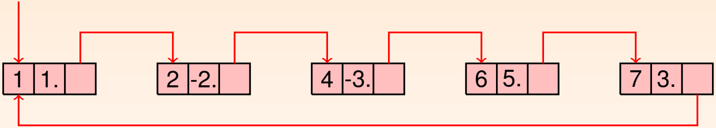


- rows/columns embedded into a larger array: emulated dynamic behavior

Sparse vectors and matrices in a computer

Sparse vector in a computer: linked lists

- Alternatively, a linked list can be used.
- Linked list - based format: stores matrix rows/columns as items connected by pointers
- Linked lists can be cyclic, one-way, two-way, etc.



- rows/columns embedded into a larger array: emulated dynamic behavior

Sparse vector

Sparse vector in a computer

- Linked list can be embedded into a large array.

Example

Possible ways of storing the sparse vector using linked lists.

Subscripts	1	2	3	4	5
Values	1.	$-2.$	$-3.$	5.	3.
Indices	1	2	4	6	7
Links	2	3	4	5	0
Header	1				

Subscripts	1	2	3	4	5
Values	5.	3.	1.	$-2.$	$-3.$
Indices	6	7	1	2	4
Links	2	0	4	5	1
Header	3				

Sparse vector in a computer: embedded linked list

- Reasons for linked lists: easy (dynamic) adds and removes.

Example

On the top, an entry -4 has been added in position 5. On the bottom, an entry -2 in position 2 has been removed. $*$ indicates the entry is not accessed. The links that have changed are in bold.

Subscripts	1	2	3	4	5	6
Values	1.	$-2.$	$-3.$	5.	3.	$-4.$
Indices	1	2	4	6	7	5
Links	2	3	4	5	6	0
Header	1					

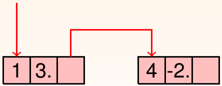
Subscripts	1	2	3	4	5
Values	1.	$*$	$-3.$	5.	3.
Indices	1	$*$	4	6	7
Links	3	$*$	4	5	0
Header	1				

Sparse vectors and matrices in a computer

Sparse matrix storage

- A set of linked lists. (Rather dynamic storage format)
- Example matrix $A \in \mathbb{R}^{5\times 5}$

$$\begin{array}{cc} & \begin{array}{ccccc} \;\;1\; & \;\;2\; & \;\;3\; & \;\;4\; & \;\;5\; \end{array} \\ \begin{array}{c} 1 \\ 2 \\ 3 \\ 4 \\ 5 \end{array} & \left(\begin{array}{ccccc} 3. & & & -2. & \\ & 1. & & & 4. \\ -1. & & 3. & & 1. \\ & & & 1. & \\ & 7. & & & 6. \end{array} \right) \end{array}.$$



Sparse vectors and matrices in a computer

Sparse matrix: linked lists: embedded in an array

- Easy dynamic matrix modifications if linked lists are used: example: A held as a collection of columns, each in a linked list. `colA_head` holds pointers to start a column list.

Example

Subscripts	1	2	3	4	5	6	7	8	9	10
`rowindA`	3	2	3	4	1	1	2	5	3	5
`valA`	3.	1.	-1.	1.	-2.	3.	4.	6.	1.	7.
`link`	0	10	0	0	4	3	9	0	8	0
`colA_head`	6	2	1	5	7					

If we consider column 4, then $\texttt{colA\_head}(4) = 5$, $\texttt{rowindA}(5) = 1$ and $\texttt{valA}(5) = -2.$, so the first entry in column 4 is $a_{1,4} = -2.$. Next, $\texttt{link}(5) = 4$, $\texttt{rowindA}(4) = 4$ and $\texttt{valA}(4) = 1.$, so the next entry in column 4 is $a_{4,4} = 1.$

Sparse vectors and matrices in a computer

Sparse matrix storage

- Coordinate (or triplet format: the individual entries of A are held as triplets (i, j, a_{ij}), where i is the row index and j is the column index of the entry $a_{ij} \neq 0$.
- This is also a dynamic storage format, often used also in other contexts (for files).

Sparse vectors and matrices in a computer

Sparse matrix in the coordinate format

- Example matrix $A \in \mathbb{R}^{5\times 5}$

$$\begin{array}{cc} & \begin{array}{ccccc} 1 & 2 & 3 & 4 & 5 \end{array} \\ \begin{array}{c} 1 \\ 2 \\ 3 \\ 4 \\ 5 \end{array} & \left(\begin{array}{ccccc} 3. & & & -2. & \\ & 1. & & & 4. \\ -1. & & 3. & & 1. \\ & & & 1. & \\ & 7. & & & 6. \end{array}\right) \end{array}. \tag{11}$$

Example

Subscripts	1	2	3	4	5	6	7	8	9	10
`rowindA`	3	2	3	4	1	1	2	5	3	5
`colindA`	3	2	1	4	4	1	5	5	5	2
`valA`	3.	1.	-1.	1.	-2.	3.	4.	6.	1.	7.

Sparse vectors and matrices in a computer

Sparse matrix storage: static formats

- CSR (Compressed Sparse Row) format. The column indices of the entries of A held by rows in an integer array (which we will call `colindA`) of length $nz(A)$, with those in row 1 followed by those in row 2, and so on (with no space between rows). Sorted or unsorted. (static storage format)
- CSC (Compressed Sparse Columns): analogously by columns instead of rows. (static storage format)
- If A is symmetric, only the lower (or upper) triangular part stored.
- Possible to store only $\mathcal{S}\{A\}$ and not numerical values.
- Very useful schemes: static, but theory helps to use them in applications

Sparse vectors and matrices in a computer

Sparse matrix in the CSR format

- CSR format represents A as follows. Here the entries within each row are in order of increasing column index.

Example

Subscripts	1	2	3	4	5	6	7	8	9	10
`rowptrA`	1	3	5	8	9	11				
`colindA`	1	4	2	5	1	3	5	4	2	5
`valA`	3.	-2.	1.	4.	-1.	3.	1.	1.	7.	6.

- Technical comment: In our codes often used: ia → rowptrA, ja → colindA, aa → valindA

Sparse vectors and matrices in a computer

Sparse matrix storage: static versus dynamic formats

- dynamic data structures:
  - – more flexible but this flexibility might not be needed
  - – difficult to vectorize
  - – difficult to keep spatial locality
  - – used preferably for storing vectors
- static data structures:
  - – ad-hoc insertions/deletions should be avoided (better algorithms)
  - – much simpler to vectorize / utilize cache
  - – efficient access to rows/columns
- level of dynamism interpreted for individual operations

Simulating dynamic storage formats by static ones

- A disadvantage of linked list storage: prohibits the fast access to rows (or columns) of the matrix. And this is needed!
- Simulated dynamism of storage schemes: storage format with some additional elbow space for new non zero entries of A is needed.
- Often the case in approximate factorizations where new non zero entries can be added and/or removed and it is hard to predict the necessary space in advance.
- In this case, the elbow space can embed new non zeros.
- The format is called the DS format.

Sparse vectors and matrices in a computer

Sparse matrix: DS formats

- Consider again the sparse matrix $A \in \mathbb{R}^{5\times 5}$ (11). The DS format represents A as follows.

Example

Subscripts	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
`rowptrA`	1	5	8	12	14										
`colindA`	1	4			2	5		1	3	5		4		2	5
`valAR`	3.	-2.			1.	4.		-1.	3.		1.		1.	7.	6.
`rowlength`	2	2	3	1	2										
`colptrA`	1	4	6	9	12										
`rowindA`	1	3		2	5	3			1	4		2	3	5	
`valAC`	3.	-1.		1.	7.	3.			-2.	1.		4.	1.	6.	
`collength`	2	2	1	2	3										

Sparse matrix: DS formats

- It can happen that the free space between row and/or column segments disappears throughout a computational algorithm. Then the DS format must be reorganized.
- In particular, a row segment can be moved to the end of the arrays `valAR` and `colindA` implying also a corresponding update in `rowptrA`. The space where the row i originally resided is then denoted as free.
- If there is no free space at the end of the arrays `valAR` and `colindA`, a compression of the row segments or full reallocation should be done.
- While the DS format seems to be complicated, it can be extremely useful in some cases. Surprisingly efficient if the amount of changes is limited as it often is in approximate factorizations.

Sparse matrices and data structures

Block formats

- Blocked formats may be used to accelerate multiplication between a sparse matrix and a dense vector.
- The Variable Block Row (VBR) format groups together similar adjacent rows and columns.
- The data structure of the VBR format uses six arrays. Integer arrays `rptr` and `cptr` hold the index of the first row in each block row and the index of the first column in each block column, respectively. In many cases, the block row and column partitionings are conformal and only one of these arrays is needed. The real array `valA` contains the entries of the matrix block-by-block in column-major order. The integer array `indx` holds pointers to the beginning of each block entry within `valA`. The index array `bindx` holds the block column indices of the block entries of the matrix and, finally, the integer array `bptr` holds pointers to the start of each row block in `bindx`.

Sparse matrices and data structures

Sparse matrix: DS formats

Example

Consider the sparse matrix $A \in \mathbb{R}^{8\times 8}$

$$\begin{array}{c} \\ 1\\2\\3\\4\\5\\6\\7\\8 \end{array}\begin{array}{c} \begin{array}{cccccccc} 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 \end{array} \\ \begin{pmatrix} 1. & 2. & & & & 3. & & \\ 4. & 5. & & & & 6. & & \\ & & 7. & 8. & 9. & 10. & & \\ 11. & 12. & & & & & 15. & 16. \\ & 13. & & & & & 17. & \\ 14. & & & & & & & 18. \\ & & 19. & & 20. & & & \\ & & 21. & 22. & & & & \end{pmatrix} \end{array}.$$

Here the row blocks comprise rows 1:2, 3, 4:6 and 7:8. The column blocks comprise columns 1:2, 3:5, 6, 7:8. The VBR format stores A as follows.

Sparse matrices and data structures

Sparse matrix: DS formats

Example

Subscripts	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22
`rptr`	1	3	4	7	9																	
`cptr`	1	3	6	7	9																	
`valA`	1.	4.	2.	5.	3.	6.	7.	8.	9.	10.	11.	14.	12.	13.	15.	17.	16.	18.	19.	21.	22.	20.
`indx`	1	5	7	10	11	15	19															
`bindx`	1	3	2	3	1	4	2															
`bptr`	1	3	5	7																		

Sparse matrices and data structures

Matrix-matrix multiplications (matmats) in CSR/CSC

(A by rows, B by columns)

$$A = \begin{pmatrix} A_{1,:} \\ \vdots \\ A_{m,:} \end{pmatrix} \in \mathbb{R}^{m \times k}, B = \big(B_{:,1}, \dots, B_{:,n}\big) \in \mathbb{R}^{k \times n}, C = AB = (c_{ij})_{ij}$$

- Each entry c_{ij}: product of a compressed row $A_{i,:}$ and compressed column $B_{:,j}$.
- Not known in advance whether $c_{ij} == 0$ or not.
- $m = k = n \Rightarrow O(n^3)$ operations, not useful for sparse matmats.

Sparse matrices and data structures

Matrix-matrix multiplications (matmats) in CSR/CSR

(A by rows, B by rows)

$$A = \begin{pmatrix} A_{1,:} \\ \vdots \\ A_{m,:} \end{pmatrix} \in \mathbb{R}^{m \times k}, B = \begin{pmatrix} B_{1,:} \\ \vdots \\ B_{n,:} \end{pmatrix} \in \mathbb{R}^{k \times n}, C = AB = (c_{ij})_{ij} \qquad (12)$$

$$\begin{array}{c|ccccc|}
 & 1 & 2 & 3 & 4 & 5 \\
1 & \ast & & & \ast & \\
2 & & \ast & & & \ast \\
3 & & & \ast & \ast & \\
4 & & & \ast & \ast & \\
5 & & \ast & & & \ast
\end{array}$$

A

$$\begin{array}{c|ccccccccc|}
 & 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 \\
1 & \ast & \ast & & & & & & \ast & \ast \\
2 & \ast & \ast & & & & & & \ast & \ast \\
3 & & & \ast & \ast & & & & \ast & \\
4 & & & \ast & \ast & & & \ast & \ast & \\
5 & & & & & \ast & \ast & \ast & &
\end{array}$$

B

Sparse matrices and data structures

Matrix-matrix multiplications (matmats) in CSR/CSR

(A by rows, B by rows)

$$A = \begin{pmatrix} A_{1,:} \\ \vdots \\ A_{m,:} \end{pmatrix} \in \mathbb{R}^{m\times k}, B = \begin{pmatrix} B_{1,:} \\ \vdots \\ B_{n,:} \end{pmatrix} \in \mathbb{R}^{k\times n}, C = AB = (c_{ij})_{ij} \qquad (13)$$

$$\begin{array}{cccccc} & 1 & 2 & 3 & 4 & 5 \\ 1 & \bigstar & & & \bigstar & \\ 2 & & * & & & * \\ 3 & & & * & * & \\ 4 & & & * & * & \\ 5 & & * & & & * \end{array}$$

A

$$\begin{array}{cccccccccc} & 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 \\ 1 & * & * & & & & & & * & * \\ 2 & * & * & & & & & & * & * \\ 3 & & & * & * & & & & * & \\ 4 & & & * & * & & & * & * & \\ 5 & & & & & * & * & * & & \end{array}$$

B

Sparse matrices and data structures

Matrix-matrix multiplications (matmats) in CSR/CSR

(A by rows, B by rows)

$$A = \begin{pmatrix} A_{1,:} \\ \vdots \\ A_{m,:} \end{pmatrix} \in \mathbb{R}^{m\times k}, B = \begin{pmatrix} B_{1,:} \\ \vdots \\ B_{n,:} \end{pmatrix} \in \mathbb{R}^{k\times n}, C = AB = (c_{ij})_{ij} \qquad (14)$$

$$\begin{array}{c|ccccc} & 1 & 2 & 3 & 4 & 5 \\ \hline 1 & \star & & & \star & \\ 2 & & * & & & * \\ 3 & & & * & * & \\ 4 & & & * & * & \\ 5 & & * & & & * \end{array}$$

A

$$\begin{array}{c|ccccccccc} & 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 \\ \hline 1 & \star & * & & & & & & \star & \star \\ 2 & * & * & & & & & & * & * \\ 3 & & & * & * & & & & * & \\ 4 & & & \star & \star & & & \star & \star & \\ 5 & & & & & * & * & * & & \end{array}$$

B

Sparse matrices and data structures

Matrix-matrix multiplications (matmats) in CSR/CSR

(A by rows, B by rows)

$$A = \begin{pmatrix} A_{1,:} \\ \vdots \\ A_{m,:} \end{pmatrix} \in \mathbb{R}^{m \times k}, B = \begin{pmatrix} B_{1,:} \\ \vdots \\ B_{n,:} \end{pmatrix} \in \mathbb{R}^{k \times n}, C = AB = (c_{ij})_{ij} \qquad (15)$$

$$\begin{array}{c|ccccc} & 1 & 2 & 3 & 4 & 5 \\ \hline 1 & * & & & * & \\ 2 & & * & & & * \\ 3 & & & * & * & \\ 4 & & & * & * & \\ 5 & & * & & & * \end{array}$$

A

$$\begin{array}{c|ccccccccc} & 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 \\ \hline 1 & \star & * & & & & & & \star & \star \\ 2 & * & * & & & & & & * & * \\ 3 & & & * & * & & & & * & \\ 4 & & & \star & \star & & & \star & \star & \\ 5 & & & & & * & * & * & & \end{array}$$

B

$\mathcal{S}(C_{1,:}) = \mathcal{S}(B_{1,:}) \cup \mathcal{S}(B_{4,:})$: optimal operation count

Sparse matrices and data structures

Matrix-matrix multiplications (matmats) in CSR/CSR

(A by columns, B by rows)

$$A = \left(A_{:,1}, \ldots, A_{:,k}\right) \in \mathbb{R}^{m \times k}, B = \begin{pmatrix} B_{1,:} \\ \vdots \\ B_{k,:} \end{pmatrix} \in \mathbb{R}^{k \times n}, C = AB = (c_{ij}) \tag{16}$$

- Simulation rows even when A is stored by columns.
- Assume that rows are in the order 1 → n

Sparse matrices and data structures

Matrix-matrix multiplications (matmats) in CSR/CSR

(A by columns, B by rows)

$$A = \left(A_{:,1}, \ldots, A_{:,k}\right) \in \mathbb{R}^{m \times k}, B = \begin{pmatrix} B_{1,:} \\ \vdots \\ B_{k,:} \end{pmatrix} \in \mathbb{R}^{k \times n}, C = AB = (c_{ij}) \tag{17}$$

- Simulation rows even when A is stored by columns.
- Assume that rows are in the order 1 → n
- More arrays needed.
  - Head pointers to first entries in columns are used
  - Entries of the same row (so far found): kept in a linked list
- Complexity of the multiplication is then linear: $O(nz(A)) + O(n)$

Sparse matrices and data structures

Matrix-matrix multiplications (matmats) in CSR/CSR

(A by columns, B by rows)

$$A = \left(A_{:,1}, \ldots, A_{:,k}\right) \in \mathbb{R}^{m\times k}, B = \begin{pmatrix} B_{1,:} \\ \vdots \\ B_{k,:} \end{pmatrix} \in \mathbb{R}^{k\times n}, C = AB = (c_{ij}) \qquad (18)$$

- A before the multiplication. Nonzeros are depicted in red.

●	○	○	○
○	●	●	○
●	○	○	●
●	●	○	●

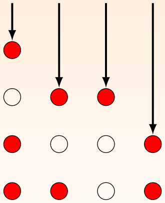
Sparse matrices and data structures

Matrix-matrix multiplications (matmats) in CSR/CSR

(A by columns, B by rows)

$$A = \left(A_{:,1}, \ldots, A_{:,k}\right) \in \mathbb{R}^{m \times k}, B = \begin{pmatrix} B_{1,:} \\ \vdots \\ B_{k,:} \end{pmatrix} \in \mathbb{R}^{k \times n}, C = AB = (c_{ij}) \qquad (19)$$

- A pointers initialization. Pointers point to first entries in a column



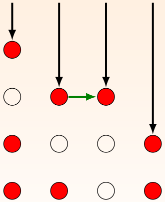
Sparse matrices and data structures

Matrix-matrix multiplications (matmats) in CSR/CSR

(A by columns, B by rows)

$$A = \left(A_{:,1}, \ldots, A_{:,k}\right) \in \mathbb{R}^{m \times k}, B = \begin{pmatrix} B_{1,:} \\ \vdots \\ B_{k,:} \end{pmatrix} \in \mathbb{R}^{k \times n}, C = AB = (c_{ij}) \qquad (20)$$

- Pointers pointing to the same row are connected by a linked list
- Potentially n linked lists, but all embedded into one array!!!



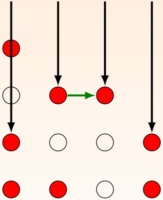
Sparse matrices and data structures

Matrix-matrix multiplications (matmats) in CSR/CSR

(A by columns, B by rows)

$$A = \left(A_{:,1}, \ldots, A_{:,k}\right) \in \mathbb{R}^{m\times k}, B = \begin{pmatrix} B_{1,:} \\ \vdots \\ B_{k,:} \end{pmatrix} \in \mathbb{R}^{k\times n}, C = AB = (c_{ij}) \qquad (21)$$

- After processing the first row (linked list for the first row contains only one entry):



- Process the second row as in the row - row scheme.

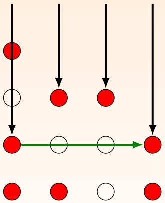
Sparse matrices and data structures

Matrix-matrix multiplications (matmats) in CSR/CSR

(A by columns, B by rows)

$$A = \left(A_{:,1}, \ldots, A_{:,k}\right) \in \mathbb{R}^{m \times k}, B = \begin{pmatrix} B_{1,:} \\ \vdots \\ B_{k,:} \end{pmatrix} \in \mathbb{R}^{k \times n}, C = AB = (c_{ij}) \qquad (22)$$

- After processing the first row (linked list for the first row contains only one entry):



- Update linked lists to be involved in further computations.

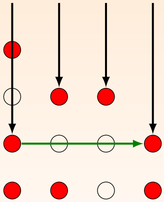
Sparse matrices and data structures

Matrix-matrix multiplications (matmats) in CSR/CSR

(A by columns, B by rows)

$$A = \left(A_{:,1}, \ldots, A_{:,k}\right) \in \mathbb{R}^{m\times k}, B = \begin{pmatrix} B_{1,:} \\ \vdots \\ B_{k,:} \end{pmatrix} \in \mathbb{R}^{k\times n}, C = AB = (c_{ij}) \qquad (23)$$

- A pointers initialization. Pointers (heads) point to first entries in a column



-
- And then: simply updates of the pointers and linked lists.
- Each relinking is cheap.

Outline

1. Introduction
2. Introductory notation and terminology
3. Factorizations
4. Graphs and sparse matrices
5. Sparse matrices and data structures
6. **Symbolic factorizations**
7. Symbolic Cholesky factorization
8. Cholesky factorization
9. Sparse LU factorization
10. Stability, ill-conditioning, indefiniteness
11. Blocks in the input matrix
12. Sparse Least Squares and QR factorization
13. Reorderings
14. Algebraic preconditioning
15. Incomplete factorizations
16. Sparse approximate inverses

Symbolic factorizations

Back to factorizations and sparsity

- First look: The matrix A is factorized and then, given the right-hand side b, the factors are used to compute the solution x.
- But, using graphs, even the factorization may be (in some cases) split:
  - 1) If care about numerical properties of A not needed (like in Cholesky): $\rightarrow$ factorization can be split into a structural part (symbolic phase) and the rest (numeric/factorization phase).
  - 2) If numerical issues have to be considered (like row interchanges in LU factorization); $\rightarrow$ some combination of symbolic and numerical factorizations in the factorization phase needed.

Symbolic factorizations

More on factorizability and factorization phases

- Pure symbolic phase typically uses only the sparsity pattern $\mathcal{S}\{A\}$ to compute the nonzero structure of the factors of A without computing numerical values of the nonzeros.
- Historically, the symbolic phase was much faster than the factorization phase. But parallelising the factorization $\rightarrow$ timings are much more closer. But, of course, if the phases are partially or fully combined, the whole factorization can be slow.
- Series of problems (like sequences of linear systems obtained from a nonlinear one) in which the numerical values of the entries of A change but $\mathcal{S}\{A\}$ does not (may not): symbolic phase performed just once.

Symbolic factorizations

More on factorizability and factorization phases

- The above comments on the factorization phases imply that we will concentrate to the Cholesky factorization and its symbolic phase, first
- Note that some of the construction may be used also in the cases of more general factorizations of symmetric matrices

Outline

1. Introduction
2. Introductory notation and terminology
3. Factorizations
4. Graphs and sparse matrices
5. Sparse matrices and data structures
6. Symbolic factorizations
7. Symbolic Cholesky factorization
8. Cholesky factorization
9. Sparse LU factorization
10. Stability, ill-conditioning, indefiniteness
11. Blocks in the input matrix
12. Sparse Least Squares and QR factorization
13. Reorderings
14. Algebraic preconditioning
15. Incomplete factorizations
16. Sparse approximate inverses

Symbolic Cholesky

Symbolic Cholesky

- Implicitly assumed that all diagonal entries of A are included in $\mathcal{S}\{A\}$ (even if they are zero – but if A is SPD, diagonal entries of A must be nonzero).

- A fundamental difference between dense and sparse Cholesky factorizations:
  If A is sparse, each column of L may depend numerically only on a subset of the previous columns.
- Symbolic dependence expressed by graphs may be even more straightforward: cheaper to obtain
- Furthermore, operations to update a computed column should be also sparse.
- We will see that the symbolic Cholesky can be clearly described using just a tree (forest) structure.

Symbolic Cholesky

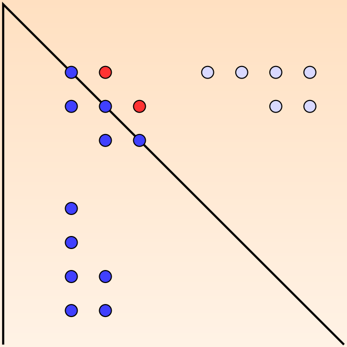
Column replication

- Let us start from the patterns of subsequent Schur complements.
- $$S^{(k)} = A_{k:n,k:n} - \sum_{j=1}^{k-1} \begin{pmatrix} l_{kj} \\ \vdots \\ l_{nj} \end{pmatrix} \begin{pmatrix} l_{kj} & \dots & l_{nj} \end{pmatrix}. \qquad (24)$$
- Example

$$
\begin{array}{c} \\ 2 \\ 3 \\ 4 \\ 5 \\ 6 \end{array}
\!\!\begin{array}{c} \begin{array}{ccccc} 2 & 3 & 4 & 5 & 6 \end{array} \\ \begin{pmatrix} \ast & & \ast & & \\ & \ast & \ast & {\color{red}\ast} & {\color{red}\ast} \\ \ast & \ast & \ast & & \ast \\ & {\color{red}\ast} & & \ast & {\color{red}\ast} \\ & {\color{red}\ast} & \ast & {\color{red}\ast} & \ast \end{pmatrix} \end{array}
=
\begin{array}{c} \\ 2 \\ 3 \\ 4 \\ 5 \\ 6 \end{array}
\!\!\begin{array}{c} \begin{array}{ccccc} 2 & 3 & 4 & 5 & 6 \end{array} \\ \begin{pmatrix} \ast & & \ast & & \\ & \ast & \ast & & \\ \ast & \ast & \ast & & \ast \\ & & & \ast & \\ & & \ast & & \ast \end{pmatrix} \end{array}
-
\begin{array}{c} \\ 2 \\ 3 \\ 4 \\ 5 \\ 6 \end{array}
\!\!\begin{array}{c} 2 \\ \begin{pmatrix} \\ \ast \\ \\ \ast \\ \ast \end{pmatrix} \end{array}
\begin{array}{cc} & \begin{array}{ccccc} 2 & 3 & 4 & 5 & 6 \end{array} \\ 2 & \begin{pmatrix} & \ast & & \ast & \ast \end{pmatrix} \end{array}
$$

Symbolic Cholesky

Column replication: as a sequence

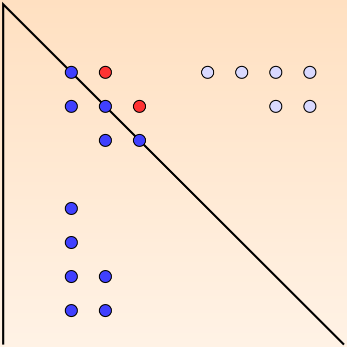


● Nonzero entries of the lower triangular part

○ Symmetric nonzeros values of L^T

Symbolic Cholesky

Column replication: as a sequence



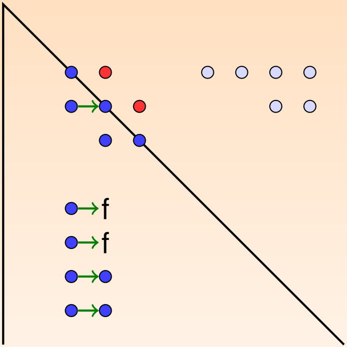
$$\begin{array}{c|cccccccc}
 & 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 \\
\hline
1 & * & * & & & & * & * & * \\
2 & * & * & * & & & & * & * \\
3 & & * & * & & & & & \\
4 & & & & * & & & & \\
5 & * & & & & * & & & \\
6 & * & & & & & * & & \\
7 & * & * & & & & & * & \\
8 & * & * & & & & & & *
\end{array}$$

● Nonzero entries of the lower triangular part

○ Symmetric nonzeros values of L^T

Symbolic Cholesky

Column replication: as a sequence



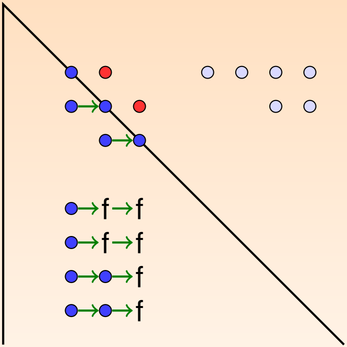
$$
\begin{array}{c|cccccccc}
 & 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 \\
\hline
1 & \ast & \ast & & & & \ast & \ast & \ast \\
2 & \ast & \ast & \ast & & f & f & \ast & \ast \\
3 & & \ast & \ast & & & & & \\
4 & & & & \ast & & & & \\
5 & \ast & f & & & \ast & & & \\
6 & \ast & f & & & & \ast & & \\
7 & \ast & \ast & & & & & \ast & \\
8 & \ast & \ast & & & & & & \ast
\end{array}
$$

● Nonzero entries of the lower triangular part

○ Symmetric nonzeros values of L^T

Symbolic Cholesky

Column replication: as a sequence



$$
\begin{array}{c|cccccccc}
 & 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 \\
\hline
1 & * & * & & & & * & * & * \\
2 & * & * & * & & f & f & * & * \\
3 & & * & * & & f & f & f & f \\
4 & & & & * & & & & \\
5 & * & f & f & & * & & & \\
6 & * & f & f & & & * & & \\
7 & * & * & f & & & & * & \\
8 & * & * & f & & & & & *
\end{array}
$$

● Nonzero entries of the lower triangular part

○ Symmetric nonzeros values of L^T

Symbolic Cholesky

Column replication: as a sequence

- Example: Consider $j = 1$. Because the first subdiagonal entry in column 1 is in row 3, first column updated by the column 1 is column 3.

$$
\begin{array}{c|ccccccc}
 & 1 & 2 & 3 & 4 & 5 & 6 & 7 \\
\hline
1 & * & & * & & & * & \\
2 & & * & & * & & * & \\
3 & * & & * & & & & \\
4 & & * & & * & & & \\
5 & & & * & & * & & \\
6 & * & * & & * & & * & \\
7 & * & & & & & * & *
\end{array}
\quad
\begin{array}{c|ccccccc}
 & 1 & 2 & 3 & 4 & 5 & 6 & 7 \\
\hline
1 & * & & * & & & * & \\
2 & & * & & * & & * & \\
3 & * & & * & & * & f & f \\
4 & & * & & * & & & \\
5 & & & * & & * & & \\
6 & * & * & f & * & & * & \\
7 & * & & f & & & * & *
\end{array}
\quad
\begin{array}{c|ccccccc}
 & 1 & 2 & 3 & 4 & 5 & 6 & 7 \\
\hline
1 & * & & * & & & * & \\
2 & & * & & * & & * & \\
3 & * & & * & & * & f & f \\
4 & & * & & * & & & \\
5 & & & * & & * & & \\
6 & * & * & f & * & f & * & \\
7 & * & & f & & f & * & *
\end{array}
$$

Figure: *Three steps: On the left are the entries in L before step 1 of a Cholesky factorization (that is, the entries in the lower triangular part of A); in the centre: the replication of the nonzeros from column 1 in the pattern of column 3. (red entries f); on the right: the subsequent replication in column 5.*

Symbolic Cholesky

Column replication formally

- Consider replications of nonzeros one by one
- Consider column j of L $(1 \leq j \leq k-1)$ and let $l_{ij} \neq 0$ for some $i > j$. We have this observation.

Observation

For any $i > j \geq 1$ *such that* $l_{ij} \neq 0$

$$\mathcal{S}\{L_{i:n,j}\} \subseteq \mathcal{S}\{L_{i:n,i}\}. \tag{25}$$

This is called the column replication principle because the pattern of column j *of* L *(rows* i *to* n*) is replicated in the pattern of column* i *of* L*.*

Symbolic Cholesky

Existence of nonzeros in columns

The following result shows that, provided A is irreducible, we always have a subdiagonal nonzero in column j, $j < n$.

Theorem

If A is SPD and irreducible then in each column j $(1 \leq j < n)$ of its Cholesky factor L there exists an entry $l_{ij} \neq 0$ with $i > j$.

Proof.

From Parter's rule, each step of the Cholesky factorization corresponds to adding new edges into the graph of the corresponding Schur complement. If A is irreducible then the graphs corresponding to the Schur complements are connected. Consequently, for any vertex j $(1 \leq j < n)$ in any of these graphs there is at least one vertex i with $i > j$ to which j is connected. This corresponds to the nonzero entry in column j of L.
□ □

Symbolic Cholesky

Column replication: final simplification

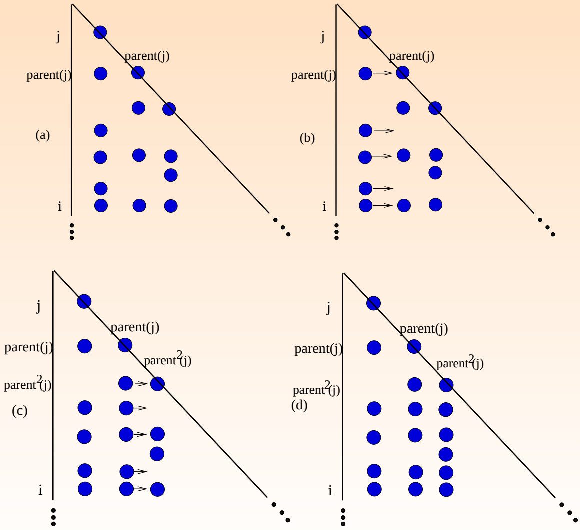
- The first subdiagonal entry plays a special role
- Denote the row index of the first subdiagonal nonzero entry in column j of L by $parent(j)$, that is,

$$parent(j) = \min\{i \mid i > j \text{ and } l_{ij} \neq 0\}. \tag{26}$$

- If there is no such entry, set $parent(j) = 0$ (For A reducible or $j = n$).
- The row index $parent(parent(j))$ is denoted by $parent^2(j)$, and so on.
- Applying the column replication recursively we get that the sparsity pattern of column j of L is replicated in that of column $parent(j)$, which in turn is replicated in the pattern of column $parent^2(j)$, and so on.

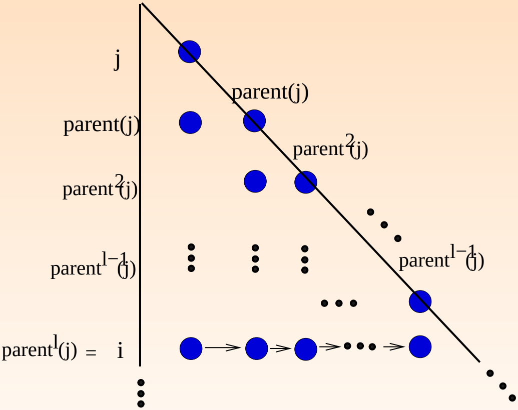
Symbolic Cholesky

Fill-in as replication of column structures



Symbolic Cholesky

Row replication: example



- Replication of column structures → replication of nonzeros in a row.

Symbolic Cholesky

Row replication: formally

With the convention $parent^1(j) = parent(j)$, the next theorem shows that if entry l_{ij} of L is nonzero then $parent^t(j) = i$ for some $t \geq 1$ and there is row replication in the sequence
$j,\ parent^1(j),\ parent^2(j),\ \ldots, parent^t(j)$.

Theorem

Let A be SPD and let L be its Cholesky factor. If $l_{ij} \neq 0$ for some $j < i \leq n$ then there exists $t \geq 1$ such that $parent^t(j) = i$ and $l_{ik} \neq 0$ for $k = j$, $parent^1(j)$, $parent^2(j),\ \ldots, parent^t(j)$.

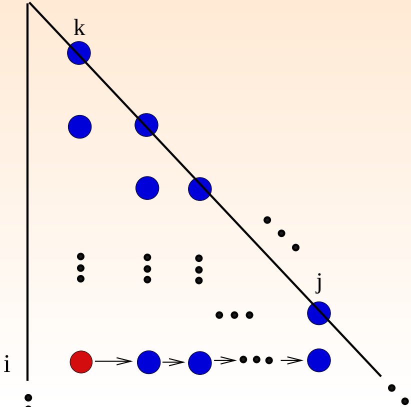
Proof.

If $i = parent^1(j)$, the result is immediate. Otherwise, there exists an index k, $j < k < i$ of a subdiagonal entry in column j of L such that $k = parent^1(j)$. Column replication implies $l_{ik} \neq 0$. Applying an inductive argument to l_{ik}, the result follows after a finite number of steps. □ □

Symbolic Cholesky

Necessary and sufficient condition for a fill-in entry

- If there is sequence of nonzeros in a row of L, it is natural to ask where the sequence begins:
- If there is no $k \geq 1$ such that $a_{ik} \neq 0$, no replication of nonzeros can start in row i.
- Consequently, there should be always a starting nonzero in a row.



Symbolic Cholesky

Necessary and sufficient condition for a fill-in entry: formally

Theorem

Let A be SPD and let L be its Cholesky factor. If $a_{ij} = 0$ for some $1 \leq j < i \leq n$ then there is a filled entry $l_{ij} \neq 0$ if and only if there exists $k < j$ and $t \geq 1$ such that $a_{ik} \neq 0$ and $parent^t(k) = j$.

Elimination tree

- The discussion of column replication is significantly simplified using elimination trees.
- The elimination tree (or etree) $\mathcal{T}(A)$ (or simply $\mathcal{T}$) of a SPD matrix has vertices $1, 2, \ldots, n$ and an edge between each pair $(j, parent(j))$, where $parent(j)$ has been defined above;
- j is a root vertex of the tree if $parent(j) = 0$.
- The edges of $\mathcal{T}$ are considered to be directed from a child to its parent, that is,

  $$\mathcal{E}(\mathcal{T}) = \{(j \longrightarrow i) \mid i = parent(j)\}.$$

  If $\mathcal{T}$ has a single component then the root vertex is n.
- Despite the terminology, the elimination tree (in our definition) need not be connected and in general is a forest. For simplicity, in our discussions, we (tacitly) assume $\mathcal{T}$ has a single component and we say that $\mathcal{T}$ is described by the vector $parent$.

Symbolic Cholesky

Elimination tree

$$
\begin{array}{c|cccccccc}
 & 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 \\
\hline
1 & * & & & & * & * & & \\
2 & & * & & * & * & & & * \\
3 & & & * & * & & & & * \\
4 & & * & * & * & f & & & f \\
5 & * & * & & f & * & f & & f \\
6 & * & & & & f & * & & f \\
7 & & & & & & & * & * \\
8 & & * & * & f & f & f & * & *
\end{array}
$$

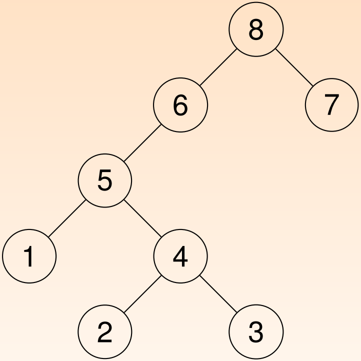


Figure: *An illustration of a sparse matrix A with a symmetric sparsity pattern and its elimination tree $\mathcal{T}(A)$. The root vertex is 8. The filled entries in $\mathcal{S}\{L + L^T\}$ are denoted by f.*

Symbolic Cholesky

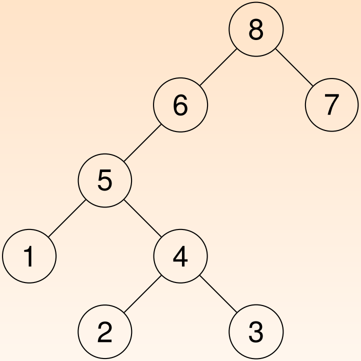
Elimination tree: terminology

- Concepts such as child, leaf, ancestor and descendant vertices for directed rooted trees can be applied to $\mathcal{T}$.
- Additionally, $anc_{\mathcal{T}}\{j\}$ and $desc_{\mathcal{T}}\{j\}$ are defined to be the sets of ancestors and descendants of vertex j in $\mathcal{T}$.
- $\mathcal{T}(j)$: the subtree of $\mathcal{T}$ induced by j and $desc_{\mathcal{T}}\{j)$; j is the root vertex of $\mathcal{T}(j)$.
- The size $|\mathcal{T}(j)|$ is the number of vertices in the subtree.
- A pruned subtree of $\mathcal{T}(j)$ is the connected subgraph induced by j and a subset of $desc_{\mathcal{T}}\{j)$. That is, for any vertex i in a pruned subtree of $\mathcal{T}(j)$, all the ancestors of i belong to $\mathcal{T}(j)$.
- A pruned subtree of $\mathcal{T}$ shares the mapping $parent$ with $\mathcal{T}$.

Symbolic Cholesky

Elimination tree: terminology

$$\begin{array}{ccccccccc}
 & 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 \\
1 & * &   &   &   & * & * &   &   \\
2 &   & * &   & * & * &   &   & * \\
3 &   &   & * & * &   &   &   & * \\
4 &   & * & * & * & f &   &   & f \\
5 & * & * &   & f & * & f &   & f \\
6 & * &   &   &   & f & * &   & f \\
7 &   &   &   &   &   &   & * & * \\
8 &   & * & * & f & f & f & * & *
\end{array}$$



- $\mathcal{T}(5)$ includes vertices $1, 2, 3, 4, 5$; $|\mathcal{T}(5)| = 5$; $anc_{\mathcal{T}}\{1\} = \{5, 6, 8\}$ etc.

Symbolic Cholesky

Elimination tree: simple properties

The following observation is straightforward (Follows from our definition).

Observation

If $i \in anc_{\mathcal{T}}\{j\}$ for some $j \neq i$ then $i > j$.

- The connection between the mapping $parent$ and the sets of ancestors and descendants can be emphasized as follows.

Observation

If i and j are vertices of the elimination tree $\mathcal{T}$ with $j < i \leq n$ then $i \in anc_{\mathcal{T}}\{j\}$ if and only if $j \in desc_{\mathcal{T}}\{i\}$ if and only if $parent^t(j) = i$ for some $t \geq 1$ (finding i going uptree from j)

Symbolic Cholesky

Elimination tree: simple properties

- Replications can be described using rooted forests (using elimination tree).
- For example, instead of stating that there exists $t \geq 1$ such that $parent^t(j) = i$, we can write that $i \in anc_{\mathcal{T}}\{j\} \setminus \{j\}$.
- Rewriting the necessary and sufficient condition in Theorem above as the following corollary we get a clear characterization of the sparsity patterns of the rows of L.

Corollary

Consider the elimination tree $\mathcal{T}$ and the Cholesky factor L of A. If i and j are vertices of $\mathcal{T}$ with $j < i \leq n$ and $a_{ij} = 0$ then $l_{ij} \neq 0$ if and only if there exists $k < j$ such that $j \in anc_{\mathcal{T}}(k)$ and $a_{ik} \neq 0$.

Symbolic Cholesky

Elimination tree: row subtrees

- The subtree of $\mathcal{T}$ with vertices that correspond to the nonzeros of row i of L is called the i-th row subtree and is denoted by $\mathcal{T}_r(i)$.
- Formally, row subtree is a pruned subtree of $\mathcal{T}$ induced by the union of the vertex set

  $$\{i\} \cup \{k \,|\, a_{ik} \neq 0 \text{ and } k < i\}$$

  with all vertices on the directed paths in $\mathcal{T}$ from k to i, that is, with all their ancestors from $\mathcal{T}_r(i)$.
- That is, the root vertex is i and the leaf vertices are a subset of the column indices in the i-th row of the lower triangular part of A.

Symbolic Cholesky

Elimination tree: row subtrees

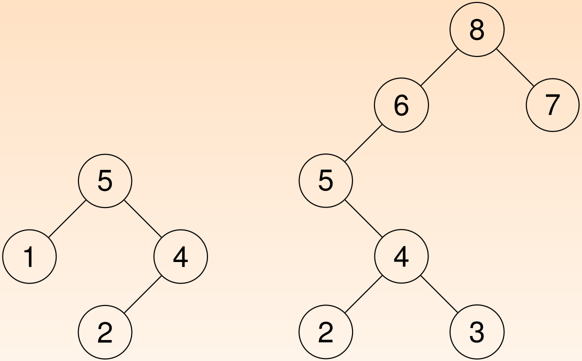
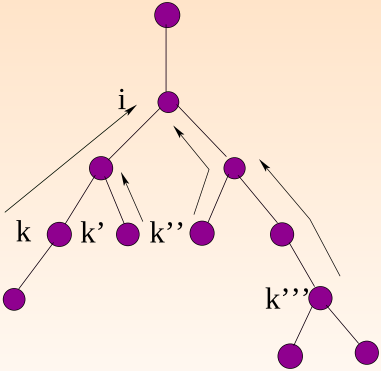


Figure: *The row subtree $\mathcal{T}_r(5)$ of the elimination tree $\mathcal{T}$ from above (left). Vertex 3 has been pruned because $a_{35} = 0$. The row subtree $\mathcal{T}_r(8)$ (right) differs from $\mathcal{T} = \mathcal{T}(A)$ because vertex 1 has been pruned ($a_{18} = 0$).*

Symbolic Cholesky

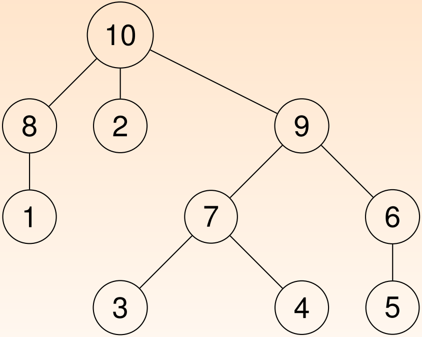
Elimination tree: row subtrees: another example



Symbolic Cholesky

Row subtrees: elimination tree

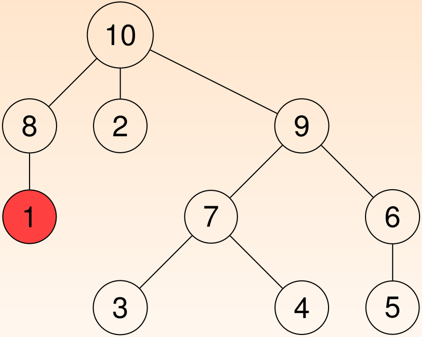
$$\begin{pmatrix}
* & & & & & & & * & & * \\
 & * & & & & & & & & * \\
 & & * & & & & * & & & * \\
 & & & * & & & * & & & \\
 & & & & * & * & & & * & \\
 & & & & * & * & & & f & * \\
 & & * & * & & & * & & * & f \\
* & & & & & & & * & & * \\
 & & & & * & f & * & & * & f \\
* & * & * & & & * & f & * & f & *
\end{pmatrix}$$



Symbolic Cholesky

Row subtrees: $\mathcal{T}_r(1)$

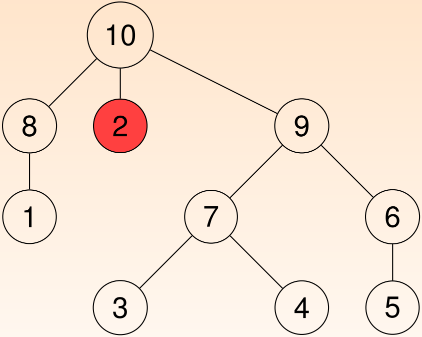
$$\begin{pmatrix}
* & & & & & & & * & & * \\
 & * & & & & & & & & * \\
 & & * & & & & * & & & * \\
 & & & * & & & * & & & \\
 & & & & * & * & & & * & \\
 & & & & * & * & & & f & * \\
 & & * & * & & & * & & * & f \\
* & & & & & & & * & & * \\
 & & & & * & f & * & & * & f \\
* & * & * & & & * & f & * & f & *
\end{pmatrix}$$



Symbolic Cholesky

Row subtrees: $\mathcal{T}_r(2)$

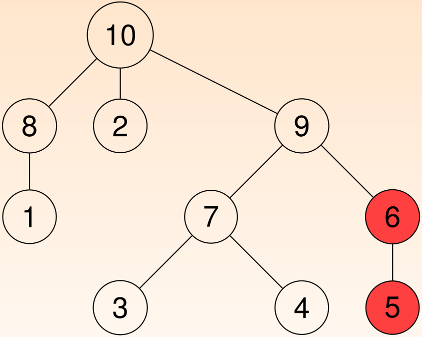
$$\begin{pmatrix}
* & & & & & & & * & & * \\
 & * & & & & & & & & * \\
 & & * & & & & * & & & * \\
 & & & * & & & * & & & \\
 & & & & * & * & & & * & \\
 & & & & * & * & & & f & * \\
 & & * & * & & & * & & * & f \\
* & & & & & & & * & & * \\
 & & & & * & f & * & & * & f \\
* & * & * & & & * & f & * & f & *
\end{pmatrix}$$



Symbolic Cholesky

Row subtrees: $\mathcal{T}_r(6)$

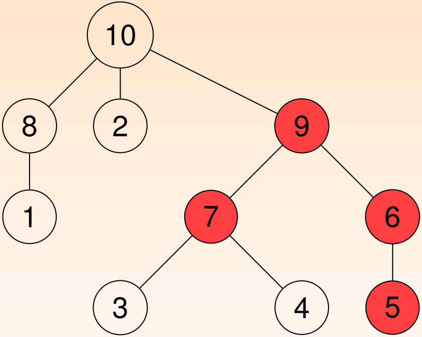
$$\begin{pmatrix}
* & & & & & & & * & & * \\
& * & & & & & & & & * \\
& & * & & & & * & & & * \\
& & & * & & & * & & & \\
& & & & * & * & & & * & \\
& & & & * & * & & & f & * \\
& & * & * & & & * & & * & f \\
* & & & & & & & * & & * \\
& & & & * & f & * & & * & f \\
* & * & * & & & * & f & * & f & *
\end{pmatrix}$$



Symbolic Cholesky

Row subtrees: $\mathcal{T}_r(9)$

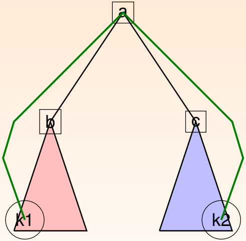
$$\begin{pmatrix}
* & & & & & & & * & & * \\
& * & & & & & & & & * \\
& & * & & & & * & & & * \\
& & & * & & & * & & & \\
& & & & * & * & & & * & \\
& & & & * & * & & & f & * \\
& & * & * & & & * & & * & f \\
* & & & & & & & * & & * \\
& & & & * & f & * & & * & f \\
* & * & * & & & * & f & * & f & *
\end{pmatrix}$$



Symbolic Cholesky

Elimination tree: row subtrees

- The row subtrees are connected subgraphs of $\mathcal{T}$, even if $\mathcal{T}$ is not connected.
- This is easy to see: for the i-th row subtree: from all vertices k that correspond to nonzeros in $A_{i,1:i-1}$ we put into T_r all vertices on the path to i and nothing more
- Formal proof: induction on the size of the subtrees.

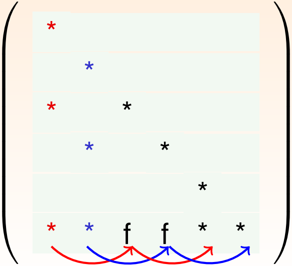


- green lines show connections in $\mathcal{G}(A)$ that must exist
- If $\mathcal{T}$ can be found without getting $\mathcal{S}(L)$ first $\Rightarrow$ $\mathcal{T}_r(i)$ can be used to get $|\mathcal{S}(L_{i,1:i-1})|$ without the need to have L or $\mathcal{S}(L)$ explicitly available.

Symbolic Cholesky

Elimination tree construction

- $\mathcal{T} = \mathcal{T}(A)$ can be constructed by stepwise extensions of the elimination trees of the principal leading submatrices of A.
- Assume we have $\mathcal{T}(A_{1:i-1,1:i-1})$ (does not need to be connected) and we want $\mathcal{T}(A_{1:i,1:i})$.
- No subdiagonal entries in row i of $A \Rightarrow$ nothing to do, only an isolated vertex i is added.
- Otherwise, i is the root of the nontrivial row subtree $\mathcal{T}_r(i)$ and an ancestor of some vertex $k, k \leq i-1$ in $\mathcal{T}$ with $a_{ik} \neq 0$.



$$\begin{pmatrix} * & & & & & \\ & * & & & & \\ * & & * & & & \\ & * & & * & & \\ & & & & * & \\ * & * & \text{f} & \text{f} & * & * \end{pmatrix}$$

Symbolic Cholesky

Elimination tree construction

- For each such vertex j of the i-th row, its ancestors k with $k < i$ are vertices of $\mathcal{T}(A_{1:i-1,1:i-1})$.
- Recall the row replication: there should be $t \geq 1$ and a directed path in $\mathcal{T}(A_{1:i,1:i})$ such that $parent^t(j) = i$.
- In the other words, $i \in anc_{\mathcal{T}}\{j\} \setminus \{j\}$.
- If $parent^t(j) \neq i$ then the new root of the subtree of $\mathcal{T}(A_{1:i,1:i})$ that contains j is either added by setting $i = parent^{t+1}(j)$. Or, i has already been added.

Symbolic Cholesky

Constructing elimination tree

Algorithm (Construction of the elimination tree)

```
for $i = 1 : n$ do
    $parent(i) = 0$
    for $j \in adj_{\mathcal{G}}\{i\}$ and $j < i$ do          ▷ For row $i$ of the lower triangular part
        $jroot = j$
        while $parent(jroot) \neq 0$ and $parent(jroot) \neq i$ do
            $jroot = parent(jroot)$
        end while
        if $parent(jroot) = 0$ then
            $parent(jroot) = i$                              ▷ Make $i$ the parent of $jroot$
        end if
    end for
end for
```

Symbolic Cholesky

Constructing elimination tree

$$\begin{pmatrix} \ast & & \ast & & \ast & \\ & \ast & & & \ast & \ast \\ \ast & & \ast & \ast & \ast & \\ & & \ast & \ast & & \ast \\ \ast & \ast & \ast & & \ast & \\ & \ast & & \ast & & \ast \end{pmatrix} \quad \begin{pmatrix} \ast & & \ast & & \ast & \\ & \ast & & & \ast & \ast \\ \ast & & \ast & \ast & \ast & \\ & & \ast & \ast & f & \ast \\ \ast & \ast & \ast & f & \ast & f \\ & \ast & & \ast & f & \ast \end{pmatrix}$$

Symbolic Cholesky

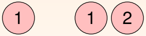
Constructing elimination tree

$$\begin{pmatrix} * & & * & & * & \\ & * & & & * & * \\ * & & * & * & * & \\ & & * & * & & * \\ * & * & * & & * & \\ & * & & * & & * \end{pmatrix} \quad \begin{pmatrix} * & & * & & * & \\ & * & & & * & * \\ * & & * & * & * & \\ & & * & * & f & * \\ * & * & * & f & * & f \\ & * & & * & f & * \end{pmatrix}$$

Symbolic Cholesky

Constructing elimination tree

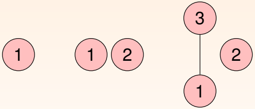
$$\begin{pmatrix} * & & * & & * & \\ & * & & & * & * \\ * & & * & * & * & \\ & & * & * & & * \\ * & * & * & & * & \\ & * & & * & & * \end{pmatrix} \quad \begin{pmatrix} * & & * & & * & \\ & * & & & * & * \\ * & & * & * & * & \\ & & * & * & f & * \\ * & * & * & f & * & f \\ & * & & * & f & * \end{pmatrix}$$



Symbolic Cholesky

Constructing elimination tree

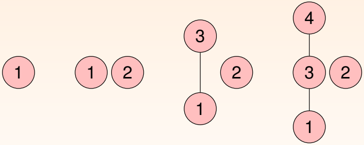
$$
\begin{pmatrix}
* & & * & & * & \\
& * & & & * & * \\
* & & * & * & * & \\
& & * & * & & * \\
* & * & * & & * & \\
& * & & * & & *
\end{pmatrix}
\quad
\begin{pmatrix}
* & & * & & * & \\
& * & & & * & * \\
* & & * & * & * & \\
& & * & * & f & * \\
* & * & * & f & * & f \\
& * & & * & f & *
\end{pmatrix}
$$



Symbolic Cholesky

Constructing elimination tree

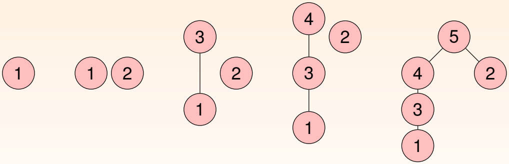
$$\begin{pmatrix} * & & * & & * & \\ & * & & & * & * \\ * & & * & * & * & \\ & & * & * & & * \\ * & * & * & & * & \\ & * & & * & & * \end{pmatrix} \quad \begin{pmatrix} * & & * & & * & \\ & * & & & * & * \\ * & & * & * & * & \\ & & * & * & {\color{red}f} & * \\ * & * & * & {\color{red}f} & * & {\color{red}f} \\ & * & & * & {\color{red}f} & * \end{pmatrix}$$



Symbolic Cholesky

Constructing elimination tree

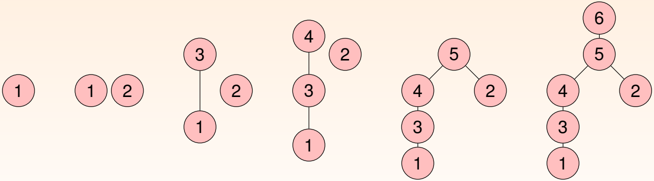
$$\begin{pmatrix} * & & * & & * & \\ & * & & & * & * \\ * & & * & * & * & \\ & & * & * & & * \\ * & * & * & & * & \\ & * & & * & & * \end{pmatrix} \qquad \begin{pmatrix} * & & * & & * & \\ & * & & & * & * \\ * & & * & * & * & \\ & & * & * & \color{red}{f} & * \\ * & * & * & \color{red}{f} & * & \color{red}{f} \\ & * & & * & \color{red}{f} & * \end{pmatrix}$$



Symbolic Cholesky

Constructing elimination tree

$$\begin{pmatrix} * & & * & & * & \\ & * & & & * & * \\ * & & * & * & * & \\ & & * & * & & * \\ * & * & * & & * & \\ & * & & * & & * \end{pmatrix} \qquad \begin{pmatrix} * & & * & & * & \\ & * & & & * & * \\ * & & * & * & * & \\ & & * & * & {\color{red}f} & * \\ * & * & * & {\color{red}f} & * & {\color{red}f} \\ & * & & * & {\color{red}f} & * \end{pmatrix}$$



Symbolic Cholesky

Constructing elimination tree: an example demonstrating a problem

$$\begin{pmatrix} * & * & * & * & * & * \\ * & * & & & & \\ * & & * & & & \\ * & & & * & & \\ * & & & & * & \\ * & & & & & * \end{pmatrix} \qquad \begin{pmatrix} * & * & * & * & * & * \\ * & * & f & f & f & f \\ * & f & * & f & f & f \\ * & f & f & * & f & f \\ * & f & f & f & * & f \\ * & f & f & f & f & * \end{pmatrix}$$

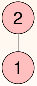
- For this example, $\mathcal{T}$ is determined by the mapping $parent(6) = 0$; $parent(i) = i + 1$ for $i = 1, \ldots, 5$.

Symbolic Cholesky

Constructing elimination tree: an example demonstrating a problem

$$\begin{pmatrix} * & * & * & * & * & * \\ * & * & & & & \\ * & & * & & & \\ * & & & * & & \\ * & & & & * & \\ * & & & & & * \end{pmatrix} \qquad \begin{pmatrix} * & * & * & * & * & * \\ * & * & f & f & f & f \\ * & f & * & f & f & f \\ * & f & f & * & f & f \\ * & f & f & f & * & f \\ * & f & f & f & f & * \end{pmatrix}$$

- For this example, $\mathcal{T}$ is determined by the mapping $parent(6) = 0$; $parent(i) = i + 1$ for $i = 1, \ldots, 5$.

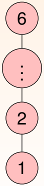


Symbolic Cholesky

Constructing elimination tree: an example demonstrating a problem

$$\begin{pmatrix} * & * & * & * & * & * \\ * & * & & & & \\ * & & * & & & \\ * & & & * & & \\ * & & & & * & \\ * & & & & & * \end{pmatrix} \qquad \begin{pmatrix} * & * & * & * & * & * \\ * & * & f & f & f & f \\ * & f & * & f & f & f \\ * & f & f & * & f & f \\ * & f & f & f & * & f \\ * & f & f & f & f & * \end{pmatrix}$$

- For this example, $\mathcal{T}$ is determined by the mapping $parent(6) = 0$; $parent(i) = i + 1$ for $i = 1, \ldots, 5$.



Symbolic Cholesky

Constructing elimination tree: improvement

- The most expensive part of Algorithm to find the elimination tree is the while loop that searches for subtree roots.
- This search is based on stepwise tracing the directed path from j to its root $parent^t(j)$.
- Because this path is unique for a given j, shortcuts can be incorporated; this is called path compression.
- Shortcuts: having a directed path from j to k, subsequent searches can be made more efficient by introducing a vector of shortcuts $ancestor$ and setting $ancestor(j) = k$.
- The modified algorithm is outlined below. It maintains both vectors $parent$ and $ancestor$. The tree described by $ancestor$ is termed the virtual tree.

Symbolic Cholesky

Algorithm (Construction of an elimination tree using path compression)

```
for $i = 1 : n$ do
    $parent(i) = 0$, $ancestor(i) = 0$
    for $j \in adj_{\mathcal{G}}\{i\}$ and $j < i$ do
        $jroot = j$
        while $ancestor(jroot)\ = 0$ and $ancestor(jroot)\ = i$ do
            $l = ancestor(jroot)$
            $ancestor(jroot) = i$        ▷ Shortcuts to accelerate future searches
            $jroot = l$
        end while
        if $ancestor(jroot) = 0$ then
            $ancestor(jroot) = i$ and $parent(jroot) = i$
        end if
    end for
end for
```

Symbolic Cholesky

Constructing elimination tree with compression

- The complexity of the first Algorithm is $O(n^2)$. This is clear from the example
- The complexity of the Algorithm with compression is $O(n)$ for A from the example..
- Formally, the complexity of the algorithm with compression is $O(nz(A)\log_2(n))$, where $nz(A)$ is the number of nonzeros of A but the logarithmic factor is rarely reached.

Symbolic Cholesky

Elimination tree construction: different way

- Can be constructed by exploiting the locality properties in another way.
- By subsequent search of paths in a subset of graph vertices.

Symbolic Cholesky

Locality: Interconnection of vertices in $\mathcal{G}(A)$

Theorem

If i and j are vertices in the elimination tree $\mathcal{T}$ with $j < i$ then $i \in anc_{\mathcal{T}}\{j\}$ if and only if there exists a path

$$j \overset{\mathcal{G}(A)}{\underset{\{1,\ldots,i\}}{\Longleftrightarrow}} i. \tag{27}$$

Proof.

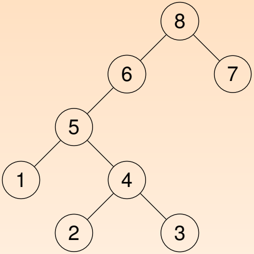
Sketch for ⇒: If such a path is, it is composed from edges of $\mathcal{T}$ that are composed from fill-paths. They correspond to the path in $\mathcal{G}(A)$.

Sketch for ⇐: If i and j are connected in $\mathcal{G}(A)$ by an edge: i is an ancestor of j. For longer paths, let m be the largest vertex on this path. If $m < j$ (note that $j < i$) is a fill-path connecting i and j and, therefore, $i \in anc_{\mathcal{T}}\{j\}$. Otherwise, for $m \geq j$, connection shorter paths: the assumption implies $i \in anc_{\mathcal{T}}\{m\}$ and $m \in anc_{\mathcal{T}}\{j\}$, that is, $i \in anc_{\mathcal{T}}\{j\}$. □

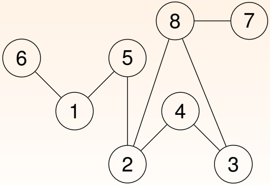
Symbolic Cholesky

Locality in characterization of ancestors: example

$$\begin{array}{c|cccccccc}
 & 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 \\
\hline
1 & * & & & & * & * & & \\
2 & & * & & * & * & & & * \\
3 & & & * & * & & & & * \\
4 & & * & * & * & f & & & f \\
5 & * & * & & f & * & f & & f \\
6 & * & & & & f & * & & f \\
7 & & & & & & & * & * \\
8 & & * & * & f & f & f & * & *
\end{array}$$



- $anc_{\mathcal{T}}\{4\}$ comprises vertices 5, 6 and 8. 7 is not in $anc_{\mathcal{T}}\{4\}$ since there is no path from 7 to 4 in $\mathcal{G}(A_{1:7,1:7})$.



Symbolic Cholesky

Locality in characterization of ancestors

- Given $j \in \mathcal{T}$, the corollary shows how can be found $parent(j)$ (if it exists).
- $anc_{\mathcal{T}}\{j\} \neq \emptyset \Rightarrow$ the lowest numbered one is $parent(j)$.

Corollary

Vertex i is the parent of vertex j in $\mathcal{T}$ if and only if i is the lowest numbered vertex satisfying $j < i \leq n$ connected by a path in $\mathcal{G}(A_{1:i,1:i})$.

- Existence of the path is equivalent to requiring i and j belong to the same component of the graph $\mathcal{G}(A_{1:i,1:i})$ corresponding to the $i \times i$ principal leading submatrix $A_{1:i,1:i}$ of A.
- Among $anc_{\mathcal{T}}\{4\}$ of previous slide vertex 5 is the smallest – it is the parent of 4.

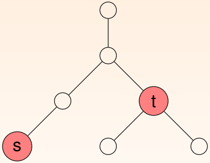
Symbolic Cholesky

Independence of subtrees: parallelism at hand

- There is no edge in $\mathcal{G}(L+L^T)$ between vertices belonging to subtrees of $\mathcal{T}$ with different vertex sets.

Theorem

Consider the elimination tree $\mathcal{T}$ and the Cholesky factor L of A. Let $\mathcal{T}(i)$ and $\mathcal{T}(j)$ be two vertex-disjoint subtrees of $\mathcal{T}$. Then for all $s \in \mathcal{T}(i)$ and $t \in \mathcal{T}(j)$ the entry l_{st} of L is zero.



- No proof: Of course, $l_{st} = 0$. Otherwise t would have to be ancestor or descendant of s.

Symbolic Cholesky

Sparsity pattern of L

- Elimination tree describes replications. Now we have to summarize nonzeros in L.
- Sometimes we need only the size $|L|$ to allocate memory, sometimes more: the whole structre of a row/column.
- Let $row_L\{i\}$: the sparsity pattern of the offdiagonals of row i of L, that is,

$$row_L\{i\} = \mathcal{S}\{L_{i,1:i-1}\} = \{j \mid j < i,\ l_{ij} \neq 0\}, \quad 1 \leq i \leq n.$$

- The number of entries in L is

$$nz(L) = \sum_{i=1}^{n} |row_L\{i\}| + n.$$

- Simple algorithm: $row_L\{i\}$ is given by the vertices of the row subtree $\mathcal{T}_r(i)$.

Symbolic Cholesky

Algorithm (Get row sparsity patterns of the Cholesky factor L)

```
for $i = 1 : n$ do                                   ▷ Loop over the rows of $A$
    $row_L\{i\} = \emptyset$                         ▷ Initialisation
    $mark(i) = i$
    for $k \in adj_{\mathcal{G}}\{i\}$ and $k < i$ do
        $j = k$
        while $mark(j) \neq i$ do                    ▷ Column $j$ not yet encountered in row $i$
            $mark(j) = i$                            ▷ Flag $j$ as encountered in row $i$
            $row_L\{i\} = row_L\{i\} \cup \{j\}$     ▷ Add $j$ to the sparsity pattern of row $i$
            $j = parent(j)$                          ▷ Move up the elimination tree
        end while
    end for
end for
```

Symbolic Cholesky

Cholesky factorization skeleton: quantitative improvement

- The complexity of our algorithm is $O(nz(L))$.
- Efficiency can be improved by employing the skeleton graph $\mathcal{G}(A^-)$ that is obtained from $\mathcal{G}(A)$ by removing every edge (i, j) for which $j < i$ and j is not a leaf vertex of $\mathcal{T}_r(i)$.
- $\mathcal{G}(A^-)$ is the smallest subgraph of $\mathcal{G}(A)$ with the same filled graph as $\mathcal{G}(A)$. The corresponding matrix is the skeleton matrix.

Symbolic Cholesky

Cholesky factorization skeleton

$$A = \begin{array}{c|ccccccc} & 1 & 2 & 3 & 4 & 5 & 6 & 7 \\ \hline 1 & \ast & & \ast & & & & \\ 2 & & \ast & \ast & & \ast & & \\ 3 & \ast & \ast & \ast & \ast & & \ast & \ast \\ 4 & & & \ast & \ast & & & \ast \\ 5 & & \ast & & & \ast & \ast & \ast \\ 6 & & & \ast & & \ast & \ast & \ast \\ 7 & & & \ast & \ast & \ast & \ast & \ast \end{array} \qquad A^{-} = \begin{array}{c|ccccccc} & 1 & 2 & 3 & 4 & 5 & 6 & 7 \\ \hline 1 & \ast & & \ast & & & & \\ 2 & & \ast & \ast & & \ast & & \\ 3 & \ast & \ast & \ast & \ast & & \ast & \ast \\ 4 & & & \ast & \ast & & & \\ 5 & & \ast & & & \ast & & \\ 6 & & & \ast & & & \ast & \\ 7 & & & \ast & & & & \ast \end{array}$$

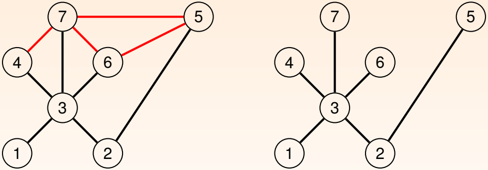


Figure: *Sparsity pattern of A and its graph $\mathcal{G}(A)$ (left) and the sparsity pattern of the skeleton matrix A^{-} and graph $\mathcal{G}(A^{-})$ (right). The entries in A and edges of $\mathcal{G}(A)$ that do not belong to the skeleton matrix and graph are depicted in red.*

Symbolic Cholesky

Cholesky factorization skeleton: quantitative improvement

- Because $nz(A^-)$ is often significantly smaller than $nz(A)$, an implementation that processes $\mathcal{G}(A^-)$ rather than $\mathcal{G}(A)$ can be substantially faster.
- The complexity of our algorithm is then still $O(nz(L))$, although with a quantitative improvement ☺

Symbolic Cholesky

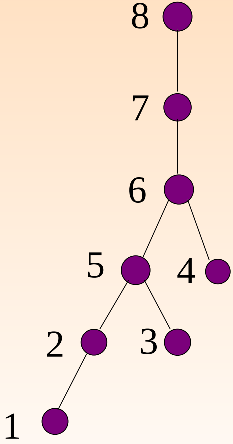
Are we satisfied? Let us summarize

- We are able to compute $row_L\{i\}$ and $|row_L\{i\}|$ for $i = 1, \ldots, n$.
- We know dependencies among columns since they are expressed by the elimination tree.
- But both submatrix and column algorithms expect, in principle, that column sparsity patterns are known ...

Not yet satisfied

Symbolic Cholesky

It would be nice to know column sparsity patterns of L as well



row structure

column structure

row subtrees

?

Symbolic Cholesky

Column sparsity patterns

- We should explicitly express the replication
- Let $col_L\{j\}$ denote the sparsity pattern of the off-diagonal part of column j of L, that is,

$$col_L\{j\} = \mathcal{S}(L_{j+1:n,j}) = \{i \mid i > j,\ l_{ij} \neq 0\}, \quad 1 \leq j \leq n.$$

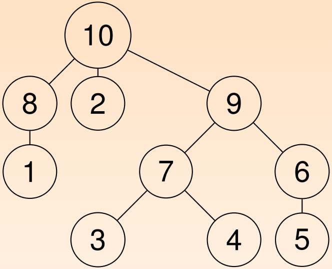
- The column replication principle can be written as

$$col_L\{j\} \subseteq col_L\{parent(j)\} \cup parent(j).$$

- And we can get column sparsity patterns using $\mathcal{T}(A)$ going up the tree: from the leaves to the root (s)

Symbolic Cholesky

Column sparsity patterns



- At the beginning, all nodes i correspond to some columns of A and have their patterns $col_A\{i\}$
- $col_L\{1\} = col_A\{1\}$
- Then, $col_L\{8\} = col_L\{1\} \cup col_A\{8\}$
- And so on, going up the tree, from the leaves to the root.

Symbolic Cholesky

Column structures

Standard sequential computation:

$$col_L\{j\} = \left(adj_{\mathcal{G}(A)}\{j\} \quad \bigcup_{\{k \,|\, k \in \mathcal{T}(j)\}} col_L\{k\} \right) \setminus \{1, \ldots, j\}.$$

Using the column replication, this can be significantly simplified

$$col_L\{j\} = \left(adj_{\mathcal{G}(A)}\{j\} \quad \bigcup_{\{k \,|\, j = parent(k)\}} col_L\{k\} \right) \setminus \{1, \ldots, j\}. \tag{28}$$

- Since any parent node contains the sparsity patterns of its sons.

Symbolic Cholesky

Column structures

- The following algorithm constructs the sparsity pattern of each column j of L as the union of the sparsity pattern of column j of A ($adj_{\mathcal{G}(A)}\{j\}$) and the patterns of the children of j in $\mathcal{T}(A)$.
- Here $child\{j\}$ denotes the set of children j.
- Because any child k of j satisfies $k < j$, the j-th outer step has the information needed to compute the sparsity pattern. Observe that $\mathcal{T}(A)$ does not need to be input.

Symbolic Cholesky

Algorithm (Determine sparsity patterns of columns of L)

```
for $j = 1 : n$ do
    $child\{j\} = \emptyset$                                                     ▷ Initialisation
end for
for $j = 1 : n$ do                                                  ▷ Loop over the columns of $L$
    $col_L\{j\} = adj_{\mathcal{G}(A)}\{j\} \ \setminus \ \{1, \ldots, j-1\}$
    for $k \in child\{j\}$ do                                          ▷ Unifying child structures
        $col_L\{j\} = col_L\{j\} \cup col_L\{k\} \ \setminus \ \{j\}$
    end for
    if $col_L\{j\} \neq \emptyset$ then
        $l = \min\{i \mid i \in col_L\{j\}\}$
        $child\{l\} = child\{l\} \cup \{j\}$                             ▷ Parent of $j$ detected
    end if
end for
```

Symbolic Cholesky

Column structures: can be also formalized

- Theorem describes $col_L\{j\}$ using the vertices of the subtree $\mathcal{T}(j)$.
- No proof: Of course, all their ancestors, nothing else can appear.

Theorem

The column sparsity pattern $col_L\{j\}$ of the Cholesky factor L of the matrix A is equal to the adjacency set of vertices of the subtree $\mathcal{T}(j)$ in $\mathcal{G}(A)$, that is,

$$col_L\{j\} = adj_{\mathcal{G}(A)}\{\mathcal{T}(j)\}. \tag{29}$$

Proof.

If $i \in col_L\{j\}$ then $j \in row_L\{i\}$ and Theorem on necessary and sufficient condition for the fill-in imply $j \in anc_{\mathcal{T}}\{k\}$ for some k such that $a_{ik} \neq 0$. That is, $i \in adj_{\mathcal{G}}\{\mathcal{T}(j)\}$. Conversely, $i \in adj_{\mathcal{G}}\{\mathcal{T}(j)\}$ implies that in row i the entry in column j of L is nonzero. Thus, $j \in row_L\{i\}$ and hence $i \in col_L\{j\}$. □ □

Symbolic Cholesky

Column structures: is that all?

- Of course, not: we can renumber the nodes.
- That is, the outer loop in the algorithm to find column sparsity patterns does not have to be performed strictly in the order $j = 1, \ldots, n$.
- Why we should be interested? There are some principal reasons:
  - Each union of patterns means some intermediate memory. And this can be minimized. This minimization is more important for computations with values, as we will see later.
  - Useful for much faster symbolic computations (to compute row structures, column structures etc.)
  - Specific orderings imply some important invariants in the computations.

Symbolic Cholesky

Topological orderings

- An ordering of the vertices in a tree (and, more generally, in a DAG) is a topological ordering if, for all i and j, $j \in desc_{\mathcal{T}}\{i\}$ implies $j < i$ (reminder)
- Observation above confirms that the ordering of vertices in the elimination tree $\mathcal{T}$ is a topological ordering.
- Then it is necessary only: for each step j, the column sparsity pattern for each child of j has already been computed. And nothing more.
- A new topological ordering of $\mathcal{T}$ defines a relabelling of its vertices corresponding to a symmetric permutation of A.

Symbolic Cholesky

Topological orderings

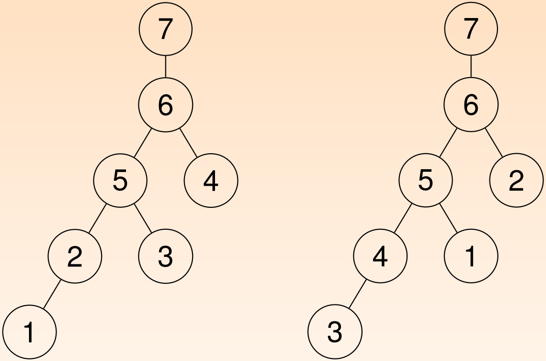


Figure: *Two topological orderings of an elimination tree.*

- Some of the column patterns need to wait longer before they are merged (related to the amount of the intermediate memory).

Symbolic Cholesky

Topological orderings

Sparsity patterns of the Cholesky factors of A and PAP^T can be different but the following result shows that the amount of fill-in is the same.

Theorem

Let $\mathcal{S}\{A\}$ be symmetric. If P is the permutation matrix corresponding to a topological reordering of the elimination tree $\mathcal{T}$ of A then the filled graphs of A and PAP^T are isomorphic.

- This implies the same fill-in.
- In the other words, the amount of fill-in is invariant under the class of topological reorderings of the elimination tree.

Symbolic Cholesky

Topological orderings and postorderings

- Let us recall: there are many topological orderings of $\mathcal{T}$.
- Specific subclass of topological orderings are postorderings
- A topological ordering of $\mathcal{T}$ is a postordering if the vertex set of any subtree $\mathcal{T}(i)$ ($i = 1, \ldots, n$) is a contiguous sublist of $1, \ldots, n$.
- Unless additional rules on how vertices are selected are imposed, a postordering is generally not unique.

Symbolic Cholesky

Postordering: example of two different postorderings

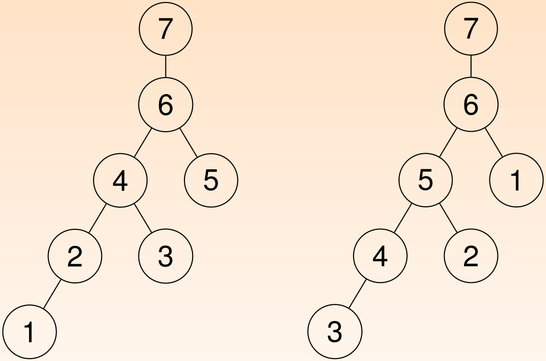


Figure: *An example to illustrate the non uniqueness of postorderings of an elimination tree.*

Symbolic Cholesky

Topological orderings and postorderings

- Postorderings can be obtained by the depth-first search (DFS). And DFS is apparently not unique. Various algorithms may prefer specific postorderings.
- DFS is applied to all components of $\mathcal{T}$ starting at their root vertices.
- Let us recall that in DFS, once vertex i has been visited, redT all the vertices of the subtree $\mathcal{T}(i)$ are visited immediately after i and i is labelled as the last vertex of $\mathcal{T}(i)$.

Symbolic Cholesky

Searching the adjacency graph $\mathcal{G}(A) = (\mathcal{V}, \mathcal{E})$

- The sequence in which the vertices are visited can be used to reorder the graph and hence permute the matrix.
- Given a start vertex, a graph search (graph traversal) performs exploration of the vertices and edges of $\mathcal{G}(A)$
- It generates sets of visited vertices and explored edges.
- $\mathcal{V}_v$: the set of visited vertices, $\mathcal{V}_n$: the set of vertices not yet visited.
- The search: selects an unexplored edge in $\mathcal{E}$ with vertices in $\mathcal{V}_v$. If the other vertex belongs to $\mathcal{V}_n$ then this vertex is moved into $\mathcal{V}_v$ and the edge is flagged as explored.
- The explored edge may be directed or undirected; in an undirected graph, the edge (v, w) formally corresponds to the pair of edges $(w \rightarrow v)$ and $(v \rightarrow w)$.

Depth-first search (DFS)

- A depth-first search (DFS) visits child vertices before visiting sibling vertices
- Starting from a chosen vertex s, the set of vertices that are visited are those vertices u for which a directed path from s to u exists in $\mathcal{G}$.
- Different results depending on s and how ties are broken.
- All vertices in $\mathcal{R}each(s)$ are visited.
- Traversed edges form a DFS spanning tree. Visiting all the edges of a graph results in a DFS forest that consists of exactly one DFS spanning tree for each connected component.

Symbolic Cholesky

Depth-first search (DFS)

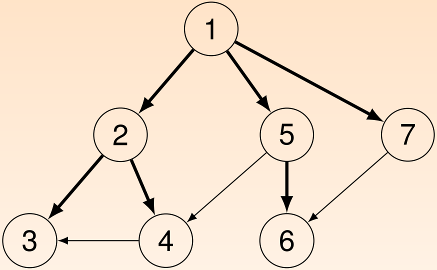
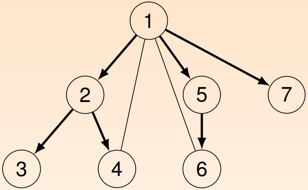


Figure: *An illustration of a DFS of a connected directed graph. The labels indicate the order in which the vertices are visited. The edges of the DFS spanning tree are in bold.*

Symbolic Cholesky

Depth-first search (DFS): Symbolic Cholesky connection

$$\begin{array}{c|ccccccc}
 & 1 & 2 & 3 & 4 & 5 & 6 & 7 \\
\hline
1 & * & * & & & * & & * \\
2 & * & * & * & * & & & \\
3 & & * & * & & & & \\
4 & & * & & * & & & \\
5 & * & & & & * & * & \\
6 & & & & & * & * & \\
7 & * & & & & & & *
\end{array}$$

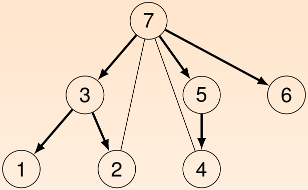


- Elimination tree of the permuted matrix can be obtained by the DFS applied to the $G(L^T)$ and reverting orientation of its edges. The permutation is given by the postordering generated by the DFS.

Symbolic Cholesky

Depth-first search (DFS): Symbolic Cholesky connection

$$
\begin{array}{c|ccccccc}
 & 1 & 2 & 3 & 4 & 5 & 6 & 7 \\
\hline
1 & * & & * & & & & \\
2 & & * & * & & & & * \\
3 & * & * & * & & & & * \\
4 & & & & * & * & & * \\
5 & & & & * & * & & * \\
6 & & & & & & * & * \\
7 & & * & * & * & * & * & *
\end{array}
$$



- Elimination tree of the permuted matrix can be obtained by the DFS applied to the $G(L^T)$ and reverting orientation of its edges. The permutation is given by the postordering generated by the DFS.

Symbolic Cholesky

Depth-first search (DFS)

- More ways to construct the output vertex order for a DFS.
- Given a start vertex s, in a preorder list, the vertices are returned in the order in which they are added into $\mathcal{V}_v$ (set of visited vertices)
- In a postorder list, the vertices are in the order in which they are last visited during the DFS algorithm. In Figure, the vertices are added into $\mathcal{V}_v$ in the order $1, 2, 3, 4, 5, 6, 7$ (preorder list).
- The sequence in which the DFS visits the vertices is $1, 2, 3, 2, 4, 2, 1, 5, 6, 5, 1, 7, 1$. Vertex 3 is the first vertex to appear for the last time $\Rightarrow$ the postordering starts with 3. The next vertex to appear for the last time is vertex 4, followed by 2, and so on, resulting in the postorder list $3, 4, 2, 6, 5, 7, 1$.
- For DFS is needed stack

Symbolic Cholesky

Stack and queue: two special lists

- Queue will be needed later.

Definition

List is called queue, if it enables efficient

- access head of a list,
- deletion of the list head (pop) and
- adding an element behind the current tail.

List is called stack, if it enables efficient

- access head of a list,
- deletion of the list head (pop) and
- adding and element before the current head (push).

Symbolic Cholesky

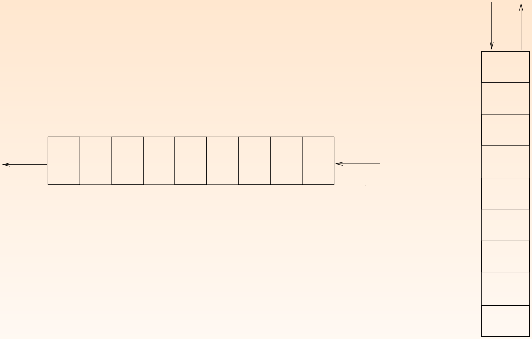
Lists

- A stack is a list in which elements can only be added to or removed from the head. A pointer locates the head of the stack. Let $S = (u_0, u_1, \ldots, u_{k-1}, u_k)$ be a stack. $push(S, v)$ denotes adding v onto the stack by incrementing the pointer by one, giving $(v, u_0, \ldots u_k)$. $pop(S, u_0)$ denotes the stack $(u_1, \ldots u_k)$ that results from decreasing the pointer by one (removing u_0 from the head).
- A queue is a list in which elements can be added to the tail (appended) or removed (popped) from the head. Two pointers locate the head and the tail. Consider the queue $\mathcal{Q} = (u_0, u_1, \ldots, u_{k-1}, u_k)$. The append operation $append(\mathcal{Q}, u_{k+1})$ results in the queue $(u_0, \ldots u_k, u_{k+1})$ and the pop operation $pop(\mathcal{Q}, u_0)$ results in the queue $(u_1, \ldots u_k)$.

Symbolic Cholesky

Stack and queue

Queue and stack are schematically depicted below. The arrows represent efficient (easily implementable) operations.



Symbolic Cholesky

Algorithm (Find preorder and postorder lists using a DFS)

```
$\mathcal{V}_v = \emptyset$, $preorder = ()$ and $postorder = ()$
for all $v \in \mathcal{V}$ do
    if $v \notin \mathcal{V}_v$ then
        $push(preorder, v)$                          ▷ Add $v$ onto the preorder stack
        $\mathcal{V}_v = \mathcal{V}_v \cup \{v\}$   ▷ Add $v$ to the set of visited vertices
        dfs_step($v$)
    end if
end for
recursive function (dfs_step($v$))
    for all $(v \to w) \in \mathcal{E}$ do
        if $w \notin \mathcal{V}_v$ then
            $push(preorder, w)$                          ▷ Add $w$ onto the preorder stack
            $\mathcal{V}_v = \mathcal{V}_v \cup \{w\}$   ▷ Add $w$ to the set of visited vertices
            dfs_step($w$)                                ▷ recursive search
        end if
    end for
    $push(postorder, v)$                             ▷ Add $v$ onto the postorder stack
end recursive function
```

Symbolic Cholesky

Leaf vertices of row subtrees

- Leaf vertices of row subtrees play a key role in graph algorithms related to sparse Cholesky factorizations.
- They can be used to find the skeleton matrix described above.
- But they can be used for more other tasks. See the next theorem

Symbolic Cholesky

Theorem

Let the elimination tree $\mathcal{T}$ of A be postordered. Let the column indices of the nonzeros in the strictly lower triangular part of row i of A be $c_1, \ldots, c_s$ with $s \geq 1$ and $0 < c_1 < \ldots < c_s < i$. Then c_t is a leaf vertex of the row subtree $\mathcal{T}_r(i)$ if and only if

$$t = 1 \text{ or } (1 < t \leq s \text{ and } c_{t-1} \notin \mathcal{T}(c_t)).$$

- Not a proof:
- Case of $t = 1$ clear.
- Otherwise $c_{t-1} \in \mathcal{T}(c_t) \iff c_t$ cannot be a leaf since
  - a) From both of c_{t-1} and c_t we should get to i uptree (they are both vertices in the same row subtree $\mathcal{T}_r(i)$
  - b) All paths from c_{t-1} would have to go through c_t, the root of $\mathcal{T}(c_t)$.

Symbolic Cholesky

- Not a proof:
- Case of $t = 1$ clear.
- Otherwise consider $c_{t-1} \in \mathcal{T}(c_t) \iff$
  - a) All paths from c_{t-1} would have to go through c_t, the root of $\mathcal{T}(c_t)$.
    This is standard $\mathcal{T}(c_t)$ subtree-related statement
  - b) We know that from both of c_{t-1} and c_t we can get to i uptree (they are both vertices in the same row subtree $\mathcal{T}_r(i)$), we have $a_{i,c_{t-1}} \neq 0$, $a_{i,c_t} \neq 0$
    This is row subtree-related statement
  - Consequently, this is equivalent with c_t cannot not a leaf of the row subtree $\mathcal{T}_r(i)$

Symbolic Cholesky

Leaf vertices of row subtrees: the distance between leaves

Corollary

Under the assumptions of the previous Theorem, c_t is a leaf vertex of $\mathcal{T}_r(i)$ if and only if

$$t = 1 \ \textit{ or } \ (1 < t \le s \ \textit{ and } \ c_{t-1} < c_t - |\mathcal{T}(c_t)| + 1).$$

- Note that this inequality directly follows for the postordering.
- Namely, each vertex (c_{t-1}) that is not a leaf of a subtree $\mathcal{T}_{c_t}$ must have its label smaller than all vertices of this subtree.
- And there are at least $|\mathcal{T}_{c_t}|$ such vertices.

Symbolic Cholesky

Leaf vertices of row subtrees

- Subtree sizes can be easily computed bottom up.
- Correctness of next Algorithm is guaranteed because $parent$ defines a topological ordering of $\mathcal{T}$.
- One warning: implementation may not be straightforward, since matrix rows may not be sorted.
- Theorem below relaxes the condition of row ordering: the leaf vertices of row subtrees can be determined in column-oriented algorithms.

Symbolic Cholesky

Leaf vertices of row subtrees by columns

Theorem

Consider the elimination tree $\mathcal{T}$ of A. Vertex j is a leaf vertex of some row subtree of $\mathcal{T}$ if and only if there exists $i \in adj_{\mathcal{G}(A)}\{j\}$, $j < i \leq n$, such that $i \notin adj_{\mathcal{G}(A)}\{k\}$ for all $k \in \mathcal{T}(j) \setminus \{j\}$.

- Going down the column j, scanning all its row indices i.
- We know that columns k from $\mathcal{T}(j) \setminus \{j\}$ have been processed before (due to postordering)
- Then, if there is such i in the column j, $i \notin adj_{\mathcal{G}(A)}\{k\}$ (that is, by the previous result, a sufficient gap between k and j), j should be a leaf of $\mathcal{T}_r(i)$.

Symbolic Cholesky

Leaf vertices of row subtrees

Algorithm (Find leaf vertices of row subtrees of $\mathcal{T}$)

Input: *A with a symmetric sparsity pattern and a corresponding postordered elimination tree $\mathcal{T}$.*
Output: *Logical vector $isleaf$ with entries set to true for leaf vertices of row subtrees.*

```
$isleaf(1:n) = false, \quad prev\_nonz(1:n) = 0$
Compute $|\mathcal{T}(1:n)|$
for $j = 1:n$ do                                        ▷ Loop over the columns of $A$
    for $i$ such that $i > j$ and $a_{ij} \neq 0$ do    ▷ Row index in strictly lower triangular part of $A$
        $k = prev\_nonz(i)$                             ▷ Column index of most recently seen entry in row $i$
        if $k < j - |\mathcal{T}(j)| + 1$ then
            $isleaf(j) = true$                          ▷ $j$ is a leaf vertex
        end if
        $prev\_nonz(i) = j$                             ▷ Flag $j$ as the most recently seen entry in row $i$
    end for
end for
```

Symbolic Cholesky

Leaf of row subtrees: technicality - needed subtree sizes

Algorithm (Find sizes of subtrees $\mathcal{T}(i)$ of $\mathcal{T}$)

Input: *Elimination tree $\mathcal{T}$ described by the vector $parent$.*
Output: *Subtree sizes $|\mathcal{T}(i)|$ ($1 \le i \le n$).*

```
$|\mathcal{T}(1:n)| = 1$
for $i = 1 : n-1$ do
    $k = parent(i)$
    $|\mathcal{T}(k)| = |\mathcal{T}(k)| + |\mathcal{T}(i)|$
end for
```

Symbolic Cholesky

Leaf vertices of row subtrees by columns

Theorem

Consider the elimination tree $\mathcal{T}$ of A. Vertex j is a leaf vertex of some row subtree of $\mathcal{T}$ if and only if there exists $i \in adj_{\mathcal{G}(A)}\{j\}$, $j < i \leq n$, such that $i \notin adj_{\mathcal{G}(A)}\{k\}$ for all $k \in \mathcal{T}(j) \setminus \{j\}$.

Proof.

Assume that for some $i \in anc_{\mathcal{T}}\{j\}$ vertex j is a leaf vertex of $\mathcal{T}_r(i)$. That is, $i \in adj_{\mathcal{G}(A)}\{j\}$, $i > j$. Suppose there exists k in $\mathcal{T}(j) \setminus \{j\}$ such that $i \in adj_{\mathcal{G}(A)}\{k\}$. Then all the ancestors of $k, k \leq i$, in particular j, belong to $\mathcal{T}_r(i)$ and j cannot be a leaf vertex of $\mathcal{T}_r(i)$. This is a contradiction.

Conversely, assume that j is not a leaf vertex of any row subtree of $\mathcal{T}$ and that there exists $i \in adj_{\mathcal{G}(A)}\{j\}$, $j < i \leq n$, such that $i \notin adj_{\mathcal{G}(A)}\{k\}$ for all $k \in \mathcal{T}(j) \setminus \{j\}$. Because j is not a leaf vertex of any such $\mathcal{T}_r(i)$, Theorem on necessary and sufficient fill-in conditions implies that there exists $k \in \mathcal{T}(j) \setminus \{j\}$ such that $a_{ik} \neq 0$, which gives a contradiction and completes the proof. □ □

Symbolic Cholesky

Leaf vertices of row subtrees: you have noticed that

- To find leaf vertices of row subtrees of $\mathcal{T}(A)$, Algorithm uses a marking scheme based on Theorem above (with the gap sizes ☺) and exploits the postordering of $\mathcal{T}(A)$.
- The auxiliary vector $prev\_nonz$ stores the column indices of the most recently encountered entries in the rows of the strictly lower triangular part of A. They can used to evaluate the gaps.

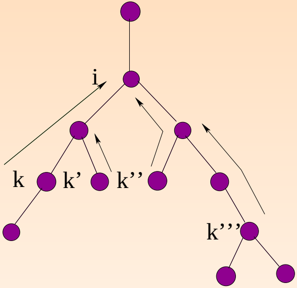
Symbolic Cholesky

Leaf vertices of row subtrees: why do we need them?

- Remind the skeleton that improves some algorithms quantitatively.
- For better algorithms to compute the elimination tree. By using
  - The skeleton matrix and its graph $\mathcal{G}(A^-)$
  - Additional and better a priori shortcuts demonstareted in the next slide
- Theoretical complexity can be reduced to $O(nz(A)\, g(nz(A), n))$, where $g(nz(A), n)$ is a very slowly increasing function called the functional inverse of Ackermann's function (nearly linear in $m = nz(A)$.

Symbolic Cholesky

Demonstrating the more sophisticated approach to get $\mathcal{T}(A)$: the idea



- Instead of following the paths: distances to junctions only used.
- Uses a fast (existing) algorithm to determine junctions of branches.
- We have a fast algorithm to find leaves of the elimination tree and use postordering.
- Then the complexity of getting the elimination tree can be $O(nz(A^{-})\, g(nz(A^{-}), n))$, where $nz(A^{-})$ is the number of entries in the skeleton matrix: nearly linear in m.

Symbolic Cholesky

Blocks

- Blocks are absolutely crucial to compute efficiently: enable to process as much data as possible for a unit of data transfer between CPU(s) and memory hierarchy.
- In BLAS terminology:

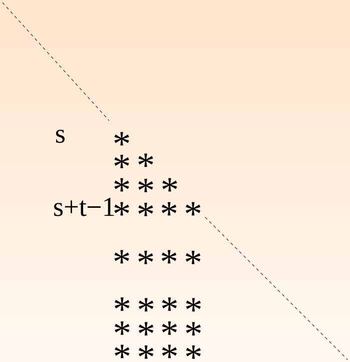
$$z = x + \alpha y \longrightarrow Z = X + \alpha Y \ (\ vector \ opes \ )$$

in general: saxpy $\longrightarrow$ dgemm

- But we have sparse matrices. It is not so straightforward to split their nonzeros into blocks.
- But, postordered elimination tree naturally reveals them.
- Although some other matrix reorderings will be mentioned later.

Symbolic Cholesky

Supernodes: look like this: in L



- A suspicion: replication will help us. ☺

Symbolic Cholesky

Supernodes

- Because of column replication, the columns of L generally become denser as the Cholesky factorization proceeds.
- To exploit this, we require the concept of supernodes. The idea: group together columns with the same sparsity structure, so that they can be treated as a dense matrix.

Symbolic Cholesky

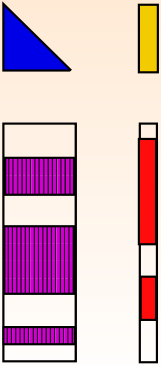
Supernodes

- Enhance the efficiency of sparse factorizations and sparse triangular solves because they enable floating-point operations to be performed on dense submatrices rather than on individual nonzeros, thus improving memory hierarchy utilization and allowing the use of highly efficient dense linear algebra kernels (such as Level 3 BLAS kernels).
- Columns within a supernode are numbered consecutively but they can be numbered within the supernode in any order without changing the number of nonzeros in L (assuming the corresponding rows are permuted symmetrically). On some architectures, particularly those using GPUs, this freedom can be exploited to improve the factorization efficiency.
- Supernode amalgamation to achieve better efficiency.

Symbolic Cholesky

Supernodes and efficient computation

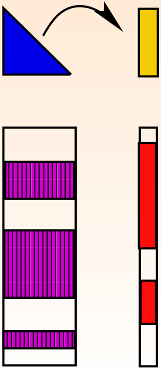
- the loop over rows has no indirect addressing: (dense BLAS1)



Symbolic Cholesky

Supernodes and efficient computation

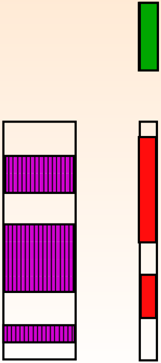
- the loop over rows has no indirect addressing: (dense BLAS1)



Symbolic Cholesky

Supernodes and efficient computation

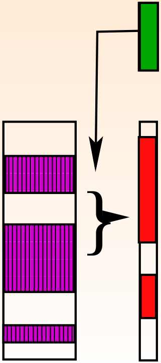
- the loop over rows has no indirect addressing: (dense BLAS1)
- the loop over columns of the updating supernode can be unrolled to save memory references (dense BLAS2)



Symbolic Cholesky

Supernodes and efficient computation

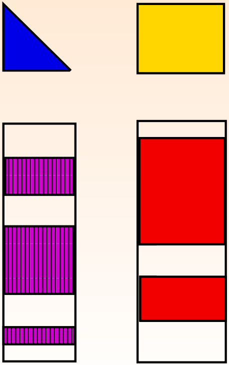
- the loop over rows has no indirect addressing: (dense BLAS1)
- the loop over columns of the updating supernode can be unrolled to save memory references (dense BLAS2)



Symbolic Cholesky

Supernodes and efficient computation

- the loop over rows has no indirect addressing: (dense BLAS1)
- the loop over columns of the updating supernode can be unrolled to save memory references (dense BLAS2)
- parts of the updating supernode can be used for blocks of updated supernode (dense BLAS3)



Symbolic Cholesky

Supernodes: another visualization

$$
\begin{array}{c|ccccccccccc}
 & \ldots & \ldots & s & \ldots & \ldots & s' & \ldots & \ldots & \ldots & \ldots & \\
\hline
\vdots & \ddots & \ldots & & & & & & & & & \\
\vdots & \vdots & \ddots & & & & & & & & & \\
s & & & * & & & & & & & & \\
\vdots & & & * & \ddots & & & & & & & \\
\vdots & & & * & * & \ddots & & & & & & \\
s' & & & * & * & * & * & & & & & \\
\vdots & & & . & \ldots & \ldots & . & \ddots & & & & \\
\vdots & & & * & * & * & * & & \ddots & & & \\
\vdots & & & * & * & * & * & & & \ddots & & \\
\vdots & & & . & \ldots & \ldots & . & & & & \ddots & \\
\vdots & & & * & * & * & * & & & & & \ddots
\end{array}
$$

Symbolic Cholesky

Supernodes: formally

- Let $1 \leq s, t \leq n$ with $s + t - 1 \leq n$. A set of contiguously numbered columns of L with indices $S = \{s, s+1, \ldots, s+t-1\}$ is a supernode of L if

  $$col_L\{s\} \cup \{s\} = col_L\{s+t-1\} \cup \{s, \ldots, s+t-1\}, \tag{30}$$

  and S cannot be extended for $s > 1$ by adding $s - 1$ or for $s + t - 1 < n$ by adding $s + t$.

- Because S cannot be extended it is a maximal subset of column indices. In graph terminology, a supernode is a maximal clique of contiguous vertices of $\mathcal{G}(L + L^T)$.

- A supernode may contain a single vertex ($t = 1$).

Symbolic Cholesky

Supernodes: an example

$$L = \begin{array}{c|cccccccc|}
 & 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 \\
\hline
1 & * & & & & & & & \\
2 & * & * & & & & & & \\
3 & & & * & & & & & \\
4 & & & * & * & & & & \\
5 & * & * & * & * & * & & & \\
6 & & & & & * & * & & \\
7 & * & * & & & * & * & * & \\
8 & & & * & * & * & * & * & *
\end{array}$$

Figure: *An example to illustrate supernodes in the whole L. The first supernode comprises columns 1 and 2, the second columns 3 and 4, and the third columns 5 to 8.*

Symbolic Cholesky

Supernodes and the assembly tree

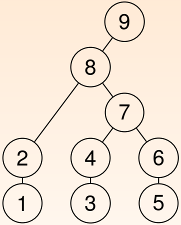
- The supernodal elimination tree or assembly tree is the elimination tree based on the supernodal structure.
- Each vertex of the elimination tree is associated with one elimination and a single integer (the index of its parent) is needed.
- Practicality: with each vertex of the assembly tree is associated an index list of the row indices of the nonzeros in the columns of the supernode. These lists (expressing the row structures of supernodes) implicitly define the sparsity pattern of L.

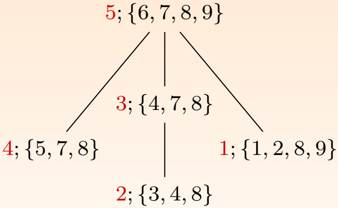
Symbolic Cholesky

Supernodes: difference of the introduced trees

- Demonstrating the difference between the elimination and assembly trees. The elimination tree is postordered, it has 5 supernodes: $\{1, 2\}$, 3, 4, 5, $\{6, 7, 8, 9\}$. For supernode 1 that comprises columns 1 and 2, the row index list is $\{1, 2, 8, 9\}$.

$$
\begin{array}{c|ccccccccc}
 & 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 \\
\hline
1 & * & * & & & & & & * & * \\
2 & * & * & & & & & & * & * \\
3 & & & * & * & & & & * & \\
4 & & & * & * & & & * & * & \\
5 & & & & & * & * & * & & \\
6 & & & & & * & * & f & * & * \\
7 & & & & * & * & f & * & f & * \\
8 & * & * & * & * & & * & f & * & f \\
9 & * & * & & & & * & * & f & *
\end{array}
$$




$5; \{6, 7, 8, 9\}$

$3; \{4, 7, 8\}$

$4; \{5, 7, 8\}$

$1; \{1, 2, 8, 9\}$

$2; \{3, 4, 8\}$

Figure: *A sparse matrix, its postordered elimination tree (left) and postordered assembly tree (right). Filled entries in $\mathcal{S}\{L + L^T\}$ are denoted by f. For the assembly tree, the vertices are in red and the index lists associated with each vertex are given.*

Symbolic Cholesky

Convenient characterization of supernodes

Theorem

The set of columns of L with indices $S = \{s, s+1, \ldots, s+t-1\}$ is a supernode of L if and only if it is a maximal set of contiguous columns such that $s+i-1$ is a child of $s+i$ for $i = 1, \ldots, t-1$ and

$$|\,col_L\{s\}\,| = |\,col_L\{s+t-1\}\,| + t - 1. \tag{31}$$

- Supernodes specified just by sizes and not by set inclusions.
- Only one stream of replications. Since other streams would surely introduce other nodes.
- But, still the supernodes can go through junctions and up.

Symbolic Cholesky

Convenient characterization of supernodes

Theorem

The set of columns of L with indices $S = \{s, s+1, \dots, s+t-1\}$ is a supernode of L if and only if it is a maximal set of contiguous columns such that $s+i-1$ is a child of $s+i$ for $i = 1, \dots, t-1$ and

$$|\, col_L\{s\} \,| = |\, col_L\{s+t-1\} \,| + t - 1. \tag{32}$$

Proof.

Let S be a supernode. For $i, j \in S$ with $i > j$ we have $i \in col_L\{j\}$. This implies that in the postordered elimination tree the vertex $i = j+1$ is the parent of j for $j = s, \dots, s+t-2$. Moreover, from Observation 25, for any $i, j \in S$ with $i > j$, $i \in col_L\{j\}$ implies $col_L\{j\} \setminus \{1, \dots, i\} \subseteq col_L\{i\}$. Therefore, $|\, col_L\{s+i\} \,| \geq |\, col_L\{s+i-1\} \,| - 1, \quad i = 1, \dots, t-1$, with equality if and only if $col_L\{s+i\} = col_L\{s+i-1\} \setminus \{s+i\}$, that is, if S is a supernode.

Conversely, assume S is a maximal set of contiguous columns such that, for $i = 1, \dots, t-1$, $s+i-1$ is a child of $s+i$ and S satisfies (32). Because of column replication, such a sequence of parent and child vertices must satisfy inequality above with equality if and only if S is a supernode. □ □

Symbolic Cholesky

Fundamental supernodes: useful restriction

- In practice, fundamental supernodes are easier to work with in the numerical factorization: do not continue through junctions.
- Let $1 \le s, t \le n$ with $s+t-1 \le n$. A maximal set of contiguously numbered columns of L with indices $S = \{s, s+1, \ldots, s+t-1\}$ is a fundamental supernode if for any successive pair $i-1$ and i in the list, $i-1$ is the only child of i in $\mathcal{T}$ and $col_L\{i\} \cup \{i\} = col_L\{i-1\}$. s is termed the starting vertex.
- The difference between the sets of supernodes and fundamental supernodes is normally not large, with the latter having (slightly) more blocks in the resulting partitioning of L.
- Note that fundamental supernodes are independent of the choice of the postordering of $\mathcal{T}$.

Symbolic Cholesky

$$\begin{array}{cc}
 & \begin{matrix} 1 & 2 & 3 & 4 & 5 & 6 \end{matrix} \\
\begin{matrix} 1 \\ 2 \\ 3 \\ 4 \\ 5 \\ 6 \end{matrix} &
\begin{pmatrix}
* & * & & & * & * \\
* & * & & f & f & \\
 & & * & * & * & * \\
 & & * & * & f & f \\
* & f & * & f & * & * \\
* & f & * & f & * & *
\end{pmatrix}
\end{array}$$

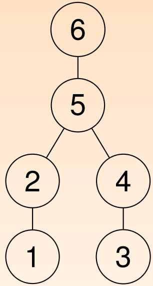


Figure: A matrix A and its postordered elimination tree $\mathcal{T}$ for which the set of supernodes $\{1, 2\}$ and $\{3, 4, 5, 6\}$ and the set of fundamental supernodes $\{1, 2\}, \{3, 4\}$ and $\{5, 6\}$ are different. The filled entries in $\mathcal{S}\{L + L^T\}$ are denoted by f.

Symbolic Cholesky

Fundamental supernodes and leaves of row subtrees: that's it

Theorem

Assume $\mathcal{T}$ is postordered. Vertex s is the starting vertex of a fundamental supernode if and only if it has at least two child vertices in $\mathcal{T}$ or it is a leaf vertex of a row subtree of $\mathcal{T}$.

Proof.

If s has at least two child vertices then, from the definition of a fundamental supernode, it must be the starting vertex of a fundamental supernode. Assume that, for some $i > s$, s is a leaf vertex of $\mathcal{T}_r(i)$. If s is also a leaf vertex of $\mathcal{T}$ then s is a starting vertex of a supernode. The remaining case is s having only one child. Because $\mathcal{T}$ is postordered, this child must be $s-1$. Theorem of necessary and sufficient condititions for the fill-in implies $a_{is} \neq 0$ and $a_{i,s-1} = 0$, that is, $i \in col_L\{s\}$ and $i \notin col_L\{s-1\}$. It follows that

$$\mathcal{S}\{L_{s-1:n,s-1}\} \subsetneq \mathcal{S}\{L_{s:n,s}\} \cup \{s-1\},$$

and vertices s and $s-1$ cannot belong to the same supernode. Hence, s is the starting vertex of a new fundamental supernode.

Conversely, assume that s is the starting vertex of a fundamental supernode S. If s has no child vertices or at least two child vertices, the result follows. If s has exactly one child vertex, postordering implies this child is $s-1$. Because S is maximal there exists i such that $i \notin col_L\{s-1\}$ and $i \in col_L\{s\}$, (otherwise S could be extended by adding $s-1$). Hence s is a leaf vertex of $\mathcal{T}_r(i)$. □

Symbolic Cholesky

Overall complexity of symbolic operations

- Because fundamental supernodes are characterized by their starting vertices, they can be found by modifying Algorithm to incorporate marking leaf vertices of the row subtrees and vertices with at least two child vertices.
- Once the elimination tree has been computed the complexity is $O(n + nz(A))$.
- The computation can be made even more efficient by using the skeleton graph $\mathcal{G}(A^{-})$.
- Overall we have the complexity close to linear.

Outline

1. Introduction
2. Introductory notation and terminology
3. Factorizations
4. Graphs and sparse matrices
5. Sparse matrices and data structures
6. Symbolic factorizations
7. Symbolic Cholesky factorization
8. Cholesky factorization
9. Sparse LU factorization
10. Stability, ill-conditioning, indefiniteness
11. Blocks in the input matrix
12. Sparse Least Squares and QR factorization
13. Reorderings
14. Algebraic preconditioning
15. Incomplete factorizations
16. Sparse approximate inverses

Cholesky Factorization

Sparse Cholesky factorization: conceptual comments

- Theoretical background based on the elimination tree $\mathcal{T} \rightarrow$ Efficient symbolic phase: done, explained.
- Namely, $\mathcal{T}(A)$ allows:
  - row/column counts of L known $\rightarrow$ storage can be allocated
  - Postordering enables a lot of other efficient algorithms
  - leaves of row subtrees: (fundamental) supernodes
  - column structures $\rightarrow$ supernodal communication (dependency) DAG
- Further tricks: as splitting large supernodes into smaller panels to embed them into computer caches, Sparse data access tricks.

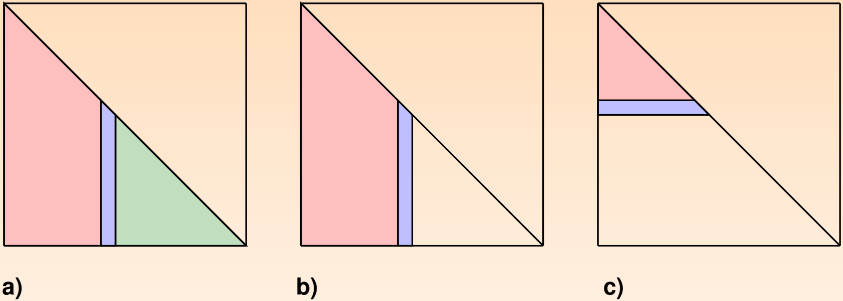
Cholesky Factorization

Sparse Cholesky factorizations: our three forms

- Several classes of sparse Cholesky algorithms (right-looking, left-looking, upward-looking).
- Major difference among them: in what relative order they schedule the computations (tasks). But, all of them are equivalent also in finite precision arithmetic.
- Large and sparse computations that exploit parallel potential/panel/block can be modified even more specifically. Compiler optimization: other source of non-equivalence.

Cholesky Factorization

Sparse Cholesky factorizations: our three forms graphically



- a) submatrix Cholesky, immediate update, data-driven, right-looking
- b) column Cholesky, delayed update, demand-driven, left-looking
- a) row Cholesky, substitution-based, up-looking

🔴 Entries computed

🔵 Entries being processed

🟢 Entries partially computed

Cholesky Factorization

Sparse Cholesky factorizations

- The entries of L satisfy in all our schemes the relationship

$$L_{j+1:n,j} = \left(A_{j+1:n,j} - \sum_{k=1}^{j-1} L_{j+1:n,k} l_{jk} \right) / l_{jj} \quad \text{with} \quad l_{jj} = \left(a_{jj} - \sum_{k=1}^{j-1} l_{jk}^2 \right)^{1/2} ,$$

- Symbolic dependencies described by replications represent typically only a fraction of numerical dependencies.

Theorem

The numerical values of the entries in column $j > k$ of L depend on the numerical values in column k of L if and only if $l_{jk} \neq 0$.

Cholesky Factorization

Sparse Cholesky factorization: left-looking

Algorithm (Simplified sparse left-looking Cholesky)

Input: *SPD matrix A and sparsity pattern $\mathcal{S}\{L\}$.*

```
$l_{ij} = a_{ij}$ for all $(i,j) \in \mathcal{S}\{L\}$        ▷ Filled entries in $L$ are initialised to zero
for $j = 1 : n$ do
    for $k \in \{k < j \,|\, l_{jk} \neq 0\}$ do
        for $i \in \{i \geq j \,|\, l_{ik} \neq 0\}$ do
            $l_{ij} = l_{ij} - l_{ik} l_{jk}$
        end for
    end for
    $l_{jj} = (l_{jj})^{1/2}$
    for $i \in \{i > j \,|\, l_{ij} \neq 0\}$ do
        $l_{ij} = l_{ij} / l_{jj}$
    end for
end for
```

- Remind: $\mathcal{S}\{L\}$ was efficiently determined $\Rightarrow$ and static storage for L can be used. This simplifies the schemes a lot. But implementation is still crucial.

Cholesky Factorization

Sparse Cholesky factorizations: right-looking

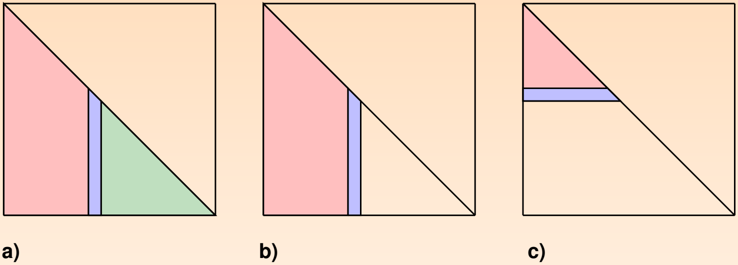
Algorithm (Simplified sparse right-looking Cholesky)

Input: *SPD matrix A and sparsity pattern $\mathcal{S}\{L\}$.*

```
For all $(i, j) \in \mathcal{S}\{L\}$ set $l_{ij} = a_{ij}$        ▷ Filled entries in $L$ are initialised to zero
for $j = 1 : n$ do
    $l_{jj} = (l_{jj})^{1/2}$
    for $i \in \{i > j \,|\, l_{ij} \neq 0\}$ do
        $l_{ij} = l_{ij} / l_{jj}$
    end for
    for $k \in \{k > j \,|\, l_{kj} \neq 0\}$ do
        for $i \in \{i \geq k \,|\, l_{ij} \neq 0\}$ do
            $l_{ik} = l_{ik} - l_{ij} l_{kj}$
        end for
    end for
end for
```

Cholesky Factorization

Numerical Cholesky factorization: from operations to tasks



- $cdiv(k)$: scaling column k by the square root of the diagonal entry
- $cmod(j,k)$: column j modified by a multiple of column k

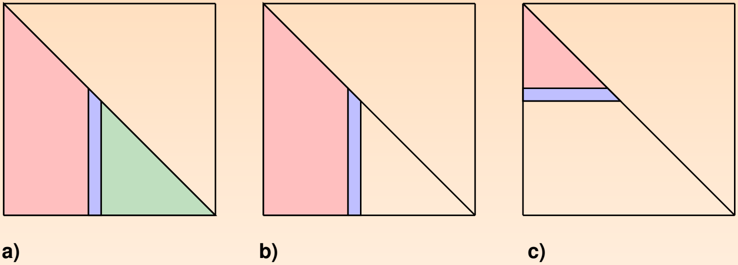
Algorithm

Sparse column (left-looking) Cholesky

```
1: for $j = 1 : n$ do
2:     for $k \in Struct(L_{j*})$ do
3:         $cmod(j,k)$
4:     end for
5:     $cdiv(j)$
6: end for
```

Cholesky Factorization

Numerical Cholesky factorization: from operations to tasks



- $cdiv(k)$: scaling column k by the square root of the diagonal entry
- $cmod(j,k)$: column j modified by a multiple of column k

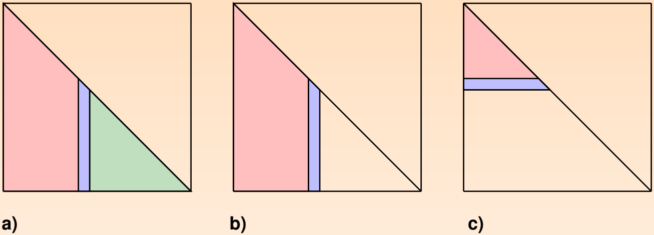
Algorithm

Sparse submatrix (right-looking) Cholesky

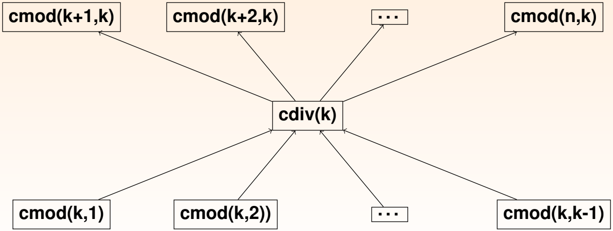
```
for $k = 1 : n$ do
    $cdiv(k)$
    for $j \in Struct(L_{*k})$ do
        $cmod(j,k)$
    end for
end for
```

Cholesky Factorization

Splitting Cholesky factorization into tasks



- $cdiv(k)$: scaling column k by the square root of the diagonal entry
- $cmod(j, k)$: column j modified by a multiple of column k



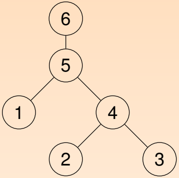
cmod(k+1,k) **cmod(k+2,k)** **···** **cmod(n,k)**

cdiv(k)

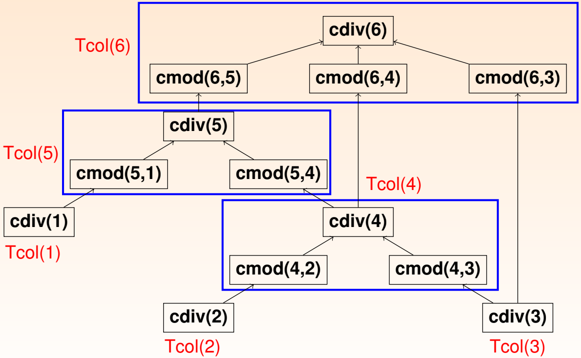
cmod(k,1) **cmod(k,2))** **···** **cmod(k,k-1)**

Cholesky Factorization

Large-grain column communication model

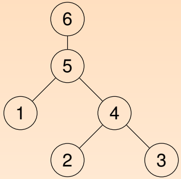


$$\begin{array}{c|cccccc} & 1 & 2 & 3 & 4 & 5 & 6 \\ \hline 1 & * & & & & * & \\ 2 & & * & & * & & \\ 3 & & & * & * & & * \\ 4 & & * & * & * & * & f \\ 5 & * & & & * & * & * \\ 6 & & & * & f & * & * \end{array}$$

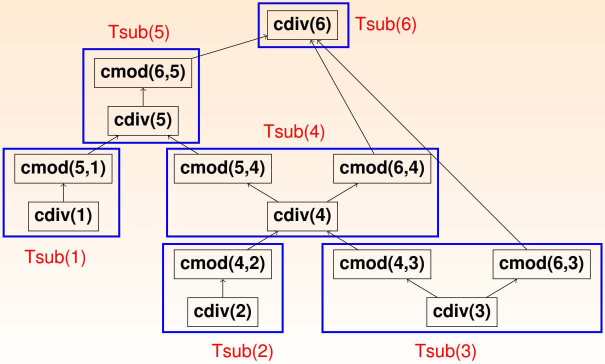


Cholesky Factorization

Large-grain submatrix communication model

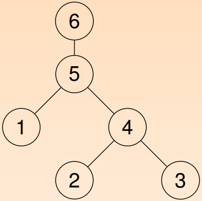


$$
\begin{array}{c|cccccc}
 & 1 & 2 & 3 & 4 & 5 & 6 \\
\hline
1 & \ast & & & & \ast & \\
2 & & \ast & & \ast & & \\
3 & & & \ast & \ast & & \ast \\
4 & & \ast & \ast & \ast & \ast & f \\
5 & \ast & & & \ast & \ast & \ast \\
6 & & & \ast & f & \ast & \ast
\end{array}
$$

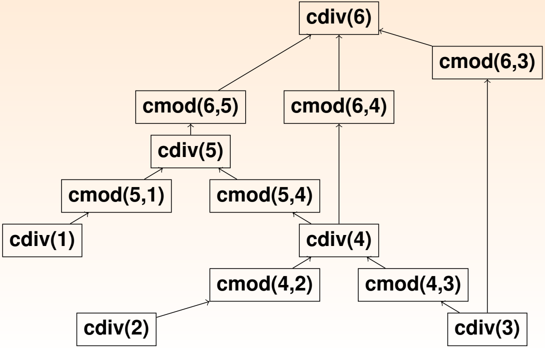


Cholesky Factorization

Medium-grain



$$\begin{array}{c|cccccc}
 & 1 & 2 & 3 & 4 & 5 & 6 \\
\hline
1 & * & & & & * & \\
2 & & * & & * & & \\
3 & & & * & * & & * \\
4 & & * & * & * & * & f \\
5 & * & & & * & * & * \\
6 & & & * & f & * & *
\end{array}$$



cdiv(6)

cmod(6,3)

cmod(6,5)

cmod(6,4)

cdiv(5)

cmod(5,1)

cmod(5,4)

cdiv(1)

cdiv(4)

cmod(4,2)

cmod(4,3)

cdiv(2)

cdiv(3)

Cholesky Factorization

Numerical factorization: blocks and panels needed

- In order to be efficient, dense blocks and dense rectangular panels should be heavily exploited.
- Remind: SPD matrices are factorizable (strongly regular), the Cholesky factorization $A = LL^T$ exists (in exact arithmetic) $\rightarrow$ permutations can foster block structure of (symmetrically) permuted A
- To show block factorization, we start with straightforward dense schemes. That can be extended to the saprse case since we know the structure of L.
- In-place algorithms: L overwrites the lower triangular part of A (reducing memory requirements, if A is no longer required).

Cholesky Factorization

In-place pointwise dense factorization

Algorithm (In-place dense left-looking Cholesky factorization)

Input: *Dense SPD matrix A.*
Output: *Factor L such that $A = LL^T$.*

```
for $j = 1 : n$ do
    $L_{j:n,j} = A_{j:n,j}$                         ▷ Only the lower triangular part of $A$ is required
    for $k = 1 : j-1$ do
        $L_{j:n,j} = L_{j:n,j} - L_{j:n,k}\, l_{jk}$   ▷ Update column $j$ using previous columns
    end for
    $l_{jj} = (l_{jj})^{1/2}$                       ▷ Overwrite the diagonal entry with its square root
    $L_{j+1:n,j} = L_{j+1:n,j} / l_{jj}$             ▷ Scale off-diagonal entries in column $j$
end for
```

Cholesky Factorization

Using panels (left-looking)

Algorithm (In-place dense left-looking panel Cholesky factorization)

Input: *Dense SPD matrix A with nb panels.*
Output: *Factor L such that $A = LL^T$.*

```
for $jb = 1 : nb$ do
    $L_{jb:nb,jb} = A_{jb:nb,jb}$
    for $kb = 1 : jb - 1$ do
        $L_{jb:nb,jb} = L_{jb:nb,jb} - L_{jb:nb,kb}\, L^T_{jb,kb}$    ▷ Update block column $jb$
    end for
    Compute in-place factorization of $L_{jb,jb}$    ▷ Overwrite $L_{jb,jb}$ with its Cholesky factor
    $L_{jb+1:nb,jb} = L_{jb+1:nb,j}\, L^{-T}_{jb,jb}$    ▷ Dense triangular solve
end for
```

Cholesky Factorization

Using panels (right-looking)

Algorithm (In-place dense right-looking panel Cholesky factorization)

Input: *Dense SPD matrix A with nb panels.*
Output: *Factor L such that $A = LL^T$.*

```
for $jb = 1 : nb$ do
    $L_{jb:nb,jb} = A_{jb:nb,jb}$
end for
for $jb = 1 : nb$ do
    Compute in-place factorization of $L_{jb,jb}$    ▷ Overwrite $L_{jb,jb}$ with its Cholesky factor
    $L_{jb+1:nb,jb} = L_{jb+1:nb,j} \, L_{jb,jb}^{-T}$    ▷ Dense triangular solve
    for $kb = jb + 1 : nb$ do
        $L_{kb:nb,kb} = L_{kb:nb,kb} - L_{kb:nb,jb} \, L_{kb,jb}^T$
    end for
end for
```

Cholesky Factorization

Using blocks (right-looking)

Algorithm (In-place dense right-looking block Cholesky factorization)

Input: *Dense SPD matrix A with $nb \times nb$ blocks.*
Output: *Factor L such that $A = LL^T$.*

```
for $jb = 1 : nb$ do
    $L_{jb:nb,jb} = A_{jb:nb,jb}$
end for
for $jb = 1 : nb$ do
    Compute in-place factorization of $L_{jb,jb}$                 ▷ Task factorize$(jb)$
    for $ib = jb + 1 : nb$ do
        $L_{ib,jb} = L_{ib,jb}\, L_{jb,jb}^{-T}$                  ▷ Task solve$(ib, jb)$
        for $kb = jb + 1 : ib$ do
            $L_{ib,kb} = L_{ib,kb} - L_{ib,jb}\, L_{kb,jb}^{T}$   ▷ Task update$(ib, jb, kb)$
        end for
    end for
end for
```

Cholesky Factorization

Tasks for efficient parallelization (still dense case motivation)

- The panel and block descriptions of the factorization enable efficient parallelization extending pointwise description above.
- The three main block operations (tasks) are factor(jb), solve(ib, jb) and update(ib, jb, kb).
- There are the following dependencies between the tasks.

factorize(jb) depends on update(jb, kb, jb) for all $kb = 1, \dots, jb - 1$.

solve(ib, jb) depends on update(ib, kb, jb) for all $kb = 1, \dots, jb - 1$, and factorize(jb).

update(ib, jb, kb) depends on solve(ib, kb), solve(jb, kb).

- A communication/dependency graph (DAG) based on blocks used to schedule the tasks.

Cholesky Factorization

Using supernodes: enhancing parallel processing

- Assume supernodal structure. Arithmetic of dense trapezoidal matrices. Blocks (obtained by shrinking - need to remember its mapping) termed nodal matrices.

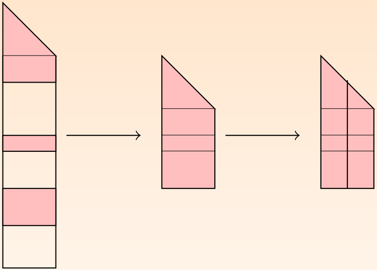


Figure: Supernode (left), the corresponding nodal matrix (centre), and the nodal matrix with two panels (right).

- Ideas applicable in both left-looking and right-looking algorithms.

Cholesky Factorization

Comments on (the right-looking) supernodal processing

- Ad our scheme: once a supernode is ready to be factorized → a dense factorize task is performed
- Then a triangular solve (solve task) is performed with the computed factor and the rectangular part of the nodal matrix.
- Iterating update tasks uptree the supernode in the assembly tree (fan-in/fan-out).
- Technically, for each parent, the rows of the current supernode for each parent columns are identified. The outer product of those rows and the subdiagonal part of the supernode (update operations).
- Can be done in the left-looking way as well (as mentioned) ☺.

Cholesky Factorization

DAG-based approach and task splittings

- Nodal matrices may be subdivided into blocks.

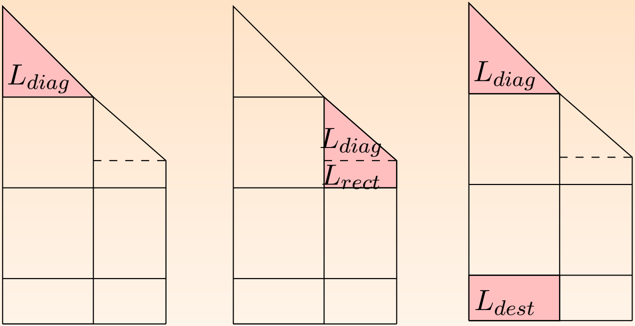


Figure: An illustration of a blocked nodal matrix with two block columns. The first block on the diagonal is triangular and the second one is trapezoidal. The task factorize_block is illustrated on the left and in the centre; the task solve_block is illustrated on the right.

Cholesky Factorization

DAG-based approach and supernodes: task splitting more formally

- Various ways to split factorization into tasks.
- Tentative classification distinguishes update_internal (the same nodal matrix), update_between (different nodal matrices).
  - factorize_block(L_{diag}) Computes the dense Cholesky factor L_{diag}. If the block is trapezoidal, it is followed by a triangular solve of its rectangular part $L_{rect} = L_{rect} L_{diag}^{-T}$ (centre).
  - solve_block(L_{dest}) A triangular solve of an off-diagonal block L_{dest} of the form $L_{dest} = L_{dest} L_{diag}^{-T}$ (rightmost).
  - update_internal (L_{dest}, L_r, L_c) The update $L_{dest} = L_{dest} - L_r L_c^T$, where L_{dest}, L_r and L_c belong to the same nodal matrix.
  - update_between (L_{dest}, L_r, L_c) Performs the update $L_{dest} = L_{dest} - L_r L_c^T$, where L_r and L_c belong to the same nodal matrix and L_{dest} belongs to a different nodal matrix.

Cholesky Factorization

DAG-based approach and the sparse case: notes

- The tasks are partially ordered: a dependency DAG is used determine the precedencies. Some tasks computed in parallel.

- Using precendencies means that, for example, updating task that uses L_{dest}, L_r, L_c has to wait until all the relevant rows of the block column are available.

- At each stage of the factorization, some tasks are being executed (in parallel) while others are held (in a stack or pool of tasks) ready for execution.

Cholesky Factorization

ariations of the Cholesky factorization approaches: sparsity and supernode

- Left-looking approach:
  - dependency DAG, sophisticated mappings for the update tasks
  - DAG paralelism, block arithmetic
- Right-looking approach:
  - a popular approach: delayed, using the supernodal elimination tree for dependencies: the multifrontal method
  - tree paralelism, block arithmetic
  - high level of memory efficiency due to computational locality: contributions to the Schur complement kept aside in a stack

Cholesky Factorization

Multifrontal method: some initial comments

Theorem

Let A be SPD and let $\mathcal{T}$ be its elimination tree. The numerical values of entries in column k of the Cholesky factor L of A only affect the numerical values of entries in column i of L for $i \in anc_{\mathcal{T}}\{k\}$ ($1 \leq k \leq n-1$).

- This is what we know, since

$$col_L\{j\} = adj_{\mathcal{G}(A)}\{\mathcal{T}(j)\}.$$

- Updates to the ancestors can be simply stored separately and passed up the tree.
-

Multifrontal method: separate updates

- The multifrontal method works with updates to the Schur complements separately.
- The updates called the frontal matrices are connected to subtrees of $\mathcal{T}$.
- Vertices of $\mathcal{T}$ are topologically ordered $\Rightarrow$ the order to apply the updates goes up the tree: from the leaves of $\mathcal{T}$ to the root vertex.
- And the Schur complements are composed from these individual updates.
- This allows the computation of $S^{(k)}$ to be rewritten as

$$S^{(k)} = A_{k:n,k:n} - \sum_{j \in \mathcal{T}(k) \setminus \{k\}} L_{k:n,j} L_{k:n,j}^T,$$

emphasizing the role of $\mathcal{T}$.

Cholesky Factorization

Multifrontal method: frontal matrix more formally

- Assume $k, k_1, \ldots, k_r$ are the row indices of the nonzeros in column k of L.
- The frontal matrix F_k of the k-th subtree $\mathcal{T}(k)$ of $\mathcal{T}$ is the dense $(r+1) \times (r+1)$ matrix defined by

$$F_k = \begin{pmatrix} a_{kk} & a_{kk_1} & \ldots & a_{kk_r} \\ a_{k_1k} & 0 & \ldots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ a_{k_rk} & 0 & \ldots & 0 \end{pmatrix} - \sum_{j \in \mathcal{T}(k) \setminus \{k\}} \begin{pmatrix} l_{kj} \\ l_{k_1j} \\ \vdots \\ l_{k_rj} \end{pmatrix} \left(l_{kj} \; l_{k_1j} \; \ldots \; l_{k_rj} \right) \tag{33}$$

- It contains all the contributions from previous columns (it is fully summed) ... and passes them subsequently up the tree.

Cholesky Factorization

Multifrontal method: processing the frontal matrix

- A step of the Cholesky factorization of F_k (factorize task and send up the tree) can be written as

$$\begin{aligned} F_k &= \begin{pmatrix} l_{kk} & 0 & \dots & 0 \\ l_{k_1k} & & & \\ \vdots & & I & \\ l_{k_rk} & & & \end{pmatrix} \begin{pmatrix} 1 & & \\ & & \\ & & V_k \\ & & \end{pmatrix} \begin{pmatrix} l_{kk} & l_{k_1k} & \dots & l_{k_rk} \\ 0 & & & \\ \vdots & & I & \\ 0 & & & \end{pmatrix} \\ &= \begin{pmatrix} l_{kk} \\ l_{k_1k} \\ \vdots \\ l_{k_rk} \end{pmatrix} \left(l_{kk}\ l_{k_1k}\ \dots\ l_{k_rk}\right) + \begin{pmatrix} 0 & & \\ & & \\ & & V_k \\ & & \end{pmatrix} \end{aligned}$$

- And V_k (termed a generated element or update matrix or a contribution block) is passed up the tree.

Cholesky Factorization

Multifrontal method: contribution block formally

- Equating the last r rows and columns yields

$$V_k = - \sum_{j \in \mathcal{T}(k)} \begin{pmatrix} l_{k_1 j} \\ \vdots \\ l_{k_r j} \end{pmatrix} \begin{pmatrix} l_{k_1 j} & \dots & l_{k_r j} \end{pmatrix}.$$

  Assume that c_j ($j = 1, \dots, s$) are the children of k in $\mathcal{T}$.

$$F_k = \begin{pmatrix} a_{kk} & a_{kk_1} & \dots & a_{kk_r} \\ a_{k_1 k} & 0 & \dots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ a_{k_r k} & 0 & \dots & 0 \end{pmatrix} \mathbin{\updownarrow\!\!\!\!\leftrightarrow} V_{c_1} \mathbin{\updownarrow\!\!\!\!\leftrightarrow} \dots \mathbin{\updownarrow\!\!\!\!\leftrightarrow} V_{c_s}.$$

- The operation $\updownarrow\!\!\!\!\leftrightarrow$ (extend-add) denotes the addition of matrices that have row and column indices belonging to subsets of the same set of indices (in this case, $k, k_1, \dots, k_r$); entries that have the same row and column indices are summed.
- Of course, the symmetry allows only the lower triangular part of

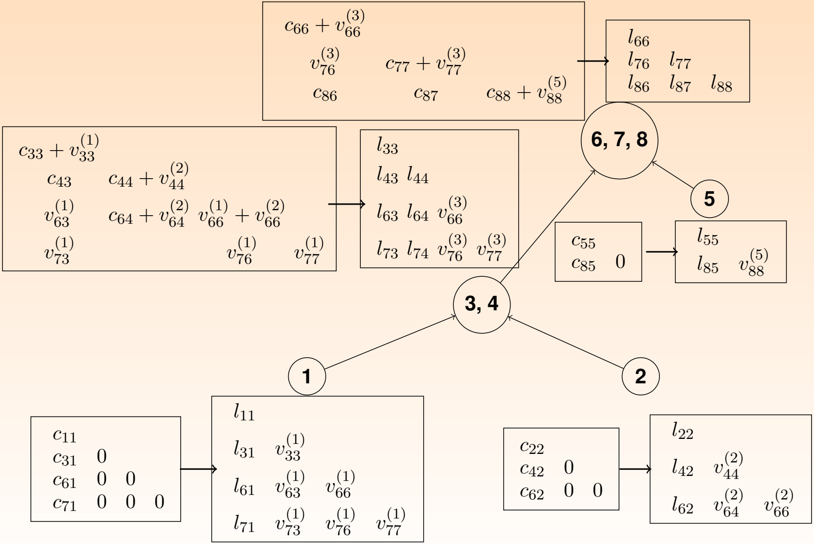
Cholesky Factorization

Multifrontal method: details

$$
\begin{array}{cc}
 & \begin{array}{cccccccc} 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 \end{array} \\
\begin{array}{c} 1 \\ 2 \\ 3 \\ 4 \\ 5 \\ 6 \\ 7 \\ 8 \end{array} &
\begin{pmatrix}
* & & & & & & & \\
 & * & & & & & & \\
* & & * & & & & & \\
 & * & * & * & & & & \\
 & & & & * & & & \\
* & * & & * & & * & & \\
* & & & & & & * & \\
 & & & & * & * & * & *
\end{pmatrix}
\end{array}
\quad
\begin{array}{cc}
 & \begin{array}{cccccccc} 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 \end{array} \\
\begin{array}{c} 1 \\ 2 \\ 3 \\ 4 \\ 5 \\ 6 \\ 7 \\ 8 \end{array} &
\begin{pmatrix}
* & & & & & & & \\
 & * & & & & & & \\
* & & * & & & & & \\
 & * & * & * & & & & \\
 & & & & * & & & \\
* & * & f & * & & * & & \\
* & & f & & & & * & \\
 & & & & * & * & * & *
\end{pmatrix}
\end{array}
\quad
\begin{array}{cc}
 & \begin{array}{cccccccc} 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 \end{array} \\
\begin{array}{c} 1 \\ 2 \\ 3 \\ 4 \\ 5 \\ 6 \\ 7 \\ 8 \end{array} &
\begin{pmatrix}
* & & & & & & & \\
 & * & & & & & & \\
* & & * & & & & & \\
 & * & * & * & & & & \\
 & & & & * & & & \\
* & * & f & * & & * & & \\
* & & f & f & & f & * & \\
 & & & & * & * & * & *
\end{pmatrix}
\end{array}
$$

Cholesky Factorization

Multifrontal method: details



$$\begin{array}{ccc} c_{66}+v_{66}^{(3)} & & \\ v_{76}^{(3)} & c_{77}+v_{77}^{(3)} & \\ c_{86} & c_{87} & c_{88}+v_{88}^{(5)} \end{array} \longrightarrow \begin{array}{ccc} l_{66} & & \\ l_{76} & l_{77} & \\ l_{86} & l_{87} & l_{88} \end{array}$$

$$\begin{array}{cccc} c_{33}+v_{33}^{(1)} & & & \\ c_{43} & c_{44}+v_{44}^{(2)} & & \\ v_{63}^{(1)} & c_{64}+v_{64}^{(2)} & v_{66}^{(1)}+v_{66}^{(2)} & \\ v_{73}^{(1)} & & v_{76}^{(1)} & v_{77}^{(1)} \end{array} \longrightarrow \begin{array}{cccc} l_{33} & & & \\ l_{43} & l_{44} & & \\ l_{63} & l_{64} & v_{66}^{(3)} & \\ l_{73} & l_{74} & v_{76}^{(3)} & v_{77}^{(3)} \end{array}$$

$$\begin{array}{cc} c_{55} & \\ c_{85} & 0 \end{array} \longrightarrow \begin{array}{cc} l_{55} & \\ l_{85} & v_{88}^{(5)} \end{array}$$

$$\begin{array}{cccc} c_{11} & & & \\ c_{31} & 0 & & \\ c_{61} & 0 & 0 & \\ c_{71} & 0 & 0 & 0 \end{array} \longrightarrow \begin{array}{cccc} l_{11} & & & \\ l_{31} & v_{33}^{(1)} & & \\ l_{61} & v_{63}^{(1)} & v_{66}^{(1)} & \\ l_{71} & v_{73}^{(1)} & v_{76}^{(1)} & v_{77}^{(1)} \end{array}$$

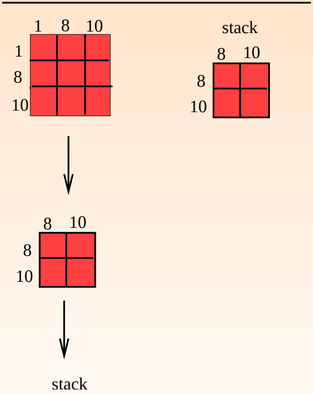
$$\begin{array}{ccc} c_{22} & & \\ c_{42} & 0 & \\ c_{62} & 0 & 0 \end{array} \longrightarrow \begin{array}{ccc} l_{22} & & \\ l_{42} & v_{44}^{(2)} & \\ l_{62} & v_{64}^{(2)} & v_{66}^{(2)} \end{array}$$

Figure: The multifrontal method applied to the matrix $C = \{c_{ij}\}$ from previous slide. The assembly tree is shown. Each vertex shows the transformation from its frontal matrix to the

Cholesky Factorization

Multifrontal method: just sketching

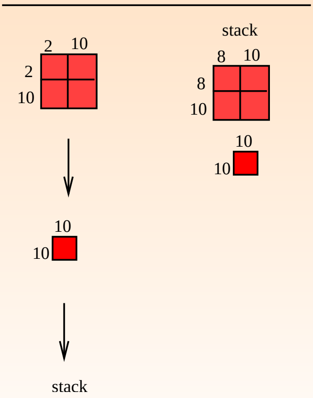
$$\begin{pmatrix}
\ast & & & & & & & \ast & & \ast \\
 & \ast & & & & & & & & \ast \\
 & & \ast & & & & \ast & & & \ast \\
 & & & \ast & & & \ast & & & \\
 & & & & \ast & \ast & & & \ast & \\
 & & & & \ast & \ast & & & f & \ast \\
 & & \ast & \ast & & & \ast & & \ast & f \\
\ast & & & & & & & \ast & & \ast \\
 & & & & \ast & f & \ast & & \ast & f \\
\ast & \ast & \ast & & & \ast & f & \ast & f & \ast
\end{pmatrix}$$



Cholesky Factorization

Multifrontal method: just sketching

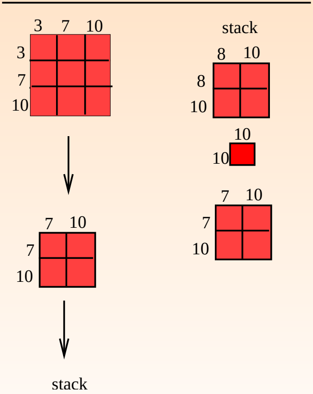
$$\begin{pmatrix}
* & & & & & & & * & & * \\
 & * & & & & & & & & * \\
 & & * & & & & * & & & * \\
 & & & * & & & * & & & \\
 & & & & * & * & & & * & \\
 & & & & * & * & & & \color{red}{f} & * \\
 & & * & * & & & * & & * & \color{red}{f} \\
* & & & & & & & * & & * \\
 & & & & * & \color{red}{f} & * & & * & \color{red}{f} \\
* & * & * & & & * & \color{red}{f} & * & \color{red}{f} & *
\end{pmatrix}$$



Cholesky Factorization

Multifrontal method: just sketching

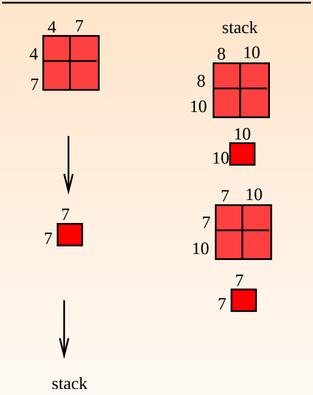
$$\begin{pmatrix}
* & & & & & & & * & & * \\
 & * & & & & & & & & * \\
 & & * & & & & * & & & * \\
 & & & * & & & * & & & \\
 & & & & * & * & & & * & \\
 & & & & * & * & & & f & * \\
 & & * & * & & & * & & * & f \\
* & & & & & & & * & & * \\
 & & & & * & f & * & & * & f \\
* & * & * & & & * & f & * & f & *
\end{pmatrix}$$



Direct methods: Multifrontal method

Multifrontal method: just sketching

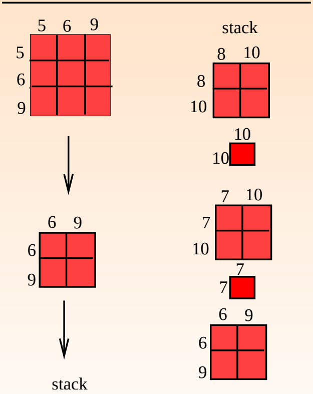
$$\begin{pmatrix}
* & & & & & & & * & & * \\
& * & & & & & & & & * \\
& & * & & & & * & & & * \\
& & & * & & & * & & & \\
& & & & * & * & & & * & \\
& & & & * & * & & & f & * \\
& & * & * & & & * & & * & f \\
* & & & & & & & * & & * \\
& & & & * & f & * & & * & f \\
* & * & * & & & * & f & * & f & *
\end{pmatrix}$$



Cholesky Factorization

Multifrontal method: just sketching

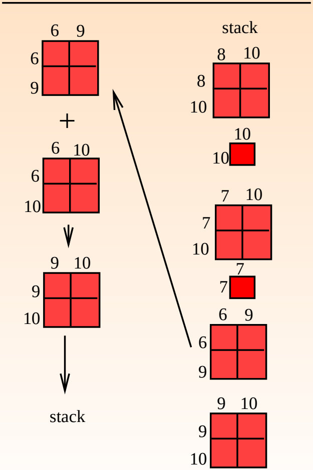
$$\begin{pmatrix}
* & & & & & & & * & & * \\
& * & & & & & & & & * \\
& & * & & & & * & & & * \\
& & & * & & & * & & & \\
& & & & * & * & & & * & \\
& & & & * & * & & & f & * \\
& & * & * & & & * & & * & f \\
* & & & & & & & * & & * \\
& & & & * & f & * & & * & f \\
* & * & * & & & * & f & * & f & *
\end{pmatrix}$$



Cholesky Factorization

Multifrontal method: just sketching

$$\begin{pmatrix}
* & & & & & & & * & & * \\
 & * & & & & & & & & * \\
 & & * & & & & * & & & * \\
 & & & * & & & * & & & \\
 & & & & * & * & & & * & \\
 & & & & * & * & & & f & * \\
 & & * & * & & & * & & * & f \\
* & & & & & & & * & & * \\
 & & & & * & f & * & & * & f \\
* & * & * & & & * & f & * & f & *
\end{pmatrix}$$



Cholesky Factorization

Multifrontal method: algorithm outline

Algorithm (**Basic multifrontal Cholesky factorization)**

Input: *SPD matrix A and its elimination tree.*
Output: *Factor L such that $A = LL^T$.*

```
for $k = 1 : n$ do
    Assemble the frontal matrix $F_k$ using extend-add
    Perform a partial Cholesky factorization of $F_k$ to obtain column $k$ of $L$ and the generated element $V_k$
end for
```

The symmetric multifrontal method satisfies:

Observation

Each generated element V_k is used only once to contribute to a frontal matrix $F_{parent(k)}$. Furthermore, the index list for the frontal matrix F_k is the set of row indices of the nonzeros in column k of the Cholesky factor L.

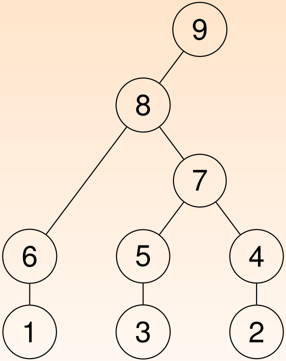
Cholesky Factorization

The multifrontal approach

$$\begin{pmatrix}
* & & & & & * & & * & * \\
 & * & & * & & & * & & \\
 & & * & & * & & & * & \\
 & * & & * & & & f & * & * \\
 & & * & & * & & * & * & \\
* & & & & & * & & * & * \\
 & * & & f & * & & * & f & * \\
* & & * & * & * & * & f & * & f \\
* & & & * & & * & * & f & *
\end{pmatrix}
\qquad
\begin{pmatrix}
* & * & & & & & & * & * \\
* & * & & & & & & * & * \\
 & & * & * & & & & * & \\
 & & * & * & & & * & * & \\
 & & & & * & * & * & & \\
 & & & & * & * & f & * & * \\
 & & & * & * & f & * & f & * \\
* & * & * & * & & * & f & * & f \\
* & * & & & & * & * & f & *
\end{pmatrix}$$

Cholesky Factorization

The multifrontal approach: elimination tree



Cholesky Factorization

The multifrontal approach

$$F_1 = \begin{array}{ccccc} & 1 & 6 & 8 & 9 \\ 1 & * & * & * & * \\ 6 & * & & & \\ 8 & * & & & \\ 9 & * & & & \end{array}, \quad V_1 = \begin{array}{cccc} & 6 & 8 & 9 \\ 6 & * & * & * \\ 8 & * & * & f \\ 9 & * & f & * \end{array}.$$

Here V_1 is dense and f denotes fill-in entries. Similarly, we have

$$F_2 = \begin{array}{cccc} & 2 & 4 & 7 \\ 2 & * & * & * \\ 4 & * & & \\ 7 & * & & \end{array}, \quad V_2 = \begin{array}{ccc} & 4 & 7 \\ 4 & * & f \\ 7 & f & * \end{array}, \quad F_3 = \begin{array}{cccc} & 3 & 5 & 8 \\ 3 & * & * & * \\ 5 & * & & \\ 8 & * & & \end{array}, V_3 = \begin{array}{ccc} & 5 & 8 \\ 5 & * & * \\ 8 & * & * \end{array}.$$

The sparsity pattern of the frontal matrix F_4 is then

$$F_4 = \begin{array}{cccc} & 4 & 8 & 9 \\ 4 & * & * & * \\ 8 & * & & \\ 9 & * & & \end{array} \mathbin{\leftrightarrow\!\!\!\!\!\!\updownarrow} V_2 = \begin{array}{ccccc} & 4 & 7 & 8 & 9 \\ 4 & * & f & * & * \\ 7 & f & * & & \\ 8 & * & & & \\ 9 & * & & & \end{array}, \quad V_4 = \begin{array}{cccc} & 7 & 8 & 9 \\ 7 & * & f & * \\ 8 & f & * & f \\ 9 & * & f & * \end{array},$$

and so on.

Cholesky Factorization

Multifrontal method: assembly tree

- Efficiency is improved by using the assembly tree (larger blocks) and via postordering

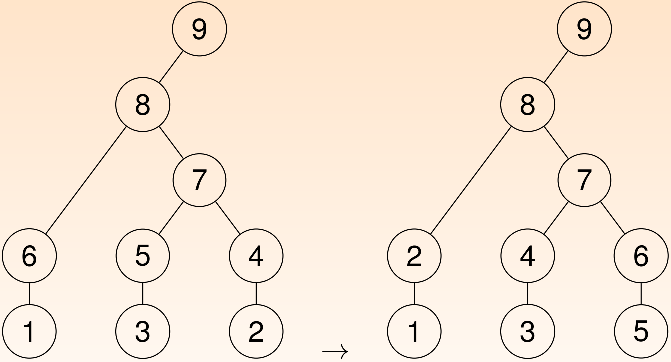
Algorithm (Multifrontal Cholesky factorization using the assembly tree)

Input: *SPD matrix A and its assembly tree.*
Output: *Factor L such that $A = LL^T$.*

```
1: $nelim = 0$                                   ▷ $nelim$ is the number of eliminations performed
2: for $kb = 1 : nsup$ do                        ▷ $nsup$ is the number of supernodes
3:     Assemble the frontal matrix $F_{kb}$; $l$: the number of fully summed variables in
   $F_{kb}$.
4:     Perform a block partial Cholesky factorization of $F_{kb}$ to obtain columns
   $nelim + 1$ to $nelim + l$ of $L$ and the generated element $V_{kb}$
5:     $nelim = nelim + l$
6: end for
```

Cholesky Factorization

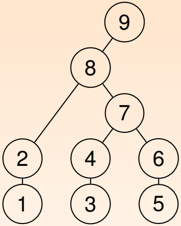
The multifrontal method: + postordering

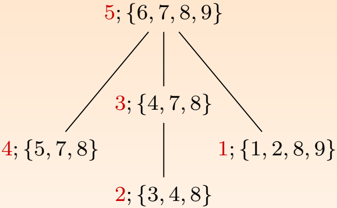


Cholesky Factorization

Multifrontal method: example with postordering and the assembly tree

$$
\begin{array}{c|ccccccccc}
 & 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 \\
\hline
1 & * & * & & & & & & * & * \\
2 & * & * & & & & & & * & * \\
3 & & & * & * & & & & * & \\
4 & & & * & * & & & * & * & \\
5 & & & & & * & * & * & & \\
6 & & & & & * & * & f & * & * \\
7 & & & & * & * & f & * & f & * \\
8 & * & * & * & * & & * & f & * & f \\
9 & * & * & & & & * & * & f & *
\end{array}
$$





5; {6, 7, 8, 9}

4; {5, 7, 8}

3; {4, 7, 8}

1; {1, 2, 8, 9}

2; {3, 4, 8}

Figure: *A sparse matrix, its postordered elimination tree (left) and postordered assembly tree (right).*

Cholesky Factorization

Multifrontal method: example with postordering and the assembly tree

- $nsup = 5$, supernodes are $\{1, 2\}, 3, 4, 5, \{6, 7, 8, 9\}$, supernode eliminated first. The frontal matrix F_1 and generated element V_1 have the structure

$$F_1 = \begin{array}{c} \\ 1 \\ 2 \\ 8 \\ 9 \end{array} \begin{array}{c} \begin{array}{cccc} 1 & 2 & 8 & 9 \end{array} \\ \begin{pmatrix} * & * & * & * \\ * & * & * & * \\ * & * & & \\ * & * & & \end{pmatrix} \end{array}, \quad V_1 = \begin{array}{c} \\ 8 \\ 9 \end{array} \begin{array}{c} \begin{array}{cc} 8 & 9 \end{array} \\ \begin{pmatrix} f & f \\ f & f \end{pmatrix} \end{array},$$

- f denotes fill-in, lower triangular entries stored in practice. Further:

$$F_2 = \begin{array}{c} \\ 3 \\ 4 \\ 8 \end{array} \begin{array}{c} \begin{array}{ccc} 3 & 4 & 8 \end{array} \\ \begin{pmatrix} * & * & * \\ * & * & * \\ * & * & * \end{pmatrix} \end{array}, \quad V_2 = \begin{array}{c} \\ 4 \\ 8 \end{array} \begin{array}{c} \begin{array}{cc} 4 & 8 \end{array} \\ \begin{pmatrix} * & * \\ * & * \end{pmatrix} \end{array}.$$

Cholesky Factorization

Multifrontal method: example with postordering and the assembly tree

The frontal matrix F_3 and generated element V_3 are given by

$$F_3 = \begin{array}{c} \\ 4 \\ 7 \\ 8 \end{array}\!\!\begin{array}{c} \begin{array}{ccc} 4 & 7 & 8 \end{array} \\ \begin{pmatrix} * & * & * \\ * & * & \\ * & & * \end{pmatrix} \end{array} \mathbin{\updownarrow\!\!\!\!\!\!\leftrightarrow} V_2, \quad V_3 = \begin{array}{c} \\ 7 \\ 8 \end{array}\!\!\begin{array}{c} \begin{array}{cc} 7 & 8 \end{array} \\ \begin{pmatrix} * & f \\ f & * \end{pmatrix} \end{array}.$$

Then

$$F_4 = \begin{array}{c} \\ 5 \\ 7 \\ 8 \end{array}\!\!\begin{array}{c} \begin{array}{ccc} 5 & 7 & 8 \end{array} \\ \begin{pmatrix} * & * & * \\ * & * & \\ * & & * \end{pmatrix} \end{array}, \quad V_4 = \begin{array}{c} \\ 7 \\ 8 \end{array}\!\!\begin{array}{c} \begin{array}{cc} 7 & 8 \end{array} \\ \begin{pmatrix} * & f \\ f & * \end{pmatrix} \end{array},$$

and, finally, with $kb = 5$ we have

$$F_5 = \begin{array}{c} \\ 6 \\ 7 \\ 8 \\ 9 \end{array}\!\!\begin{array}{c} \begin{array}{cccc} 6 & 7 & 8 & 9 \end{array} \\ \begin{pmatrix} * & & * & * \\ & * & & * \\ * & & * & \\ * & * & & * \end{pmatrix} \end{array} \mathbin{\updownarrow\!\!\!\!\!\!\leftrightarrow} V_4 \mathbin{\updownarrow\!\!\!\!\!\!\leftrightarrow} V_3 \mathbin{\updownarrow\!\!\!\!\!\!\leftrightarrow} V_1.$$

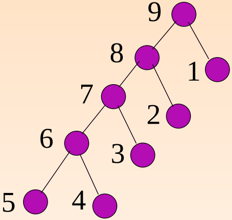
Cholesky Factorization

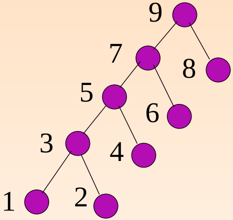
The multifrontal method: memory considerations

-
- If the vertices of the assembly tree are postordered, the generated elements required at each stage are the most recently computed ones amongst those that have not yet been assembled: see the next slide
- This makes convenient to use a stack to store the generated elements.
- The memory demands of the multifrontal method can be very large. Auxiliary out-of-core storage can be used.
- The ordering of the children of a vertex (choice of postordering) in the assembly tree can significantly affect the required stack size: next slide

Cholesky Factorization

The multifrontal method: postordering choice demonstration





- Left: Maximum stack size may be 1×1+2×2+3×3+4×4 (note that we store only triangular matrices)
- Right: Maximum stack size may be 4×4
- Conclusion: Postorderings may strongly influence intermediate memory demands

Cholesky Factorization

The multifrontal method: stack considerations

Theorem

Let $\mathcal{T}$ be postordered. Assume each computed generated element (V_x) is pushed onto a stack. Then when constructing the (frontal matrix) F_j, the required generated elements are on the top of the stack. They can be directly popped from the stack and assembled into F_j.

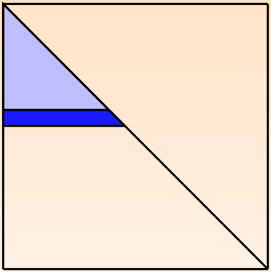
Proof sketch: 1) Vertices of each subtree of the postordered $\mathcal{T}$ form an interval. 2) Denote $c_l,\ l = 1, \ldots, s$ children of j in $\mathcal{T}$. 3) Each c_l is the root of a subtree $\mathcal{T}(c_l)$. 4) Once the frontal matrix F_{c_l} for a leaf of $\mathcal{T}(c_l)$ is constructed, all its children have been processed and the generated V_{c_l} is pushed onto the stack. 5) That is, all subtrees $\mathcal{T}_{c_l},\ l = 1, \ldots, s$ are fully assembled into the generated elements before F_j can be constructed. 6) If F_j is ready to be assembled (step j), the s generated $V_{c_l},\ l = 1, \ldots, s$ are on the top of the stack.

Cholesky Factorization

The multifrontal method: summary

- Right-looking (submatrix) method
- Does not form the Schur complement directly: the updates are moved to a stack as dense matrices and used when needed.
- The processing order is based on the elimination/assembly tree
- To have the generated elements readily available, stack is used, enabled by a postordering.
- Specific postorderings may minimize intermediate memory size.
- Tree and node parallelism combined in parallel implementation.
- Modifications and enhancements of the basic concept can be used.

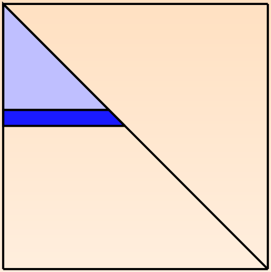
Sparse Cholesky factorizations: up-looking factorization



- An alternative for sparse matrices is to compute L one row at a time. This is sometimes called an up-looking factorization.
- Asymptotically optimal, but difficult to incorporate high level BLAS.

Cholesky Factorization

Sparse Cholesky factorizations: up-looking factorization



- The following relation holds for the i-th row of L

$$L^T_{i,1:i-1} = L^{-1}_{1:i-1,1:i-1} A_{1:i-1,i} \quad \text{with} \quad l^2_{ii} = a_{ii} - L_{i,1:i-1} L^T_{i,1:i-1}.$$

$L^{-1}_{1:i-1,1:i-1}$ can be applied as solving the triangular system

$$L_{1:i-1,1:i-1} y = A_{1:i-1,i},$$

and setting $L^T_{i,1:i-1} = y$.

- How can be sparse triangular systems solved?

Cholesky Factorization

Triangular systems and sparsity pattern of L

- Using graphs to find the solution pattern to solve $Ly = b$.
- The key to see this is $\mathcal{G}(L^T)$ and the reachabilities of vertices in $\mathcal{S}\{b\} = \{2, 4\} \rightarrow \mathcal{S}\{y\} = \{2, 4, 5, 6\}$.

$$L = \begin{array}{c} \\ 1 \\ 2 \\ 3 \\ 4 \\ 5 \\ 6 \end{array} \begin{array}{c} \begin{array}{cccccc} 1 & 2 & 3 & 4 & 5 & 6 \end{array} \\ \left(\begin{array}{cccccc} * & & & & & \\ * & * & & & & \\ & & * & & & \\ * & & * & * & & \\ & & & * & * & \\ & * & * & & * & * \end{array} \right) \end{array} \qquad b = \begin{array}{c} \\ 1 \\ 2 \\ 3 \\ 4 \\ 5 \\ 6 \end{array} \begin{array}{c} 1 \\ \left(\begin{array}{c} \\ * \\ \\ * \\ \\ \\ \end{array} \right) \end{array}$$

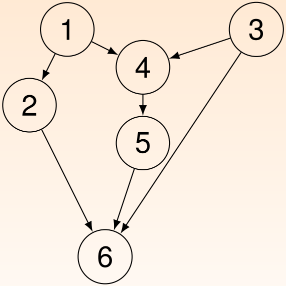


Figure: *An example to illustrate L, b and $\mathcal{G}(L^T)$.*

Cholesky Factorization

Triangular systems and sparsity pattern of L

Theorem

Consider a sparse lower triangular matrix L and the DAG $\mathcal{G}(L^T)$ with vertex set $\{1, 2, \ldots, n\}$ and edge set $\{(j \longrightarrow i) \mid l_{ij} \neq 0\}$. The sparsity pattern $\mathcal{S}\{y\}$ of the solution y of the system $Ly = b$ is the set of all vertices reachable in $\mathcal{G}(L^T)$ from $\mathcal{S}\{b\}$.

Proof.

From factorization Algorithm and assuming the non-cancellation assumption, we see that (a) if $b_i \neq 0$ then $y_i \neq 0$ and (b) if for some $j < i$, $y_j \neq 0$ and $l_{ij} \neq 0$ then $y_i \neq 0$. These two conditions can be expressed as a graph transversal problem in $\mathcal{G}(L^T)$. (a) adds all vertices in $\mathcal{S}\{b\}$ to the set of visited vertices and (b) states that if vertex j has been visited then all its neighbours in $\mathcal{G}(L^T)$ are added to the set of visited vertices. Thus $\mathcal{S}\{y\} = \mathcal{R}each(\mathcal{S}\{b\}) \cup \mathcal{S}\{b\}$. □ □

Cholesky Factorization

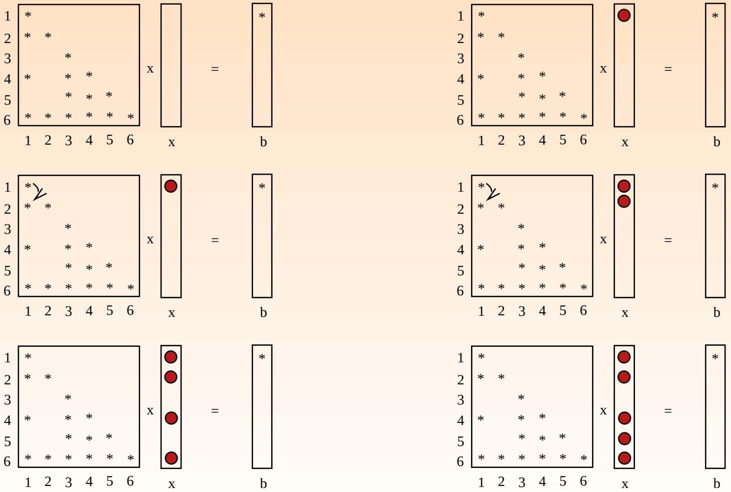
Triangular systems and sparsity pattern of L: example

$$\begin{pmatrix} * & & & & \\ & * & & & \\ * & & * & & \\ & & & * & \\ & & * & & * \end{pmatrix} \begin{pmatrix} {\color{red}*} \\ \\ {\color{red}*} \\ \\ {\color{red}*} \end{pmatrix} = \begin{pmatrix} * \\ \\ \\ \\ \\ \end{pmatrix} \tag{34}$$

- The only nonzero of the right-hand side implies the three nonzeros in the solution (in red)

Cholesky Factorization

Triangular systems and sparsity pattern of L: another example



Cholesky Factorization

Up-looking Cholesky - further comments

- Sparse row Cholesky factorization is based on the repeated solution of triangular linear systems.
- Theorem above can be used to determine the sparsity pattern of row i at Step 3, that is, by finding all the vertices that are reachable in $\mathcal{G}(L^T_{1:j-1,1:j-1})$ from the set $\{i \,|\, a_{ij} \neq 0,\ i < j\}$.
- A depth-first search of $\mathcal{G}(L^T_{1:j-1,1:j-1})$ determines the vertices in row sparsity patterns in topological order, and performing the numerical solves in that order correctly preserves the numerical dependencies in the factorization.
- An option is to use the row subtrees using $\mathcal{T}(A)$ to get row sparsity patterns.

Cholesky Factorization

Up-looking Cholesky: algorithm

Algorithm (Sparse up-looking Cholesky factorization)

Input: *SPD matrix A.*
Output: *Factor L such that $A = LL^T$.*

```
$l_{11} = (a_{11})^{1/2}$
for $i = 2 : n$ do
    Find $\mathcal{S}\{L_{i,1:i-1}\}$                              ▷ Sparsity pattern of $L_{i,1:i-1}$
    $L^T_{i,1:i-1} = L^{-1}_{1:i-1,1:i-1} A_{1:i-1,i}$             ▷ Sparse triangular solve
    $l_{ii} = a_{ii} - L_{i,1:i-1} L^T_{i,1:i-1}$
    $l_{ii} = (l_{ii})^{1/2}$
end for
```

Outline

1. Introduction
2. Introductory notation and terminology
3. Factorizations
4. Graphs and sparse matrices
5. Sparse matrices and data structures
6. Symbolic factorizations
7. Symbolic Cholesky factorization
8. Cholesky factorization
9. Sparse LU factorization
10. Stability, ill-conditioning, indefiniteness
11. Blocks in the input matrix
12. Sparse Least Squares and QR factorization
13. Reorderings
14. Algebraic preconditioning
15. Incomplete factorizations
16. Sparse approximate inverses

Sparse LU: models and methods

Graphs and LU factorization

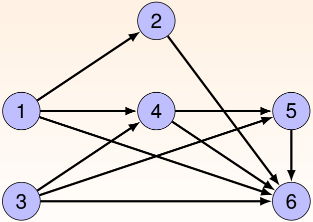
- From structural descrption of a sparse symmmetric (Cholesky) factorization to the symbolic LU factorization.
- Differences (roughly):
  - Two factors: more general graph models (directed, bipartite) needed to describe A and the factors
  - Problems with factorizability: symbolic and numerical steps cannot be always separated
  - Sometimes stronger assumptions needed, sometimes on-the-fly changes: pivoting

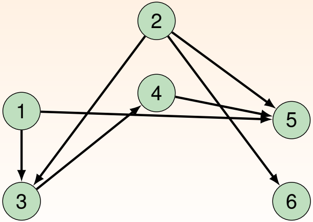
Sparse LU factorization of generally nonsymmetric matrices

LU factorization: first symbolic model: DAGs

$$\begin{array}{c|cccccc} & 1 & 2 & 3 & 4 & 5 & 6 \\ \hline 1 & * & & * & & * & \\ 2 & * & * & & & * & * \\ 3 & & & * & * & & \\ 4 & * & & * & * & & \\ 5 & & & * & & * & \\ 6 & * & * & & & & * \end{array} \qquad \begin{array}{c|cccccc} & 1 & 2 & 3 & 4 & 5 & 6 \\ \hline 1 & * & & * & & * & \\ 2 & * & * & f & & * & * \\ 3 & & & * & * & & \\ 4 & * & & * & * & f & \\ 5 & & & * & f & * & \\ 6 & * & * & f & f & f & * \end{array}$$

- Directed acyclic graphs for the factors capture their structure. We use $G(L^T)$ (L by columns, left) and $G(U)$ (U by rows, right).





Sparse LU: models and methods

LU factorization and DAGs: replication

- The two DAGs express the structure of factors with the fill-in.
- Symmetric replication can be generalized to the nonsymmetric case. Note that we do not remind factorizability at this moment.

Observation

If $i > j$ and $u_{ji} \neq 0$ then the **column replication principle** *states*

$$\mathcal{S}\{L_{i:n,j}\} \subseteq \mathcal{S}\{L_{i:n,i}\},$$

that is, the pattern of column j of L (rows i to n) is replicated in the pattern of column i of L.
Analogously, if $i > j$ and $l_{ij} \neq 0$ then the **row replication principle** *states*

$$\mathcal{S}\{U_{j,i:n}\} \subseteq \mathcal{S}\{U_{i,i:n}\},$$

that is, the pattern of row j of U (columns i to n) is replicated in the pattern of row i of U.

Sparse LU: models and methods

LU factorization and DAGs: replication example

$$
\begin{array}{c|ccccccc}
 & 1 & 2 & 3 & 4 & 5 & 6 & 7 \\
\hline
1 & * & * & & & & & \\
2 & & * & * & * & & & * \\
3 & * & & * & & * & & \\
4 & * & & & * & & & \\
5 & & & * & & * & & \\
6 & & & & * & * & * & * \\
7 & * & & & & & & *
\end{array}
\qquad
\begin{array}{c|ccccccc}
 & 1 & 2 & 3 & 4 & 5 & 6 & 7 \\
\hline
1 & * & * & & & & & \\
2 & & * & * & * & & & * \\
3 & * & f & * & & * & & \\
4 & * & f & & * & & & \\
5 & & & * & & * & & \\
6 & & & & * & * & * & * \\
7 & * & f & & & & & *
\end{array}
\qquad
\begin{array}{c|ccccccc}
 & 1 & 2 & 3 & 4 & 5 & 6 & 7 \\
\hline
1 & * & * & & & & & \\
2 & & * & * & * & & & * \\
3 & * & f & * & f & * & & f \\
4 & * & f & f & * & & & f \\
5 & & & * & & * & & \\
6 & & & & * & * & * & * \\
7 & * & f & f & f & & & *
\end{array}
$$

- Column and row replication principles in the sparse LU factorizations.
- Left: the matrix A. Centre: showing one column replication. Right: also a row replication.
- Filled entries not involved so far and resulting from subsequent steps of the factorization are denoted in black.

Sparse LU: models and methods

Basic sparse LU (numerical) factorization: algorithm

- Assumed A being factorizable so that pivoting is not needed.

Algorithm (**Basic sparse LU factorization**)

Input: *Nonsymmetric and factorizable matrix* $A = L_A + D_A + U_A$.
Output: *LU factorization* $A = LU$.

```
$L = I + L_A$                                        ▷ Strictly lower triangular part of $A$
$U = D_A + U_A$                                      ▷ Diagonal plus strictly upper triangular part of $A$
for $k = 1 : n - 1$ do
    for $i \in \{i > k \,|\, l_{ik} \neq 0\}$ do
        $l_{ik} = l_{ik}/u_{kk}$
        $U_{i,i:n} = U_{i,i:n} - U_{k,i:n} l_{ik}$   ▷ Update row $i$ of $U$
    end for
    for $j \in \{j > k \,|\, u_{kj} \neq 0\}$ do
        $L_{j+1:n,j} = L_{j+1:n,j} - L_{j+1:n,k} u_{kj}$   ▷ Update column $j$ of $L$
    end for
end for
```

Sparse LU: models and methods

Recursive application of replications

- Symmetric factorization: the recursive replications driven by the first nonzero entries in columns of L (parents).
- In LU it is more interesting ☺: for L we use directed paths in U and for the rows of U we use directed paths in $\mathcal{G}(L^T)$.

Theorem

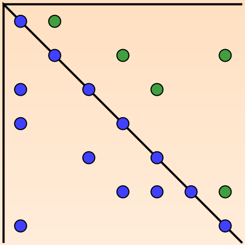
Assume that for some $k < j$ there is a directed path $k \xRightarrow{\mathcal{G}(U)} j$. Then

$$\mathcal{S}\{L_{j:n,k}\} \subseteq \mathcal{S}\{L_{j:n,j}\}.$$

Moreover, if $l_{ik} \neq 0$ for some $i > j$ then $l_{is} \neq 0$ for all vertices s on this path.

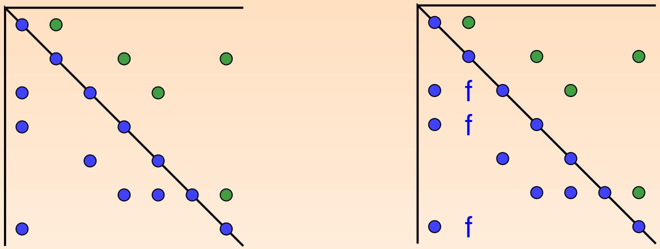
Sparse LU: models and methods

Column replication in LU



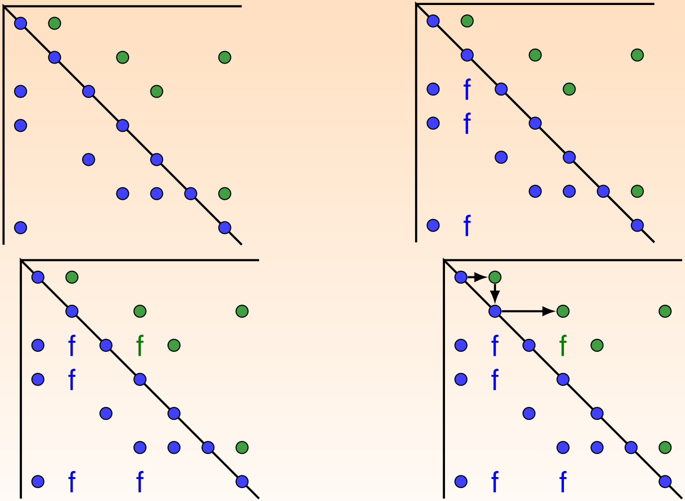
Sparse LU: models and methods

Column replication in LU



Sparse LU: models and methods

Column replication in LU



Sparse LU: models and methods

Sparse LU: both replications (for columns and rows)

Theorem

If $a_{ij} = 0$ and $i > j$ then there is a filled entry $l_{ij} \neq 0$ if and only if there exists $k < j$ such that $a_{ik} \neq 0$ and there is a directed path $k \xrightarrow{\mathcal{G}(U)} j$.

Theorem

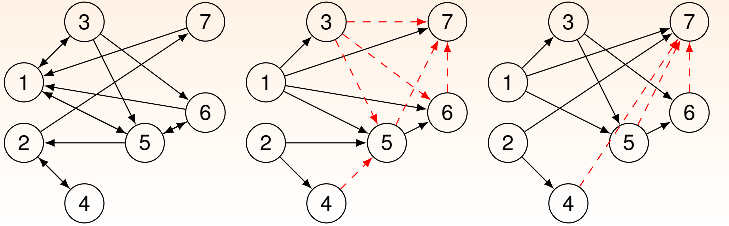
If $a_{ij} = 0$ and $i < j$ then there is a filled entry $u_{ij} \neq 0$ if and only if there exists $k < i$ such that $a_{kj} \neq 0$ and there is a directed path $k \xrightarrow{\mathcal{G}(L^T)} i$.

Sparse LU: models and methods

Sparse LU: replications: funny game to detect paths

- The path $1 \to 3 \to 5 \to 6$ in $\mathcal{G}(U)$. It implies the fill-in in L, first in column 3, then in columns 5 and 6.
- $2 \to 4 \to 5 \to 6$ in $\mathcal{G}(L^T) \Rightarrow$ fill-in at $(4,7)$, $(5,7)$ and $(6,7)$ in U.

$$\begin{array}{c|ccccccc} & 1 & 2 & 3 & 4 & 5 & 6 & 7 \\ \hline 1 & * & & * & & * & & \\ 2 & & * & & * & & & * \\ 3 & * & & * & & * & * & \\ 4 & & * & & * & & & \\ 5 & * & * & & & * & * & \\ 6 & * & & & & * & * & \\ 7 & * & & & & & & * \end{array} \qquad \begin{array}{c|ccccccc} & 1 & 2 & 3 & 4 & 5 & 6 & 7 \\ \hline 1 & * & & * & & * & & \\ 2 & & * & & * & & & * \\ 3 & * & & * & & * & * & \\ 4 & & * & & * & & & f \\ 5 & * & * & f & f & * & * & f \\ 6 & * & & f & & * & * & f \\ 7 & * & & f & & f & f & * \end{array}$$



Sparse LU: models and methods

Transitive reduction

- To employ $\mathcal{G}(L^T)$ and $\mathcal{G}(U)$ in efficient algorithms, they need to be simplified. They must be sparser and preserve reachability.
- Similarly as the elimination tree $\mathcal{T}$ is a simplification.
- In LU, the analogy is the transitive reduction of a DAG.
- A subgraph $\mathcal{G}^0 = (\mathcal{V}, \mathcal{E}^0)$ is a transitive reduction of $\mathcal{G} = (\mathcal{V}, \mathcal{E})$ if the following conditions hold:

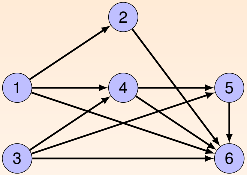
  $(T1)$ there is a path from vertex i to vertex j in $\mathcal{G}$ if and only if there is a path from i to j in $\mathcal{G}^0$ (reachability condition), and

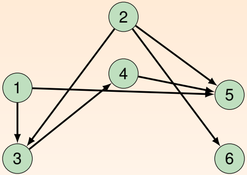
  $(T2)$ there is no subgraph with vertex set $\mathcal{V}$ that satisfies $(T1)$ and has fewer edges (minimality condition).

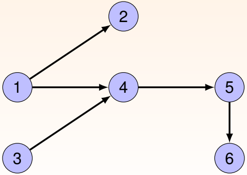
Sparse LU factorization of generally nonsymmetric matrices

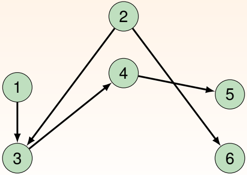
Transitive reduction

$$\begin{array}{c|cccccc} & 1 & 2 & 3 & 4 & 5 & 6 \\ \hline 1 & * & & * & & * & \\ 2 & * & * & & & * & * \\ 3 & & & * & * & & \\ 4 & * & & * & * & & \\ 5 & & & * & & * & \\ 6 & * & * & & & & * \end{array} \qquad \begin{array}{c|cccccc} & 1 & 2 & 3 & 4 & 5 & 6 \\ \hline 1 & * & & * & & * & \\ 2 & * & * & f & & * & * \\ 3 & & & * & * & & \\ 4 & * & & * & * & f & \\ 5 & & & * & f & * & \\ 6 & * & * & f & f & f & * \end{array}$$









Sparse LU: models and methods

Theorem

Let $\mathcal{G}$ be a DAG. The transitive reduction $\mathcal{G}^0$ of $\mathcal{G}$ is unique, is the subgraph that has an edge for every path in $\mathcal{G}$ and has no proper subgraph with this property.

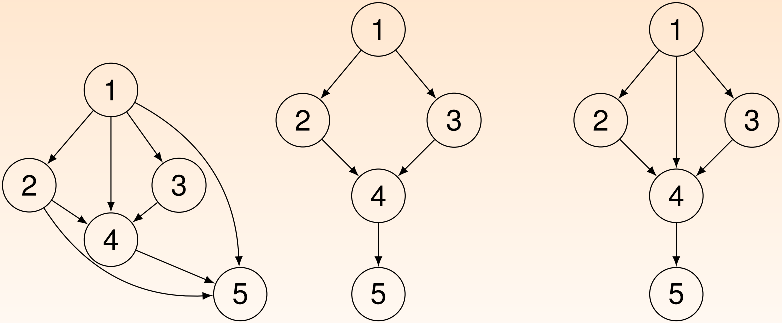
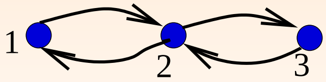


Figure: *Example to show the transitive reduction of a DAG. $\mathcal{G}$ is on the left, its transitive reduction $\mathcal{G}^0$ is in the centre, and one possible $\mathcal{G}'$ that is equireachable with $\mathcal{G}$ is on the right.*

Transitive reduction

- In general, the Theorem above does not hold for general graphs.
- In the following figure, its transitive reduction is the loop interconnecting vertices $1, 2$ and 3.
- The transitive reduction has only three edges and is not unique!



Sparse LU: models and methods

Transitive reduction for A symmetric

- If $\mathcal{S}\{A\}$ is symmetric, the role of the transitive reduction is played by the elimination tree (as we might guess ☺).

Theorem

If A is symmetrically structured then the transitive reduction of the DAG $\mathcal{G}(L^T)$ (= $\mathcal{G}(U)$) is the elimination tree $\mathcal{T}(A)$.

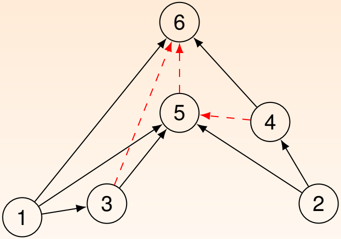
$$
\begin{array}{c|cccccc|}
 & 1 & 2 & 3 & 4 & 5 & 6 \\
\hline
1 & * & & * & & * & * \\
2 & & * & & * & * & \\
3 & * & & * & & * & * \\
4 & & * & & * & * & * \\
5 & * & * & * & * & * & * \\
6 & * & & * & * & * & *
\end{array}
\qquad
\begin{array}{c|cccccc|}
 & 1 & 2 & 3 & 4 & 5 & 6 \\
\hline
1 & * & & * & & * & * \\
2 & & * & & * & * & \\
3 & * & & * & & * & * \\
4 & & * & & * & * & * \\
5 & \bigstar & \bigstar & * & * & * & * \\
6 & \bigstar & & \bigstar & \bigstar & * & *
\end{array}
$$

Entries denoted $\bigstar$ do not belong to the transitive reduction of L.

Sparse LU: models and methods

Transitive reduction for A symmetric: example with fill-in

$$\begin{array}{c|cccccc}
 & 1 & 2 & 3 & 4 & 5 & 6 \\
\hline
1 & * & & * & & * & * \\
2 & & * & & * & * & \\
3 & * & & * & & * & f \\
4 & & * & & * & f & * \\
5 & * & * & * & f & * & f \\
6 & * & & f & * & f & *
\end{array}$$



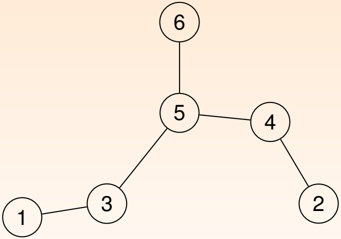


Figure: *The sparsity patterns of $L+U$ of a symmetrically structured A, the DAG $\mathcal{G}(L^T)$ (left), $\mathcal{T}(A)$ (right). $\mathcal{T}(A)$ is the transitive reduction of $\mathcal{G}(L^T)$.*

Sparse LU: models and methods

Transitive reduction may be expensive to obtain

- Namely, obtaining exact transitive reduction of a DAG can be expensive.
- Instead, approximate reductions without the minimality condition may be computed.
- A directed graph $\mathcal{G}'$ with the same vertex set as $\mathcal{G}$ that satisfies condition $(T1)$ is said to be equireachable with $\mathcal{G}$.
- Thus we look for something useful between the DAG and transitive reduction.

Theorem

Assume $\mathcal{G}'$ is equireachable with $\mathcal{G}(U)$ and for some $k < j$ there is a directed path $k \overset{\mathcal{G}'}{\Longrightarrow} j$. Then the replication theorem can use paths in these reduced DAGs. Moreover, row replication: if $l_{ik} \neq 0$ for some $i > j$ then $l_{is} \neq 0$ for all vertices s on the directed path.

Sparse LU: models and methods

Equireachability and efficiency

- Equireachability enables sparse triangular linear systems to be solved more efficiently since they are sparser.
- The necessary and sufficient conditions for the fill-in from above:

Theorem

If $a_{ij} = 0$ and $i > j$ then there is a filled entry $l_{ij} \neq 0$ if and only if there exists $k < j$ such that $a_{ik} \neq 0$ and a directed path $k \xRightarrow{\mathcal{G}'(U)} j$, where $\mathcal{G}'(U)$ is equireachable with $\mathcal{G}(U)$.

Theorem

If $a_{ij} = 0$ and $i < j$ then there is a filled entry $u_{ij} \neq 0$ if and only if there exists $k < i$ such that $a_{kj} \neq 0$ and a directed path $k \xRightarrow{\mathcal{G}'(L^T)} i$, where $\mathcal{G}'(L^T)$ is equireachable with $\mathcal{G}(L^T)$.

Sparse LU: models and methods

Equireachability: example

Figure depicts $\mathcal{G}(U)$ and $\mathcal{G}'(U)$ for the matrix in Figure above.

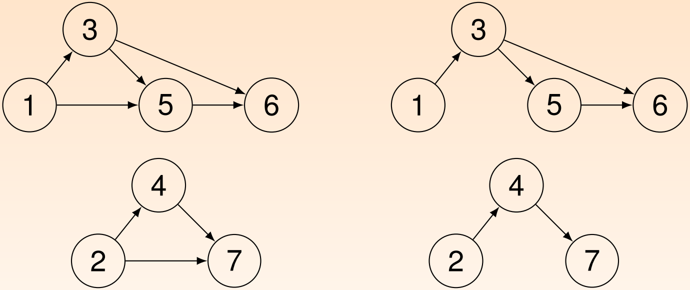


Figure: *The DAG $\mathcal{G}(U)$ (left) and $\mathcal{G}'(U)$ which is equireachable with $\mathcal{G}(U)$ (right).*

Sparse LU: models and methods

Column sparsity patterns (for L)

- Getting to the factor (column/row) sparsity patterns
- Schur complement description

$$\mathcal{S}\{L_{j:n,j}\} = \mathcal{S}\{A_{j:n,j}\} \bigcup_{k<j, u_{kj}\neq 0} \mathcal{S}\{L_{j:n,k}\}, \quad 1 \le j \le n.$$

- But, similarly as in the symmetric case, not all the terms in this union are needed to obtain $\mathcal{S}\{L_{j:n,j}\}$.

Sparse LU: models and methods

Column sparsity patterns (for L)

Theorem

If $\mathcal{G}'(U)$ is equireachable with $\mathcal{G}(U)$ then

$$\mathcal{S}\{L_{j:n,j}\} = \mathcal{S}\{A_{j:n,j}\} \bigcup_{(k\to j)\in\mathcal{E}(\mathcal{G}'(U))} \mathcal{S}\{L_{j:n,k}\}, \quad 1 \le j \le n.$$

Proof.

Consider an edge $(k \to j)$ in $\mathcal{G}(U)$ but not in $\mathcal{G}'(U)$. Repeatedly applying replication results along the directed path $k \xRightarrow{\mathcal{G}'(U)} j$, we see that $L_{j:n,k}$ is contained in the right-hand side of the predicted structure and therefore $\mathcal{S}\{L_{j:n,j}\}$ is contained in the right-hand side of this structure as well. Because the right-hand side of the formula is trivially contained in the left-hand side, the result follows. □ □

Sparse LU: models and methods

Row sparsity patterns (for U)

An analogous result holds for the rows of U.

Theorem

If $\mathcal{G}'(L)$ is equireachable with $\mathcal{G}(L)$ then

$$\mathcal{S}\{U_{i,i:n}\} = \mathcal{S}\{A_{i,i:n}\} \bigcup_{(k \to i) \in \mathcal{E}(\mathcal{G}'(L^T))} \mathcal{S}\{U_{k,i:n}\}, \quad 1 \leq i \leq n.$$

- But, we do not have the DAGs, nor their equireachable reductions to use this.

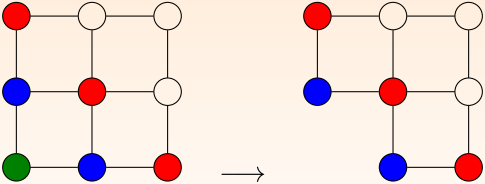
Sparse LU: models and methods

Pruning of the elimination DAGs

- Getting cheaper equireachable DAGs: pruning

Theorem

If for some $j < s$ both $l_{sj} \neq 0$ and $u_{js} \neq 0$, then there are no edges $(j \to k)$ with $k > s$ in the transitive reductions of $\mathcal{G}(U)$ and $\mathcal{G}(L^T)$.



Pruning in $\mathcal{G}(L^T)$: green and blue nodes represent edges

Pruning of the elimination DAGs

Theorem

If for some $j < s$ both $l_{sj} \neq 0$ and $u_{js} \neq 0$, then there are no edges $(j \to k)$ with $k > s$ in the transitive reductions of $\mathcal{G}(U)$ and $\mathcal{G}(L^T)$.

- The theorem implies that if for some $s > 1$ there are edges

$$j \xrightarrow{\mathcal{G}(L^T)} s \quad \text{and} \quad j \xrightarrow{\mathcal{G}(U)} s,$$

  then all edges $(j \to k)$ in $\mathcal{G}(U)$ and $\mathcal{G}(L^T)$ with $k > s$ can be pruned.
- Resulting DAGs $\mathcal{G}'(U)$ and $\mathcal{G}'(L^T)$ are equireachable with $\mathcal{G}(U)$ and $\mathcal{G}(L^T)$, respectively.
- This is called symmetric pruning.

Sparse LU: models and methods

More general pruning of the elimination DAGs

- If for some $s > 1$ there are paths

$$j \xRightarrow{\mathcal{G}(L^T)} s \quad \text{and} \quad j \xRightarrow{\mathcal{G}(U)} s,$$

  then for all $k > s$ symmetric path pruning removes the edges $(j \rightarrow k)$ from $\mathcal{G}(U)$ and $\mathcal{G}(L^T)$.

- The key is to find such paths. This may be expensive.

Sparse LU: models and methods

Pruning of the elimination DAGs: example

- The matrix (after symmetric pruning) in the centre: the entries from positions $(4, 6)$ and $(6, 4)$ have been pruned.
- Both matrices have the same sets of reachable vertices in $\mathcal{G}(L^T)$ and $\mathcal{G}(U)$. This suggests how to find $\mathcal{G}'(L^T)$ and $\mathcal{G}'(U)$ that are equireachable with $\mathcal{G}(L^T)$ and $\mathcal{G}(U)$, respectively.

$$
\begin{array}{c|cccccc|}
 & 1 & 2 & 3 & 4 & 5 & 6 \\
1 & * & & * & & & * \\
2 & * & * & & & & \\
3 & & & * & & & * \\
4 & & * & & * & * & * \\
5 & & & & * & * & * \\
6 & * & & & * & * & *
\end{array}
\quad
\begin{array}{c|cccccc|}
 & 1 & 2 & 3 & 4 & 5 & 6 \\
1 & * & & * & & & * \\
2 & * & * & & & & \\
3 & & & * & & & * \\
4 & & * & & * & * & \\
5 & & & & * & * & * \\
6 & * & & & & * & *
\end{array}
\quad
\begin{array}{c|cccccc|}
 & 1 & 2 & 3 & 4 & 5 & 6 \\
1 & * & & * & & & \\
2 & * & * & & & & \\
3 & & & * & & & * \\
4 & & * & & * & * & \\
5 & & & & * & * & * \\
6 & & & & & * & *
\end{array}
$$

Figure: *$\mathcal{S}\{L + U\}$ (left), reduced sparsity pattern from symmetric pruning (centre), after additional symmetric path pruning ($1 \Rightarrow 6$).*

Sparse LU: models and methods

Factorization by bordering: useful approach to apply this theory

- Pruning for this can be done one by one (one column, one row, another column ...) on-the-fly as we factorize. And then more factorization ways can be fine ☺.
- Factorization by bordering: $\mathcal{S}\{L\}$ by rows, $\mathcal{S}\{U\}$ by columns.
- Assume the sparsity patterns of the first $k-1$ rows of L and the first $k-1$ columns of U $(1 < k \leq n)$ have been computed.
- At step k, the matrix $A_{1:k,1:k}$ is

$$\begin{pmatrix} A_{1:k-1,1:k-1} & A_{1:k-1,k} \\ A_{k,1:k-1} & a_{kk} \end{pmatrix} = \begin{pmatrix} L_{1:k-1,1:k-1} & 0 \\ L_{k,1:k-1} & 1 \end{pmatrix} \begin{pmatrix} U_{1:k-1,1:k-1} & U_{1:k-1,k} \\ 0 & u_{kk} \end{pmatrix}$$

Sparse LU: models and methods

Factorization by bordering

- Factorization by bordering: $\mathcal{S}\{L\}$ by rows, $\mathcal{S}\{U\}$ by columns.
- Assume the sparsity patterns of the first $k-1$ rows of L and the first $k-1$ columns of U $(1 < k \leq n)$ have been computed.
- At step k, the matrix $A_{1:k,1:k}$ is

$$\begin{pmatrix} A_{1:k-1,1:k-1} & A_{1:k-1,k} \\ A_{k,1:k-1} & a_{kk} \end{pmatrix} = \begin{pmatrix} L_{1:k-1,1:k-1} & 0 \\ L_{k,1:k-1} & 1 \end{pmatrix} \begin{pmatrix} U_{1:k-1,1:k-1} & U_{1:k-1,k} \\ 0 & u_{kk} \end{pmatrix}$$

- Equating terms for the $(2,1)$ block, row k of L satisfies

$$L_{k,1:k-1}U_{1:k-1,1:k-1} = A_{k,1:k-1},$$

or, equivalently, if y denotes the off-diagonal part of the column k of L^T then it is the solution of the lower triangular system

$$U^T_{1:k-1,1:k-1}y = A^T_{k,1:k-1}.$$

Sparse LU: models and methods

Factorization by bordering

- The sparsity pattern of y (row vector in L) is the set of all vertices reachable in the DAG $\mathcal{G}(U_{1:k-1,1:k-1})$ (or in a graph that is equireachable with it) from the nonzeros in $A_{k,1:k-1}$.
- Similarly, for rows of U: their sparsity patterns can be determined by searching the DAG $\mathcal{G}(L^T_{1:k-1,1:k-1})$.
- The diagonal entry u_{kk} is then computed as $a_{kk} - L_{k,1:k-1}U_{1:k-1,k}$.
- And the DAGs to determine the column/row sparsity patterns can be pruned on-the-fly.

Sparse LU: models and methods

Another graph model: column elimination tree

- An attractive idea for constructing $\mathcal{S}\{L+U\}$ is based on using the column elimination tree $\mathcal{T}(A^TA)$.

Theorem

Assume all the diagonal entries of A are nonzero and let $\widehat{L}\widehat{L}^T$ be the Cholesky factorization of A^TA. Then for any row permutation matrix P such that $PA = LU$

$$\mathcal{S}\{L+U\} \subseteq \mathcal{S}\{\widehat{L}+\widehat{L}^T\}.$$

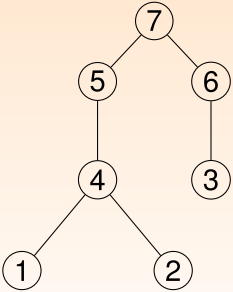
- This is very strong result (theoretically).

Sparse LU: models and methods

Another graph model: column elimination tree: example

- Standard elimination tree $\mathcal{T}(A)$.

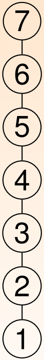
$$\begin{array}{cc}
 & \begin{array}{ccccccc} 1 & 2 & 3 & 4 & 5 & 6 & 7 \end{array} \\
\begin{array}{c} 1 \\ 2 \\ 3 \\ 4 \\ 5 \\ 6 \\ 7 \end{array} &
\left(\begin{array}{ccccccc}
* & & & * & & & \\
* & * & f & * & & & \\
 & & * & & & * & \\
* & * & & * & f & & * \\
 & * & * & f & * & * & * \\
 & & * & & & * & \\
* & & & * & * & f & *
\end{array}\right)
\end{array}$$



Sparse LU: models and methods

- The elimination tree $\mathcal{T}(A^TA)$: much more dependencies, much less parallelism.

$$
\begin{array}{c|ccccccc}
 & 1 & 2 & 3 & 4 & 5 & 6 & 7 \\
\hline
1 & * & * & & * & * & & * \\
2 & * & * & * & * & * & * & * \\
3 & & * & * & f & * & * & * \\
4 & * & * & f & * & * & f & * \\
5 & * & * & * & * & * & * & * \\
6 & & * & * & f & * & * & * \\
7 & * & * & * & * & * & * & *
\end{array}
$$



Sparse LU: models and methods

The column elimination tree

- A potential problem with the column elimination tree is then:
- $\mathcal{S}\{A^TA\}$ can have significantly more nonzero entries than $\mathcal{S}\{L+U\}$.
- An extreme example is when A has one or more dense rows because A^TA is then fully dense.
- So, using it or not using it, it depends.

Sparse LU: models and methods

Supernodes and LU

- Supernodes group together columns of the factors with the same nonzero structure, allowing them to be treated as dense for storage and computation.
- For nonsymmetric matrices, supernodes are harder to characterize. Or, rather, we have more options to define them.
- The need to incorporate pivoting means it may not be possible to predict the sparsity structures of the factors before the numerical factorization and they must be identified on the fly.
- Cholesky solver: fundamental supernodes are made contiguous by symmetrically permuting the matrix according to a postordering of its elimination tree; this does not change the sparsity of the Cholesky factor.
- For nonsymmetric A, this may be different.
- We may sometimes do a dirty trick: distinguish A with a nearly symmetric pattern from other situations.

Sparse LU: models and methods

Multifrontal LU method

- Generalization to nonsymmetric A: modifying the definitions of the frontal matrices and generated elements to conform to an LU factorization.
- Natural generalizations to rectangular frontal and generated element matrices do not simultaneously satisfy the statement above. The statement rewritten for the LU factorization is:
  - (a) Each generated element V_j is used only once to contribute to a frontal matrix.
  - (b) The row and column index lists for the rectangular frontal matrix F_j correspond to the nonzeros in column $L_{j:n,j}$ and nonzeros in row $U_{j,j:n}$, respectively.
- An approach that satisfies (a) can be based on the sparsity pattern of $\mathcal{S}\{A + A^T\}$ (SS) and storing some explicit zeros if $\mathcal{S}\{A\}$ is not structurally symmetric.
- (a)-based approach performs well if $\mathcal{S}\{A\}$ is close to that (SS).

Sparse LU: preprocessing to get full transversal

Preprocessing for LU

- Up to now we have dealt with graph models and methods to factorize.
- Some of the text indicates the algorithms as extensions from the symmetric case
- The deep problem behind is the need to solve the problem of non-factorizability
- But there exist preprocessing techniques that may alleviate this problem.

Sparse LU: preprocessing to get full transversal

Transversal

- The transversal of a matrix A is the set of its nonzero diagonal entries.
- Full or maximum transversal of A: all its diagonal entries are nonzero.
- If A is nonsingular (even structurally only) then it can be nonsymmetrically permuted to have a full transversal.
- The converse is clearly not true (for example, a matrix with all its entries equal to one has a full transversal but it is singular).

Sparse LU: preprocessing to get full transversal

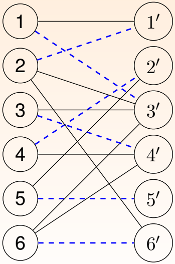
Preprocessing for LU: transversal

- Consider the case when A does not have a full transversal (that is, it has one or more zeros on the diagonal).
- For numerical stability and to reduce the number of permutations required during the factorization, it can be useful to permute A such that its transversal is full. It exists, since A is assumed to be nonsingular.
- Such permutation can be found using matchings.

Sparse LU: preprocessing to get full transversal

Preprocessing for LU: transversal

- Given a graph $\mathcal{G} = (\mathcal{V}, \mathcal{E})$, an edge subset $\mathcal{M} \subseteq \mathcal{E}$ is called a matching (or assignment) if no two edges in $\mathcal{M}$ are incident to the same vertex.
- The cardinality of a matching is the number of edges in it. A maximum cardinality matching (or maximum matching) is a matching of maximum cardinality. A matching is perfect if all the vertices are matched.



Sparse LU: preprocessing to get full transversal

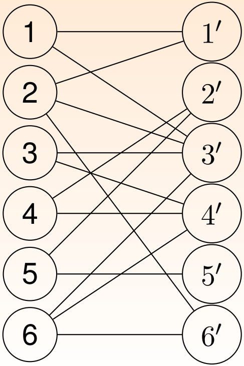
Preprocessing for LU: transversal

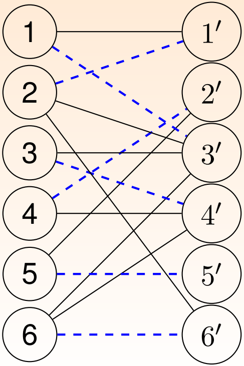
- Consider $A \in \mathbb{R}^{n \times n} = \{a_{ij'}\}$.
- Its bipartite graph model is denoted by $\mathcal{G}_b = (\mathcal{V}_{row}, \mathcal{V}_{col}, \mathcal{E})$, its row vertex set $\mathcal{V}_{row} = \{i \,|a_{ij'} \neq 0\}$ and the column vertex set $\mathcal{V}_{col} = \{j' \,|a_{ij'} \neq 0\}$ correspond to the rows and columns of A. There is an (undirected) edge $(i, j') \in \mathcal{E}$ if and only if $a_{ij'} \neq 0$.

Sparse LU: preprocessing to get full transversal

Preprocessing for LU: transversal via matching

$$\begin{array}{c|cccccc} & 1' & 2' & 3' & 4' & 5' & 6' \\ \hline 1 & * & & \color{blue}{*} & & & \\ 2 & \color{blue}{*} & & * & & & * \\ 3 & & & * & \color{blue}{*} & & \\ 4 & & \color{blue}{*} & & * & & \\ 5 & & * & & & \color{blue}{*} & \\ 6 & & & * & * & & \color{blue}{*} \end{array}$$





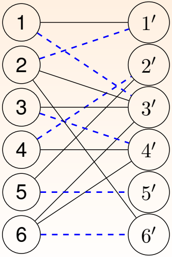
Sparse LU: preprocessing to get full transversal

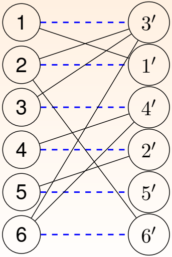
Preprocessing for LU: transversal via matching

- A is structurally nonsingular $\Rightarrow$ a matching $\mathcal{M}$ in $\mathcal{G}_b$ is perfect if it has cardinality n.
- A perfect matching defines an $n \times n$ permutation matrix Q with entries q_{ij} given by

$$q_{ij} = \begin{cases} 1, & \text{if } (j, i') \in \mathcal{M}, \\ 0, & \text{otherwise.} \end{cases}$$

- Both QA and AQ have the matching entries on the diagonal.





Sparse LU: preprocessing to get full transversal

Preprocessing for LU: transversal via matching

- Back from the bipartite graph to the nonsymmetrically permuted matrix.
- Q and the column permuted matrix AQ for the example in Figure above.

$$Q = \begin{array}{c|cccccc} & 1 & 2 & 3 & 4 & 5 & 6 \\ \hline 1 & & 1 & & & & \\ 2 & & & & 1 & & \\ 3 & 1 & & & & & \\ 4 & & & 1 & & & \\ 5 & & & & & 1 & \\ 6 & & & & & & 1 \end{array} \qquad AQ = \begin{array}{c|cccccc} & 3' & 1' & 4' & 2' & 5' & 6' \\ \hline 1 & * & * & & & & \\ 2 & * & * & & & & * \\ 3 & * & & * & & & \\ 4 & & & * & * & & \\ 5 & & & & * & * & \\ 6 & * & & * & & & * \end{array}$$

Figure: *The permutation matrix Q, the column permuted matrix AQ corresponding to the matrix above. The matched entries are on the diagonal of AQ.*

Sparse LU: preprocessing to get full transversal

Preprocessing for LU: transversal via matching

- If a perfect matching exists, it can be found using augmenting paths.
- A path $\mathcal{P}$ in a graph is an ordered set of edges in which successive edges are incident to the same vertex. $\mathcal{P}$ is called an $\mathcal{M}$-alternating path if the edges of $\mathcal{P}$ are alternately in $\mathcal{M}$ and not in $\mathcal{M}$.
- An $\mathcal{M}$-alternating path is an $\mathcal{M}$-augmenting path in $\mathcal{G}_b$ if it connects an unmatched column vertex with an unmatched row vertex. Note that the length of an $\mathcal{M}$-augmenting path is an odd integer.

Sparse LU: preprocessing to get full transversal

Preprocessing for LU: transversal via matching

- Let $\mathcal{M}$ and $\mathcal{P}$ be subsets of $\mathcal{E}$ and define the symmetric difference

$$\mathcal{M} \oplus \mathcal{P} := (\mathcal{M} \setminus \mathcal{P}) \cup (\mathcal{P} \setminus \mathcal{M}),$$

- This is set of edges that belongs to either $\mathcal{M}$ or $\mathcal{P}$ but not to both.
- If $\mathcal{M}$ is a matching and $\mathcal{P}$ is an $\mathcal{M}$-augmenting path, then $\mathcal{M} \oplus \mathcal{P}$ is a matching with cardinality $|\mathcal{M}|+1$.
- Growing the matching in this way is called augmenting along $\mathcal{P}$.

Theorem

A matching $\mathcal{M}$ in an undirected graph is a maximum matching if and only if there is no $\mathcal{M}$-augmenting path

- That is, maximum matching can be found in a straightforward way.

Sparse LU: preprocessing to get full transversal

Preprocessing for LU: matching algorithm for bipartite graphs

Algorithm (Maximum matching algorithm)

Input: *An undirected graph.*
Output: *Output maximum matching.*

```
1: Find an initial matching $\mathcal{M}$                         ▷ For example, $\mathcal{M} = \emptyset$
2: while there exists a $\mathcal{M}$-augmenting path $\mathcal{P}$ do
3:     $\mathcal{M} = \mathcal{M} \oplus \mathcal{P}$                      ▷ Augment the matching along $\mathcal{P}$
4: end while
```

Sparse LU: preprocessing to get full transversal

Augmenting paths: demonstration

- On the left is a bipartite graph with a matching with cardinality 5. An augmenting path $2 \Longrightarrow 3' \Longrightarrow 3 \Longrightarrow 4' \Longrightarrow 4 \Longrightarrow 2'$ shown.
- Augmenting the matching along this path, the cardinality of the matching increases to 6 and $\mathcal{M} \oplus \mathcal{P}$ is a perfect matching.

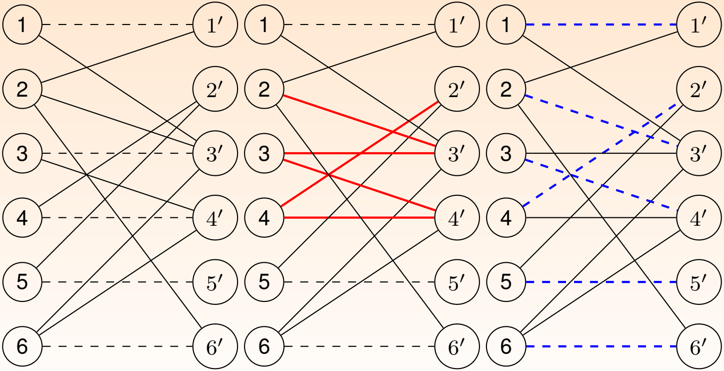


Figure: *Search for a perfect matching using augmenting paths.*

Sparse LU: preprocessing to get full transversal

Transversals: weighted matchings

- More sophisticated variations that aim to ensure the absolute values of the diagonal entries of the permuted matrix (or their product) are in some sense large.
- Formally: if $\mathcal{G}_b = (\mathcal{V}_{row}, \mathcal{V}_{col}, \mathcal{E})$ is the bipartite graph associated with A then let $\mathcal{G}_b(C) = (\mathcal{V}_{row}, \mathcal{V}_{col}, \mathcal{E})$ be the corresponding weighted bipartite graph in which each edge $(i, j') \in \mathcal{E}$ has a weight $c_{ij'} \geq 0$.
- The weight (or cost) of a matching $\mathcal{M}$ in $\mathcal{G}_b(C)$, denoted by $csum(\mathcal{M})$, is the sum of its edge weights; i.e.,

$$csum(\mathcal{M}) = \sum_{(i,j') \in \mathcal{M}} c_{ij'}.$$

- A perfect matching $\mathcal{M}$ in $\mathcal{G}_b(C)$ is said to be a minimum weight perfect matching if it has smallest possible weight i.e., $csum(\mathcal{M}) \leq csum(\widehat{\mathcal{M}})$ for all possible perfect matchings $\widehat{\mathcal{M}}$.

Sparse LU: preprocessing to get full transversal

Transversals: weighted matchings: useful approach

- The problem: given A, find a matching of the rows to the columns such that the product of the matched entries is maximized.
- That is, find a permutation vector q that maximizes

$$\prod_{i=1}^{n} |a_{iq_i}|. \tag{35}$$

- Define a matrix C corresponding to A with entries $c_{ij'} \geq 0$ as follows:

$$c_{ij'} = \begin{cases} \log(\max_i |a_{ij'}|) - \log |a_{ij'}|, & \text{if } a_{ij'} \neq 0 \\ \infty, & \text{otherwise.} \end{cases} \tag{36}$$

Sparse LU: preprocessing to get full transversal

Transversals: weighted matchings

- It is straightforward to see that finding a q that solves the problem is equivalent to finding a q that minimizes

$$\sum_{i=1}^{n} |c_{iq_i}|, \tag{37}$$

- This is equivalent to finding a minimum weight perfect matching in an edge weighted bipartite graph.
- This is a well-studied problem and is known as the bipartite weighted matching or linear sum assignment problem.
- The key concept for finding a minimum weight perfect matching is that of a shortest augmenting path.

Sparse LU: preprocessing to get full transversal

Weighted matchings: primal-dual algorithm

- A matching $\mathcal{M}_e$ is extreme if and only if there exist u_i and $v_{j'}$ (which are termed dual variables) satisfying

$$\begin{cases} c_{ij'} = u_i + v_{j'}, & \text{if } (i, j') \in \mathcal{M}_e, \\ c_{ij'} \geq u_i + v_{j'}, & \text{otherwise.} \end{cases} \tag{38}$$

- This is employed by the `MC64` algorithm.
- The dual variables can be used for scaling.
- The algorithm starts with a feasible solution and corresponding extreme matching and then proceeds to iteratively increase its cardinality by one by constructing a sequence of shortest augmenting paths until a perfect extreme matching is found.

Sparse LU: preprocessing to get full transversal

Weighted matchings: primal-dual algorithm

- Can be more efficient if a large initial matching is used.
- For example, Step 3 can be replaced by setting $u_i = \min\{c_{ij'} | \, j' \in \mathcal{S}\{A_{i,1:n}\}\}$ for $i \in \mathcal{V}_{row}$ and $v_{j'} = \min\{c_{ij'} - u_i | \, i \in \mathcal{S}\{A_{1:n,j'}\}\}$ for $j' \in \mathcal{V}_{col}$. In Step 4, an initial extreme matching can be determined from the edges for which $c_{ij'} - u_i - v_{j'} = 0$.

Algorithm (Outline of the `MC64` algorithm)

Input: *Matrix A.*
Output: *A matching $\mathcal{M}$ and dual variables $u_i, v_{j'}$.*

```
1: Define the weights $c_{ij'}$ using (36)
2: Construct the weighted bipartite graph $\mathcal{G}_b(C) = (\mathcal{V}_{row}, \mathcal{V}_{col}, \mathcal{E})$
3: Set $u_i = 0$ for $i \in \mathcal{V}_{row}$ and $v_{j'} = \min\{c_{ij'} : (i, j') \in \mathcal{E}\}$ for $j' \in \mathcal{V}_{col}$     ▷ Initial solution
4: Set $\mathcal{M} = \{(i, j') | \, u_i + v_{j'}\}$     ▷ Initial extreme matching
5: while $\mathcal{M}$ is not perfect do
6:     Find the shortest augmenting path $\mathcal{P}$ with respect to $\mathcal{M}$
7:     Augment the matching $\mathcal{M} = \mathcal{M} \oplus \mathcal{P}$
8:     Update $u_i, v_{j'}$ so that (38) is satisfied for new $\mathcal{M}$     ▷ make $\mathcal{M}$ extreme
9: end while
```

Sparse LU: preprocessing to get full transversal

Weighted matchings: primal-dual algorithm

- Number of potential problems with the MC64 algorithm.
  - The runtime is hard to predict and depends on the initial ordering of A.
  - It is a serial algorithm and as such it can represent a significant fraction of the total factorization time of a direct solver.
- Because the complexity of Step 6 of Algorithm is $O((n + nz(A)) \log n)$ and the complexity of Step 7 is $O(n)$ and of Step 8 is $O(n + nz(A)$, MC64 has a worst-case complexity of $O(n(n + nz(A)) \log n)$.
- In practice, this bound is not achieved and the algorithm is widely used.

Sparse LU: preprocessing to get full transversal

Transversals: extension to non-square matrices

- The Dulmage-Mendelsohn (DM) decomposition is based on matchings and it generalizes the block triangular form.
- It comprises row and column permutations P and Q such that

$$PAQ = \begin{array}{c} \\ \mathcal{R}_1 \\ \mathcal{R}_2 \\ \mathcal{R}_3 \end{array} \begin{array}{c} \begin{array}{ccc} \mathcal{C}_1 & \mathcal{C}_2 & \mathcal{C}_3 \end{array} \\ \begin{pmatrix} A_1 & A_4 & A_6 \\ 0 & A_2 & A_5 \\ 0 & 0 & A_3 \end{pmatrix} \end{array}. \qquad (39)$$

Here A_1 is an $m_1 \times n_1$ underdetermined matrix ($m_1 < n_1$ or $m_1 = n_1 = 0$), A_2 is an $m_2 \times m_2$ square matrix and A_3 is an $m_3 \times n_3$ overdetermined matrix ($m_3 > n_3$ or $m_3 = n_3 = 0$).

Sparse LU: preprocessing to get full transversal

Transversals: extension to non-square matrices

An example decomposition for a 10×10 matrix. Here $\mathcal{R} = \{1, 2, \ldots, 9\}$ and $\mathcal{C} = \{2, 3, \ldots, 10\}$.

$$\begin{array}{c|cccccccccc}
 & 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 & 10 \\
1 & * & & * & & & & & & & \\
2 & * & * & & & * & & & * & & * \\
3 & * & & * & * & & & & & & \\
4 & & & * & * & & & & & * & * \\
5 & & & & * & * & * & & * & & \\
6 & & * & & & & * & & & & \\
7 & & & & * & & & * & & & \\
8 & * & * & & & & & & * & & \\
9 & & & & & & & & & * & * \\
10 & & & & & * & & * & & & *
\end{array}, PAQ = \left(\begin{array}{cccc|cccc|cc}
* & {\color{blue}*} & * & * & * & & & & & \\
 & & {\color{blue}*} & * & & * & * & & * & \\
 & * & & {\color{blue}*} & & & & & & \\
\hline
 & & & & {\color{blue}*} & & & & & * \\
 & & & & * & {\color{blue}*} & & * & & \\
 & & & & & & {\color{blue}*} & * & & \\
 & & & & & & * & {\color{blue}*} & * & \\
\hline
 & & & & & & & & {\color{blue}*} & * \\
 & & & & & & & & & {\color{blue}*} \\
 & & & & & & & & * &
\end{array}\right).$$

Figure: An example of a (coarse) DM decomposition. The blue entries belong to the maximum matching. $m_1 = 3$, $m_2 = 4$, $m_3 = 3$, $n_1 = 4$, $n_2 = 4$, $n_3 = 2$. Column 1 and row 10 are unmatched.

From factorization to solution

The solve

- The solve phase uses the factors to solve for a single b or for a block of multiple right-hand sides or for a sequence of right-hand sides one-by-one.
- Once an LU factorization has been computed, the solution x of the linear system $Ax = b$ is computed by solving the lower triangular system

  $$Ly = b, \tag{40}$$

  followed by the upper triangular system

  $$Ux = y. \tag{41}$$

- Triangular solves with a dense right-hand side vector are straightforward.
- Typically much cheaper, but parallel computation may change the perspective.

From factorization to solution

The solve

- First step is forward substitution: the component y_1 is determined from the first equation. Substitution into the second equation to obtain y_2, and so on.
- Once y is available, the solution can be obtained by back substitution in which the last equation is used to obtain x_n, which is then substituted into equation $n - 1$ to obtain x_{n-1}, and so on.

From factorization to solution

The solve

Algorithm (Forward substitution: simple $Ly = b$ with b dense)

Input: *Lower triangular matrix L with nonzero diagonal entries and dense right-hand side b.*
Output: *The dense solution vector y.*

```
Initialise $y = b$
for $j = 1 : n$ do
    $y_j = y_j / l_{jj}$
    for $i = j + 1 : n$ do
        if $l_{ij} \neq 0$ then
            $y_i = y_i - l_{ij} y_j$
        end if
    end for
end for
```

From factorization to solution

The sparse solve

- When b is sparse, the solution y is (may be) also sparse.
- In particular, if in Algorithm $y_k = 0$, then the outer loop with $j = k$ can be skipped.
- Furthermore, if $b_1 = b_2 = \ldots = b_k = 0$ and $b_{k+1} \neq 0$, then $y_1 = y_2 = \ldots = y_k = 0$. Scanning y to check for zeros adds $O(n)$ to the complexity.
- But if the set of indices $\mathcal{J} = \{j \mid y_j \neq 0\}$ is known beforehand then we can use the following Algorithm,
- A way to determine $\mathcal{J}$ is discussed later.

From factorization to solution

The sparse solve

Algorithm (Forward substitution: simple solve $Ly = b$ with b sparse)

Input: *Lower triangular matrix L with nonzero diagonal entries, sparse vector b and the set $\mathcal{J}$.*
Output: *The sparse solution vector y.*

```
Initialise $y = b$
for $j \in \mathcal{J}$ do                ▷ Take indices from $\mathcal{J}$ in increasing order
    $y_j = y_j / l_{jj}$
    for $i = j + 1 : n$ do
        if $l_{ij} \neq 0$ then
            $y_i = y_i - l_{ij} y_j$
        end if
    end for
end for
```

Sparse LU: preprocessing to get BTF shape

Enhancements due to permuting into a block form

- Permuting to block form is closely connected to matrix reducibility.
- A is said to be reducible if there is a permutation matrix P such that
$$PAP^T = \begin{pmatrix} A_{p_1,p_1} & A_{p_1,p_2} \\ 0 & A_{p_2,p_2} \end{pmatrix},$$
where A_{p_1,p_1} and A_{p_2,p_2} are non trivial square matrices (that is, they are of order at least 1).
- If A is not reducible, it is irreducible. Matrices of order 1 are irreducible.
- If $\mathcal{S}\{A\}$ is symmetric then $A_{p_1,p_2} = 0$ and PAP^T is block diagonal.
- Wy do we need this? To be more happy ☺: factorize only blocks, do solves only with blocks.

Sparse LU: preprocessing to get BTF shape

Using BTF

Algorithm (Solve a sparse linear system in upper BTF)

Input: *Upper block triangular matrix and a conformally partitioned right-hand side vector c.*
Output: *The conformally partitioned solution vector y.*

```
for $ib = 1 : nb$ do
    Compute $P_{ib} A_{ib,ib} = L_{ib} U_{ib}$
end for
Solve $L_{nb} U_{nb}\, y_{nb} = P_{nb} c_{nb}$                ▷ Perform forward and back substitutions
for $ib = nb - 1 : 1$ do
    for $jb = ib + 1 : nb$ do
        $c_{ib} = c_{ib} - A_{ib,jb} y_{jb}$                  ▷ Sparse matrix-vector operation
    end for
    Solve $L_{ib} U_{ib}\, y_{ib} = P_{ib} c_{ib}$                ▷ Perform substitutions
end for
```

Sparse LU: preprocessing to get BTF shape

Theorem

Given a nonsingular nonsymmetric matrix A there exists a permutation matrix P such that

$$PAP^T = \begin{pmatrix} A_{1,1} & A_{1,2} & \cdots & A_{1,nb} \\ 0 & A_{2,2} & \cdots & A_{2,nb} \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \cdots & A_{nb,nb} \end{pmatrix}, \qquad (42)$$

where the square matrices $A_{ib,ib}$ on the diagonal are irreducible. The set $\{A_{ib,ib} \,|\, 1 \leq ib \leq nb\}$ is uniquely determined (but the blocks may appear on the diagonal in a different order). The order of the rows and columns within each $A_{ib,ib}$ may not be unique.

- This upper block triangular form (BTF) is also known as the Frobenius normal form. It is said to be non trivial if $nb > 1$.

Sparse LU: preprocessing to get BTF shape

Permutation to BTF

An example of a matrix that can be symmetrically permuted to block triangular form with $nb = 2$ is given below.

$$
\begin{array}{c|cccccc|}
 & 1 & 2 & 3 & 4 & 5 & 6 \\
\hline
1 & * & * & & * & * & \\
2 & & * & & & * & \\
3 & & * & * & & * & * \\
4 & & * & & * & & \\
5 & * & & & & * & \\
6 & & * & * & & & *
\end{array}
\qquad
\begin{array}{c|cccccc|}
 & 6 & 3 & 5 & 4 & 1 & 2 \\
\hline
6 & * & * & & & & * \\
3 & * & * & * & & & * \\
5 & & & * & & * & \\
4 & & & & * & & * \\
1 & & & * & * & * & * \\
2 & & & * & & & *
\end{array}
$$

- Why symmetrically? Using just relabelling of vertices.

Figure: *The sparsity patterns of A (left) and the upper block triangular form PAP^T with two blocks $A_{ib,ib}$, $i = 1, 2$, of order 2 and 4 (right).*

- Why relabelling? Since we just find somehow a new vertex order in a digraph.

Sparse LU: preprocessing to get BTF shape

Permutation to BTF

- Finding P_s is identical to finding the strongly connected components (SCCs) of the digraph $\mathcal{G}(A) = (\mathcal{V}, \mathcal{E})$. Mentioned above.
- To find the SCCs, $\mathcal{V}$ is partitioned into non-empty subsets $\mathcal{V}_i$ with each vertex belonging to exactly one subset. Each vertex i in the quotient graph corresponds to a subset $\mathcal{V}_i$ and there is an edge in the quotient graph with endpoints i and j if $\mathcal{E}$ contains at least one edge with one endpoint in $\mathcal{V}_i$ and the other in $\mathcal{V}_j$.
- The condensation (or component graph) of a digraph is a quotient graph in which the SCCs form the subsets of the partition, that is, each SCC is contracted to a single vertex. This reduction provides a simplified view of the connectivity between components.

Sparse LU: preprocessing to get BTF shape

Permutation to BTF

An example of SCCs. Five of them: $\{p, q, r\}$, $\{s, t, u\}$, $\{v\}$, $\{w\}$, $\{x\}$.

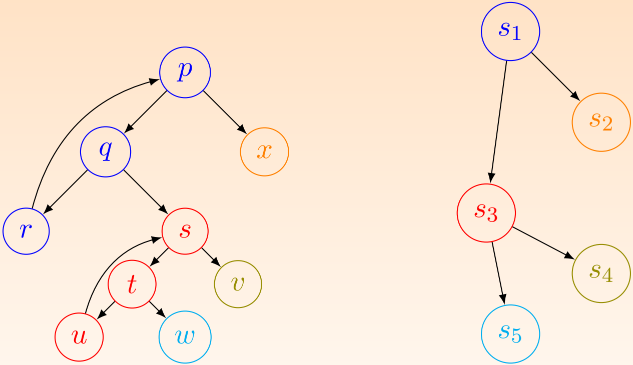


Figure: *An illustration of the strong components of a digraph. On the left, the five SCCs are denoted using different colours and on the right is the condensation DAG $\mathcal{G}_C$ formed by the SCCs.*

Sparse LU: preprocessing to get BTF shape

Relationship between SCCs and DAGs

Theorem

The condensation $\mathcal{G}_C$ of a digraph is a DAG (directed acyclic graph).

- Any DAG can be topologically ordered. (Directed edges connect smaller to larger (labelled) vertices).
- Consequently, $\mathcal{G}_C = (\mathcal{V}_C, \mathcal{E}_C)$ can be topologically ordered and if $\mathcal{V}_i$ and $\mathcal{V}_j$ are contracted to s_i and s_j and $(s_i \longrightarrow s_j) \in \mathcal{E}_C$ then $s_i < s_j$.
- It follows that to permute A to block triangular form it is sufficient to find the SCCs of $\mathcal{G}(A)$.
- That is, topologically ordering the vertices of the condensation $\mathcal{G}_C$ induced by the SCCs is the quotient graph that implies the block triangular form.

Outline

1. Introduction
2. Introductory notation and terminology
3. Factorizations
4. Graphs and sparse matrices
5. Sparse matrices and data structures
6. Symbolic factorizations
7. Symbolic Cholesky factorization
8. Cholesky factorization
9. Sparse LU factorization
10. Stability, ill-conditioning, indefiniteness
11. Blocks in the input matrix
12. Sparse Least Squares and QR factorization
13. Reorderings
14. Algebraic preconditioning
15. Incomplete factorizations
16. Sparse approximate inverses

Stability and ill-conditioning

Backward stability

- Computational algorithm

  $$z = g(d)$$

  for computing z as a function g of given data d.
- Backward stable algorithm: the computed solution $\hat{z}$ is the exact solution of $\hat{z} = g(d + \Delta d)$, and the perturbation Δd is “small” for all possible inputs d.
- What is meant by small depends on the context. For example, if d is based on physical measurements (necessarily inaccurate), Δd is small if it is of the same absolute value as the inaccuracies in determining d or smaller.

Stability and ill-conditioning

Forward and backward errors

$$z = g(d) \to \hat{z} = g(d + \Delta d)$$

- Minimum $|\Delta d|$: the absolute backward error
- Minimum $|\Delta d|/|d|$: the relative backward error.
- Absolute and relative errors of $\hat{z}$: called forward errors.

Stability and ill-conditioning

Ill-conditioning

- A related concept is ill-conditioning.
- We say that the problem $z = g(d)$ is ill-conditioned if small perturbations in the data d can lead to large changes in $\hat{z}$.
- The condition number measures how sensitive the output of a function is to its input.

Observation

Backward stability is a property of the computational algorithm. To compute solutions with a small backward error we need to consider stable algorithms. Ill-conditioning is a property of input problem data.

Stability and ill-conditioning

Ill-conditioning: relation between forward and backward errors

- The following approximate inequality holds:

$$\text{forward error } \lessapprox \text{ condition number} \times \text{backward error.}$$

- The computed solution to an ill-conditioned problem can have a large forward error even if the computed solution has a small backward error since the backward error can be amplified by a large condition number.

Stability and ill-conditioning

Backward error result for LU

ϵ denotes the machine precision.

Theorem

Let the computed LU factorization of a matrix A be $A + \Delta A = \widehat{L}\,\widehat{U}$. The perturbation ΔA that results from using finite precision arithmetic satisfies

$$||\Delta A||_\infty \leq \; n\, O(\epsilon)\, ||\widehat{L}||_\infty ||\widehat{U}||_\infty + O(\epsilon^2). \tag{43}$$

Moreover, the computed solution $\hat{x}$ of the linear system $Ax = b$ satisfies $(A + \Delta' A)\hat{x} = b$ with

$$||\Delta' A||_\infty \leq n\, O(\epsilon)\, ||\widehat{L}||_\infty ||\widehat{U}||_\infty + O(\epsilon^2). \tag{44}$$

Stability and ill-conditioning

Stability of Cholesky

Theorem

Let the computed Cholesky factorization of a SPD matrix A be $A + \Delta A = \widehat{L}\widehat{L}^T$. The perturbation ΔA that results from using finite precision arithmetic satisfies

$$||\Delta A||_\infty \leq n^2 \, O(\epsilon) \, ||A||_\infty .$$

Moreover, the computed solution $\hat{x}$ of the linear system $Ax = b$ satisfies $(A + \Delta' A)\hat{x} = b$ with

$$||\Delta' A||_\infty \leq n^2 \, O(\epsilon) \, ||A||_\infty .$$

Stability and ill-conditioning

Sidestep: using the inverse instead of factorization

- No such stability results: The computed inverse is typically not the exact inverse of a nearby matrix $A + \Delta A$ for any small perturbation ΔA.
- Impractical to compute and store A^{-1}, regardless of how sparse A is: see below.
- But there is another point: matrix sparsity. See the next slide.

Stability and ill-conditioning

Sidestep: using the inverse instead of factorization

Theorem

A irreducible $\Rightarrow$ the sparsity pattern $\mathcal{S}\{A^{-1}\}$ of its inverse is fully dense.

Proof.

Assume A is factorizable. (If not, there is a permutation matrix P such that the LU factorization of the row permuted matrix PA is factorizable). Then consider PA instead of A because for any permutation matrix P: $(PA)^{-1}$ is fully dense if and only if A is fully dense. Consider the matrix K of order $2n$

$$K = \begin{pmatrix} A & I_n \\ I_n & 0 \end{pmatrix}.$$

After applying n elimination steps to $K = K^{(1)}$, the order n active submatrix of $K^{(n+1)}$ is $-A^{-1}$. Consider entry $(A^{-1})_{ij}$ $(1 \le i, j \le n)$. Because A is irreducible and the off-diagonal $(1,2)$ and $(2,1)$ blocks of K are equal to the identity matrix, there is a fill path $i \Longrightarrow j$ in $\mathcal{G}(K)$ (the indices of all the intermediate vertices on the path are less than or equal to n). Theorem on fill paths and the non-cancellation assumption imply $(A^{-1})_{ij} \neq 0 \Rightarrow A^{-1}$ is fully dense. □ □

Stability and ill-conditioning

Ill-conditioning and backward stability for factorizations

- Ill-conditioning can be improved by preprocessing, better solution also by postprocessing.
- Backward stability can be improved by better algorithms .
  - One of the tools (making algorithms better) is pivoting.
  - Needed for LU: Cholesky factorization of A is unconditionally backward stable.

Stability and ill-conditioning

Pivoting to improve stability of factorizations

- At step k of GE, the computed (not distinguishing them by notation) $a_{kk}^{(k)}$ is termed the pivot $(1 \leq k < n)$.
- It must be nonzero. To avoid this, use row interchanges: partial pivoting.
- If $|a_{kk}^{(k)}|$ is very small (relatively to other entries in the active submatrix), it can cause difficulties in finite precision arithmetic since $l_{ik} = a_{ik}^{(k)}/a_{kk}^{(k)}$ can be very large. Formally, this increases the growth in the factor U.
- Partial pivoting ensures

$$|l_{ik}| \leq 1 \Longrightarrow \max_{i>k} |a_{ik}^{(k)}| \leq |a_{kk}^{(k)}|.$$

Stability and ill-conditioning

Pivoting to improve stability of factorizations

- If P_k is the row permutation at stage k and $P = P_{n-1}P_{n-2}\dots P_1$ then the computed factors of PA satisfy

$$||\widehat{L}||_\infty \le n \quad \text{and} \quad ||\widehat{U}||_\infty \le n\,\rho_{growth}||A||_\infty,$$

where the growth factor ρ_{growth} is defined to be

$$\rho_{growth} = \max_{i,j,k}\left(|a_{ij}^{(k)}|\,/\,|a_{ij}|\right). \tag{45}$$

- The bounds above can be rewritten as

$$||\Delta A||_\infty \le n^3\,\rho_{growth}\,O(\epsilon)\,||A||_\infty, \quad ||\Delta' A||_\infty \le n^3\,\rho_{growth}\,O(\epsilon)\,||A||_\infty.$$

- In practice, these bounds are pessimistic.
- The growth factor is influenced both by the initial ordering of A and the pivoting strategy ⇒ LU factorization conditionally backward stable.

Stability and ill-conditioning

Partial pivoting

- Recall: partial pivoting interchanges rows at each step of the factorization to select the entry of largest absolute value in its column as the next pivot.
- If partial pivoting is used, it is straightforward to show that the growth factor satisfies

$$\rho_{growth} \leq 2^{n-1}.$$

Stability and ill-conditioning

Partial pivoting: an example of growth

- An example that shows that the growth factor satisfies

$$\rho_{growth} \leq 2^{n-1}.$$

$$\begin{array}{c} \\ 1\\2\\3\\4\\5 \end{array}\!\!\begin{array}{ccccc} 1&2&3&4&5 \\ \end{array}$$

$$\begin{array}{c|ccccc} & 1 & 2 & 3 & 4 & 5 \\ \hline 1 & 1 & & & & 1 \\ 2 & -1 & 1 & & & 1 \\ 3 & -1 & -1 & 1 & & 1 \\ 4 & -1 & -1 & -1 & 1 & 1 \\ 5 & -1 & -1 & -1 & -1 & 1 \end{array} = \begin{array}{c|ccccc} & 1 & 2 & 3 & 4 & 5 \\ \hline 1 & 1 & & & & 0 \\ 2 & -1 & 1 & & & 0 \\ 3 & -1 & -1 & 1 & & 0 \\ 4 & -1 & -1 & -1 & 1 & 0 \\ 5 & -1 & -1 & -1 & -1 & 1 \end{array} \begin{array}{c|ccccc} & 1 & 2 & 3 & 4 & 5 \\ \hline 1 & 1 & & & & 1 \\ 2 & & 1 & & & 2 \\ 3 & & & 1 & & 4 \\ 4 & & & & 1 & 8 \\ 5 & & & & & 16 \end{array}$$

- The bound can be achieved also in nontrivial cases.
- Although generally pessimistic, it is a useful approach.

Stability and ill-conditioning

Complete pivoting

- A much smaller bound obtained if complete (or full) pivoting is used: chooses the pivot as an entry of the largest magnitude in the active submatrix (Schur complement of the previous step),
- That is, at stage k the pivot $a_{kk}^{(k)}$ is chosen so that

$$\max_{i\ge k, j\ge k} |a_{ij}^{(k)}| \le |a_{kk}^{(k)}|.$$

- Then

$$\rho_{growth} \le n^{1/2}(2.\, 3^{1/2}.\, 4^{1/3} \ldots n^{1/(n-1)})^{1/2}. \qquad (46)$$

- Can be expensive even outside parallel environment.
- Relaxations (simplifications) not searching the whole matrix used in practice.

Stability and ill-conditioning

Rook pivoting

- Rook pivoting: more restrictive than partial pivoting but cheaper than complete pivoting
- The pivot is chosen as the largest entry in its row *and* its column, that is,
$$\max_{i>k}\left(\,|a_{ik}^{(k)}|, |a_{ki}^{(k)}|\,\right) \leq |a_{kk}^{(k)}|.$$
- In practice, the cost is usually a small multiple of the cost of partial pivoting and significantly less than that of complete pivoting.
- The growth factor for rook pivoting satisfies
$$\rho_{growth} \leq 1.5\, n^{(3/4)\log n}\,.$$

Stability and ill-conditioning

Threshold partial pivoting

- The pivot $a_{kk}^{(k)}$ chosen to satisfy

$$\max_{i>k} |a_{ik}^{(k)}| \leq \gamma^{-1} |a_{kk}^{(k)}|,$$

where $\gamma \in (0,1]$ is a chosen threshold parameter.

- Straightforward to see that

$$\max_{i} |a_{ij}^{(k)}| \leq (1+\gamma^{-1}) \max_{i} |a_{ij}^{(k-1)}|,$$

and

$$\max_{i} |a_{ij}^{(k)}| \leq (1+\gamma^{-1})^{nz_j} \max_{i} |a_{ij}|,$$

where nz_j is the number of off-diagonal entries in the j-th column of the U factor. Furthermore,

$$\rho_{growth} \leq (1+\gamma^{-1})^{nz_{cmax}},$$

where $nz_{cmax} = \max_j nz_j \leq n-1$.

Stability and ill-conditioning

2×2 pivoting

- A symmetric, but indefinite: it may not be possible to use diagonal pivots (all of them can be zero).

$$A = \begin{pmatrix} 0 & 1 \\ 1 & 0 \end{pmatrix}.$$

- If only rows (columns) of A are permuted (so that off-diagonal entries are selected as pivots): symmetry is destroyed ⇒ LU factorization is an option and this essentially doubles the cost of the factorization in terms of both storage and operation counts.
- Symmetry preserved by generalizing the LDL^T factorization using as pivots also 2×2 blocks.
- It can be shown that this is sufficient.

Stability and ill-conditioning

Symmetric indefinite: example

- Consider

$$A = \begin{pmatrix} \delta & 1 \\ 1 & 0 \end{pmatrix}.$$

- $\delta = 0 \Rightarrow$ LDLT with D diagonal does not exist.
- $\delta \ll 1 \Rightarrow$ LDLT with D diagonal is not stable since $\rho_{growth} = 1/\delta$.
- LDLT factorization generalized to allow D block diagonal matrix with 1×1 and $2 \times 2 \Rightarrow$ factorization preserves symmetry and is nearly as stable as an LU factorization.

$$A = \begin{pmatrix} 1 & 1 & 0 \\ 1 & 1 & 1 \\ 0 & 1 & 0 \end{pmatrix} = \begin{pmatrix} 1 & 0 & 0 \\ 1 & 1 & 0 \\ 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} 1 & 0 & 0 \\ 0 & 0 & 1 \\ 0 & 1 & 0 \end{pmatrix} \begin{pmatrix} 1 & 1 & 0 \\ 0 & 1 & 1 \\ 0 & 0 & 1 \end{pmatrix} = LDL^T.$$

Here D has one 1×1 block and one 2×2 block.

Stability and ill-conditioning

Symmetric indefinite: balancing 1×1 pivots and 2×2 pivots

- Asumming symmetric pivots (from the diagonal or block diagonal).
- Small growth for 1×1 pivot if $|a_{kk}|$ (a diagonal entry) is large.
- If such pivot not found, consider large off-diagonals
- Consider the inverse of the 2×2 block

$$\begin{pmatrix} a & b \\ b & d \end{pmatrix}^{-1} = \frac{1}{ad - b^2} \begin{pmatrix} d & -b \\ -b & a \end{pmatrix}$$

- $\Rightarrow$ if $|a|, |d|$ small with respect to $|b|$, 2×2 pivot may be used.
- The standard rule to choose between the cases: requiring the same potential maximal growth in a 2×2 pivot versus two consecutive 1×1 pivots.
- An appropriate threshold $(1 + \sqrt{17})/8$ to choose between the pivots (see the next slide)
- $$\rho_{growth} < 3n\sqrt{2\ 3^{1/2} 4^{1/3} \dots n^{1/(n-1)}},$$

Stability and ill-conditioning

Indefinite factorization: classical schemes

Algorithm (One step of full indefinite pivoting)

```
Set $\alpha = (1+\sqrt{17})/8 \approx 0.64$
Find $a_{kk}$: diagonal entry of maximum size
Find $a_{ij}$: off-diagonal entry of maximum size ($i < j$)
if $|a_{kk}| \geq \alpha |a_{ij}|$ then
    use $a_{kk}$ as $1 \times 1$ pivot (ready for $a_{kk} = 0$)
else
    use $\begin{pmatrix} a_{ii} & a_{ij} \\ a_{ji} & a_{jj} \end{pmatrix}$ as $2 \times 2$ the pivot
end if
```

- Full pivoting: choosing pivots based on global search can be expensive despite: the growth factor bound is only a slightly worse than for LU
- Needed to combine with sparsity considerations

Stability and ill-conditioning

Indefinite factorization: classical scheme of symmetric partial pivoting

- The following scheme shows entries sufficient to be checked

$$\begin{pmatrix} d & . & . & \lambda & . & . & . \\ . & . & . & . & . & . & . \\ . & . & . & . & . & . & . \\ \lambda & . & . & c & . & \sigma & . \\ . & . & . & . & . & . & . \\ . & . & . & \sigma & . & . & . \\ . & . & . & . & . & . & . \end{pmatrix}$$

- λ, σ: maximum absolute value in its row and column, respectively.
- That is: only two rows and columns of A searched.
- The main price for less searches: slightly larger growth factor bound than in LU

Stability and ill-conditioning

Indefinite factorization: classical schemes

Algorithm (One step of symmetric partial pivoting)

```
$\alpha = (1+\sqrt{17})/8 \approx 0.64, i = 1$ (or, e.g., $i$ satisfies $|a_{ii}| \geq \alpha|a_{kk}|$ for all $k$)
Find $j \neq i$ such that $a_{ji} = \max\{|a_{ki}|, k \neq i\} =: \lambda$
if $|a_{ii}| \geq \alpha\lambda$ then
    use $a_{ii}$ as $1 \times 1$ pivot
else
    $\sigma = \max\{|a_{kj}|, k \neq j\}$
    if $|a_{ii}|\sigma \geq \alpha\lambda^2$ then
        use $a_{ii}$ as $1 \times 1$ pivot
    else if $|a_{jj}| \geq \alpha\sigma$ then
        use $a_{jj}$ as a $1 \times 1$ pivot
    else
        use $\begin{pmatrix} a_{ii} & a_{ij} \\ a_{ji} & a_{jj} \end{pmatrix}$ as $2 \times 2$ pivot
    end if
end if
```

Stability and ill-conditioning

Threshold extensions for 2×2 (symmetric) pivots

- In the sparse symmetric indefinite case, the stability test for a 1×1 pivot $a_{tt}^{(k)}$ in column t of the active submatrix at stage k is as follows (γ is the threshold).

$$\max_{i \neq t,\, i \geq k} |a_{it}^{(k)}| \leq \gamma^{-1} |a_{tt}^{(k)}|$$

- For a 2×2 pivot in rows and columns s and t the test is

$$\left| \begin{pmatrix} a_{ss}^{(k)} & a_{st}^{(k)} \\ a_{st}^{(k)} & a_{tt}^{(k)} \end{pmatrix}^{-1} \right| \begin{pmatrix} \max_{i \neq s,t; i \geq k} |a_{is}^{(k)}| \\ \max_{i \neq s,t; i \geq k} |a_{it}^{(k)}| \end{pmatrix} \leq \gamma^{-1} \begin{pmatrix} 1 \\ 1 \end{pmatrix},$$

where the absolute value of the matrix is interpreted element-wise.

Stability and ill-conditioning

Indefinite factorization: other comments

- Additional thresholds can be used also in classical pivoting schemes (here symmetric)
- For example, only $|d| \geq \gamma|\lambda|$ for some $\gamma \in (0, 1\rangle$ (useful if only a part of the active matrix is available).

Algorithm (One step of modified symmetric partial pivoting)

```
if $|d| \geq \gamma|\lambda|$ then use $d$ as $1 \times 1$ pivot
else if $|d\gamma| \geq \alpha|\lambda|^2$ then use $d$ as $1 \times 1$ pivot
else if $|e| \geq \alpha|\gamma|$ then use $e$ as $1 \times 1$ pivot
else
    use $\begin{pmatrix} d & f \\ f & e \end{pmatrix}$ as $2 \times 2$ pivot
end if
```

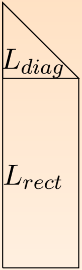
Symmetric indefinite factorization

Indefinite factorization: other comments

- In addition to bounding magnitudes of the entries in L, the ability to stably apply the inverse of D to a vector is required.
- For 2×2 pivots necessary to check that the determinant $|a_{ss}^{(k)}a_{tt}^{(k)} - a_{st}^{(k)}a_{st}^{(k)}|$ is sufficiently large.
- Limiting the size of the entries of L: $|l_{ij}| \leq \gamma^{-1}$ for all i, j plus a backward stable scheme for solving 2×2 linear systems suffices to show backward stability for the entire solution process.
- But a pivot satisfying the stability criteria may not exist.
- This may lead to loops to find a pivot from more starting factorization points.

Stability and ill-conditioning

Stability and supernodes



- Pivots can only be chosen from the block L_{diag} on the diagonal, but entries in the off-diagonal block L_{rect} are involved in the stability tests.
- Possible delaying columns and backtrack to previous supernodes or modifying A (regularization).

Stability and ill-conditioning

Segregated approaches for saddle-point problems

- Consider the following system of the form

$$\begin{pmatrix} A & B^T \\ B & 0 \end{pmatrix} \begin{pmatrix} x \\ y \end{pmatrix} = \begin{pmatrix} b \\ c \end{pmatrix} \tag{47}$$

  where $A \in \mathbb{R}^{n_1 \times n_1}$, $R \in \mathbb{R}^{n_2 \times n_1}$ with $n_1 \geq n_2$.
- Assuming regularity of the system and A, full rank B
- Saddle-point problem.
- Standard solution techniques: substituting from one block row to another one
  - Schur complement approach
  - Null-space approach

Stability and ill-conditioning

Schur complement approach to solve the saddle-point problem

$$\begin{aligned} Ax + B^T y &= b \\ Bx &= c \end{aligned}$$

- Set $x = A^{-1}(b - B^T y)$
- Substitute $BA^{-1}b - BA^{-1}B^T y = c$
- Get y from $BA^{-1}B^T y = BA^{-1}b - c$
- Get x from the first block equation.

Stability and ill-conditioning

Null-space approach to solve the saddle-point problem

$$\begin{aligned} Ax + B^T y &= b \\ Bx &= c \end{aligned}$$

- Find the null-space of B ($BZ = 0$, $\mathcal{R}(Z) = \mathcal{N}(B)$)
- Find $\hat{x}$ that solves $B\hat{x} = c$ (QR, LU; easy if n_2 is small)
- Full solution of the 2nd block equation: $x = Zv + \hat{x}$, $v \in \mathbb{R}^{n_1 - n_2}$
- Substitute this x into the first block equation:

$$A(Zv + \hat{x}) = b - B^T y$$

- Getting $Z^T A Z v = Z^T (b - A\hat{x})$ (multiplication by Z^T from left)
- Get $x = Zv + \hat{x}$, get y from $BB^T y = B(b - Ax)$ ($BB^T \in \mathbb{R}^{n_2 \times n_2}$)

Stability and ill-conditioning

Solving ill-conditioned problems

- $$\kappa(A) = \|A\| \, \|A^{-1}\| \tag{48}$$
  is the condition number of the matrix A.
- The inequality from above:
  $$\text{forward error } \lessapprox \text{condition number} \times \text{backward error.}$$
  shows that the condition number is an error magnification factor.
- A large condition number means that A is close to being singular ($\kappa(A)$ tends to infinity as A tends to singularity).
- More sophisticated condition numbers can be considered.
- Ways to mitigate ill-conditioning by preprocessing/postprocessing: iterative refinement , scaling

Stability and ill-conditioning

Iterative refinement (IR)

- IR can be used to overcome matrix ill-conditioning and improve the accuracy of the computed solution.

Algorithm (Iterative refinement of the computed solution of $Ax = b$)

Input: *The vector b and matrix A.*
Output: *A sequence of approximate solutions $x^{(0)}, x^{(1)}, \ldots$.*

```
Solve $Ax^{(0)} = b$                                  ▷ $x^{(0)}$ is the initial computed solution
for $k = 0, 1, \ldots$ do
    Compute $r^{(k)} = b - Ax^{(k)}$                  ▷ Residual on iteration $k$
    Solve $A\,\delta x^{(k)} = r^{(k)}$               ▷ Solve correction equation
    $x^{(k+1)} = x^{(k)} + \delta x^{(k)}$
end for
```

Stability and ill-conditioning

Iterative refinement

- Fixed precision IR: all computations use the same precision. Mixed precision IR: the most expensive parts in lower precision, residual computation in double precision.
- single: fp32: significant memory and data movement reduction.
- half: fp16: even greater potential, problems due to the small range of representable numbers

Theorem

One step of fp32 IR enough for obtaining componentwise relative backward error to the order of $O(\epsilon)$ under weak assumptions. Strong bound for the error norm using fp64 IR.

- A possible variation: Krylov subspace solver as IR

Stability and ill-conditioning

Scaling to reduce ill-conditioning

- Consider two nonsingular $n \times n$ diagonal matrices S_r and S_c. Diagonal scaling of the system $Ax = b$ transforms it to

$$S_r\, A\, S_c\, y = S_r\, b, \qquad y = S_c^{-1}\, x. \tag{49}$$

Theorem

Let the matrix A be SPD and let D_A be the diagonal matrix with entries a_{ii} ($1 \le i \le n$). Then for all diagonal matrices D with positive entries

$$\kappa(D_A^{-1/2}\, A\, D_A^{-1/2}) \le nz_{rmax}\, \kappa(D^{-1/2}\, A\, D^{-1/2}),$$

where nz_{rmax} is the maximum number of entries in a row of A.

Stability and ill-conditioning

Equilibration scaling

- Finding an appropriate scaling is an open question, but a number of heuristics have been proposed.
- An obvious choice is to seek to balance among entries of the scaled matrix $S_r A S_c$: to have approximately equal absolute values. This is called (approximate) equilibration scaling.

Stability and ill-conditioning

Equilibration scaling: an algorithm to find it

Algorithm (**Equilibration scaling in the infinity norm**)

Input: *The matrix A and convergence tolerance $\delta > 0$.*
Output: *Diagonal scaling matrices S_r and S_c.*

```
$B^{(1)} = A$, $D^{(1)} = I$, $E^{(1)} = I$
for $k = 1, 2, \ldots$ do
    Compute $\|B^{(k)}_{i,1:n}\|_\infty$ and $\|B^{(k)}_{1:n,i}\|_\infty$, $1 \le i \le n$    ▷ $i$-th row and column of $B^{(k)}$
    if $max_i \left\{ |1 - \|B^{(k)}_{i,1:n}\|_\infty| \right\} \le \delta$ and $max_i \left\{ |1 - \|B^{(k)}_{1:n,i}\|_\infty| \right\} \le \delta$ exit for loop
    $R = diag\left(\sqrt{\|B^{(k)}_{i,1:n}\|_\infty}\right)$ and $C = diag\left(\sqrt{\|B^{(k)}_{1:n,i}\|_\infty}\right)$
    $B^{(k+1)} = R^{-1} B^{(k)} C^{-1}$,   $D^{(k+1)} = D^{(k)} R^{-1}$, $E^{(k+1)} = E^{(k)} C^{-1}$
end for
$S_r = D^{(k+1)}$ and $S_c = E^{(k+1)}$
```

Outline

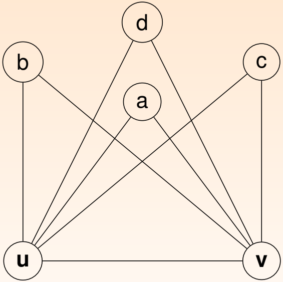
1. Introduction
2. Introductory notation and terminology
3. Factorizations
4. Graphs and sparse matrices
5. Sparse matrices and data structures
6. Symbolic factorizations
7. Symbolic Cholesky factorization
8. Cholesky factorization
9. Sparse LU factorization
10. Stability, ill-conditioning, indefiniteness
11. Blocks in the input matrix
12. Sparse Least Squares and QR factorization
13. Reorderings
14. Algebraic preconditioning
15. Incomplete factorizations
16. Sparse approximate inverses

Blocks in the input matrix

Finding blocks: indistinguishability

Vertices u and v of $\mathcal{G}$ are indistinguishable if

$$Adj_{\mathcal{G}}(u) \cup \{u\} = Adj_{\mathcal{G}}(v) \cup \{v\}.$$



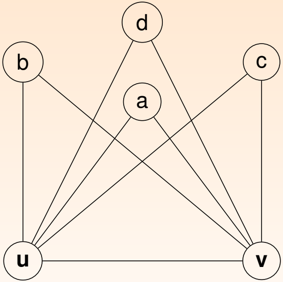
- On the right we see what happens if u or v is eliminated (keeping u and v in the figure)

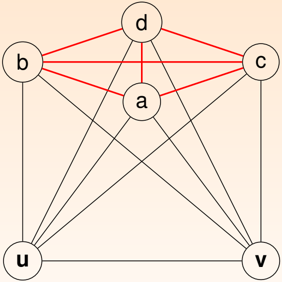
Blocks in the input matrix

Finding blocks: indistinguishability

Vertices u and v of $\mathcal{G}$ are indistinguishable if

$$Adj_{\mathcal{G}}(u) \cup \{u\} = Adj_{\mathcal{G}}(v) \cup \{v\}.$$





- On the right we see what happens if u or v is eliminated (keeping u and v in the figure)

Blocks in the input matrix

Finding blocks: using indistinguishability

- Indistinguishability is an equivalence relation on $\mathcal{V}$
- Maximal indistinguishable vertex sets represent its classes $\rightarrow$ a partitioning of $\mathcal{V}$ into $nsup \geq 1$ non-empty disjoint subsets

$$\mathcal{V} = \mathcal{V}_1 \cup \mathcal{V}_2 \cup \ldots \cup \mathcal{V}_{nsup}.$$

- Assume $\mathcal{S}\{A\}$ symmetric, $\mathcal{G}(A) = (\mathcal{V}, \mathcal{E})$.
- Let $\mathcal{V}$ be partitioned into indistinguishable vertex sets and reorder the vertices such that those belonging to each subset $\mathcal{V}_1, \ldots, \mathcal{V}_{nsup}$ are numbered consecutively, with those in $\mathcal{V}_i$ preceding those in $\mathcal{V}_{i+1}$ $(1 \leq i < nsup)$.
- Then then PAP^T has a block structure in which the blocks are dense, where P corresponds to the reordering above.

Blocks in the input matrix

A simple way to find indistinuishable sets: the approach by Ashcraft

1. The main idea: finding reasons why vertices cannot be in the same set
2. Number graph nodes/vertices (use numbers as their labels)
3. Compute vertex checksums:

$$chksum(u) = \sum_{\{u,v\} \in E} v$$

4. Sort vertices by their checksums: in O(|E|+|V|log(|V|)) time
5. Different checksum means different block
6. First tie-breaking rule: if $chksum(u) = chksum(v)$: compare $|adj(u)|$ and $|adj(v)|$
7. Second tie-breaking rule: compare adjacency sets of u and v (the most time consuming)

Blocks in the input matrix

Approximate indistinguishability: Saad

- Using symbolic dot products between the rows of the matrix.
- Here we assume that $\mathcal{S}\{A\}$ is symmetric but modifications exist.
- Rewrite A as row vectors

  $$A = \left(a_1^T, \ldots, a_n^T\right)^T ,$$

  and consider $\mathcal{G}(A) = (\mathcal{V}, \mathcal{E})$.
- A partition $\mathcal{V} = \mathcal{V}_1 \cup \ \ldots \ \cup \ \mathcal{V}_{nb}$ is constructed using row products $a_i^T a_k$ between different rows of A that express the level of orthogonality between the rows;
- if $a_i^T a_k$ is large then i and k (very similar pattern) are assigned to the same vertex set.

Blocks in the input matrix

Approximate indistinguishability: notes on the subsequent algorithm

- Algorithm treats all entries of A as unity (symbolically)
- The symbolic row products: a generalization of the angles between rows expressed by their cosines.
- On output, if $adjmap(i_1) = adjmap(i_2)$ then vertices i_1 and i_2 belong to the same vertex set.
- Symmetry of $\mathcal{S}\{A\}$ simplifies the computation of the symbolic row products because for row i only $k > i$ is considered: only the symbolic row products that correspond to a triangular part of $A^T A$ are checked.
- Approximativeness is controlled by a threshold parameter $\tau \in (0, 1]$.
- Remind: procedure is sequential and thus similar to approximate sequential coloring schemes.

Blocks in the input matrix

Algorithm

Find approximately indistinguishable vertex sets

```
$nb = 0$, $adjmap(1:n) = 0$, $cosine(1:n) = 0$
for $i = 1:n$ do
    if $adjmap(i) = 0$ then
        $nb = nb + 1$                                        ▷ Start a new set
        $adjmap(i) = ib$
        for $(i,j) \in \mathcal{E}$ do                       ▷ Corresponds to an entry in $A_{i,1:n}$
            for $(k,j) \in \mathcal{E}$ with $k > i$ do      ▷ Both rows $i$ and $k$ have an entry in column $j$
                if $adjmap(k) = 0$ then                      ▷ $k$ has not been yet added to some partitioning set
                    $cosine(k) = cosine(k) + 1$              ▷ Increase partial dot product
                end if
            end for
            for $k$ with $cosine(k) \neq 0$ do
                if $cosine(k)^2 \geq \tau^2 * nz_i * nz_k$ then   ▷ Test similarity of row patterns
                    $adjmap(k) = nb$
                end if
                $cosine(k) = 0$
            end for
        end for
    end if
```

Blocks in the input matrix

Approximate indistinguishability: an example

$$\begin{array}{c|cccccc} & 1 & 2 & 3 & 4 & 5 & 6 \\ \hline 1 & \ast & & \ast & & \ast & \\ 2 & & \ast & & & \ast & \ast \\ 3 & \ast & & \ast & & \ast & \\ 4 & & & & \ast & & \\ 5 & \ast & \ast & \ast & & \ast & \ast \\ 6 & & \ast & & & \ast & \ast \end{array} \qquad \begin{array}{c|cccccc} & 1 & 3 & 2 & 6 & 4 & 5 \\ \hline 1 & \ast & \ast & & & & \ast \\ 3 & \ast & \ast & & & & \ast \\ 2 & & & \ast & \ast & & \ast \\ 6 & & & \ast & \ast & & \ast \\ 4 & & & & & \ast & \\ 5 & \ast & \ast & \ast & \ast & & \ast \end{array} \qquad \begin{array}{c|cccccc} & 1 & 3 & 5 & 2 & 6 & 4 \\ \hline 1 & \ast & \ast & \ast & & & \\ 3 & \ast & \ast & \ast & & & \\ 5 & \ast & \ast & \ast & \ast & \ast & \\ 2 & & & \ast & \ast & \ast & \\ 6 & & & \ast & \ast & \ast & \\ 4 & & & & & & \ast \end{array}$$

Figure: *The original matrix is given (left) together with the permuted matrix with indistinguishable vertex sets* $\mathcal{V} = \{1,3\} \cup \{2,6\} \cup \{4\} \cup \{5\}$ *obtained using* $\tau = 1$ *(centre) and the permuted matrix with approximately indistinguishable vertex sets* $\mathcal{V} = \{1,3,5\} \cup \{2,6\} \cup \{4\}$ *obtained using* $\tau = 0.5$ *(right). The threshold* $\tau = 0.5$ *results in putting row* 5 *into the same set as row* 1*, making the vertex sets only approximately indistinguishable. The permuted matrix has an approximate block form.*

Blocks in the input matrix

Finding supervariables: another splitting idea

$$\begin{array}{c c}
 & \begin{array}{ccccc} 1 & 2 & 3 & 4 & 5 \end{array} \\
\begin{array}{c} 1 \\ 2 \\ 3 \\ 4 \\ 5 \end{array} & \begin{pmatrix} * & * & & & * \\ * & * & & & * \\ & & * & * & * \\ & & * & * & * \\ * & * & * & * & * \end{pmatrix}.
\end{array}$$

- Initially, $1, 2, 3, 4, 5$ are put into a single vertex set $\mathcal{V}_1$.
- Vertices $i = 1, 2$ and 5 belong to $adj_{\mathcal{G}}\{1\} \cup \{1\}$; they are moved from $\mathcal{V}_1$ into a new vertex set.
- There is no further splitting of the vertex sets for $j = 2$.
- $adj_{\mathcal{G}}\{3\} \cup \{3\} = \{3, 4, 5\}$. $i = 3$ and 4 are moved from $\mathcal{V}_1$ into a new vertex set. $\mathcal{V}_1$ is now empty and can be discarded. Vertex $i = 5$ is moved from the vertex set that holds vertices 1 and 2 into a new vertex set. For $j = 4$ and 5 no additional splitting is performed.
- Three supervariables are found, namely $\{1, 2\}$, $\{3, 4\}$ and $\{5\}$.

Blocks in the input matrix

Splitting idea and supervariables: more formally

- Initially place all the vertices in a single vertex set (that is, into a single supervariable).
- Split into two supervariables by taking the first vertex $j = 1$ and moving those vertices that are in the adjacency set of j into a new vertex set (a new supervariable).
- Each vertex j is considered in turn and each vertex set $\mathcal{V}_{sv}$ that contains a vertex in $adj_{\mathcal{G}}\{j\} \cup j$ is split into two by moving the vertices in $adj_{\mathcal{G}}\{j\} \cup j$ that belong to $\mathcal{V}_{sv}$ into a new vertex set.
- As a result of the splitting and moving of vertices a vertex set can become empty, in which case it is discarded.
- Once the supervariables have been determined, the permuted matrix PAP^T can be condensed to a matrix of order equal to $nsup$.

Blocks in the input matrix

Finding supervariables

Algorithm (**Find the supervariables of an undirected graph)**

Input: *Graph $\mathcal{G}$ of a symmetrically structured matrix.*
Output: *Partitioning of $\mathcal{V}$ into indistinguishable vertex sets.*

```
$\mathcal{V}_1 = \{1, 2, \ldots, n\}$
for $j = 1 : n$ do
    for $i \in adj_{\mathcal{G}}\{j\} \cup j$ do
        Find $sv$ such that $i \in \mathcal{V}_{sv}$
        if this is the first occurrence of $sv$ for the current index $j$ then
            Establish a new set $\mathcal{V}_{nsv}$ and move $i$ from $\mathcal{V}_{sv}$ to $\mathcal{V}_{nsv}$
        else
            Move $i$ from $\mathcal{V}_{sv}$ to $\mathcal{V}_{nsv}$
        end if
        Discard $\mathcal{V}_{sv}$ if it is empty
    end for
end for
```

Outline

1. Introduction
2. Introductory notation and terminology
3. Factorizations
4. Graphs and sparse matrices
5. Sparse matrices and data structures
6. Symbolic factorizations
7. Symbolic Cholesky factorization
8. Cholesky factorization
9. Sparse LU factorization
10. Stability, ill-conditioning, indefiniteness
11. Blocks in the input matrix
12. **Sparse Least Squares and QR factorization**
13. Reorderings
14. Algebraic preconditioning
15. Incomplete factorizations
16. Sparse approximate inverses

Sparse Least Squares and QR factorization

Least squares: our setting

$$\boxed{\textit{Given } A \in \mathbb{R}^{m\times n} \textit{ of rank } k \leq \min(m,n) \textit{ and } b \in \mathbb{R}^m}$$

$$\boxed{\textit{find } x \in \mathbb{R}^n \textit{ that minimises } \|b - Ax\|_2.}$$

- Assuming (mostly) overdetermined problems: $m > n$. But we still like to put them into a more general context.
- In this case we have

Theorem

x is a solution of such least squares (LS) problem $\Leftrightarrow$ x satisfies the $n \times n$ normal equations

$$Cx = A^T b, \quad C = A^T A,$$

Sparse Least Squares and QR factorization

Least squares: special case and rank deficiency

Lemma

Let $m > n$. The normal matrix $C = A^T A$ is SPD $\Leftrightarrow rank(A) = n$. In this case, the unique least squares solution and corresponding residual are

$$x = (A^T A)^{-1} A^T b \qquad \textit{and} \qquad r = b - A(A^T A)^{-1} A^T b.$$

$A^+ = (A^T A)^{-1} A^T$ is the pseudoinverse of A.

- If $rank(A) < n$ (rank deficient) then A has a null space of dimension $n - rank(A) > 0$ (solution is not unique) In this case, we can seek the least-norm solution, that is, we solve

$$\min_{x \in \mathcal{S}} \|x\|_2, \quad \mathcal{S} = \{x \in \mathbb{R}^n \mid \|b - Ax\|_2 = \min\}.$$

This solution is always unique.

Sparse Least Squares and QR factorization

Least squares: an augmented indefinite system

- Another equivalent formulation
- The normal equations are equivalent to the linear equations $A^T r = 0$, and $r = b - Ax$. Together: the $(m+n) \times (m+n)$ augmented system

$$K \begin{pmatrix} z \\ x \end{pmatrix} = \begin{pmatrix} b \\ c \end{pmatrix} \quad \text{with} \quad K = \begin{pmatrix} I & A \\ A^T & 0 \end{pmatrix},$$

  with $z = r$ and $c = 0$.
- The symmetric indefinite matrix K is non singular if and only if $rank(A) = n$.
- And symmetric indefinite solvers can be used.

Sparse Least Squares and QR factorization

Least squares: sensitivity

- Sensitivity quantification: the condition number
- For the normal matrix:

$$\kappa_2(A) = \kappa_2(R) = (\kappa_2(A^T A))^{1/2}.$$

- But, sensitivity of LS also determined by the right-hand side vector b and more complex condition numbers are then introduced. Like $\kappa_{LS}(A,b)$ that considers only model (and not data) perturbations.

$$\kappa_{LS}(A,b) = \kappa_2(A)\left(1 + \kappa_2(A)\frac{\|r\|_2}{||A||_2||x||_2}\right), \text{with} \quad r = \|r\|_2/\|b\|_2;$$

where x minimizes $\|b - Ax\|_2$: that is, we have residual norm dependence.

Sparse Least Squares and QR factorization

Least squares: rank deficiency

- Rank deficiency due to small singular values of A plus the noise are serious practical problems.
- Rank-deficiency difficult to detect.
- There exist rank-revealing approaches but they may be expensive.
- We do not want to deal with problematic problems.
- Our assumption was to have A regular: but, rank deficiency often encountered. And solvers must be ready to cope with such situations.

Sparse Least Squares and QR factorization

Least squares: regularization

- To cope with the rank deficiency: LS problems have to be modified (regularized)
- Regularization: extract the linearly independent information from A and noisy b
- Solving such systems we get only approximate solutions: recall the interplay between direct and iterative methods.
- Consider the two solution approaches so far at hand, taking into account also regularization.

- We will see that: forming A^TA squares the condition number. But solving symmetric indefinite systems may be problematic as well.

Sparse Least Squares and QR factorization

Least squares: two solution approaches so far

- 1. SPD (Cholesky) factorization of $A^T A$
-
$$\min_{x \in \mathbb{R}^n} (\|b - Ax\|_2^2 + \gamma^2 \|x\|_2^2),$$
- If $\gamma > \sigma_{min}(A)$, we have
$$\kappa(A^T A + \gamma^2 I) \approx (\|A\|_2/\gamma)^2$$
- Not a big progress, since γ should be kept small.

Sparse Least Squares and QR factorization

Least squares: two solution approaches so far

- 2. Symmetric indefinite factorizations of K
- Equivalent to

$$\min_{x\in\mathbb{R}^n} \left\| \begin{pmatrix} b \\ 0 \end{pmatrix} - \begin{pmatrix} A \\ \gamma\, I \end{pmatrix} x \right\|_2 .$$

- This leads to

$$\begin{pmatrix} I & A \\ A^T & -\gamma^2 I \end{pmatrix} \begin{pmatrix} r \\ x \end{pmatrix} = \begin{pmatrix} b \\ 0 \end{pmatrix} \text{ or } K_\gamma \begin{pmatrix} s \\ x \end{pmatrix} = \begin{pmatrix} b \\ 0 \end{pmatrix},\ K_\gamma = \begin{pmatrix} \gamma I & A \\ A^T & -\gamma I \end{pmatrix},\ r = \gamma s.$$

- If $\gamma > \sigma_{min}(A)$, we have $\kappa(K_\gamma) \approx \|A\|_2/\gamma$.
- Seems to be better, but indefiniteness.

Sparse Least Squares and QR factorization

Least squares: an additional solution approach

- One additional solution strategy: QR factorization based approach.
-
$$A = (Q_1 \;\; Q_2) \begin{pmatrix} R \\ 0 \end{pmatrix} = Q_1 R,$$
- $Q = (Q_1 \;\; Q_2)$ is orthogonal, $R \in \mathbb{R}^{m \times n}$ is upper triangular.

Sparse Least Squares and QR factorization

Least squares: QR factorization

- $$A = (Q_1 \;\; Q_2) \begin{pmatrix} R \\ 0 \end{pmatrix} = Q_1 R,$$
- Multiplication by orthogonal matrices does not change the Euclidean norm ⇒
  $$\|b - Ax\|_2^2 = \|Q^T(b - Ax)\|_2^2 = \|Q_1^T b - Rx\|_2^2 + \|Q_2^T b\|_2^2.$$
- The solution of LS problem and the residual can be computed:
  $$Rx = d_1, \; r = Q \begin{pmatrix} 0 \\ d_2 \end{pmatrix} \quad \text{where} \quad Q^T b = \begin{pmatrix} d_1 \\ d_2 \end{pmatrix}.$$
- $Q^T b$ can formally be obtained by applying the QR factorization to $\begin{pmatrix} A & b \end{pmatrix}$ and storing Q can then be avoided.

Sparse Least Squares and QR factorization

Least squares: problematic aspects

- Structurally (sparsity point of view), an effect similar to forming normal equations also in the QR factorization:

$$A^T A = (Q_1 R)^T Q_1 R = R^T (Q_1^T Q_1) R = R^T R,$$

- That is, R factor is mathematically equivalent to the transpose of the L factor of the Cholesky factorization of the normal matrix.

- Consequently, no royal way, we need to develop more solution approaches, any of them can be better in particular situations.
- So, let us discuss also the sparse QR factorization.

Sparse Least Squares and QR factorization

Least squares: orthogonalization approaches

- 1. Givens rotations
- A counterclockwise rotation of a nonzero vector $w = (w_1\ w_2)^T \in \mathbb{R}^2$ through an angle θ such that y_2 of the rotated vector $y = (y_1\ y_2)^T$ is zero.

$$G \begin{pmatrix} w_1 \\ w_2 \end{pmatrix} = \begin{pmatrix} c & -s \\ s & c \end{pmatrix} \begin{pmatrix} w_1 \\ w_2 \end{pmatrix} = \begin{pmatrix} y_1 \\ y_2 \end{pmatrix} = \begin{pmatrix} d \\ 0 \end{pmatrix},$$

$$\text{where } s = w_2/d,\ c = -w_1/d,\ d = \|w\|_2.$$

- Can be expressed as a plane rotation in the extended space.

Sparse Least Squares and QR factorization

QR factorization: Givens rotations

- If the two axes correspond to row indices i and j of A then the $m \times m$ matrix $G(i,j)$ given by

$$G(i,j) = \begin{array}{c} \\ \\ \\ i \\ \\ j \\ \\ \\ \end{array}\!\!\begin{array}{c} \begin{array}{ccccccc} & & i & & j & & \end{array} \\ \begin{pmatrix} 1 & \dots & 0 & \dots & 0 & \dots & 0 \\ \vdots & \ddots & \vdots & & \vdots & & \vdots \\ 0 & \dots & c & \dots & s & \dots & 0 \\ \vdots & & \vdots & \ddots & \vdots & & \vdots \\ 0 & \dots & -s & \dots & c & \dots & 0 \\ \vdots & & \vdots & & \vdots & \ddots & \vdots \\ 0 & \dots & 0 & \dots & 0 & \dots & 1 \end{pmatrix} \end{array},$$

with 1's on the diagonal except rows i and j.

- This enables to apply it to A as a matrix transformation.

Sparse Least Squares and QR factorization

QR factorization: Givens rotations

- Can be used to systematically eliminate individual subdiagonal entries of A by applying them one-by-one to pairs of rows.
- For example

$$G(n,m)\dots G(2,m)\dots G(2,3)\dots G(1,m)\dots G(1,2)A = R,$$

- Q is the transpose of the product of the Givens rotations.
- A sparse ⇒ sufficient to eliminate only the nonzero entries.
- The order of applying the rotations must satisfy some rules to avoid creating nonzeros (intermediate fill-in) as much as possible.

.

Sparse Least Squares and QR factorization

Least squares: orthogonalization approaches

- 2. Householder reflectors (HR)
- HR is a symmetric orthogonal matrix of the form

  $$H = I - \beta w w^T,$$

  β is a scalar, $w \neq 0$ satisfies

  $$y = Hx \Rightarrow |y_1| = ||x||_2, y_{2:n} = 0$$

- w is called a Householder vector.
- The application of an $m \times m$ Householder reflector $H^{(1)}$ to the matrix A with a_1 as its first column can be written as

$$H^{(1)}A = H^{(1)} \begin{pmatrix} a_1 & A_{1:m,2:n} \end{pmatrix} = (I - \beta_1 w^{(1)} (w^{(1)})^T) \begin{pmatrix} a_1 & A_{1:m,2:n} \end{pmatrix} = \begin{pmatrix} R_{1,1} & R_{1,2:n} \\ 0 & A^{(1)} \end{pmatrix}.$$

Sparse Least Squares and QR factorization

QR factorization: getting Householder reflectors (HR)

- Elimination in the first column of A
- Applying an $(m-1)\times(m-1)$ HR $H_2 = I - \beta_2 w^{(2)}(w^{(2)})^T$ to $A^{(1)}$ such that its $(1,1)$ entry becomes zero.
- H_2 extended to the full dimension (to an $m \times m$ matrix) by setting

$$H^{(2)} = \begin{pmatrix} 1 & \\ & H_2 \end{pmatrix}.$$

- The sequence of HR and their applications

$$A^{(0)} = A, \ldots, A^{(j)} = H^{(j)} A^{(j-1)}, j = 1, \ldots, n-1.$$

Sparse Least Squares and QR factorization

QR factorization: Householder reflectors

- Repeating the process yields $H^{(n)} \dots H^{(2)} H^{(1)} A = A^{(n)}$
- That is $A = QR, Q = H^{(1)} H^{(2)} \dots H^{(n)}$ and $R = A^{(n)}$.
- Complexity: compared with Givens rotations, Householder reflectors reduce the flops count by a third (still three-times more than LU) for a dense square A.

Sparse Least Squares and QR factorization

QR factorization: Householder reflectors

- Sparse A: more considerations needed:
  - Q can be held explicitly as the product of the HRs $H^{(1)}H^{(2)}\dots H^{(n)}$
  - Or implicitly as the sequence of Householder vectors $w^{(1)}, w^{(2)}\dots$.
- The latter is generally more memory efficient because $w^{(i)}$ are typically sparse.
- Storing blocks of Householder vectors allows the use of Level 3 BLAS.

Sparse Least Squares and QR factorization

QR factorization: Gram-Schmidt orthogonalisation

- The Gram-Schmidt process computes $q_1, q_2, \ldots, q_n$
- Each column a_k, $k = 1, \ldots, n$ of A can be expressed as a linear combination

$$a_k = r_{1k}q_1 + r_{2k}q_2 + \ldots r_{nk}q_n, r_{kk} \neq 0, \langle q_i,\ q_i \rangle = 1.$$

- This provides the QR factorization

$$A = Q_1 R, Q_1 = (q_1, q_2, \ldots, q_n) \in \mathbb{R}^{n \times n}$$

- Q_1 is orthonormal, obtained and held explicitly.
- More ways to orthogonalise by algorithms equivalent in exact arithmetic.

Sparse Least Squares and QR factorization

Gram-Schmidt orthogonalisation: two principial ways

- The classical Gram-Schmidt (CGS) process generates q_k by orthonormalising a_k against $Q_{k-1} = (q_1, q_2, \dots, q_{k-1})$, that is

$$a = a_k - Q_{k-1}(q_1^T a_k \dots q_{k-1}^T a_k) a_k = a_k - Q_{k-1} Q_{k-1}^T a_k, r_{kk} = \|a\|_2, q_k = a/r_{kk}$$

- The modified Gram-Schmidt (MGS) algorithm: q_k is obtained by first projecting a_k onto the subspace orthogonal to q_1, then the result is projected onto $span\{q_1, q_2\}^\perp$, and so on, up to the projection onto the subspace $span\{q_1, q_2, \dots, q_{k-1}\}^\perp$.

$$a = (I - q_{k-1} q_{k-1}^T) \dots (I - q_1 q_1^T) a_k), r_{kk} = \|a\|_2, q_k = a/r_{kk}.$$

Sparse Least Squares and QR factorization

Gram-Schmidt orthogonalisation: and finite precision arithmetic

- Finite-precision arithmetic: MGS is not equivalent to CGS. The loss of orthogonality:

  proportional to $\kappa(A)$ (MGS), proportional to $\kappa(A)^2$ (CGS).

- CGS more suited to parallel computations, but often produces a non-orthogonal set of vectors.

- CGS can be improved by reorthogonalisation(s). Then no significant difference between this approach and MGS.

- MGS limits amplification of the rounding errors at the expense of less exploitable parallelism.

Sparse Least Squares and QR factorization

Contemporary sparse QR

- Consider a symbolic phase predicting R or Q, (GR, H vectors) first
- A Givens rotation $G(i,j)$ applied to $A_{i,i:n}$ and $A_{j,i:n}$ of A:

$$\begin{pmatrix} c & -s \\ s & c \end{pmatrix} \begin{pmatrix} A_{i,i:n} \\ A_{j,i:n} \end{pmatrix} = \begin{pmatrix} A'_{i,i:n} \\ A'_{j,i:n} \end{pmatrix}, \; A'_{j,i} = 0.$$

- An example that emphasizes sparsity patterns:

$$\begin{pmatrix} A_{i,i:n} \\ A_{j,i:n} \end{pmatrix} = \begin{pmatrix} * & * & * & & & * & & * \\ * & & & & * & & & \end{pmatrix}.$$

Applying $G(i,j)$ gives

$$\begin{pmatrix} c & -s \\ s & c \end{pmatrix} \begin{pmatrix} A_{i,i:n} \\ A_{j,i:n} \end{pmatrix} = \begin{pmatrix} * & * & * & & * & * & & * \\ & * & * & & * & * & & * \end{pmatrix} = \begin{pmatrix} A'_{i,i:n} \\ A'_{j,i:n} \end{pmatrix}.$$

Sparse Least Squares and QR factorization

Contemporary sparse QR: symbolic phase

$$\begin{pmatrix} c & -s \\ s & c \end{pmatrix} \begin{pmatrix} A_{i,i:n} \\ A_{j,i:n} \end{pmatrix} = \begin{pmatrix} * & * & * & & * & * & & & * \\ & * & * & & * & * & & & * \end{pmatrix} = \begin{pmatrix} A'_{i,i:n} \\ A'_{j,i:n} \end{pmatrix}.$$

- The $(1,1)$ entry $A'_{i,i}$ seems to remain nonzero (it is the Euclidean norm of the vector $(A_{ii}\ A_{ji})^T$) and the sparsity patterns of columns 2 to n satisfy

$$\mathcal{S}(A'_{i,i+1:n}) = \mathcal{S}(A_{i,i+1:n}) \cup \mathcal{S}(A_{j,i+1:n}), 1 \le i \le n-1.$$

Sparse Least Squares and QR factorization

Contemporary sparse QR: symbolic phase

- This way of fill-in description in the process is called the local merge rule.

- The rule based on merging of patterns is understandable since if $(u\ v)^T$ is transformed by an arbitrary Givens rotation

$$\begin{pmatrix} c & -s \\ s & c \end{pmatrix} \begin{pmatrix} u \\ v \end{pmatrix} = \begin{pmatrix} cu - sv \\ su + cv \end{pmatrix} = \begin{pmatrix} u' \\ v' \end{pmatrix},$$

- Then both entries u' and v' are generally nonzero (unless θ is a multiple of a right angle).

Contemporary sparse QR: symbolic phase

- However, the fill-in can be overestimated. Consider $a, b \neq 0$

$$\begin{pmatrix} * & a & b & & \\ * & & & * & * \\ * & & & * & * \end{pmatrix} \to \begin{pmatrix} * & c'ca & c'cb & & \\ & sa & sb & * & * \\ & s'ca & s'cb & * & * \end{pmatrix}$$

$$\to \begin{pmatrix} * & a & b & & \\ & c''sa - s''s'ca & c''sb - s''s'cb & * & * \\ & s''sa + c''s'ca & s''sb + c''s'cb & * & * \end{pmatrix}.$$

- First step: apply $G(2,1)$ with c, s to eliminate the $(2,1)$ entry.
- Second step: apply $G(3,1)$ with c', s' to eliminate the $(3,1)$ entry.
- Then, eliminate the fill-in at $(3,2)$ by rotation with c'', s''.
- We have $s''sa + c''s'ca = 0$.
- But this a nonzero multiple of the entry $s''sb + c''s'cb$ at $(3,3)$, independently of general values of a and b.
- The row merge rule is not able to predict that the $(3,3)$ entry also always becomes zero.

Sparse Least Squares and QR factorization

Contemporary sparse QR: HR

- Similarly, as by Givens rotations, the sparsity pattern of any row of $\begin{pmatrix} R_{1,2:n} \\ A^{(1)} \end{pmatrix}$ can be obtained as

$$\mathcal{S}(row) = \mathcal{S}(R_{1,2:n}) \bigcup_k \mathcal{S}(A^{(1)}_{k,1:n-1}).$$

- This extends the row merge rule for Householder reflectors.
- That is, $\mathcal{S}(row)$ unifies sparsity patterns of all the rows involved in the outer product update.
- But, the extended row merge rule may also overestimate the actual fill in.

Sparse Least Squares and QR factorization

Contemporary sparse QR: another possibility for symbolic QR

- Potentially possible to predict fill-in in R using $\mathcal{S}(A^TA) = \mathcal{S}(LL^T)$.
- That is, uniqueness of the Cholesky factorization $\Rightarrow$ the Cholesky factor of A^TA can be used to predict $\mathcal{S}(R)$. However, this can again lead to an overestimate. Consider

$$A = \begin{pmatrix} * & * & * \\ & * & \\ & & * \\ & & * \end{pmatrix}. \tag{50}$$

- A^TA is dense and $\mathcal{S}(L)$ is thus predicted to be dense.
- But the QR factorization of A requires only to eliminate the $(4,3)$ entry. Givens rotation applied to the last two rows of A does not add any fill-in and $\mathcal{S}(R) = \mathcal{S}(A_{1:3,:})$.

Sparse Least Squares and QR factorization

Contemporary sparse QR: another possibility for symbolic QR

- The relationship between the different predictions is summarised by the following crucial result which is independent of the numerical values of the nonzero entries of A.

Lemma

$\mathcal{S}(R) \subseteq \{$prediction of $\mathcal{S}(R)$ based on row merge rule $\} \subseteq \{$prediction of $\mathcal{S}(R)$ based on $A^T A\}$.

Sparse Least Squares and QR factorization

Contemporary sparse QR: predictions and the Hall properties

- $A \in \mathbb{R}^{m \times n}$ is said to be a Hall matrix (or to have the Hall property) if every set of k columns has nonzeros in at least k rows $(1 \le k \le n)$.
- A full-rank matrix must have the Hall property.
- A is a strong Hall matrix (or to have the strong Hall property) if every set of k columns $(1 \le k < n)$ has nonzeros in at least $k+1$ rows.
- The following matrix does not have the strong Hall property: its first column has a single nonzero entry.

$$A = \begin{pmatrix} * & * & * \\ & * & \\ & & * \\ & & * \end{pmatrix}.$$

Sparse Least Squares and QR factorization

Contemporary sparse QR: predictions and the Hall properties

Lemma

If A has the strong Hall property then $\mathcal{S}(R)$ is exactly predicted by the local merge rule and the Cholesky factorization of A^TA.

- There is a bunch of nice results on predictions like:
- Exact predictions for Q and for the matrix W whose columns are the Householder vectors. Such predictions are possible even for A without the strong Hall property.
- But of limited use: in practical QR factorizations also have to consider the rank-deficiency and close-to-rank-deficiency.

Sparse Least Squares and QR factorization

Contemporary sparse QR: predictions and the Hall properties

- Recall the Dulmage-Mendelsohn (DM) decomposition of A: obtained using maximum matching algorithms, provides a precise structural characterization of rectangular matrices.
- For an overdetermined A, the DM decomposition is

$$P_1 A P_2 = \begin{pmatrix} A_1 & A_2 \\ 0 & A_3 \end{pmatrix}, P_1 \text{ and } P_2 \text{ are permutation matrices.}$$

- A_1 is square, A_3 is overdetermined matrix with the strong Hall property: possible to predict R. An example:

$$P_1 A P_2 = \left(\begin{array}{cccc|cc} * & & & & & * \\ * & * & & * & & \\ & & * & * & & \\ & & * & * & * & \\ \hline & & & & * & * \\ & & & & & * \\ & & & & * & \end{array} \right).$$

Sparse Least Squares and QR factorization

Contemporary sparse QR: predictions and the Hall properties

- A_1 is square, A_3 is overdetermined matrix with the strong Hall property: possible to predict R. An example:

$$P_1AP_2 = \left(\begin{array}{cccc|cc} * & & & & & * \\ * & * & & * & & \\ & & * & * & & \\ & & * & * & * & \\ \hline & & & & * & * \\ & & & & & * \\ & & & & * & \end{array}\right).$$

- But: if A is ill conditioned or close to rank deficient: this may not be sufficient to factorize only the strong Hall blocks.
- In the other words, the structure is not enough also for QR.

Sparse Least Squares and QR factorization

Contemporary sparse QR: row order of A

- The row order does not influence $\mathcal{S}(R)$.
- But, row ordering can significantly affect the intermediate fill (and the work needed to compute the factorization).
- An example

$$A = \begin{pmatrix} * & * & * \\ * & & \\ * & & \\ * & & \\ & * & \\ & & * \end{pmatrix}, \qquad PA = \begin{pmatrix} * & & \\ * & & \\ * & & \\ * & * & * \\ & * & \\ & & * \end{pmatrix}.$$

- Eliminating the $(2,1)$ and $(3,1)$ entries using GR $G(1,2)$ and $G(1,3)$, there is intermediate fill-in in all remaining columns.
- If rows 1 and 4 of A are exchanged (PA): this fill does not occur.
- Heuristic algorithms to find a suitable row ordering needed.

Sparse Least Squares and QR factorization

Contemporary sparse QR: row order of A

Algorithm

Row ordering of A

Input: *The column indices $f_i(A)$ and $l_i(A)$ of the first and last nonzeros in row i of A.*

```
1: Order the rows by increasing $f_i(A)$.
2: for $k = 1 : \max_i f_i(A)$ do
3:     Order all rows with $f_i(A) = k$ by increasing $l_i(A)$        ▷ Needed to resolve ties
4: end for
```

- This is a simple row ordering approach. Ties at Line 3: ordering the rows in ascending order of the number of new fill-in entries.
- There are alternative strategies, like order the rows in ascending order of $l_i(A)$.

Sparse Least Squares and QR factorization

Contemporary sparse QR: row merge tree

- Row merge tree generalises GR to submatrix rotations that merge triangular submatrices.
- The elimination tree $\mathcal{T}(A^TA)$ can be used to control the order in which the triangular submatrices are merged.
- Illustrating the basic principle

$$A = \begin{array}{c} \\ 1 \\ 2 \\ 3 \\ 4 \\ 5 \\ 6 \\ 7 \end{array} \!\!\begin{array}{c} \begin{array}{cccc} 1 & 2 & 3 & 4 \end{array} \\ \begin{pmatrix} * & & * & \\ * & & * & \\ * & & * & * \\ & * & * & \\ & * & * & \\ & * & * & * \\ & & * & * \end{pmatrix} \end{array}.$$

Sparse Least Squares and QR factorization

Contemporary sparse QR: row merge tree

- Illustrating the basic principle

$$A = \begin{array}{c} \\ 1 \\ 2 \\ 3 \\ 4 \\ 5 \\ 6 \\ 7 \end{array} \begin{array}{c} \begin{array}{cccc} 1 & 2 & 3 & 4 \end{array} \\ \begin{pmatrix} * & & * & \\ * & & * & \\ * & & * & * \\ & * & * & \\ & * & * & \\ & * & * & * \\ & & * & * \end{pmatrix} \end{array}.$$

- Let A_1 be the submatrix comprising rows $1, 2, 3$ and columns $1, 3, 4$ of A and A_2 be the submatrix comprising rows $4, 5, 6$ and columns $2, 3, 4$.

Sparse Least Squares and QR factorization

Contemporary sparse QR: row merge tree

- Illustrating the basic principle

$$A = \begin{array}{c|cccc|}
 & 1 & 2 & 3 & 4 \\
1 & {\color{red}\star} & & {\color{red}\star} & \\
2 & {\color{red}\star} & & {\color{red}\star} & \\
3 & {\color{red}\star} & & {\color{red}\star} & {\color{red}\star} \\
4 & & {\color{blue}\star} & {\color{blue}\star} & \\
5 & & {\color{blue}\star} & {\color{blue}\star} & \\
6 & & {\color{blue}\star} & {\color{blue}\star} & {\color{blue}\star} \\
7 & & & {\color{blue}\star} & {\color{blue}\star}
\end{array}.$$

- Let A_1 be the submatrix comprising rows $1, 2, 3$ and columns $1, 3, 4$ of A and A_2 be the submatrix comprising rows $4, 5, 6$ and columns $2, 3, 4$.

Sparse Least Squares and QR factorization

Contemporary sparse QR: row merge tree

- Illustrating the basic principle

$$A = \begin{array}{c} \\ 1 \\ 2 \\ 3 \\ 4 \\ 5 \\ 6 \\ 7 \end{array} \begin{array}{c} \begin{array}{cccc} 1 & 2 & 3 & 4 \end{array} \\ \begin{pmatrix} \star & & \star & \\ \star & & \star & \\ \star & & \star & \star \\ & \star & \star & \\ & \star & \star & \\ & \star & \star & \star \\ & & \star & \star \end{pmatrix} \end{array}.$$

- Let A_1 be the submatrix comprising rows $1, 2, 3$ and columns $1, 3, 4$ of A and A_2 be the submatrix comprising rows $4, 5, 6$ and columns $2, 3, 4$.
- Perform the QR factorizations $A_1 = Q'_1 R'_1$, $A_2 = Q'_2 R'_2$. Can be computed independently (different orthogonalization can be used).

Sparse Least Squares and QR factorization

Contemporary sparse QR: row merge tree

- if Q_1 and Q_2 denote the orthogonal matrices extended to $\mathbb{R}^{7\times 7}$, we get a partial orthogonal transformation of A

$$A = Q_1 Q_2 \begin{pmatrix} \star & & \star & \star \\ & & \star & \star \\ & & & \star \\ & \star & \star & \star \\ & & \star & \star \\ & & & \star \\ & & * & * \end{pmatrix}.$$

At this point, the last row $A_{7,1:4}$ has been unchanged yet (its first nonzero in in column 3).

- Next: start the merge step: permute the rows of the partial transformation corresponding to the first row of R_1' and the first row of R_2' to be rows 1 and 2 of the final factor R of A.

Sparse Least Squares and QR factorization

Contemporary sparse QR: row merge tree

- The merge step: permute the rows of the partial transformation corresponding to the first row of R_1' and the first row of R_2' to be rows 1 and 2 of the final factor R of A.
- This gives

$$\begin{array}{c} \\ 1 \\ 4 \\ 2 \\ 3 \\ 5 \\ 6 \\ 7 \end{array}\begin{array}{c} \begin{array}{cccc} 1 & 2 & 3 & 4 \end{array} \\ \left(\begin{array}{cccc} {\color{red}\bigstar} & & {\color{red}\bigstar} & {\color{red}\bigstar} \\ & {\color{blue}\bigstar} & {\color{blue}\bigstar} & {\color{blue}\bigstar} \\ & & {\color{red}\bigstar} & {\color{red}\bigstar} \\ & & & {\color{red}\bigstar} \\ & & {\color{blue}\bigstar} & {\color{blue}\bigstar} \\ & & & {\color{blue}\bigstar} \\ & & * & * \end{array}\right) \end{array}.$$

Sparse Least Squares and QR factorization

Contemporary sparse QR: row merge tree

- The remaining rows are just order by the row order algorithm above (or its variant) using their f_i. to give

$$\begin{array}{c|cccc} & 1 & 2 & 3 & 4 \\ \hline 1 & {\color{red}\star} & & {\color{red}\star} & {\color{red}\star} \\ 4 & & {\color{blue}\star} & {\color{blue}\star} & {\color{blue}\star} \\ 2 & & & {\color{red}\star} & {\color{red}\star} \\ 5 & & & {\color{blue}\star} & {\color{blue}\star} \\ 7 & & & * & * \\ 3 & & & & {\color{red}\star} \\ 6 & & & & {\color{blue}\star} \end{array}.$$

- Next: orthogonalization applied to the remaining rows.
- Can be generalised to more than two blocks.

Sparse Least Squares and QR factorization

Contemporary sparse QR: row merge tree

- The key observation: the precedence relations among the computed triangular factors of such blocks of rows are determined by $\mathcal{S}(\mathcal{R})$.
- Namely, before factorizing a block of rows with the first nonzero in column k, all block factorizations that result in upper triangular factors with a nonzero in column k must already have been performed.
- That is, this order is determined by precedence from the elimination tree $\mathcal{T}(A^T A)$.

Sparse Least Squares and QR factorization

Contemporary sparse QR: multifrontal QR factorization

- The strategy above using ns supernodes and $\mathcal{T}(A^TA)$ leads to the multifrontal algorithm.

Algorithm

Multifrontal QR factorization

Input: *Matrix A of full column rank and the postordered assembly tree of A^TA.*
Output: *Upper triangular factor R of the QR factorization, orthogonal transformations used to transform A stored implicitly or as their product Q.*

```
1: for $js = 1 : ns$ do                                        ▷ Follow the postordering of the tree
2:     Assemble frontal matrix $F(js)$ using rows of $A$ for which the index of the first
       nonzero entry belongs to $js$ and the QR contribution blocks from children of $js$
3:     Compute QR factorization of $F(js)$                    ▷ Results in a block row of $R$ and
       contribution block $R(js)$
4:     Push $R(js)$ onto the stack.          ▷ $R(js)$ will be popped from the stack when
       assembling $F(parent(js))$
5: end for
```

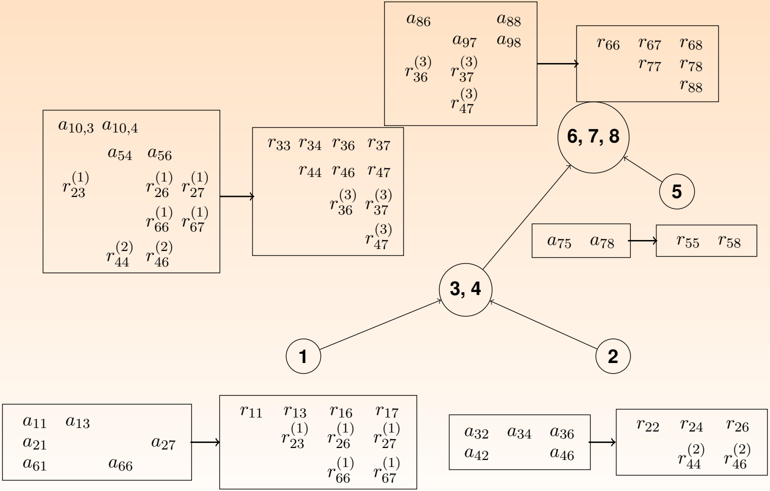
Contemporary sparse QR: multifrontal QR factorization

- Example of the multifrontal method

$$A = \begin{array}{c|cccccccc|}
 & 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 \\
1 & * & & * & & & & & \\
2 & * & & & & & & * & \\
3 & & * & & * & & * & & \\
4 & & * & & & & * & & \\
5 & & & & * & & * & & \\
6 & * & & & & & * & & \\
7 & & & & & * & & & * \\
8 & & & & & & * & & * \\
9 & & & & & & & * & * \\
10 & & & * & * & & & &
\end{array}$$

Sparse Least Squares and QR factorization

Contemporary sparse QR: multifrontal QR factorization


a_{86} a_{88}
a_{97} a_{98}
$r_{36}^{(3)}$ $r_{37}^{(3)}$
$r_{47}^{(3)}$
r_{66} r_{67} r_{68}
r_{77} r_{78}
r_{88}
6, 7, 8
5
a_{75} a_{78}
r_{55} r_{58}
$a_{10,3}$ $a_{10,4}$
a_{54} a_{56}
$r_{23}^{(1)}$ $r_{26}^{(1)}$ $r_{27}^{(1)}$
$r_{66}^{(1)}$ $r_{67}^{(1)}$
$r_{44}^{(2)}$ $r_{46}^{(2)}$
r_{33} r_{34} r_{36} r_{37}
r_{44} r_{46} r_{47}
$r_{36}^{(3)}$ $r_{37}^{(3)}$
$r_{47}^{(3)}$
3, 4
1
2
a_{11} a_{13}
a_{21} a_{27}
a_{61} a_{66}
r_{11} r_{13} r_{16} r_{17}
$r_{23}^{(1)}$ $r_{26}^{(1)}$ $r_{27}^{(1)}$
$r_{66}^{(1)}$ $r_{67}^{(1)}$
a_{32} a_{34} a_{36}
a_{42} a_{46}
r_{22} r_{24} r_{26}
$r_{44}^{(2)}$ $r_{46}^{(2)}$

Sparse Least Squares and QR factorization

Contemporary sparse QR: rank deficiency

- The QR factorization is backward stable but if A is (close to) rank deficient then the computed R factor is ill conditioned.
- If $rank(A) = rk < n$ then theoretically there is a column permutation matrix P (which is not necessarily unique) and an orthogonal matrix Q such that

$$AP = Q \begin{pmatrix} R_{11} & R_{12} \\ 0 & 0 \end{pmatrix},$$

Sparse Least Squares and QR factorization

Contemporary sparse QR: rank revealing QR

- More interesting problem: A diagonal entry $r_{rk+1,rk+1}$ of large absolute value computed after rk steps of the QR factorization may hide the fact that the rank of A is rk.
- Handling rank deficiency and revealing rank is an important component of QR factorizations.
- Rank revealing QR (RRQR) formally: $\sigma_i(A)$ of A ordered decreasingly, $\sigma_{rk}(R_{11})$ the smallest singular value of the first rk columns of AP and $\sigma_1(R_{22})$ be the largest singular value of R_{22}. RRQR if

  $$\sigma_{rk}(R_{11}) \geq \sigma_{rk}(A)/c \quad \text{and} \quad \sigma_1(R_{22}) \leq c\, \sigma_{rk+1}(A),$$

  where $c = c(k, n) > 0$ is bounded by a low-degree polynomial in rk and n.

Outline

1. Introduction
2. Introductory notation and terminology
3. Factorizations
4. Graphs and sparse matrices
5. Sparse matrices and data structures
6. Symbolic factorizations
7. Symbolic Cholesky factorization
8. Cholesky factorization
9. Sparse LU factorization
10. Stability, ill-conditioning, indefiniteness
11. Blocks in the input matrix
12. Sparse Least Squares and QR factorization
13. **Reorderings**
14. Algebraic preconditioning
15. Incomplete factorizations
16. Sparse approximate inverses

Reorderings

Minimizing the fill-in: reorderings

- Key problem: minimizing the fill-in
- Our tools: permutations.
- The problem of finding a permutation to minimize fill-in is NP complete. Thus heuristics are used to determine orderings that limit the amount of fill-in; we refer to these as fill-reducing orderings.
- Frequently, this is done using the sparsity pattern $\mathcal{S}\{A\}$ alone,
- If the matrix is not SPD, additional permutations of A may be needed to make the matrix factorizable.

Reorderings

Minimizing the fill-in: reorderings

Two main classes of reorderings that work with $\mathcal{S}\{A\}$ are commonly used.

Local orderings attempt to limit fill-in by repeated local decisions based on $\mathcal{G}(A)$ (or a relevant quotient graph).

Global orderings consider the whole sparsity pattern of A and seek to find a permutation using a divide-and-conquer approach. Such methods are normally used in conjunction with a local fill-reducing ordering, as the latter generally works well for problems that are not really large.

Minimizing the fill-in: reorderings

- Assumed that A is irreducible. If not,
  - If $\mathcal{S}\{A\}$ is symmetric, the algorithms are applied to each component of $\mathcal{G}(A)$ independently and n is then the number of vertices in the component.
  - If $\mathcal{S}\{A\}$ is nonsymmetric, we assume that A is in block triangular form and the algorithms are used on the graph of each block on the diagonal.
- We also assume that A has no rows or columns that are (almost) dense. If so, such rows and/or columns should be treated independently.

Reorderings

Minimizing the fill-in: reorderings

- Historically, ordering the matrix A before using a direct solver to factorize it was generally cheap compared to the numerical factorization cost.
- It is not the case nowadays due to the development of the computational tools.
- In the symmetric case, the diagonal entries of A are required to be present in $\mathcal{S}\{A\}$ (thus zeros on the diagonal are included in the sparsity structure). The aim is to limit fill-in in the L factor of an LL^T (or LDL^T) factorization of A.
- Two greedy heuristics are the minimum degree (MD) criterion and the local minimum fill (MF) criterion.

Reorderings

Minimum fill-in (MF) criterion

- One way to reduce fill-in is to use a local minimum fill-in (MF) criterion:
  - Select as the next variable in the ordering one that will introduce the least fill-in in the factor at that step.
- This is sometimes called the minimum deficiency approach.
- MF can produce good orderings, its cost is often considered to be prohibitive.
- An approximate variant (AMF).

Reorderings

Basic minimum degree (MD) algorithm

- The best-known and most widely-used greedy heuristic for limiting fill-in.
- It seeks to find a permutation such that at each step of the factorization the number of entries in the corresponding column of L is approximately minimized.
- Less expensive to compute compared to that used by the minimum fill-in criterion.
- The MD algorithm can be implemented using $\mathcal{G}(A)$ and it can predict the required factor storage without generating the structure of L.
- At step k, the number of off-diagonal nonzeros in a row or column of the active submatrix is the current degree of the corresponding vertex in the elimination graph $\mathcal{G}^k$. The algorithm selects a vertex of minimum current degree in $\mathcal{G}^k$ and labels it v_k, i.e., next for elimination.

Reorderings

Minimum degree algorithm

Algorithm (Basic minimum degree (MD) algorithm)

Input: *Graph $\mathcal{G}$ of a symmetrically structured matrix.*
Output: *A permutation vector p that defines a new labelling of the vertices of $\mathcal{G}$.*

```
1: Set $\mathcal{G}^1 = \mathcal{G}$ and compute the degree $deg_{\mathcal{G}^1}(u)$ of all $u \in \mathcal{V}(\mathcal{G}^1)$
2: for $k = 1 : n-1$ do
3:     Compute $mdeg = \min\{deg_{\mathcal{G}^k}(u) \,|\, u \in \mathcal{V}(\mathcal{G}^k)\}$   ▷ $mdeg$ is the current minimum degree
4:     Choose $v_k \in \mathcal{V}(\mathcal{G}^k)$ such that $deg_{\mathcal{G}^k}(v_k) = mdeg$
5:     $p(k) = v_k$   ▷ $v_k$ is the next vertex in the elimination order
6:     Construct $\mathcal{G}^{k+1}$ and update the current degrees of its vertices
7: end for
8: $p(n) = v_n$ where $v_n$ is the only vertex in $\mathcal{G}^n$
```

Reorderings

Practical problems: storing and using the fill-in

- A clique with m vertices has $m(m-1)/2$ edges. This may be too many! It can be must be represented by storing a list of its vertices, without any reference to edges.
- This leads to implicit storing of the elimination graphs with significant consequences.
- As the elimination process progresses, cliques grow or more than one clique joins to form larger cliques: clique amalgamation.
- Note that in the nonsymmetric case we need to store edges orientation in addition to be happy from cliques.
- Edges of the elimination graphs can be then expressed through the reachable sets.

Reorderings

From Parter's rule to reachable sets

Lemma

(Edges of $\mathcal{G}^k$ in terms of reachable sets) Assume $\mathcal{S}\{A\}$ is symmetric. Let $\mathcal{V}^k$ be the set of $k-1$ vertices of $\mathcal{G}(A)$ that have already been eliminated and let v be a vertex in the elimination graph $\mathcal{G}^k$. Then the set of vertices adjacent to v in $\mathcal{G}^k$ is the set $\mathcal{R}each(v, \mathcal{V}^k)$ of vertices reachable from v through $\mathcal{V}^k$ in $\mathcal{G}(A)$.

- Can be modified for A nonsymmetric.
- This just points out that the actual graph elimination model may be more complicated ☺

Reorderings

From Parter's rule to reachable sets

- Figure: graph $\mathcal{G}(A)$. The adjacency sets of the vertices in $\mathcal{G}^4$ that result from eliminating vertices $\mathcal{V}^4 = \{1,2,3\}$ are
  $adj_{\mathcal{G}^4}\{4\} = \mathcal{R}each(4, \mathcal{V}^4) = \{5\}$,
  $adj_{\mathcal{G}^4}\{5\} = \mathcal{R}each(5, \mathcal{V}^4) = \{4,6,7\}$,
  $adj_{\mathcal{G}^4}\{6\} = \mathcal{R}each(6, \mathcal{V}^4) = \{5,7\}$,
  $adj_{\mathcal{G}^4}\{7\} = \mathcal{R}each(7, \mathcal{V}^4) = \{5,6,8\}$,
  $adj_{\mathcal{G}^4}\{8\} = \mathcal{R}each(8, \mathcal{V}^4) = \{7\}$.

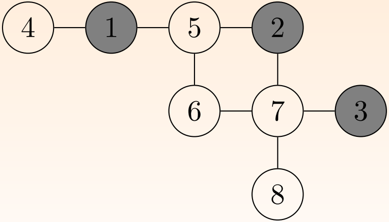


Figure: *The grey vertices 1, 2, and 3 are eliminated in the first three elimination steps* $(\mathcal{V}^4 = \{1,2,3\})$.

Reorderings

From Parter's rule to reachable sets

Lemma

(Edges of $\mathcal{G}^k$ in terms of reachable sets) Assume $\mathcal{S}\{A\}$ is symmetric. Let $\mathcal{V}^k$ be the set of $k-1$ vertices of $\mathcal{G}(A)$ that have already been eliminated and let v be a vertex in the elimination graph $\mathcal{G}^k$. Then the set of vertices adjacent to v in $\mathcal{G}^k$ is the set $\mathcal{R}each(v, \mathcal{V}^k)$ of vertices reachable from v through $\mathcal{V}^k$ in $\mathcal{G}(A)$.

Proof.

The proof is by induction on k. The result holds trivially for $k = 1$ because the $\mathcal{R}each(v, \mathcal{V}^1) = adj_{\mathcal{G}(A)}\{v\}$. Assume the result holds for $\mathcal{G}^1, \ldots, \mathcal{G}^k$ with $k \geq 1$ and let v be a vertex in the graph $\mathcal{G}^{k+1}$ that is obtained after eliminating v_k from $\mathcal{G}^k$. If v is not adjacent to v_k in $\mathcal{G}^k$ then $\mathcal{R}each(v, \mathcal{V}^{k+1}) = \mathcal{R}each(v, \mathcal{V}^k)$. Otherwise, if v is adjacent to v_k in $\mathcal{G}^k$ then $adj_{\mathcal{G}^{k+1}}\{v\} = \mathcal{R}each(v, \mathcal{V}^k) \cup \mathcal{R}each(v_k, \mathcal{V}^k)$. In both cases Parter's rule implies that the new adjacency set is exactly equal to the vertices that are reachable from v through $\mathcal{V}^{k+1}$, that is, $\mathcal{R}each(v, \mathcal{V}^{k+1})$. □ □

Reorderings

Minimum degree algorithm

- The set of vertices adjacent to v_k in $\mathcal{G}(A)$ is $\mathcal{R}each(v_k, \mathcal{V}_k)$, where $\mathcal{V}_k$ is the set of $k-1$ vertices that have already been eliminated.
- If $u \in \mathcal{R}each(v_k, \mathcal{V}_k)$, $u \neq v_k$, then its updated current degree is $|\mathcal{R}each(u, \mathcal{V}_{k+1})|$, where $\mathcal{V}_{k+1} = \mathcal{V}_k \cup v_k$.
- A tie-breaking strategy is needed when there is more than one vertex of current minimum degree.
- It is possible to construct artificial matrices showing that some systematic tie-breaking choices can lead to a large amount of fill-in but such behaviour is not typical.

Reorderings

Minimum degree algorithm

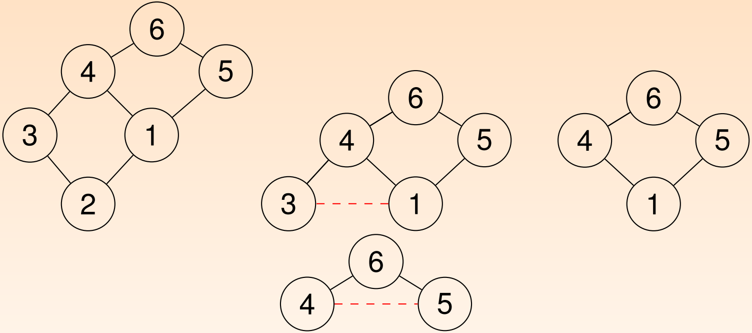


Figure: *An illustration of three steps of the MD algorithm. The original graph $\mathcal{G}$ and the elimination graphs $\mathcal{G}^2$, $\mathcal{G}^3$ and $\mathcal{G}^4$ that result from eliminating vertex 2, then vertex 3 and then vertex 1. The red dashed lines denote fill edges.*

Reorderings

Minimum degree algorithm

- The construction of each elimination graph $\mathcal{G}^{k+1}$ is central to the implementation.
- Because eliminating a vertex potentially creates fill-in, an efficient representation is needed.
- Moreover, recalculating the current degrees is time consuming.
- Using supervariables is a must.
- $\mathcal{G}_v$ denotes the elimination graph obtained from $\mathcal{G}$ when vertex $v \in \mathcal{V}(\mathcal{G})$ is eliminated.

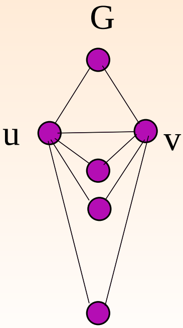
Reorderings

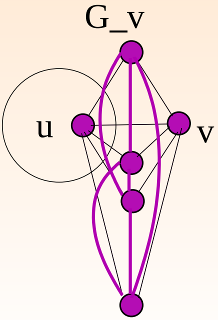
Indistinguishability (reminder)

Definition

Two different vertices **u** and **v** of G are called **indistinguishable** if

$$Adj_G(u) \cup \{u\} = Adj_G(v) \cup \{v\}. \qquad (51)$$





Reorderings

Indistinguishability

Theorem

Let u and w be indistinguishable vertices in $\mathcal{G}$. If $v \in \mathcal{V}(\mathcal{G})$ with $v \neq u, w$, then u and w are indistinguishable in $\mathcal{G}_v$.

Proof.

Two cases must be considered. First, let $u \notin adj_{\mathcal{G}}\{v\}$. Then $w \notin adj_{\mathcal{G}}\{v\}$ and if v is eliminated, the adjacency sets of u and w are unchanged and they remain indistinguishable in the resulting elimination graph $\mathcal{G}_v$. Second, let $u, w \in adj_{\mathcal{G}}\{v\}$. When v is eliminated, because u and w are indistinguishable in $\mathcal{G}$, their adjacency sets in $\mathcal{G}_v$ will be modified in the same way, by adding the entries of $adj_{\mathcal{G}}\{v\}$ that are not already in $adj_{\mathcal{G}}\{u\}$ and $adj_{\mathcal{G}}\{w\}$. Consequently, u and w are indistinguishable in $\mathcal{G}_v$. □ □

Indistinguishability

Figure demonstrates the two cases in the proof of Theorem above. Here, u and w are indistinguishable vertices in $\mathcal{G}$. Setting $v \equiv v'$ illustrates $u \notin adj_{\mathcal{G}}\{v\}$. If v' is eliminated then the adjacency sets of u and w are clearly unchanged. Setting $v \equiv v''$ illustrates $u, w \in adj_{\mathcal{G}}\{v\}$. In this case, if v'' is eliminated then vertices s and t are added to both $adj_{\mathcal{G}}\{u\}$ and $adj_{\mathcal{G}}\{w\}$.

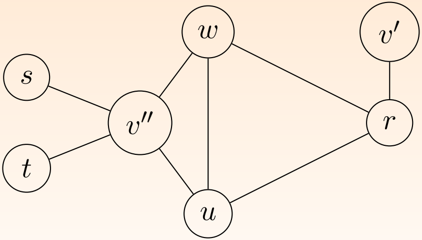


Figure: *An example to illustrate the Theorem. u and w are indistinguishable vertices in $\mathcal{G}$; $adj_{\mathcal{G}}\{u\} = \{r, w, v''\}$ and $adj_{\mathcal{G}}\{w\} = \{r, u, v''\}$.*

Reorderings

Indistinguishability

Theorem

Let u and w be indistinguishable vertices in $\mathcal{G}$. If w is of minimum degree in $\mathcal{G}$ then u is of minimum degree in $\mathcal{G}_w$.

Proof.

Let $deg_{\mathcal{G}}(w) = mdeg$. Then $deg_{\mathcal{G}}(u) = mdeg$. Indistinguishable vertices are always neighbours. Eliminating w gives $deg_{\mathcal{G}_w}(u) = mdeg - 1$ because w is removed from the adjacency set of u and there is no neighbour of u in $\mathcal{G}_w$ that was not its neighbour in $\mathcal{G}$. If $x \neq w$ and $x \in adj_{\mathcal{G}}\{u\}$ then the number of neighbours of x in $\mathcal{G}_w$ is at least $mdeg - 1$. Otherwise, if $x \notin adj_{\mathcal{G}}\{u\}$ then its adjacency set in $\mathcal{G}_w$ is the same as in $\mathcal{G}$ and is of the size at least $mdeg$. The result follows. □

□

Indistinguishability

Theorem above is illustrated in Figure.

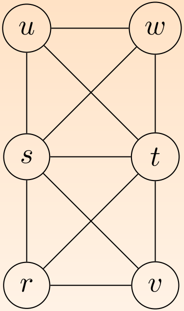


Figure: *An illustration of Theorem. Vertices u and w are of minimum degree (with degree $mdeg = 3$) and are indistinguishable in $\mathcal{G}$. After elimination of w, the current degree of u is $mdeg - 1$ and the current degree of each of the other vertices is at most $mdeg - 1$. Therefore, u is of current minimum degree in $\mathcal{G}_w$. Note that vertices r and v are also of minimum degree and indistinguishable in $\mathcal{G}$; they are not neighbours of w and their degrees do not change when w is eliminated.*

Reorderings

Indistinguishability

- The results can be extended to more than two indistinguishable vertices, which allows indistinguishable vertices to be selected one after another in the MD ordering.
- This is referred to as mass elimination.
- Treating indistinguishable vertices as a single supervariable cuts the number of vertices and edges in the elimination graphs, which reduces the work needed for degree updates.
- The external degree of a vertex is the number of vertices adjacent to it that are not indistinguishable from it. Using this leads to algorithmic efficiency.

Reorderings

Degree outmatching

- A concept that is closely related to that of indistinguishable vertices is degree outmatching.
- This avoids computing the degrees of vertices that are known not to be of current minimum degree.
- Vertex w is said to be outmatched by vertex u if

$$adj_{\mathcal{G}}\{u\} \cup \{u\} \subseteq adj_{\mathcal{G}}\{w\} \cup \{w\}.$$

- It follows that $deg_{\mathcal{G}}(u) \leq deg_{\mathcal{G}}(w)$.

Reorderings

Degree outmatching

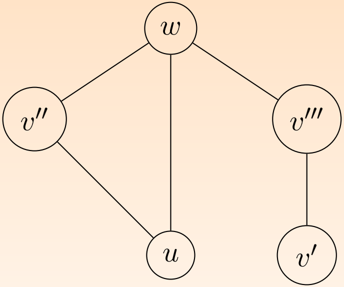


Figure: *An example $\mathcal{G}$ in which vertex w is outmatched by vertex u. v' is not a neighbour of u or w; vertex v'' is a neighbour of both u and w; v''' is a neighbour of w but not of u.*

Degree outmatching

- Importantly, degree outmatching is preserved when vertex $v \in \mathcal{G}$ of minimum degree is eliminated, as stated in the following result.

Theorem

In the graph $\mathcal{G}$ let vertex w be outmatched by vertex u and vertex v ($v \neq u, w$) be of minimum degree. Then w is outmatched in $\mathcal{G}_v$ by u.

Proof.

Three cases must be considered. First, if $u \notin adj_{\mathcal{G}}\{v\}$ and $w \notin adj_{\mathcal{G}}\{v\}$ then the adjacency sets of u and w in $\mathcal{G}_v$ are the same as in $\mathcal{G}$. Second, if v is a neighbour of both u and w in $\mathcal{G}$ then any neighbours of v that were not neighbours of u and w are added to their adjacency sets in $\mathcal{G}_v$. Third, if $u \notin adj_{\mathcal{G}}\{v\}$ and $w \in adj_{\mathcal{G}}\{v\}$ then the adjacency set of u in $\mathcal{G}_v$ is the same as in $\mathcal{G}$ but any neighbours of v that were not neighbours of w are added to the adjacency set of w in $\mathcal{G}_v$. In all three cases, w is still outmatched by u in $\mathcal{G}_v$. □ □

Reorderings

Degree outmatching

- The three possible cases for v in the proof of Theorem are illustrated in Figure above by setting $v \equiv v'$, v'' and v''', respectively.
- If w is outmatched by u then it is not necessary to consider w as a candidate for elimination and
- all updates to the data structures related to w can be postponed until u has been eliminated.

Reorderings

Cliques and quotient graphs

- From Parter's rule, if vertex v is selected at step k then the elimination matrix that corresponds to $\mathcal{G}^{k+1}$ contains a dense submatrix of size equal to the number of off-diagonal entries in row and column v in the matrix that corresponds to $\mathcal{G}^{k}$.
- For large matrices, creating and explicitly storing the edges in the sequence of elimination graphs is impractical and a more compact and efficient representation is needed.
- Each elimination graph can be interpreted as a collection of cliques, including the original graph $\mathcal{G}$, which can be regarded as having $|\mathcal{E}|$ cliques, each consisting of two vertices (or, equivalently, an edge).

Reorderings

Cliques and quotient graphs

- Let $\{\mathcal{V}_1, \mathcal{V}_2, \ldots, \mathcal{V}_q\}$ be the set of cliques for the current graph and let v be a vertex of current minimum degree that is selected for elimination. Let $\{\mathcal{V}_{s_1}, \mathcal{V}_{s_2}, \ldots, \mathcal{V}_{s_t}\}$ be the subset of cliques to which v belongs. Two steps are then required.
  1. Remove the cliques $\{\mathcal{V}_{s_1}, \mathcal{V}_{s_2}, \ldots, \mathcal{V}_{s_t}\}$ from $\{\mathcal{V}_1, \mathcal{V}_2, \ldots, \mathcal{V}_q\}$.
  2. Add the new clique $\mathcal{V}_v = \{\mathcal{V}_{s_1} \cup \ldots \cup \mathcal{V}_{s_t}\} \setminus \{v\}$ into the set of cliques.
- Hence

$$deg_{\mathcal{G}}(v) = |\mathcal{V}_v| < \sum_{i=1}^{t} |\mathcal{V}_{s_i}|,$$

  and because $\{\mathcal{V}_{s_1}, \mathcal{V}_{s_2}, \ldots, \mathcal{V}_{s_t}\}$ can now be discarded, the storage required for the representation of the sequence of elimination graphs never exceeds that needed for $\mathcal{G}(A)$.
- The storage to compute an MD ordering is therefore known beforehand in spite of the rather dynamic nature of the elimination process.

Reorderings

Cliques and quotient graphs

- The index of the eliminated vertex can be used as the index of the new clique. This is called an element or enode (the terminology comes from finite-element methods), to distinguish it from an uneliminated vertex, which is termed an snode.
- A sequence of special quotient graphs $\mathcal{G}^{[1]} = \mathcal{G}(A), \mathcal{G}^{[2]}, \ldots, \mathcal{G}^{[n]}$ with the two types of vertices is generated in place of the elimination graphs.
- Each $\mathcal{G}^{[k]}$ has n vertices that satisfy

  $$\mathcal{V}(\mathcal{G}) = \mathcal{V}_{snodes} \cup \mathcal{V}_{enodes}, \qquad \mathcal{V}_{snodes} \cap \mathcal{V}_{enodes} = \emptyset,$$

  where $\mathcal{V}_{snodes}$ and $\mathcal{V}_{enodes}$ are the sets of snodes and enodes, respectively.
- When v is eliminated, any enodes adjacent to it are no longer required to represent the sparsity pattern of the corresponding active submatrix and so they can be removed. This is called element absorption.

Reorderings

Cliques and quotient graphs

- Working with the special quotient graphs can be demonstrated by considering the computation of the vertex degrees.
- To compute the degree of an uneliminated vertex, the set of neighbouring snodes is counted.
- Then, if a neighbour of one of these snodes is an enode, its neighbours are also counted (avoiding double counting).
- More formally, if $v \in \mathcal{V}_{snodes}$ then the adjacency set of v is the union of its neighbours in $\mathcal{V}_{snodes}$ and the vertices reachable from v via its neighbours in $\mathcal{V}_{enodes}$.
- In this way, vertex degrees are computed by considering fill-paths
- Amalgamation improves this strategy: mass elimination.

Reorderings

Cliques and quotient graphs: mass elimination model

Definition

Mass elimination graph Γ of the graph $G = (V, E)$ is a ordered triple $(\mathcal{S}, \mathcal{E}, E)$, where $\mathcal{S} \cup \mathcal{E} = V, \mathcal{S} \cap \mathcal{E} = \emptyset$ and $E \subseteq \binom{\mathcal{S}}{2} \cup \binom{\mathcal{E}(\Gamma)}{2}$ are its edges.

- Edge set $\mathcal{E}$ captures eliminated vertices.
- Edge set $\mathcal{S}$ captures non-eliminated vertices.
- Neighbors of non-eliminated vertices are found as reachability sets.
- Search through the reachability sets can be pruned: $\rightarrow$ approximate minimum degree (AMD) algorithm.

Reorderings

Cliques and quotient graphs

- After elimination of 1, a new edge is added, getting a clique. The elimination of 2 creates no additional fill and $\mathcal{G}^3$ represents the sparsity structure of the corresponding active submatrix $A^{(3)}$.
- Then, 1 is an enode, the fill edge is represented implicitly. After the second step, the enodes 1 and 2 can be amalgamated as well as snodes 3 and 4 being indistinguishable.

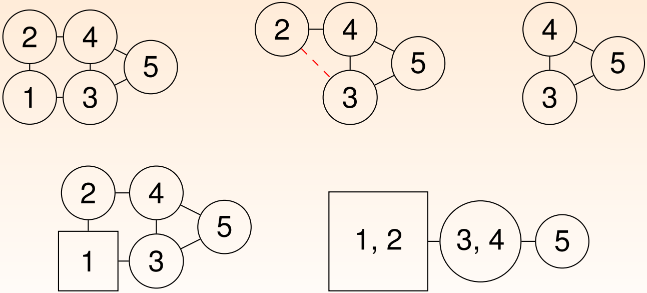


Figure: *The top line shows $\mathcal{G} = \mathcal{G}^1$, $\mathcal{G}^2$ and $\mathcal{G}^3$. The bottom line shows the quotient graphs $\mathcal{G}^{[2]}$ and $\mathcal{G}^{[3]}$ after the first and second elimination steps.*

Reorderings

Multiple minimum degree (MMD) algorithm

- The multiple minimum degree (MMD) algorithm aims to improve efficiency by processing several independent vertices that are each of minimum current degree together in the same step, before the degree updates are performed.
- At each outer loop, $t \geq 1$ denotes the number of vertices of minimum current degree that are mutually non-adjacent and so can be put into the elimination order one after another.
- An example follows.

Reorderings

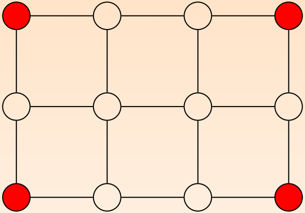


Figure: *The red (corner) vertices of $\mathcal{G}$ are each of degree 2 and are ordered consecutively during the first step of Algorithm 13.2.*

Multiple minimum degree (MMD) algorithm

Algorithm (Basic multiple minimum degree (MMD) algorithm)

Input: *Graph $\mathcal{G}$ of a symmetrically structured matrix.*
Output: *A permutation vector p that defines a new labelling of the vertices of $\mathcal{G}$.*

```
Set $k = 1$, $\mathcal{G}^1 = \mathcal{G}$ and compute the degree $deg_{\mathcal{G}^1}(u)$ of all $u \in \mathcal{V}(\mathcal{G}^1)$
while $k \leq n$ do
    Compute $mdeg = \min\{deg_{\mathcal{G}^k}(u) \mid u \in \mathcal{V}(\mathcal{G}^k)\}$
    Find all mutually non-adjacent $v_j \in \mathcal{V}(\mathcal{G}^k)$, $j = 1, \dots, t$ with $deg_{\mathcal{G}^k}(v_j) = mdeg$
    for $j = 1 : t$ do
        $p(k) = v_j$                    ▷ Vertex $v_j$ is the next vertex in the elimination order
        $k = k + 1$
    end for
    if $k < n$ then
        Construct $\mathcal{G}^{k+1}$ and update the current degrees of its vertices
    end if
end while
```

Reorderings

MD and MMD: complexity

- The complexity of the MD and MMD algorithms is $O(nz(A)n^2)$ but because for MMD the outer loop of the algorithm update is performed fewer times, it can be significantly faster then MD.
- MMD orderings can also lead to less fill-in, possibly a consequence of introducing some kind of regularity into the ordering sequence.

Reorderings

Approximate minimum degree (AMD) algorithm

- The idea behind the widely-used approximate minimum degree (AMD) algorithm is to inexpensively compute an upper bound on a vertex degree in place of the degree, and to use this bound as an approximation to the external degree.
- The quality of the orderings obtained using the AMD algorithm are competitive with those computed using the MD algorithm and can surpass them.
- The complexity of AMD is $O(nz(A)n)$ and, in practice, its runtime is typically significantly less than that of the MD and MMD approaches.

Reorderings

Minimizing the bandwidth and profile

- An alternative way of reducing the fill-in locally is to add another criterion to the relabelling of the vertices, such as restricting the nonzeros of the permuted matrix to specific positions.
- The most popular approach is to force them to lie close to the main diagonal.
- All fill-in then takes place between the first entry of a row and the diagonal or between the first entry of a column and the diagonal.
- This allows straightforward implementations of Gaussian elimination that employ static data structures.
- Here we again consider symmetric $\mathcal{S}\{A\}$; generalizations are possible.

The band and envelope of a matrix

Denote:

$$\eta_i(A) = \min\{j \mid 1 \le i \le j, \text{ with } a_{ij} \ne 0\}, \quad 1 \le i \le n, \tag{52}$$

that is, $\eta_i(A)$ is the column index of the first entry in the i-th row of A.

Define

$$\beta_i(A) = i - \eta_i(A), \quad 1 \le i \le n.$$

The semibandwidth of A is

$$\max\{\beta_i(A) | \, 1 \le i \le n\},$$

and the bandwidth is

$$\beta(A) = 2 * \max\{\beta_i(A) \mid 1 \le i \le n\} + 1.$$

The band of A is the following set of index pairs in A

$$band(A) = \{(i,j) \mid 0 < i - j \le \beta(A)\}.$$

Reorderings

The band and envelope of a matrix

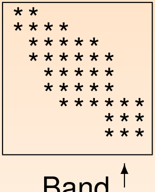
- The envelope is the set of index pairs that lie between the first entry in each row and the diagonal:

$$env(A) = \{(i,j) \mid \; 0 < i - j \leq \beta_i(A)\}.$$

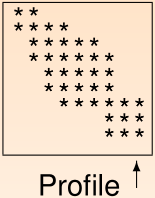
- Note that the band and envelope of a sparse symmetrically structured matrix A include only entries of the strict lower triangular part of A.
- The envelope is easily visualized: picture the lower triangular part of A, and remove the diagonal and the leading zero entries in each row. The remaining entries (whether nonzero or zero) comprise the envelope of A.
- The profile of A is defined to be the number of entries in the envelope (the envelope size) plus n.

Reorderings

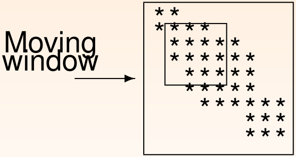
The band and envelope of a matrix: shape pushers



Band



Profile



Frontal method - dynamic band

Reorderings

The band and envelope of a matrix

An illustrative example: Here $\eta_1(A) = 1$, $\beta_1(A) = 0$, $\eta_2(A) = 1$, $\beta_2(A) = 1$, $\eta_3(A) = 2$, $\beta_3(A) = 1$, and so on.

$$
\begin{pmatrix}
* & * & & & & & \\
* & * & * & * & & & \\
& * & * & * & & & \\
& * & * & * & & * & \\
& & & & * & * & \\
& & & * & * & * & * \\
& & & & & * & *
\end{pmatrix}
\begin{pmatrix}
* & * & * & & & & \\
\circledast & * & * & * & & & \\
\circledast & \circledast & * & * & * & & \\
& \circledast & \circledast & * & * & * & \\
& & \circledast & \circledast & * & * & * \\
& & & \circledast & \circledast & * & * \\
& & & & \circledast & \circledast & *
\end{pmatrix}
\begin{pmatrix}
* & * & & & & & \\
\circledast & * & * & * & & & \\
& \circledast & * & * & & & \\
& \circledast & \circledast & * & & * & \\
& & & & * & * & \\
& & & \circledast & \circledast & * & * \\
& & & & & \circledast & *
\end{pmatrix}
$$

Figure: *Illustration of the band and envelope of a matrix A whose sparsity pattern is on the left. In the centre, the positions of $band(A)$ are circled and on the right, the positions of $env(A)$ are circled. The bandwidth is 5 and the envelope size and the profile are 7 and 14, respectively.*

The band and envelope of a matrix

- Static structures!

Theorem

If L is the Cholesky factor of A then

$$env(A) = env(L).$$

Proof.

The proof uses mathematical induction on the principal leading submatrices of A of order k. The result is clearly true for $k = 1$ and $k = 2$. Assume it holds for $2 \leq k < n$ and consider the block factorization

$$\begin{pmatrix} A_{1:k,1:k} & u_{1:k} \\ u_{1:k}^T & \alpha \end{pmatrix} = \begin{pmatrix} L_{1:k,1:k} & 0 \\ v_{1:k}^T & \beta \end{pmatrix} \begin{pmatrix} L_{1:k,1:k}^T & v_{1:k} \\ 0 & \beta \end{pmatrix},$$

where α and β are scalars. Equating the left and right sides, $L_{1:k,1:k} v_{1:k} = u_{1:k}$. Because $u_j = 0$ for $j < \eta_{k+1}(A)$ and $u_{\eta_{k+1}} \neq 0$, it follows that $v_j = 0$ for $j < \eta_{k+1}(A)$ and $v_{\eta_{k+1}} \neq 0$. This proves the induction step. □

□

A straightforward corollary is that $band(A) = band(L)$.

Reorderings

The band and envelope of a matrix

- Finding a permutation P to minimize the band or profile of PAP^T is again combinatorially hard and again heuristics are used to efficiently find an acceptable P.
- The popular Cuthill McKee (CM) approach chooses a suitable starting vertex s and labels it 1.
- Then, for $i = 1, 2, \dots, n-1$, all vertices adjacent to vertex i that are still unlabelled are labelled successively in order of increasing degree, as described in Algorithm below.
- A very important variation is the Reverse Cuthill McKee (RCM) algorithm, which incorporates a final step in which the CM ordering is reversed.

Reorderings

Level-based orderings

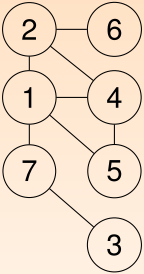
Algorithm (CM and RCM algorithms for band and profile reduction)

Input: *Graph $\mathcal{G}$ of a symmetrically structured matrix and a starting vertex s.*
Output: *Permutation vectors p_{cm} and p_{rcm} that define new labellings of the vertices of $\mathcal{G}(A)$.*

```
$label(1:n) = false$
Compute $adj_{\mathcal{G}}\{u\}$ and $deg_{\mathcal{G}}(u)$ for all $u \in \mathcal{V}(\mathcal{G})$
$k = 1,\ v_1 = s,\ p_{cm}(1) = v_1,\ label(v_1) = true$
for $i = 1 : n-1$ do
    for $w \in adj_{\mathcal{G}}\{v_i\}$ with $label(w) = false$ in order of increasing degree do
        $k = k+1,\ v_k = w,\ p_{cm}(k) = v_k,\ label(v_k) = true$
    end for
end for
For the RCM ordering, $p_{rcm}(i) = p_{cm}(n-i+1)$, $i = 1, 2, \ldots, n$.
```

Reorderings

- The CM- and RCM-permuted matrices have the same bandwidth but the latter can decrease the envelope.



$$
\begin{array}{c|ccccccc}
 & 1 & 2 & 3 & 4 & 5 & 6 & 7 \\
\hline
1 & * & * & & * & * & & * \\
2 & * & * & & * & & * & \\
3 & & & * & & & & * \\
4 & * & * & & * & * & & \\
5 & * & & & * & * & & \\
6 & & * & & & & * & \\
7 & * & & * & & & & *
\end{array},
\quad
\begin{array}{c|ccccccc}
 & 3 & 7 & 1 & 5 & 2 & 4 & 6 \\
\hline
3 & * & * & & & & & \\
7 & * & * & * & & & & \\
1 & & * & * & * & * & * & \\
5 & & & * & * & & * & \\
2 & & & * & & * & * & * \\
4 & & & * & * & * & * & \\
6 & & & & & * & & *
\end{array},
\quad
\begin{array}{c|ccccccc}
 & 6 & 4 & 2 & 5 & 1 & 7 & 3 \\
\hline
6 & * & & * & & & & \\
4 & & * & * & * & * & & \\
2 & * & * & * & & * & & \\
5 & & * & & * & * & & \\
1 & & * & * & * & * & * & \\
7 & & & & & * & * & * \\
3 & & & & & & * & *
\end{array}
,
$$

Reorderings

Level-based orderings

The importance of the CM and RCM orderings is also expressed by:

Theorem

Let A be symmetrically structured and irreducible. If P corresponds to the CM labelling obtained from Algorithm and L is the Cholesky factor of $P^T AP$ then $env(L)$ is full, that is, all entries of the envelope are nonzero.

- The full envelope of the Cholesky factor of the permuted matrix implies cache efficiency when performing the triangular solves once the factorization is complete.
- A crucial difference between profile reduction ordering algorithms and minimum degree strategies is that the former is based solely on $\mathcal{G}$: the costly construction of quotient graphs is not needed.
- However, unless the profile after reordering is very small, there can be significantly more fill-in in the factor.

Reorderings

Level-based orderings

- Key to the success is the choice of the starting vertex s.
- A good candidate is a vertex for which the maximum distance between it and some other vertex in $\mathcal{G}$ is large.
- Formally, the eccentricity $\epsilon(u)$ of the vertex u in the connected undirected graph $\mathcal{G}$ is defined to be

  $$\epsilon(u) = \max_{v \in \mathcal{V}} d(u, v),$$

  where $d(u, v)$ is the distance between the vertices u and v (the length of the shortest path between these vertices).
- The maximum eccentricity taken over all the vertices is the diameter of $\mathcal{G}$ (that is, the maximum distance between any pair of vertices). The endpoints of a diameter (also termed peripheral vertices) provide good starting vertices.
- The complexity of finding a diameter is $O(n^3)$: approximation (pseudo-preferal vertices) are needed.

Graphs and their matrices

Breadth-first search (BFS)

- Starting from a chosen start vertex s, a breadth-first search (BFS) explores all the vertices adjacent to s.
- Then all the vertices whose shortest path from s is of length 2, and then length 3, and so on
- A queue is used in its implementation.
- The search terminates when there are no unexplored edges (v, w) with $v \in \mathcal{V}_v$ and $w \in \mathcal{V}_n$ that are reachable from s.

Graphs and their matrices

Breadth-first search (BFS)

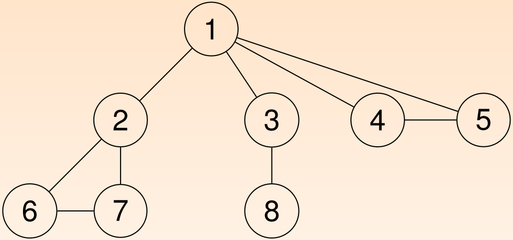


Figure: *An illustration of a BFS of a connected undirected graph, with the labels indicating the order in which the vertices are visited.*

Reorderings

Level-based orderings

- A heuristic algorithm is used to find pseudo-peripheral vertices. A commonly-used approach is based on level sets. A level structure rooted at a vertex r is defined as the partitioning of $\mathcal{V}$ into disjoint levels $\mathcal{L}_1(r), \mathcal{L}_2(r), \ldots, \mathcal{L}_h(r)$ such that

  (i) $\mathcal{L}_1(r) = \{r\}$ and

  (ii) for $1 < i \leq h$, $\mathcal{L}_i(r)$ is the set of all vertices that are adjacent to vertices in $\mathcal{L}_{i-1}(r)$ but are not in $\mathcal{L}_1(r), \mathcal{L}_2(r), \ldots, \mathcal{L}_{i-1}(r)$.
- The level structure rooted at r may be expressed as the set $\mathcal{L}(r) = \{\mathcal{L}_1(r), \mathcal{L}_2(r), \ldots, \mathcal{L}_h(r)\}$, where h is the total number of levels and is termed the depth.
- The level sets can be found using a breadth-first search that starts at the root r.

Reorderings

Level-based orderings: GPS

Algorithm (GPS algorithm to find pseudo-peripheral vertices)

```
Construct $\mathcal{L}(r)$ and set $flag = false$
while $flag = false$ do
    $flag = true$
    for $i = 1 : |\mathcal{L}(r)|$ do
        $w_i \in \mathcal{L}(r)$                          ▷ Select vertex $w_i$ from last level set
        if $flag = true$ then
            Construct $\mathcal{L}(w_i)$
            if $depth(\mathcal{L}(w_i)) > depth(\mathcal{L}(r))$ then
                $flag = false$                ▷ Flag that $w_i$ will be used as new initial vertex
            end if
        end if
    end for
    if $flag = true$ then
        $s = r$ and $t = w_i$        ▷ $s$ has been chosen; while loop will terminate algorithm
    else
        $r = w_i$
    end if
end while
```

Reorderings

Level-based orderings

A simple example: starting with $r = 2$, after two passes through the while loop, the GPS algorithm returns $s = 8$ and $t = 1$ as pseudo-peripheral vertices.

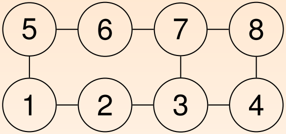


Figure: *An example to illustrate Algorithm 13.4 for finding pseudo-peripheral vertices. With root vertex* $r = 2$*, the first level set structure is* $\mathcal{L}(2) = \{\{2\}, \{1, 3\}, \{4, 5, 7\}, \{6, 8\}\}$*. Setting* $r = 8$ *at Step 16, the second level set structure is* $\mathcal{L}(8) = \{\{8\}, \{4, 7\}, \{3, 6\}, \{2, 5\}, \{1\}\}$ *and the algorithm terminates with* $s = 8$ *and* $t = 1$*.*

Reorderings

Spectral orderings

- The spectral algorithm associates a positive semidefinite Laplacian matrix L_p with the symmetric matrix A as follows:

$$(L_p)_{ij} = \begin{cases} -1 & \text{if } i \neq j \text{ and } a_{ij} \neq 0, \\ deg_{\mathcal{G}}(i) & \text{if } i = j, \\ 0 & \text{otherwise.} \end{cases}$$

- An eigenvector corresponding to the smallest positive eigenvalue of the Laplacian matrix is called a Fiedler vector. If $\mathcal{G}$ is connected, L_p is irreducible and the second smallest eigenvalue is positive.
- The vertices of $\mathcal{G}$ are ordered by sorting the entries of the Fiedler vector into monotonic order. Applying the permutation symmetrically to A yields the spectral ordering.

Reorderings

Spectral orderings

- The use of the Fiedler vector for reordering A comes from considering the matrix envelope with the size
$$|env(A)| = \sum_{i=1}^{n} \beta_i = \sum_{i=1}^{n} \max_{\substack{k<i \\ (k,i)\in\mathcal{G}}} (i-k).$$
The asymptotic upper bound on the operation count for the factorization based on $env(A)$ is
$$work_{env} = \sum_{i=1}^{n} \beta_i^2 = \sum_{i=1}^{n} \max_{\substack{k<i \\ (k,i)\in\mathcal{G}}} (i-k)^2.$$
Ordering the vertices using the Fiedler vector is closely related to minimizing $weight_{env}$ over all possible vertex reorderings, where
$$weight_{env} = \sum_{i=1}^{n} \sum_{\substack{k<i \\ (k,i)\in\mathcal{G}}} (i-k)^2.$$

Reorderings

Spectral orderings

- Thus, while minimizing the profile and envelope is related to the infinity norm, minimizing $weight_{env}$ is related to the Euclidean norm of the distance between graph vertices.
- Although computing the Fiedler vector can be computationally expensive it does have the advantage of easy vectorization and parallelization and the resulting ordering can give small profiles and low operation counts.

Reorderings

Local fill-reducing orderings for nonsymmetric $\mathcal{S}\{A\}$

- If $\mathcal{S}\{A\}$ is nonsymmetric then an often-used strategy is to apply the minimum degree algorithm (or one of its variants) or a band or profile-reducing ordering to the undirected graph $\mathcal{G}(A + A^T)$.
- This can work well if the symmetry index $s(A)$ is close to 1. But if A is highly nonsymmetric, another approach is required.
- Markowitz pivoting generalizes the MD algorithm by choosing the pivot entry based on vertex degrees computed directly from the nonsymmetric $\mathcal{S}\{A\}$; the result is a nonsymmetric permutation.

Reorderings

Markowitz pivoting

- At step k of the LU factorization, consider the $(n-k+1) \times (n-k+1)$ active submatrix, that is, the Schur complement $S^{(k)}$. Let $nz(row_i)$ and $nz(col_j)$ denote the number of entries in row i and column j of $S^{(k)}$ $(1 \le i, j \le n-k+1)$. Markowitz pivoting selects as the k-th pivot the entry of $S^{(k)}$ that minimizes the Markowitz count given by the product

$$(nz(row_i) - 1)(nz(col_j) - 1).$$

- It can be described using a sequence of bipartite graphs of the active submatrices but here we use a matrix-based description that permutes A on the fly.
- Markowitz pivoting is generally incorporated into the numerical factorization phase of an LU solver, rather than being used to derive an initial reordering of A.

Markowitz pivoting

Implementation of the algorithm requires access to the rows and the columns of the matrix.

Algorithm (Markowitz pivoting)

Input: *Matrix A with a nonsymmetric sparsity pattern.*
Output: *$A' = PAQ$, where P and Q are permutation matrices chosen to limit fill in.*

```
Set $S^{(1)} = A$ and $A' = A$
for $k = 1 : n-1$ do
    Compute $nz(row_i)$ and $nz(col_j)$ ($1 \leq i, j \leq n-k+1$)
    Find an entry $s_{ij}^{(k)}$ of $S^{(k)}$ that minimizes $(nz(row_i)-1)(nz(col_j)-1)$
    Permute the rows and columns so that $s_{ij}^{(k)}$ is the $(1,1)$ entry of the permuted $S^{(k)}$
    Compute Schur complement $S^{(k+1)}$ of the permuted $S^{(k)}$ with respect to its $(1,1)$ entry
end for
```

Reorderings

Markowitz pivoting

Example: the first pivot is a_{24} with Markowitz count 1; it does not cause fill-in. The second pivot has Markowitz count 2 in $S^{(2)}$; it results in one filled entry.

$$\begin{array}{c|ccccc} & 1 & 2 & 3 & 4 & 5 \\ \hline 1 & * & * & * & & * \\ 2 & & * & & \circledast & \\ 3 & * & & * & & * \\ 4 & & * & * & * & \\ 5 & * & * & & & * \end{array} \qquad \begin{array}{c|ccccc} & 4 & 1 & 2 & 3 & 5 \\ \hline 2 & * & & * & & \\ 1 & & * & * & * & * \\ 3 & & * & & * & * \\ 4 & * & & \circledast & * & \\ 5 & & * & * & & * \end{array} \qquad \begin{array}{c|ccccc} & 4 & 2 & 1 & 3 & 5 \\ \hline 2 & * & * & & & \\ 4 & * & * & & * & \\ 1 & & * & * & * & * \\ 3 & & & * & * & * \\ 5 & & * & * & f & * \end{array}$$

Figure: *Illustration of Markowitz pivoting. The first and second pivots are circled. The sparsity pattern of $A = S^{(1)}$ is on the left. In the centre is the sparsity pattern after permuting the pivot in position $(2, 4)$ to the $(1, 1)$ position of $S^{(1)}$. There is no fill-in after the first factorization step. On the right is the sparsity pattern after selecting the second pivot that has the original position $(4, 2)$ and permuting it to the $(1, 1)$ position of $S^{(2)}$. The resulting filled entry is denoted by f. Note that the nonsymmetric permutations transform the originally irreducible matrix into a reducible one.*

Reorderings

Local fill-reducing orderings for nonsymmetric $\mathcal{S}\{A\}$

- Markowitz pivoting as described here only considers the sparsity of A and the subsequent Schur complements.
- In practice, small pivots should be avoided.
- Practical implementations: relaxations: restriction to a limited number of rows and columns.
- Dynamic sparse formats like DS needed.

Reorderings

Global nested dissection orderings

- Nested dissection (ND) is the most important and widely-used global ordering strategy for direct methods when $\mathcal{S}\{A\}$ is symmetric; it is particularly effective for ordering very large matrices.
- Identifying a small set of vertices $\mathcal{V}_\mathcal{S}$ (known as a vertex separator)
- If removed separates the graph into two disjoint subgraphs described by the vertex subsets $\mathcal{B}$ and $\mathcal{W}$.
- The rows and columns belonging to $\mathcal{B}$ are labelled first, then those belonging to $\mathcal{W}$ and finally those in $\mathcal{V}_\mathcal{S}$. The reordered matrix has the form

$$\begin{pmatrix} A_{\mathcal{B},\mathcal{B}} & 0 & A_{\mathcal{B},\mathcal{V}_\mathcal{S}} \\ 0 & A_{\mathcal{W},\mathcal{W}} & A_{\mathcal{W},\mathcal{V}_\mathcal{S}} \\ A^T_{\mathcal{B},\mathcal{V}_\mathcal{S}} & A^T_{\mathcal{W},\mathcal{V}_\mathcal{S}} & A_{\mathcal{V}_\mathcal{S},\mathcal{V}_\mathcal{S}} \end{pmatrix}. \tag{53}$$

Reorderings

Global nested dissection orderings

Definition

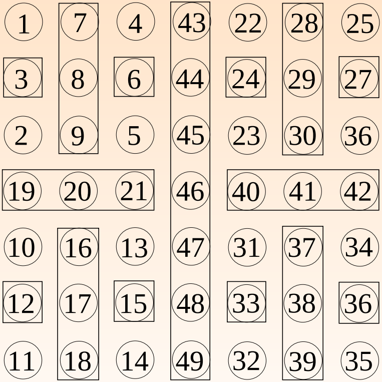
Vertex separator of an undirected $G = (V, E)$ is subset S of its vertices such that the subgraph induced by $V \setminus S$ has more components than G.

- Induced reordering

$$A = \begin{pmatrix} A_{11} & 0 & A_{31}^T \\ 0 & A_{22} & A_{32}^T \\ A_{31} & A_{32} & A_{33} \end{pmatrix} \tag{54}$$

Initial reordering

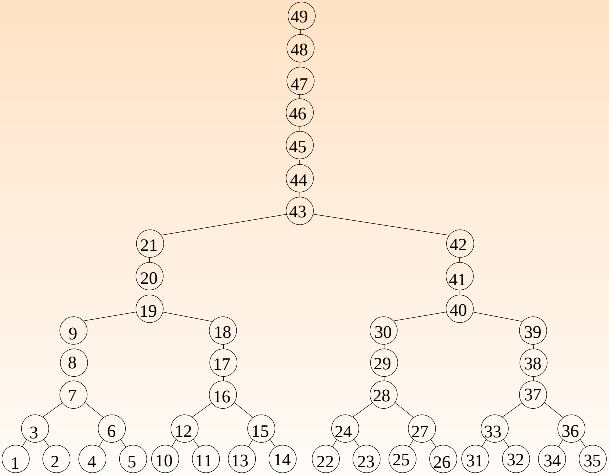
Global nested dissection orderings



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

Initial reordering

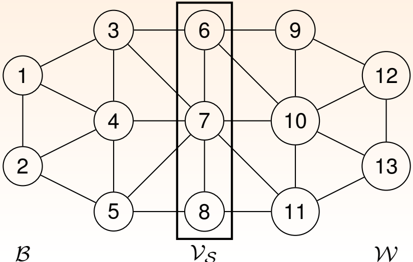
Global nested dissection orderings



Reorderings

Global nested dissection orderings

$$
\begin{array}{r}
1 \\ 2 \\ 3 \\ 4 \\ 5 \\ 6 \\ 7 \\ 8 \\ 9 \\ 10 \\ 11 \\ 12 \\ 13
\end{array}
\begin{pmatrix}
* & * & * & * &   &   &   &   &   &   &   &   &   \\
* & * &   & * & * &   &   &   &   &   &   &   &   \\
* &   & * & * &   & * & * &   &   &   &   &   &   \\
* & * & * & * & * &   & * & * &   &   &   &   &   \\
  & * &   & * & * &   &   & * &   &   &   &   &   \\
  &   & * &   &   & * & * &   & * & * &   &   &   \\
  &   & * & * &   & * & * & * &   & * & * &   &   \\
  &   &   & * & * &   & * & * &   &   & * &   &   \\
  &   &   &   &   & * &   &   & * & * &   & * &   \\
  &   &   &   &   & * & * &   & * & * & * & * & * \\
  &   &   &   &   &   & * & * &   & * & * &   & * \\
  &   &   &   &   &   &   &   & * & * &   & * & * \\
  &   &   &   &   &   &   &   &   & * & * & * & *
\end{pmatrix}
$$



Reorderings

Global nested dissection orderings

$$\begin{array}{c} 1 \\ 2 \\ 3 \\ 4 \\ 5 \\ 9 \\ 10 \\ 11 \\ 12 \\ 13 \\ 6 \\ 7 \\ 8 \end{array}
\left(\begin{array}{ccccc|ccccc|ccc}
* & * & * & * & & & & & & & & & \\
* & * & & * & * & & & & & & & & \\
* & & * & * & & & & & & & * & * & \\
* & * & * & * & * & & & & & & & * & \\
& * & & * & * & & & & & & & * & * \\
\hline
& & & & & * & * & & * & & * & & \\
& & & & & * & * & * & * & * & * & * & \\
& & & & & & * & * & & * & & * & * \\
& & & & & * & * & & * & * & & & \\
& & & & & & * & * & * & * & & & \\
\hline
& & * & & & * & * & & & & * & * & \\
& & * & * & * & & * & * & & & * & * & * \\
& & & & * & & & * & & & & * & *
\end{array}\right)$$

Reorderings

Global nested dissection orderings

- Provided the variables are eliminated in the permuted order, no fill occurs within the zero off-diagonal blocks.
- If $|\mathcal{V}_{\mathcal{S}}|$ is small and $|\mathcal{B}|$ and $|\mathcal{W}|$ are similar, these zero blocks account for approximately half the possible entries in the matrix.
- The reordering can be applied recursively to the submatrices $A_{\mathcal{B},\mathcal{B}}$ and $A_{\mathcal{W},\mathcal{W}}$ until the vertex subsets are of size less than some prescribed threshold.
- Combinations with local reorderings.

Reorderings

Global nested dissection orderings

Algorithm (Nested dissection algorithm)

Input: *Graph $\mathcal{G}$ of a symmetrically structured matrix A and a partitioning algorithm PartitionAlg.*
Output: *A permutation vector p that defines a new labelling of the vertices of $\mathcal{G}$.*

```
recursive function ($p$ = nested_dissection($A$, PartitionAlg))
    if dissection has terminated then        ▷ Vertex subsets are smaller than some threshold
        $p = \text{AMD}(\mathcal{V}, \mathcal{E})$                                   ▷ Compute an AMD ordering
    else
        Use PartitionAlg$(\mathcal{V}, \mathcal{E})$ to obtain the vertex partitioning $(\mathcal{B}, \mathcal{W}, \mathcal{V}_S)$
        $p_\mathcal{B}$ = nested_dissection($A_{\mathcal{B},\mathcal{B}}$, PartitionAlg)
        $p_\mathcal{W}$ = nested_dissection($A_{\mathcal{W},\mathcal{W}}$, PartitionAlg)
        $p_{\mathcal{V}_S}$ is an ordering of $\mathcal{V}_S$
        Set $p = \begin{pmatrix} p_\mathcal{B} \\ p_\mathcal{W} \\ p_{\mathcal{V}_S} \end{pmatrix}$
    end if
end recursive function
```

Reorderings

Bordered forms

- The matrix (53) is an example of a doubly bordered block diagonal (DBBD) form. More generally, a matrix is said in DBBD form if it has the block structure

$$A_{DB} = \begin{pmatrix} A_{1,1} & & & & C_1 \\ & A_{2,2} & & & C_2 \\ & & ... & & . \\ & & & A_{Nb,Nb} & C_{Nb} \\ R_1 & R_2 & ... & R_{Nb} & B \end{pmatrix}, \tag{55}$$

- The blocks can have very different sizes. A nested dissection ordering can be used to permute a symmetrically structured matrix A to a symmetrically structured DBBD form ($\mathcal{S}\{R_i\} = \mathcal{S}\{C_i^T\}$).
- If $\mathcal{S}\{A\}$ is close to symmetric then nested dissection can be applied to $\mathcal{S}\{A + A^T\}$. In finite-element applications, the DBBD form corresponds to partitioning the underlying finite-element domain into non-overlapping subdomains.

Reorderings

Bordered forms

- Coarse-grained parallel approaches aim to factorize the $A_{lb,lb}$ blocks in parallel before solving the interface problem that connects the blocks.
- The block factorization of A_{DB} is

$$A_{DB} = \begin{pmatrix} L_1 & & & & \\ & L_2 & & & \\ & & \dots & & \\ & & & L_{Nb} & \\ \widehat{R}_1 & \widehat{R}_2 & \dots & \widehat{R}_{Nb} & L_S \end{pmatrix} \begin{pmatrix} U_1 & & & & \widehat{C}_1 \\ & U_2 & & & \widehat{C}_2 \\ & & \dots & & . \\ & & & U_{Nb} & \widehat{C}_{Nb} \\ & & & & U_S \end{pmatrix},$$

where

$$\widehat{R}_{lb} = R_{lb} U_{lb}^{-1},\ \widehat{C}_{lb} = L_{lb}^{-1} C_{lb}\ (1 \le lb \le Nb),\ L_S U_S = B - \sum_{lb=1}^{Nb} \widehat{R}_{lb} \widehat{C}_{lb}.$$

- Here, for simplicity, no permutations emphasized.

Reorderings

Bordered forms

Algorithm (Coarse-grained parallel LU factorization using DBBD form)

Input: *Matrix A_{DB} in DBBD form (55).*
Output: *Block LU factorization.*

```
Initialise $S = B$
for $lb = 1 : Nb$ do
    $A_{lb,lb} = L_{lb} U_{lb}$                          ▷ $LU$ factorization of square block on diagonal
    $\widehat{R}_{lb} = R_{lb} U_{lb}^{-1}$              ▷ Triangular solve for bottom-border blocks
    $\widehat{C}_{lb} = L_{lb}^{-1} C_{lb}$              ▷ Triangular solve for right-border blocks
end for
$S = S - \sum_{lb=1}^{Nb} \widehat{R}_{lb} \widehat{C}_{lb}$   ▷ Assemble updates to interface block
$S = L_S U_S$                                         ▷ Factorize updated interface block (Schur complement)
```

Reorderings

Bordered forms

- Factorization of each individual $A_{lb,lb}$ and solve steps can be parallelized.
- The assembly of the interface block S and its LU can be partially parallelized.
- S is generally significantly denser than the other blocks.
- If A is not SPD then factorizing the $A_{lb,lb}$ blocks without considering the entries in the border can potentially lead to stability problems. Consider the first step in factorizing $A_{lb,lb}$ and the threshold pivoting test for a sparse LU factorization. The pivot candidate $(A_{lb,lb})_{11}$ must satisfy

  $$\max\{\max_{i>1} |(A_{lb,lb})_{i1}|, \max_{k} |(R_{lb})_{k1}|\} \le \gamma^{-1} |(A_{lb,lb})_{11}|,$$

  where $\gamma \in (0, 1]$ is the threshold parameter.
- Large entries in the row border matrix R_{lb} can prevent pivots being selected within $A_{lb,lb}$.

Reorderings

Singly bordered form

- Singly bordered block diagonal (SBBD) form

$$A_{SB} = \begin{pmatrix} A_{1,1} & & & & C_1 \\ & A_{2,2} & & & C_2 \\ & & \dots & & . \\ & & & A_{Nb,Nb} & C_{Nb} \end{pmatrix},$$

$A_{lb,lb}$ are rectangular $m_{lb} \times n_{lb}$. The linear system becomes

$$\begin{pmatrix} A_{1,1} & & & & C_1 \\ & A_{2,2} & & & C_2 \\ & & \dots & & . \\ & & & A_{Nb,Nb} & C_{Nb} \end{pmatrix} \begin{pmatrix} x_1 \\ \vdots \\ x_{Nb} \\ x_I \end{pmatrix} = \begin{pmatrix} b_1 \\ b_2 \\ \vdots \\ b_{Nb} \end{pmatrix}, \qquad (56)$$

x_{lb} is of length n_{lb}, x_I is a vector of length n_I of interface variables, and the right-hand side vectors b_{lb} are of length m_{lb}, such that

$$\begin{pmatrix} A_{lb,lb} & C_{lb} \end{pmatrix} \begin{pmatrix} x_{lb} \\ x_I \end{pmatrix} = b_{lb}, \quad 1 \leq lb \leq Nb.$$

Reorderings

Singly bordered form

- A partial factorization of each block matrix is performed, that is,

$$\begin{pmatrix} A_{lb,lb} & C_{lb} \end{pmatrix} = P_{lb} \begin{pmatrix} L_{lb} & \\ \bar{L}_{lb} & I \end{pmatrix} \begin{pmatrix} U_{lb} & \bar{U}_{lb} \\ & S_{lb} \end{pmatrix} Q_{lb}, \tag{57}$$

- Pivots can only be chosen from the columns of $A_{lb,lb}$ because the columns of C_{lb} have entries in at least one other border block C_{jb} $(jb \neq lb)$.
- The pivot candidate $(A_{lb,lb})_{11}$ at the first elimination step must satisfy

$$\max_{i>1} |(A_{lb,lb})_{i1}| \leq \gamma^{-1} |(A_{lb,lb})_{11}|,$$

Reorderings

Singly bordered form

Algorithm (Coarse-grained parallel LU factorization and solve using SBBD form)

Input: *Linear system in SBBD form (56).*
Output: *Block LU factorization and computed solution x.*

```
$S = 0$ and $z_I = 0$
for $lb = 1 : Nb$ do
    Perform a partial $LU$ factorization (57) of $(A_{lb,lb},\ C_{lb})$.
    Solve $P_{lb} \begin{pmatrix} L_{lb} & \\ \bar{L}_{lb} & I \end{pmatrix} \begin{pmatrix} y_{lb} \\ \bar{y}_{lb} \end{pmatrix} = b_{lb}$
    $S = S + S_{lb}$ and $z_I = z_I + \bar{y}_{lb}$                     ▷ Assemble $S$ and $z_I$
end for
$S = P_s L_s U_s Q_s$                       ▷ $P_s$ and $Q_s$ are permutation matrices
Solve $P_s L_s\, y_I = z_I$ and then $U_s Q_s\, x_I = y_I$    ▷ Forward then back substitution
for $lb = 1 : Nb$ do
    Solve $U_{lb}\, Q_{lb}\, x_{lb} = y_{lb} - \bar{U}_{lb}\, Q_{lb}\, x_I$
end for
```

Reorderings

Ordering to singly bordered form

- The objective is to permute A to an SBBD form with a narrow column border.
- One way to do this is to choose the number $Nb > 1$ of required blocks and use ND to compute a vertex separator $\mathcal{V}_S$ of $\mathcal{G}(A + A^T)$ such that removing $\mathcal{V}_S$ and its incident edges splits $\mathcal{G}(A + A^T)$ into Nb components.
- Then initialise the set $\mathcal{S}_\mathcal{C}$ of border columns to $\mathcal{V}_S$ and let $\mathcal{V}_{1b}, \mathcal{V}_{2b}, \ldots, \mathcal{V}_{Nb}$ be the subsets of column indices of A that correspond to the Nb components and let $n_{i,kb}$ be the number of column indices in row i that belong to $\mathcal{V}_{kb}$. If $lb = \arg\max_{1 \le kb \le Nb} |n_{i,kb}|$ then row i is assigned to partition lb. All column indices in row i that do not belong to $\mathcal{V}_{lb}$ are moved into $\mathcal{S}_\mathcal{C}$.

Reorderings

Ordering to singly bordered form

Algorithm (SBBD ordering of a general matrix)

Input: *Matrix A, the number $Nb > 1$ of blocks, vertex separator $\mathcal{V}_{\mathcal{S}}$ of $\mathcal{G}(A + A^T)$.*
Output: *Vector $block$ such that $block(i)$ denotes the partition in the SBBD form to which row i is assigned ($1 \le i \le n$) and $\mathcal{S}_{\mathcal{C}}$ is the set of border columns.*

```
Initialise $\mathcal{S}_{\mathcal{C}} = \mathcal{V}_{\mathcal{S}}$ and $block(1:n) = 0$
Initialise $\mathcal{V}_{kb}$ to hold the column indices of $A$ that correspond to component $kb$ of $\mathcal{G}(A + A^T)$
  after the removal of $\mathcal{V}_{\mathcal{S}}$, $1 \le kb \le Nb$
for each row $i$ do
    Add up the number $n_{i,kb}$ of column indices belonging to $\mathcal{V}_{kb}$, $1 \le kb \le Nb$
    Find $lb = \arg\max_{1 \le kb \le Nb} \; n_{i,kb}$
    $block(i) = lb$
    for each column index $j$ in row $i$ do
        if $j \in \mathcal{V}_{kb}$ and $kb \ne lb$ then
            Remove $j$ from $\mathcal{V}_{kb}$ and add to $\mathcal{S}_{\mathcal{C}}$
        end if
    end for
end for
Assign the rows $i$ for which $block(i) = 0$ equally between the $Nb$ partitions.
If some column $j \in \mathcal{S}_{\mathcal{C}}$ has nonzero entries only in rows belonging to partition $kb$ then
```

Complexity

Complexity of factorizations

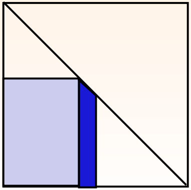
- Starting by showing complexity of LU and Cholesky
- The complexity of the most critical steps in the factorization is highly dependent on the amount of fill-in, as can be seen from the following observation.

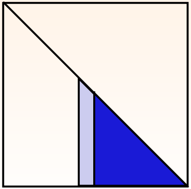
Observation

The operations to perform the sparse LU factorization $A = LU$ and the sparse Cholesky factorization $A = LL^T$ are $O(\sum_{j=1}^{n} |col_L\{j\}||row_U\{j\}|)$ and $O(\sum_{j=1}^{n} |col_L\{j\}|^2)$ respectively, where $|row_U\{j\}|$ and $|col_L\{j\}|$ are the number of off-diagonal entries in row j of U and column j of L, respectively.

Complexity

- Overall time dominated by time for the factorization
- General dense matrices
  - Space: $O(n^2)$
  - Time: $O(n^3)$
- General sparse matrices
  - Space: $\eta(L) = n + \sum_{i=1}^{n-1}(\eta(L_{*i}) - 1)$
  - Time in the i-th step: $\eta(L_{*i}) - 1$ divisions, $1/2(\eta(L_{*i}) - 1)\eta(L_{*i})$ multiple-add pairs
  - Time totally: $1/2\sum_{i=1}^{n-1}(\eta(L_{*i}) - 1)(\eta(L_{*i}) + 2)$

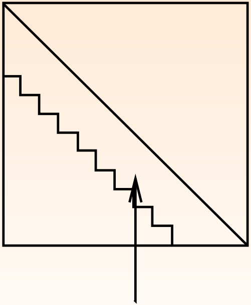




Complexity

Complexity

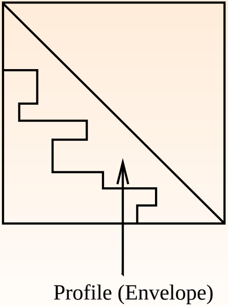
- Band schemes ($\beta << n$)
  - Space: $O(\beta n)$
  - Time: $O(\beta^2 n)$



Band

Complexity

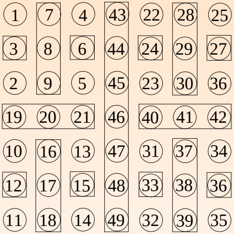
- Profile/envelope schemes
  - Space: $\sum_{i=1}^{n} \beta_i$
  - Frontwidth: $\omega_i(A) = |\{k|k > i \wedge a_{kl} \neq 0 \ for \ some \ l \leq i\}|$
  - Time: $1/2 \sum_{i=1}^{n-1} \omega_i(A)(\omega_i(A) + 3)$



From direct to iterative methods

Complexity

- General sparse schemes can be analyzed in some cases
  - Nested dissection



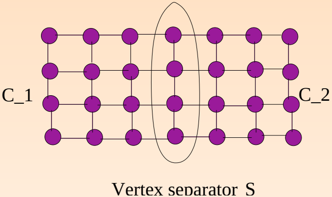
1	7	4	43	22	28	25
3	8	6	44	24	29	27
2	9	5	45	23	30	36
19	20	21	46	40	41	42
10	16	13	47	31	37	34
12	17	15	48	33	38	36
11	18	14	49	32	39	35

Definition

(α, σ) separation of a graph with n vertices: each its subgraph can be separated by a vertex separator S such that its size is of the order $O(n^{\sigma})$ and the separated subgraphs components have sizes $\leq \alpha n$,$1/2 \leq \alpha < 1$.

From direct to iterative methods

Complexity: Generalized nested dissection



- Planar graphs, 2D finite element graphs (bounded degree)
  - $\sigma = 1/2$, $\alpha = 2/3$
  - Space: $O(n \log n)$
  - Time: $O(n^{3/2})$
- 3D Finite element graphs
  - $\sigma = 2/3$
  - Space: $O(n^{4/3})$
  - Time: $O(n^2)$
- Lipton, Rose, Tarjan (1979), Teng (1997).

Outline

1. Introduction
2. Introductory notation and terminology
3. Factorizations
4. Graphs and sparse matrices
5. Sparse matrices and data structures
6. Symbolic factorizations
7. Symbolic Cholesky factorization
8. Cholesky factorization
9. Sparse LU factorization
10. Stability, ill-conditioning, indefiniteness
11. Blocks in the input matrix
12. Sparse Least Squares and QR factorization
13. Reorderings
14. Algebraic preconditioning
15. Incomplete factorizations
16. Sparse approximate inverses

Algebraic preconditioning

Algebraic preconditioning

- Finite precision arithmetic: computed factors are not exact.
- Moreover, the effort to obtain more accurate results can lead to complex coding and unavoidable inefficiencies magnified by modern computer architectures.
- Potential solution: intentionally relaxing the required accuracy of the computed factors.
- Simpler, cheaper, sparser approximate factorization of A (or A^{-1}): preconditioners.

Algebraic preconditioning

Algebraic preconditioning: use in combination with iterations

- Using the preconditioner in combination with an iterative solver.
- Our terminology: an algebraic approximate factorization is called an incomplete factorization to distinguish it from a complete factorization of a direct method.
- Used with iterative methods for solving $Ax = b$ from their two main classes:
  - stationary (relaxation) iterative methods and
  - Krylov subspace methods.

Algebraic preconditioning

Stationary iterative methods

- Stationary iterative methods work by splitting A as follows:

$$A = M - N,$$

- The matrix M is chosen to be nonsingular and easy to invert. An initial guess $x^{(0)}$, the iterations are then given by

$$x^{(k+1)} = M^{-1}Nx^{(k)} + M^{-1}b, \quad k = 0, 1, \ldots \tag{58}$$

This can be rewritten as

$$x^{(k+1)} = x^{(k)} + M^{-1}(b - Ax^{(k)}) = x^{(k)} + M^{-1}\,r^{(k)}, \quad k = 0, 1, \ldots \tag{59}$$

where the vector $r^{(k)} = b - Ax^{(k)}$ is the residual in the k-th iteration.

Algebraic preconditioning

Stationary iterative methods

- By substituting $b = r^{(k)} + Ax^{(k)}$ into $x = A^{-1}\, b$, we obtain

$$x = A^{-1}(r^{(k)} + Ax^{(k)}) = x^{(k)} + A^{-1}\, r^{(k)}.$$

  If M is used to approximate A, we again get the iteration above.

- Further

$$r^{(k+1)} = b - A(x^{(k)} + M^{-1}\, r^{(k)}) = (I - AM^{-1})\, r^{(k)} = \ldots = (I - AM^{-1})^{k+1}\, r^{(0}$$

  for the error vector on iteration k: $e^{(k)} = x - x^{(k)}$:

$$e^{(k+1)} = M^{-1}N\, e^{(k)} = \ldots = (M^{-1}N)^{k+1}\, e^{(0)} = (I - M^{-1}A)^{k+1}\, e^{(0)}.$$

- The matrix $I - M^{-1}A$ or $I - AM^{-1}$ is called the iteration matrix. In general, monitoring residuals is more practical.

Algebraic preconditioning

Stationary iterative methods

Theorem

For any initial $x^{(0)}$ and vector b, the stationary iteration converges if and only if the spectral radius of $(I - M^{-1}A)$ is less than unity.

Proof.

The spectral radius of an $n \times n$ matrix C with eigenvalues $\lambda_1, \lambda_2, \ldots, \lambda_n$ is defined to be

$$\rho(C) = \max\{|\lambda_i| \,|\, 1 \leq i \leq n\}.$$

Furthermore, the sequence of matrix powers C^k, $k = 0, 1, \ldots,$ converges to zero if and only if $\rho(C) < 1$. It follows that if the spectral radius of $(I - M^{-1}A)$ is less than unity then the iteration converges for any $x^{(0)}$ and b. Conversely, the relation

$$x^{(k+1)} - x^{(k)} = (I - M^{-1}N)(x^{(k)} - x^{(k-1)}) = \ldots = (I - M^{-1}N)^k M^{-1}(b - Ax^{(0)})$$

shows that if the iteration converges for any $x^{(0)}$ and b then $(I - M^{-1}N)^k v$ converges to zero for any v. Consequently, $\rho(I - M^{-1}A)$ must be less than unity, and the result follows. □ □

Algebraic preconditioning

Stationary iterative methods

- Generally impractical to compute the spectral radius and sufficient conditions that guarantee convergence are used.
- A sufficient condition for convergence is $\|I - M^{-1}A\| < 1$.
- Consider splitting (diagonal, strict lower triangular, strict upper triangular)

$$A = D_A + L_A + U_A.$$

- For $\omega > 0$ is a scalar parameter, classical methods include:
  - Richardson method: $M = \omega^{-1} I$,
  - Jacobi and damped Jacobi methods: $M = D_A$ and $M = \omega^{-1} D_A$,
  - Gauss-Seidel and SOR methods: $M = D_A + L_A$ and $M = \omega^{-1} D_A + L_A$.

Algebraic preconditioning

Stationary iterative methods

Theorem

If $A \in \mathbb{R}^{n\times n}$ is strongly diagonally dominant then Jacobi method and Gauss-Seidel method are convergent.

Theorem

If $A \in \mathbb{R}^{n\times n}$ is symmetric with positive diagonal D_A then the Jacobi method is convergent iff A and $2D_A - A$ are positive definite.

Theorem

If $A \in \mathbb{R}^{n\times n}$ is symmetric and positive definite then the Gauss-Seidel method is convergent.

Algebraic preconditioning

Krylov subspace methods

- Non-stationary iterative methods are of the form
$$x^{(k+1)} = x^{(k)} + \omega^{(k)} M^{-1} r^{(k)}, \quad k = 0, 1, \ldots.$$
where the $\omega^{(k)}$ are scalars.
- In this class, Krylov subspace methods are the most effective.
- Given a vector y, the k-th Krylov subspace $\mathcal{K}^{(k)}(A, y)$ generated by A from the vector y is defined to be
$$\mathcal{K}^{(k)}(A, y) = span(y, Ay, \ldots, A^{k-1}\, y).$$
- Generate a sequence of approximate solutions $x^{(k)} \in x^{(0)} + \mathcal{K}^{(k)}(A, r^{(0)})$ such that the norm of the corresponding residuals $r^{(k)} \in \mathcal{K}^{(k+1)}(A, r^{(0)})$ converge to zero.
- SPD systems the conjugate gradient method (CG); nonsymmetric systems: GMRES, BiCG, no single method of choice.
- At each iteration only matrix-vector products with A (and possibly with A^T) required.

Algebraic preconditioning

Krylov subspace methods

- Powerful, if (and only if) combined with a preconditioner: the most widely-used class of preconditioned iterative methods.
- Unfortunately, for a given A, b and $x^{(0)}$, it is usually not possible to predict the rate of convergence.
- If A is a SPD matrix then for CG

$$\|x - x^{(k)}\|_A \leq 2 \left(\frac{\sqrt{\kappa(A)} - 1}{\sqrt{\kappa(A)} + 1} \right)^k \|x - x^{(0)}\|_A,$$

  where $\kappa(A)$ is the spectral condition number.
- Often highly pessimistic. Does not show the potential for CG to converge superlinearly and that the rate of convergence depends on the distribution of all the eigenvalues of A.

Algebraic preconditioning

Krylov subspace methods

- For non-SPD matrices, less is known.
- Some favourable properties like: the minimal residual method (MINRES) for solving symmetric indefinite systems in exact arithmetic, has the norm of the residual monotonically decreasing.
- No general descriptive convergence theory is available for Krylov subspace methods for nonsymmetric systems (including GMRES).

Algebraic preconditioning

Krylov subspace methods

- Preconditioning: application of a matrix (or linear operator) to the linear system to yield a different linear system with more favourable properties.
- Consider the preconditioned linear system

$$M^{-1}Ax = M^{-1}b.$$

Here M^{-1} is applied to A from the left. We say that A is preconditioned from the left and M is a left preconditioner. Analogously, from the right:

$$AM^{-1}y = b, \qquad x = M^{-1}y.$$

Algebraic preconditioning

Krylov subspace methods

- It is not possible to determine a priori which variant is the best.

Theorem

Let δ and Δ be positive numbers. Then for any $n \geq 3$ there exist nonsingular $n \times n$ matrices A and M such that all the entries of $M^{-1}A - I$ have absolute value less than δ and all the entries of $AM^{-1} - I$ have absolute values greater than Δ.

- The choice between left and right preconditioning may be based on properties of the coupling of the preconditioner with the iterative method or on the distribution of the eigenvalues of A.

Algebraic preconditioning

Krylov subspace methods

- The computed quantities readily available during a preconditioned iterative method depend on how the preconditioner is applied and this may influence the choice. These quantities may be used, for example, to decide when to terminate the iterations.
- An obvious advantage of right preconditioning is that in exact arithmetic, the residuals for the right preconditioned system are identical to the true residuals, enabling convergence to be monitored accurately.
- In some cases, the numerical properties of an implementation and/or the computer architecture may also play a part.

Algebraic preconditioning

Krylov subspace methods

- For M in factorized form $M = M_1 M_2$, two-sided (or split) preconditioning is an option. The iterative method then solves the transformed system

$$M_1^{-1} A M_2^{-1} y = M_1^{-1} b, \qquad x = M_2^{-1} y.$$

- If A and M are SPD matrices then we would like the preconditioned matrix $M_1^{-1} A M_1^{-T}$ to be SPD. However, it is not necessary to use a two-sided transformation with the preconditioned conjugate gradient (PCG) method because it can be formulated using the M-inner product in which the matrix $M^{-1}A$ is self-adjoint.

Algebraic preconditioning

Krylov subspace methods

Theorem

Let A and M be SPD matrices. Then $M^{-1}A$ is self-adjoint in the M-inner product.

Proof.

Self-adjointness is implied by the following chain of equivalences.

$$\langle M^{-1}Ax, y\rangle_M = \langle Ax, y\rangle = \langle x, Ay\rangle = \langle x, MM^{-1}Ay\rangle = \langle Mx, M^{-1}Ay\rangle = \langle x, M^{-1}Ay\rangle_M \,.$$

□ □

- Left preconditioned CG with the M-inner product is mathematically equivalent to right preconditioned CG with the M^{-1}-inner product.
- If A is symmetric but not PD, the PCG method can breakdown (division by a zero quantity).

Algebraic preconditioning

Krylov subspace methods

- An obvious goal: to achieve rapid convergence.
- The preconditioner should aim to reduce the condition number. This may not be sufficient to give fast convergence.
- For general matrices, despite the lack of theoretical guarantees regarding convergence, many useful preconditioners motivated by bounding the condition number of the preconditioned matrix.
- Choosing a preconditioner is often based on how costly it is to compute and on some indicators that potentially reflect its quality.
- In particular, the accuracy of a preconditioner M can be assessed using the norm of the error matrix

  $$\|E\| = \|M - A\|,$$

  and its stability can be measured using

  $$\|M^{-1}E\| = \|I - M^{-1}A\| \quad \text{or} \quad \|EM^{-1}\| = \|I - AM^{-1}\|.$$
- In some cases, the inverse M^{-1} is computed directly. In this case we have an approximate inverse preconditioner.

Algebraic preconditioning

Simple preconditioners

- The simplest preconditioner consists of the diagonal of the matrix $M = D_A$. This is known as the (point) Jacobi preconditioner.
- Block versions can be derived by partitioning $\mathcal{V} = \{1, 2, \ldots, n\}$ into mutually disjoint subsets $\mathcal{V}_1, \ldots, \mathcal{V}_l$ and then setting

$$m_{ij} = \begin{cases} a_{ij} & \text{if } i \text{ and } j \text{ belong to the same subset } \mathcal{V}_k \text{ for some } k, 1 \\ 0 & \text{otherwise.} \end{cases}$$

- The SSOR preconditioner, like the Jacobi preconditioner, can be derived from A without any work. If A is symmetric then using the notation (592), the SSOR preconditioner is defined to be

$$M = (D_A + L_A) D_A^{-1} (D_A + L_A)^T,$$

or, using a parameter $0 < \omega < 2$, as

$$M = \frac{1}{2 - \omega} (\frac{1}{\omega} D_A + L_A) (\frac{1}{\omega} D_A)^{-1} (\frac{1}{\omega} D_A + L_A)^T.$$

- Finding optimal value of ω typically expensive.

Algebraic preconditioning

The Eisenstat trick

- Generally cheaper to apply M^{-1} and A separately,
- But: let M be given by

$$M = (D + L_A)\,[D^{-1}(D + U_A)] = M_1\, M_2,$$

  where D is a nonsingular diagonal matrix.
- The SSOR matrix is one example in the symmetric case but more generally $D \neq D_A$.
- Using two-sided preconditioning, this becomes

$$A'y = M_1^{-1} A M_2^{-1}\, y = (D{+}L_A)^{-1} A[D^{-1}(D{+}U_A)]^{-1}\, y = (D{+}L_A)^{-1} b.$$

- Setting

$$\bar{L} = D^{-1} L_A, \;\; \bar{U} = D^{-1} U_A, \;\; \bar{A} = D^{-1} A, \;\; \text{and} \;\; \bar{b} = (I + \bar{L})^{-1} D^{-1}\, b,$$

  we obtain

$$\begin{aligned} A' &= (D + L_A)^{-1} A[D^{-1}(D + U_A)]^{-1} = [(D + L_A)^{-1} D] D^{-1} A[D^{-1}(D + U_A)]^{-1} \\ &= [D^{-1}(D + L_A)]^{-1} D^{-1} A(I + D^{-1} U_A)^{-1} = (I + \bar{L})^{-1} \bar{A}(I + \bar{U})^{-1}. \end{aligned}$$

Algebraic preconditioning

The Eisenstat trick

- That is, the system becomes

$$A' y = (I+\bar{L})^{-1} \bar{A} (I+\bar{U})^{-1} \, y = (I+\bar{L})^{-1} D^{-1} \, b = \bar{b}.$$

  If y solves (605) then the solution x of $(I+\bar{U}) \, x = y$ solves $Ax = b$.
- Further

$$\begin{aligned} A' &= (I+\bar{L})^{-1} \, (I+\bar{L}+D^{-1} D_A - 2I + I + \bar{U})(I+\bar{U})^{-1} \\ &= (I+\bar{L})^{-1} \, [(I+\bar{L})(I+\bar{U})^{-1} + (D^{-1} D_A - 2I)(I+\bar{U})^{-1} + I] \\ &= (I+\bar{U})^{-1} + (I+\bar{L})^{-1} \, [(D^{-1} D_A - 2I)(I+\bar{U})^{-1} + I]. \end{aligned}$$

- Thus to compute $z = A' w = (I+\bar{L})^{-1} \bar{A} (I+\bar{U})^{-1} w$ for a given w, it is necessary only to solve two triangular systems

  $(I+\bar{U}) \, z_1 = w$ followed by $(I+\bar{L}) \, z_2 = (D^{-1} D_A - 2I) \, z_1 + w,$

  and then set $z = z_1 + z_2$.
- This trick is not a preconditioner: it is a way of applying the preconditioner of special shape.

Algebraic preconditioning

Some special classes of matrices

- The development of algebraic preconditioners has been closely connected to solving discretized PDEs.
- Two-dimensional Poisson problem, discretized using a uniform regular grid, finite differences, zero Dirichlet conditions on the boundary, natural ordering.

$$A = \begin{pmatrix} 4 & -1 & & -1 & & & & & \\ -1 & 4 & -1 & & -1 & & & & \\ & -1 & 4 & & & -1 & & & \\ -1 & & & 4 & -1 & & -1 & & \\ & -1 & & -1 & 4 & -1 & & -1 & \\ & & -1 & & -1 & 4 & & & -1 \\ & & & -1 & & & 4 & -1 & \\ & & & & -1 & & -1 & 4 & -1 \\ & & & & & -1 & & -1 & 4 \end{pmatrix}.$$

- If the spatial discretization on the domain is characterized by the mesh parameter h then the size of A is inversely proportional to h.
- $\kappa(A)$ depends asymptotically on h^{-2}.

Algebraic preconditioning

Some special classes of matrices

- Matrices with similar banded sparsity patterns with nonzeros on only a small number of subdiagonals arise from simple finite difference or finite element discretizations of other partial differential equations.
- Particular cases of special classes of matrices help to describe the theoretical background behind the discretized systems.
- Let the off-diagonal entries of the nonsingular matrix A be nonpositive (that is, $a_{ij} \leq 0$ for all $i \neq j$). Then A is a (nonsingular) M-matrix if one of the following holds:
    - $A + D$ is nonsingular for any diagonal matrix D with nonnegative entries;
    - all the entries of A^{-1} are nonnegative;
    - all principal minors of A are positive.
- The matrix above is an example of an M-matrix. A symmetric M-matrix is known as a Stieltjes matrix, and such a matrix is positive definite.

Some special classes of matrices

- The class of nonsingular H-matrices includes matrices coming from simple discretizations of convection-diffusion problems. The comparison matrix $C(A)$ of A is defined to have entries

$$C(A)_{ij} = \begin{cases} -|a_{ij}|, & i \neq j, \\ |a_{ij}|, & i = j. \end{cases}$$

- If $C(A)$ is a nonsingular M-matrix then A is a nonsingular H-matrix.
- A is diagonally dominant by rows if

$$\sum_{j=1,\, j\neq i}^{n} |a_{ij}| \leq |a_{ii}|, \quad 1 \leq i \leq n. \tag{60}$$

- A is strictly diagonally dominant by rows if strict inequality holds for all i. A is (strictly) diagonally dominant by columns if A^T is (strictly) diagonally dominant by rows.

Algebraic preconditioning

Some special classes of matrices

- A is said to be irreducibly diagonally dominant if it is irreducible and the inequalities are satisfied with strict inequality for at least one row i. If A is strictly diagonally dominant by rows or columns or is irreducibly diagonally dominant then it is nonsingular and factorizable.
- The class of diagonally dominant matrices is closely connected to that of nonsingular H-matrices. For example, the property that there exists a diagonal matrix D with positive entries such that AD is strictly diagonally dominant is equivalent to A being a nonsingular H-matrix.

Algebraic preconditioning

Some special classes of matrices: once more

- Still theoretical assumptions are rather strong.
- Concept of special matrices

Theorem

Matrix A is called a **regular M-matrix** *if $a_{ij} \leq 0, i \neq j$, is regular and $A^{-1} \geq 0$.*

Theorem

A is a **H-matrix** *if $B = |D_A| - |A - D_A|$ is an M-matrix.*

- Many equivalent definitions

Incomplete factorization breakdown

- Dropping entries can lead to breakdown of the incomplete factorization, that is, a zero pivot may be encountered during the factorization (or a non positive pivot in the Cholesky case).
- It is only possible to predict when this will happen in special cases, as stated in the following theorem, which is a consequence of the fact that being an M-matrix or an H-matrix is preserved in the sequence of the Schur complements during the factorization.

Theorem

Let A be a nonsingular M-matrix or H-matrix. If the target sparsity pattern of the incomplete factors contains the positions of the diagonal entries then the incomplete factorization of A does not break down.

Algebraic preconditioning

Incomplete factorization breakdown

- To illustrate the error accumulation in the incomplete factorization of an M-matrix using dropping, consider the following example.
- Let E be the error matrix. E is initialised to zero and at each stage of the factorization the dropped entries are added into it.
- After one step of the complete factorization of A the partially eliminated matrix $A^{(2)}$ is

$$A^{(2)} = \begin{pmatrix} 4 & -1 & & -1 & & & & & \\ & 3.75 & -1 & -0.25 & -1 & & & & \\ & -1 & 4 & & & -1 & & & \\ & -0.25 & & 3.75 & -1 & & -1 & & \\ & -1 & & -1 & 4 & -1 & & -1 & \\ & & -1 & & -1 & 4 & & & -1 \\ & & & -1 & & & 4 & -1 & \\ & & & & -1 & & -1 & 4 & -1 \\ & & & & & -1 & & -1 & 4 \end{pmatrix}.$$

Algebraic preconditioning

Incomplete factorization breakdown

- Suppose the filled entries -0.25 in positions $(2,4)$ and $(4,2)$ are dropped. Then the values of the corresponding diagonal entries in the subsequent elimination matrices are larger than they would have been without any dropping.
- Furthermore, as all the off-diagonal nonzero entries are negative, for any target sparsity pattern the dropped entries are negative.
- The M-matrix property applies to all subsequent Schur complements, which implies that all the entries added into E are negative and so the absolute values of the entries in E grow as the factorization proceeds (the contributions can never cancel each other out).
- Thus, although the factorization does not break down, the growth in the error is potentially a problem for the accuracy of an incomplete factorization of an M-matrix.
- Modifying the diagonal entries of A is a common approach to avoid breakdown in an incomplete factorization.

Algebraic preconditioning

Incomplete factorization breakdown

$$A = \begin{pmatrix} 3 & -2 & & 2 \\ -2 & 3 & -2 & \\ & -2 & 3 & -2 \\ 2 & & -2 & 8 \end{pmatrix}, \; L = \begin{pmatrix} 1 & & & \\ -2/3 & 1 & & \\ & -6/5 & 1 & \\ 2/3 & 4/5 & -2/3 & 1 \end{pmatrix}, \; D = \begin{pmatrix} 3 & & & \\ & 5/3 & & \\ & & 3/5 & \\ & & & 16/3 \end{pmatrix}.$$

$$\widetilde{L} = \begin{pmatrix} 1 & & & \\ -2/3 & 1 & & \\ & -6/5 & 1 & \\ 2/3 & & -10/3 & 1 \end{pmatrix}, \; \widetilde{D} = \begin{pmatrix} 3 & & & \\ & 5/3 & & \\ & & 3/5 & \\ & & & 0 \end{pmatrix}.$$

Figure: An example to illustrate breakdown. The matrix A and its square-root free factors are given together with the incomplete factors $\widetilde{L}$ and $\widetilde{D}$ that result from dropping the entry l_{24} during the factorization. $\tilde{d}_{44} = 0$ means the incomplete factorization has broken down.

- Remedy: perturb the diagonal value causing the breakdown.
- Practice of making simple ad hoc modifications not very positive. If breakdown (or near-breakdown) occurs, it may be too late.

Algebraic preconditioning

Incomplete factorization breakdown

- An alternative and more effective strategy to avoid breakdown is to modify all the diagonal entries of A a priori and then compute an incomplete factorization of $A + \alpha I$, where the shift $\alpha > 0$ is a scalar parameter.
- It is always possible to find α such that $A + \alpha I$ is nonsingular and diagonally dominant and is thus an H-matrix.
- However, being an H-matrix is not a necessary condition for a matrix to be factorizable and, in practice, much smaller values of α can provide incomplete factorizations for which $\|E\|$ is small.
- A simple trial-and-error procedure for choosing a shift is given below.

Algebraic preconditioning

Incomplete factorization breakdown

Algorithm (Trial-and-error global shifted incomplete factorization)

Input: *Matrix A, incomplete factorization algorithm, initial shift $\alpha^{(0)}$*
Output: *Shift α such that $A + \alpha I \approx \widetilde{L}\widetilde{U}$.*

```
for $k = 0, 1, 2, \ldots$ do
    $A + \alpha^{(k)} I \approx \widetilde{L}\widetilde{U}$        ▷ Perform incomplete factorization
    If successful, $\alpha = \alpha^{(k)}$ and return
    $\alpha^{(k+1)} = 2\alpha^{(k)}$
end for
```

Algebraic preconditioning

Incomplete factorization breakdown

- An alternative approach to avoid small pivots:to follow what is done in sparse direct solvers and incorporate partial or threshold pivoting within the incomplete factorization algorithm: preprocessing by reordering, scaling etc.
- One way to attempt to minimize the norm of the error matrix E is to select the pivot candidate to minimize the sum of the absolute values of the dropped (discarded) entries. However, this minimum discarded fill ordering is typically too expensive to be useful in practice.

Algebraic preconditioning

Introduction to incomplete factorizations

- Incomplete factorizations fall into three main classes:
  - Threshold-based methods: locations of permissible fill-in are determined in conjunction with the numerical factorization of A; entries of the computed factors of absolute value greater than a prescribed threshold $\tau > 0$ are dropped.
  - Memory-based methods in which the amount of memory available for the incomplete factorization is prescribed and only the largest entries in each row (or column) are retained.
  - Structure-based methods: an initial symbolic factorization phase determines the location of permissible entries using only $\mathcal{S}\{A\}$. This allows the memory requirements to be determined before an incomplete numerical factorization is performed. The specified set of positions is called the target sparsity pattern.

Algebraic preconditioning

Introduction to incomplete factorizations

- The basic dropping approaches can be combined and employed in conjunction with sparsifying A before the factorization commences.
- Sparsification of A after permuting reveals a block structure.

$$\begin{array}{ccccccc} & 1 & 2 & 3 & 4 & 5 & 6 \\ 1 & \ast & \ast & \ast & & & \\ 2 & \ast & \ast & f & \ast & \ast & \ast \\ 3 & \ast & f & \ast & f & f & f \\ 4 & & \ast & f & \ast & \ast & \ast \\ 5 & & \ast & f & \ast & \ast & \ast \\ 6 & & \ast & f & \ast & \ast & \ast \end{array} \qquad \begin{array}{ccccccc} & 2 & 4 & 1 & 3 & 5 & 6 \\ 2 & \ast & \ast & \ast & & \ast & \ast \\ 4 & \ast & \ast & f & & \ast & \ast \\ 1 & \ast & f & \ast & \ast & f & f \\ 3 & & & \ast & \ast & f & f \\ 5 & \ast & \ast & f & f & \ast & \ast \\ 6 & \ast & \ast & f & f & \ast & \ast \end{array} \qquad \begin{array}{ccccccc} & 2 & 4 & 1 & 3 & 5 & 6 \\ 2 & \ast & \ast & & & \ast & \ast \\ 4 & \ast & \ast & & & \ast & \ast \\ 1 & & & \ast & \ast & & \\ 3 & & & \ast & \ast & & \\ 5 & \ast & \ast & & & \ast & \ast \\ 6 & \ast & \ast & & & \ast & \ast \end{array}$$

Figure: Illustration of matrix sparsification. f denotes filled entries in the factors. On the left is the original matrix A with its filled entries; in the centre is the permuted matrix with its filled entries; on the right is the sparsified permuted matrix after dropping the entries of A in positions $(1,3)$ and $(3,1)$ (it has no filled entries).

Algebraic preconditioning

Incomplete factorization breakdown

Polynomial preconditioning

- Polynomial preconditioning selects a polynomial ϕ and applies a Krylov subspace method to solve either

  $$\phi(A)Ax = \phi(A)\,b$$

  (left preconditioning) or

  $$A\,\phi(A)\,y = b, \quad x = \phi(A)\,y$$

  (right preconditioning). ϕ should be of small degree and chosen to enhance convergence.
- Consider the characteristic polynomial $\phi_n(\mu) = \det(A - \mu I)$ of A ($\det$ denotes the determinant).
- The Cayley-Hamilton theorem states that A satisfies its own characteristic equation so that

  $$\phi_n(A) = \sum_{j=0}^{n} \beta_j\, A^j = 0,$$

Algebraic preconditioning

Polynomial preconditioning

- Provided A is nonsingular,

$$A^{-1} = (-1)^{n+1} \frac{1}{\det(A)} \sum_{j=1}^{n} \beta_j \, A^{j-1}.$$

A preconditioner can be constructed by taking the first k terms, possibly weighted by some suitable scalar coefficients, that is,

$$M^{-1} = \sum_{j=0}^{k} \gamma_j \, A^{k}.$$

Algebraic preconditioning

Polynomial preconditioning

- An important question is why such a preconditioner can help in the presence of the optimality properties of Krylov subspace methods.
- For example, at iteration $k+1$ of the CG method, $x^{(k+1)}$ satisfies

  $$x^{(k+1)} = x^{(0)} + \phi_k(A)\, r^{(0)},\ k = 0, 1, \dots,$$

  where ϕ_k is a monic polynomial of degree k. This polynomial is optimal in the sense that $x^{(k+1)}$ minimizes

  $$\|x - x^{(k+1)}\|_A^2. \tag{61}$$

- A preconditioner that is a polynomial in A cannot speed the convergence because the resulting iteration again forms the new $x^{(k+1)}$ as $x^{(0)}$ plus a polynomial in A times $r^{(0)}$, and thus the same or a higher degree polynomial is needed to achieve the same value of the A-norm of the error.
- Consequently, the number of matrix-vector multiplications cannot decrease.

Algebraic preconditioning

Polynomial preconditioning

- Nevertheless, polynomial preconditioning can be useful for a number of reasons.
  - The polynomial can improve the eigenvalue distribution of the preconditioned matrix and result in a reduction in the number of iterations required for convergence (even though the overall complexity may increase).
  - It requires very little memory and its implementation can be straightforward.
  - It can decrease the number of synchronization points in iterative methods as represented by inner products. This is potentially important for message-passing parallel architectures.

Algebraic preconditioning

Polynomial preconditioning

- Even if only a small number of terms are used in approximating A^{-1}, a crucial issue is getting $\gamma_0, \dots, \gamma_k$.
- A straightforward way of doing this: based on the Neumann series of a matrix C given by $\sum_{j=0}^{+\infty} C^j$, which is convergent if and only if $\rho(C) < 1$.
- In this case,

$$(I - C)^{-1} = \sum_{j=0}^{+\infty} C^j. \tag{62}$$

- Now let $\bar{M}$ be a nonsingular matrix and $\omega > 0$ a scalar such that the matrix $C = I - \omega \bar{M}^{-1} A$ satisfies $\rho(C) < 1$.
- Using (62),

$$A^{-1} = \omega(\omega \bar{M}^{-1} A)^{-1} \bar{M}^{-1} = \omega \, (I - C)^{-1} \bar{M}^{-1} = \omega \left(\sum_{j=0}^{+\infty} C^j \right) \bar{M}^{-1}.$$

Algebraic preconditioning

Polynomial preconditioning

- Truncating the summation gives as a possible preconditioner

$$M^{-1} = \omega \left(\sum_{j=0}^{k} C^j \right) \bar{M}^{-1}.$$

- Observe that

$$I - M^{-1}A = I - \omega \left(\sum_{j=0}^{k} C^j \right) \bar{M}^{-1}A = I - \left(\sum_{j=0}^{k} C^j \right) (I - C) = C^{k+1},$$

  which shows the positive effect of increasing k. If A and $\bar{M}$ are SPD matrices then M can be used with the CG method preconditioned from the left because $M^{-1}A$ is self-adjoint in the $\bar{M}$-inner product.
- Generalizations of the approach weight the powers of C in M^{-1} using additional scalars. The choice of $\bar{M}$ is crucial for the effectiveness of the approach.

Outline

1 Introduction
2 Introductory notation and terminology
3 Factorizations
4 Graphs and sparse matrices
5 Sparse matrices and data structures
6 Symbolic factorizations
7 Symbolic Cholesky factorization
8 Cholesky factorization
9 Sparse LU factorization
10 Stability, ill-conditioning, indefiniteness
11 Blocks in the input matrix
12 Sparse Least Squares and QR factorization
13 Reorderings
14 Algebraic preconditioning
15 Incomplete factorizations
16 Sparse approximate inverses

Incomplete factorizations

World of incomplete factorizations

- Direct factorizations may not be feasible (data structures and pivoting, operation counts, stability)
- Even by direct factorizations improving solution when using less accurate arithmetic (smaller ϵ) may be needed.
- The incomplete factors denoted here by $\widetilde{L}$ and $\widetilde{U}$;
- SPD case: $\widetilde{U} = \widetilde{L}^T$.
- We assume that the sparsity patterns of A and its incomplete factors always include the positions of the diagonal entries.
- Notation (other mentioned later): ILU(0) factorization (or an IC(0) factorization if A is SPD): $\mathcal{S}\{\widetilde{L} + \widetilde{U}\} = \mathcal{S}\{A\}$.

Incomplete factorizations

Exactness within the target sparsity pattern

Theorem

Consider the incomplete LU factorization $A + E = \widetilde{L}\widetilde{U}$ with sparsity pattern $\mathcal{S}\{\widetilde{L} + \widetilde{U}\}$. The entries of the error matrix E are zero at positions $(i, j) \in \mathcal{S}\{\widetilde{L} + \widetilde{U}\}$.

Proof.

The result clearly holds for $j = 1$. Let $(i, j) \in \mathcal{S}\{\widetilde{L} + \widetilde{U}\}$ and assume without loss of generality that $i > j > 1$. The (i, j) entry of $\widetilde{L}$ is computed as

$$\tilde{l}_{ij} = \left(a_{ij} - \sum_{k=1}^{j-1} \tilde{l}_{ik}\, \tilde{u}_{kj} \right) / \tilde{u}_{jj}$$

with the sums over k implying $(i, k) \in \mathcal{S}\{\widetilde{L} + \widetilde{U}\}$ and $(k, j) \in \mathcal{S}\{\widetilde{L} + \widetilde{U}\}$. This gives

$$a_{ij} = \widetilde{L}_{i,1:j-1}\widetilde{U}_{1:j-1,j} + \tilde{l}_{ij}\tilde{u}_{jj} = \widetilde{L}_{i,1:j}\widetilde{U}_{1:j,j} = L_{i,1:j}U_{1:j,j},$$

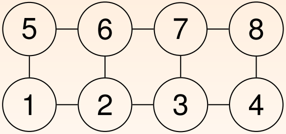
and the corresponding entry of E is zero. □

Incomplete factorizations

Incomplete factorizations and patterns

- Theorem ⇒ extending $\mathcal{S}\{\widetilde{L}+\widetilde{U}\}$ gives a larger set of entries of A for which $(E)_{ij}=0$.
- In some situations, there are straightforward ways to extend $\mathcal{S}\{\widetilde{L}+\widetilde{U}\}$. In simple discretizations of a PDE may be a natural choice is to allow $\mathcal{S}\{\widetilde{L}+\widetilde{U}\}$ to include fill-in along a few additional diagonals within the band.

$$A = \begin{array}{c|cccccccc}
 & 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 \\
\hline
1 & * & * & & & * & & & \\
2 & * & * & * & & & * & & \\
3 & & * & * & * & & & * & \\
4 & & & * & * & & & & * \\
5 & * & & & & * & * & & \\
6 & & * & & & * & * & * & \\
7 & & & * & & & * & * & * \\
8 & & & & * & & & * & *
\end{array}$$



Incomplete factorizations

Incomplete factorizations and patterns

$$\begin{pmatrix} * & * & & & * & & & \\ * & * & * & & f & * & & \\ & * & * & * & & & * & \\ & & * & * & & & & * \\ * & f & & & * & * & & \\ & * & & & * & * & * & \\ & & * & & & * & * & * \\ & & & * & & & * & * \end{pmatrix} \to \begin{pmatrix} * & * & & & * & & & \\ * & * & * & & f & * & & \\ & * & * & * & f & f & * & \\ & & * & * & & & & * \\ * & f & f & & * & * & & \\ & * & f & & * & * & * & \\ & & * & & & * & * & * \\ & & & * & & & * & * \end{pmatrix} \to$$

$$\begin{pmatrix} * & * & & & * & & & \\ * & * & * & & f & * & & \\ & * & * & * & f & f & * & \\ & & * & * & f & f & f & * \\ * & f & f & f & * & * & f & \\ & * & f & f & * & * & * & \\ & & * & f & f & * & * & * \\ & & & * & & & * & * \end{pmatrix}$$

Figure: An 8×8 banded sparse SPD matrix A and its graph $\mathcal{G}(A)$. The first three steps of a Cholesky factorization are shown. Filled entries are denoted by f.

Incomplete factorizations

Crout incomplete factorizations

- The Crout variant: computes $\widetilde{U}$ (by rows) and $\widetilde{L}$ (by columns).

Algorithm (Crout incomplete LU factorization)

Input: *Matrix A and, optionally, a target sparsity pattern $\mathcal{S}\{\widetilde{L}+\widetilde{U}\}$.*
Output: *Incomplete LU factorization $A \approx \widetilde{L}\widetilde{U}$.*

```
for $j = 1 : n$ do
    $\tilde{l}_{jj} = 1,\ \widetilde{L}_{j+1:n,j} = A_{j+1:n,j}$
    $\widetilde{U}_{j,j:n} = A_{j,j:n}$
    for $k = 1 : j-1$ such that $(j,k) \in \mathcal{S}\{\widetilde{L}\}$ do
        $\widetilde{U}_{j,j:n} = \widetilde{U}_{j,j:n} - \tilde{l}_{jk}\,\widetilde{U}_{k,j:n}$          ▷ Sparse linear combination
    end for
    Sparsify $\widetilde{U}_{j,j+1:n}$                                   ▷ Drop entries from row $j$ of $\widetilde{U}$
    for $k = 1 : j-1$ such that $(k,j) \in \mathcal{S}\{\widetilde{U}\}$ do
        $\widetilde{L}_{j+1:n,j} = \widetilde{L}_{j+1:n,j} - \tilde{u}_{kj}\,\widetilde{L}_{j+1:n,k}$      ▷ Sparse linear combination
    end for
    Sparsify $\widetilde{L}_{j+1:n,j}$                                   ▷ Drop entries from column $j$ of $\widetilde{L}$
    $\widetilde{L}_{j+1:n,j} = \widetilde{L}_{j+1:n,j}/\tilde{u}_{jj}$
end for
```

Incomplete factorizations

Row incomplete factorizations

Algorithm (Row incomplete LU factorization)

Input: *Matrix A and, optionally, a target sparsity pattern $\mathcal{S}\{\widetilde{L}+\widetilde{U}\}$.*
Output: *Incomplete LU factorization $A \approx \widetilde{L}\widetilde{U}$.*

```
for $i = 1 : n$ do
    $\tilde{l}_{ii} = 1,\ \widetilde{L}_{i,1:i-1} = A_{i,1:i-1}$
    $\widetilde{U}_{i,i:n} = A_{i,i:n}$
    Sparsify $\widetilde{L}_{1,1:i-1}$ and $\widetilde{U}_{i,i+1:n}$
    for $k = 1 : i-1$ such that $(i,k) \in \mathcal{S}\{\widetilde{L}\}$ do
        $\tilde{l}_{ik} = \tilde{l}_{ik}/\tilde{u}_{kk}$
        $\widetilde{L}_{i,k+1:i-1} = \widetilde{L}_{i,k+1:i-1} - \tilde{l}_{ik}\,\widetilde{U}_{k,k+1:i-1}$
        Sparsify $\widetilde{L}_{i,k+1:i-1}$
        $\widetilde{U}_{i,i:n} = \widetilde{U}_{i,i:n} - \tilde{l}_{ik}\,\widetilde{U}_{k,i:n}$
        Sparsify $\widetilde{U}_{i,i+1:n}$
    end for
end for
```

Incomplete factorizations

Incomplete factorizations based on shortest fill-paths

- Entries of the factors that correspond to nonzero entries of A are assigned the level 0 while each potential filled entry in position (i, j) is assigned a level as follows:

$$level(i,j) = \min_{1 \le k < \min\{i,j\}} (level(i,k) + level(k,j) + 1). \qquad (63)$$

- Given $\ell \geq 0$, during the factorization a filled entry is permitted at position (i, j) provided $level(i,j) \leq \ell$.
- The resulting level-based incomplete factorization is denoted by ILU(ℓ) (or IC(ℓ)); the basic row variant is given below.

Incomplete factorizations

Algorithm (Level-based incomplete LU factorization)

```
Initialise $level$ to 0 for nonzeros and diagonal entries of $A$ and to $n+1$ otherwise
for $i = 1 : n$ do                                                    ▷ Loop over rows
    $\tilde{l}_{ii} = 1$, $\widetilde{L}_{i,1:i-1} = A_{i,1:i-1}$ and $\widetilde{U}_{i,i:n} = A_{i,i:n}$    ▷ Initialise row $i$ of $\widetilde{L}$ and $\widetilde{U}$
    for $k = 1 : i-1$ such that $level(i,k) \leq \ell$ do
        $\tilde{l}_{ik} = \tilde{l}_{ik} / \tilde{u}_{kk}$
        for $j = k+1 : i-1$ do
            $\tilde{l}_{ij} = \tilde{l}_{ij} - \tilde{l}_{ik}\,\tilde{u}_{kj}$ and update $level(i,j)$
        end for
        for $j = i : n$ do
            $\tilde{u}_{ij} = \tilde{u}_{ij} - \tilde{l}_{ik}\,\tilde{u}_{kj}$ and update $level(i,j)$
        end for
    end for
    for $k = 1 : i-1$ do          ▷ Drop factor entries in row $i$ for which $level$ is too high
        if $level(i,k) > \ell$ then $\tilde{l}_{ik} = 0$
    end for
    for $k = i : n$ do
        if $level(i,k) > \ell$ then $\tilde{u}_{ik} = 0$
    end for
end for
```

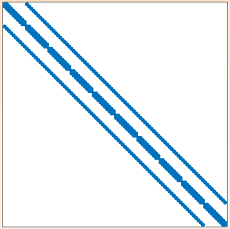
Incomplete factorizations

Incomplete factorizations based on shortest fill-paths

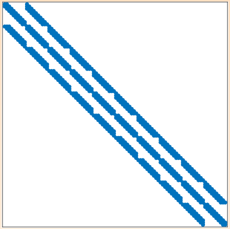
- Figure depicts $\mathcal{S}\{\widetilde{L} + \widetilde{L}^T\}$ for the IC(ℓ) factorization of A from the discretized Laplace equation on a square grid and for a matrix with a more general symmetric sparsity structure.
- The fill-in is typically generated irregularly throughout the factorization: initially few updates are needed but later steps involve many updates, leading to large amounts of dropping.
- The amount of fill-in can grow quickly with increasing ℓ and, as a result, l is typically small.
- Level-based dropping is often combined with threshold-based dropping or with sparsifying A before the factorization commences.

Incomplete factorizations

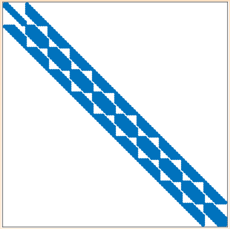
Incomplete factorizations based on shortest fill-paths



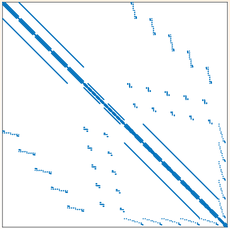
IC(0)



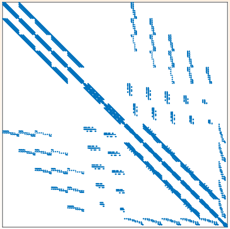
IC(2)



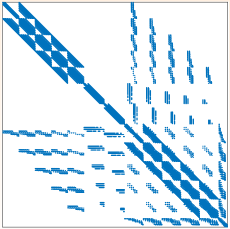
IC(4)



IC(0)



IC(2)



IC(4)

Incomplete factorizations

Incomplete factorizations based on shortest fill-paths

- The level-based strategy comes from observing that in practical examples the absolute values of the entries in the factors in positions for which $level$ is large are often small. This is the case for model problems arising from discretized PDEs.
- Theoretical understanding follows.

Theorem

Consider the ILU(ℓ) factorization of A. $level(i,j) = k$ for some $k \leq \ell$ if and only if there is a shortest fill path $i \Longrightarrow j$ of length $k+1$ in the adjacency graph $\mathcal{G}(A)$.

Incomplete factorizations based on shortest fill-paths

Algorithm (Find the sparsity pattern of row i of the ILU(ℓ) factor $\widetilde{U}$ of A: breadth first search)

```
$\mathcal{S}\{\widetilde{U}_{i,i:n}\} = \{i\}$, $\mathcal{Q} = \{i\}$          ▷ Queue holds $i$ initially
$length(i) = 0$
$visited(i) = i$
while $\mathcal{Q}$ is not empty do
    $pop(\mathcal{Q}, k)$          ▷ Take $k$ from the queue
    for $j \in adj_{\mathcal{G}(A)}(k)$ with $visited(j) \neq i$ do
        $visited(j) = i$
        if $j < i$ and $length(k) < \ell$ then
            $append(\mathcal{Q}, j)$          ▷ Add $j$ to the queue
            $length(j) = length(k) + 1$
        else if $j > i$ then
            $\mathcal{S}\{\widetilde{U}_{i,i:n}\} = \mathcal{S}\{\widetilde{U}_{i,i:n}\} \cup \{j\}$          ▷ Add $j$ to the sparsity pattern of row $i$ of $\widetilde{U}$
        end if
    end for
end while
```

Incomplete factorizations

Modifications based on maintaining row sums

- Assume that the target sparsity pattern $\mathcal{S}\{\widetilde{L}+\widetilde{U}\}$ contains $\mathcal{S}\{A\}$.
- Modified incomplete factorizations (MILU or MIC in the SPD case) seek to maintain equality between the row sums of A and $\widetilde{L}\widetilde{U}$, that is, $\widetilde{L}\widetilde{U}e = Ae$ (e is the vector of all ones).

Incomplete factorizations

Algorithm (Modified incomplete factorization (MILU))

```
Initialise $\tilde{l}_{ij} = (I + L_A)_{ij}$ for all $(i,j) \in \mathcal{S}(\widetilde{L})$          ▷ $\mathcal{S}(L_A) \subseteq \mathcal{S}(\widetilde{L})$
Initialise $\tilde{u}_{ij} = (D_A + U_A)_{ij}$ for all $(i,j) \in \mathcal{S}(\widetilde{U})$      ▷ $\mathcal{S}(U_A) \subseteq \mathcal{S}(\widetilde{U})$
for $k = 1 : n-1$ do
    for $i = k+1 : n$ such that $(i,k) \in \mathcal{S}\{\widetilde{L}\}$ do
        $\tilde{l}_{ik} = \tilde{l}_{ik} / \tilde{u}_{kk}$                                  ▷ Check that $\tilde{u}_{kk}$ is nonzero
        for $j = i : n$ such that $(k,j) \in \mathcal{S}\{\widetilde{U}\}$ do
            if $(i,j) \in \mathcal{S}\{\widetilde{U}\}$ then
                $\tilde{u}_{ij} = \tilde{u}_{ij} - \tilde{l}_{ik}\,\tilde{u}_{kj}$
            else
                $\tilde{u}_{ii} = \tilde{u}_{ii} - \tilde{l}_{ik}\,\tilde{u}_{kj}$          ▷ Modify diagonal instead of creating fill-in
            end if
        end for
        for $j = k+1 : i-1$ such that $(k,j) \in \mathcal{S}\{\widetilde{U}\}$ do
            if $(i,j) \in \mathcal{S}\{\widetilde{L}\}$ then
                $\tilde{l}_{ij} = \tilde{l}_{ij} - \tilde{l}_{ik}\,\tilde{u}_{kj}$
            else
                $\tilde{u}_{ii} = \tilde{u}_{ii} - \tilde{l}_{ik}\,\tilde{u}_{kj}$          ▷ Modify diagonal instead of creating fill-in
            end if
        end for
    end for
```

Incomplete factorizations

Modifications based on maintaining row sums

- Equality of row sums: If all the filled entries are retained (that is, $\mathcal{S}\{\widetilde{L}+\widetilde{U}\} = \mathcal{S}\{L+U\}$) then the claim holds trivially.
- Otherwise, if an entry in column j of row i of A belongs to the target sparsity pattern then its value is modified in Step 8 if $i \le j$ or in Step 15 if $i > j$. Otherwise, the i-th diagonal entry of $\widetilde{U}$ is modified (Step 10 or Step 17). In each case, $\tilde{l}_{ik}\,\tilde{u}_{kj}$ is subtracted from entries of the i-th row of the incomplete factors.
- Consider row i of $\widetilde{L}\widetilde{U}$. This product is given by

$$\begin{aligned}
\sum_{j=1}^{i-1}\tilde{l}_{ij}\sum_{k=j}^{n}\tilde{u}_{jk} &= \sum_{j=1}^{i-1}\tilde{l}_{ij}\tilde{u}_{jj} + \sum_{j=1}^{i-1}\tilde{l}_{ij}\sum_{k=j+1}^{n}\tilde{u}_{jk} + \sum_{k=i}^{n}\tilde{u}_{ik} = \\
&= \sum_{j=1}^{i-1}\left(a_{ij} - \sum_{k=1}^{j-1}\tilde{l}_{ik}\tilde{u}_{kj}\right) + \sum_{j=1}^{i-1}\tilde{l}_{ij}\sum_{k=j+1}^{n}\tilde{u}_{jk} + \sum_{k=i}^{n}\left(a_{ik} - \sum_{j=1}^{i-1}\tilde{l}_{ij}\tilde{u}_{jk}\right) \\
&= \sum_{j=1}^{n}a_{ij} + \sum_{j=1}^{i-1}\tilde{l}_{ij}\sum_{k=j+1}^{n}\tilde{u}_{jk} - \left(\sum_{j=1}^{i-1}\sum_{k=1}^{j-1}\tilde{l}_{ik}\tilde{u}_{kj} + \sum_{k=i}^{n}\sum_{j=1}^{i-1}\tilde{l}_{ij}\tilde{u}_{jk}\right).
\end{aligned}$$

Incomplete factorizations

Modifications based on maintaining row sums

- Rearranging the indices in the double summations, the last three sums cancel out.
- Moreover, the added double summation is the sum of all the modification terms $\tilde{l}_{ik}\,\tilde{u}_{kj}$ in the MILU Algorithm, and the sum of the two subtracted double summations also comprises all the modification terms.
- Consequently, the row sums of A are preserved in the product of the incomplete factors.

Incomplete factorizations

Modifications based on maintaining row sums

- Theorem above provides motivation for maintaining constant row sums in the case of a model PDE problem.
- The result is also valid for Neumann or mixed boundary conditions, and there are extensions to three-dimensional problems and MIC(ℓ) with $\ell > 0$. However, although Theorem holds for MILU factorizations, the approach may not be useful for general A.

Theorem

Let A come from a discretized Poisson problem on a uniform two-dimensional rectangular grid with Dirichlet boundary conditions and discretization parameter h. Then the condition number $\kappa((\widetilde{L}\widetilde{U})^{-1}A)$ for the level-based MIC(0) preconditioner is $O(h^{-1})$.

Incomplete factorizations

Modifications based on maintaining row sums

- RILU/RIC: the update term $\tilde{l}_{ik}\,\tilde{u}_{kj}$ may be multiplied by a parameter θ $(0 < \theta < 1)$ before it is subtracted from the diagonal entry $\tilde{u}_{ii}$.
- This is a practical way to extend MILU to linear systems not coming from discretized PDEs. Clearly, using $\theta < 1$ reduces the amount by which the diagonal entries are modified.

Incomplete factorizations

Dynamic compensation

- Instead of accepting a filled entry in position (i, j), the idea is to add a (weighted) multiple of its absolute value to the corresponding diagonal entries $\tilde{u}_{ii}$ and $\tilde{u}_{jj}$.
- Provided the number of modifications is small, this can be useful if A is diagonally dominant and scaled so that its diagonal entries are nonnegative.
- The parameter ω controls the amount by which the diagonal entries of $\widetilde{U}$ are modified but if $\omega < 1$ then breakdown can still occur.
- Dynamic compensation can be successful when incorporated into an IC factorization of a SPD matrix A because the resulting local modifications correspond to adding positive semidefinite matrices to A.
- In practice, the behaviour of the resulting preconditioner can be very different from that computed using the MIC approach.

Incomplete factorizations

Algorithm (ILU factorization with dynamic compensation)

```
$\tilde{l}_{ij} = (I + L_A)_{ij}$ for all $(i,j) \in \mathcal{S}(\widetilde{L})$
$\tilde{u}_{ij} = (D_A + U_A)_{ij}$ for all $(i,j) \in \mathcal{S}(\widetilde{U})$
for $k = 1 : n-1$ do
    for $i = k+1 : n$ such that $(i,k) \in \mathcal{S}\{\widetilde{L}\}$ do
        $\tilde{l}_{ik} = \tilde{l}_{ik} / \tilde{u}_{kk}$
        for $j = i : n$ such that $(k,j) \in \mathcal{S}\{\widetilde{U}\}$ do
            if $(i,j) \in \mathcal{S}\{\widetilde{U}\}$ then
                $\tilde{u}_{ij} = \tilde{u}_{ij} - \tilde{l}_{ik}\, \tilde{u}_{kj}$
            else
                $\rho_{ij} = (\tilde{u}_{ii} / \tilde{u}_{jj})^{1/2}$
                $\tilde{u}_{ii} = \tilde{u}_{ii} + \omega \rho_{ij}\, |\tilde{l}_{ik}\, \tilde{u}_{kj}|, \;\; \tilde{u}_{jj} = \tilde{u}_{jj} + \omega |\tilde{l}_{ik}\, \tilde{u}_{kj}| / \rho_{ij}, \;\; \tilde{u}_{ij} = 0.$
            end if
        end for
        for $j = k+1 : i-1$ such that $(k,j) \in \mathcal{S}\{\widetilde{U}\}$ do
            if $(i,j) \in \mathcal{S}\{\widetilde{L}\}$ then
                $\tilde{l}_{ij} = \tilde{l}_{ij} - \tilde{l}_{ik}\, \tilde{u}_{kj}$
            else
                $\rho_{ij} = (\tilde{u}_{ii} / \tilde{u}_{jj})^{1/2}$
                $\tilde{u}_{ii} = \tilde{u}_{ii} + \omega \rho_{ij}\, |\tilde{l}_{ik}\, \tilde{u}_{kj}|, \; \tilde{u}_{jj} = \tilde{u}_{jj} + \omega |\tilde{l}_{ik}\, \tilde{u}_{kj}| / \rho_{ij}, \;\; \tilde{l}_{ij} = 0.$
            end if
```

Incomplete factorizations

Dynamic compensation: getting closer to special matrices

- A related scheme, called diagonally compensated reduction, modifies A before the factorization begins by adding the values of all of its positive off-diagonal entries to the corresponding diagonal entries and then setting these off-diagonal entries to zero.
- If A is SPD then the resulting matrix is a symmetric M-matrix and the incomplete factorization of an M-matrix is breakdown-free. However, the modified matrix may be too far from A for its incomplete factors to be useful.

Incomplete factorizations

Memory-limited incomplete factorizations

- A be SPD, consider the decomposition

$$A = (\widetilde{L} + \widetilde{R})\,(\widetilde{L} + \widetilde{R})^T - E.$$

- The error matrix E is $E = \widetilde{R}\widetilde{R}^T$.
- On step j of the incomplete factorization, the first column of the Schur complement $S^{(j)}$ is split into the sum

$$\widetilde{L}_{j:n,j} + \widetilde{R}_{j:n,j},$$

  where $\widetilde{L}_{j:n,j}$ contains the entries that are retained in column j of the final incomplete factorization and $\widetilde{R}_{jj} = 0$ and $\widetilde{R}_{j+1:n,j}$ contains the entries that are discarded.

Incomplete factorizations

Memory-limited incomplete factorizations

- If a complete factorization was being computed then the Schur complement would be updated by subtracting

$$(\widetilde{L}_{j+1:n,j} + \widetilde{R}_{j+1:n,j})\,(\widetilde{L}_{j+1:n,j} + \widetilde{R}_{j+1:n,j})^T.$$

- However, the incomplete factorization discards the term

$$E^{(j)} = \widetilde{R}_{j+1:n,j}\,\widetilde{R}^T_{j+1:n,j}.$$

  Thus, $E^{(j)}$ is implicitly added to A and because $E^{(j)}$ is positive semidefinite, the approach is naturally breakdown-free.

- The obvious choice for $\widetilde{R}_{j+1:n,j}$ is the smallest off-diagonal entries in the column.

Incomplete factorizations

Memory-limited incomplete factorizations

- Figure depicts the first step of this approach. In the first row and column, $*$ and δ denote the entries of $\widetilde{L}_{1:n,1}$ and $\widetilde{R}_{1:n,1}$, respectively.
- Standard sparsification scheme: no fill (left)
- Using intermediate memory: right.

$$\begin{pmatrix} * & * & * & \delta & \delta \\ * & f & f & & \\ * & f & f & & \\ \delta & & & & \\ \delta & & & & \end{pmatrix} \quad \begin{pmatrix} * & * & * & \delta & \delta \\ * & f & f & f & f \\ * & f & f & f & f \\ \delta & f & f & & \\ \delta & f & f & & \end{pmatrix}$$

Figure: An illustration of the fill-in in a standard sparsification-based IC factorization (left) and in the approach that uses intermediate memory (right) after one step of the factorization. Entries with small absolute value in row and column 1 are denoted by δ. The filled entries are denoted by f.

Incomplete factorizations

Memory-limited incomplete factorizations

- Enables the structure of the complete factorization to be followed more closely than is possible using a standard approach. If the small entries at positions $(1,3)$ and $(3,1)$ are not discarded then there is a filled entry in position $(3,2)$ and this allows the incomplete factorization using intermediate memory to involve the (large) off-diagonal entries in positions $(5,2)$ and $(6,2)$ in the second step of the IC factorization.

$$\begin{pmatrix} * & * & \delta & & & \\ * & * & & & * & * \\ \delta & & * & * & & \\ & & * & * & & \\ & * & & & * & * \\ & * & & & * & * \end{pmatrix} \begin{pmatrix} * & & & & & \\ * & * & & & & \\ & & * & & & \\ & & * & * & & \\ & * & & & * & \\ & * & & & * & * \end{pmatrix} \begin{pmatrix} * & & & & & \\ * & * & & & & \\ \delta & f & * & & & \\ & & * & * & & \\ & * & & & * & \\ & * & & & * & * \end{pmatrix}$$

Incomplete factorizations

Memory-limited incomplete factorizations

- Unfortunately, because the column $\widetilde{R}_{j+1:n,j}$ must be retained to perform the updates on the next step, the total memory requirements are essentially as for a complete factorization.
- Relaxations are needed: e.g., introducing two drop tolerances so that only entries of absolute value at least τ_1 are kept in $\widetilde{L}$ and entries smaller than τ_2 are dropped from $\widetilde{R}$.
- Or, limiting the fill-in.
- But, then, no longer breakdown-free approach.

Incomplete factorizations

Algorithm (Crout memory-limited IC factorization)

```
$w(1:n) = 0$
for $j = 1:n$ do
    for $i = j:n$ such that $a_{ij} \neq 0$ do
        $w_i = a_{ij}$                                ▷ $w$ is a vector of length $n$
    end for
    for $k < j$ such that $\tilde{l}_{jk} \neq 0$ do
        for $i = j:n$ such that $\tilde{l}_{ik} \neq 0$ do
            $w_i = w_i - \tilde{l}_{ik}\,\tilde{l}_{jk}$
        end for
        for $i = j:n$ such that $\tilde{r}_{ik} \neq 0$ do
            $w_i = w_i - \tilde{r}_{ik}\,\tilde{l}_{jk}$
        end for
    end for
    for $k < j$ such that $\tilde{r}_{jk} \neq 0$ do
        for $i = j:n$ such that $\tilde{l}_{ik} \neq 0$ do
            $w_i = w_i - \tilde{l}_{ik}\,\tilde{r}_{jk}$
        end for
    end for
    Copy into $\widetilde{L}_{j:n,j}$ the $lsize + nz(A_{j:n,j})$ entries of $w$ of largest absolute value
    Copy into $\widetilde{R}$ the entries of $w$ that are the next largest in absolute value
```

Incomplete factorizations

Fixed-point iterations for computing ILU factorizations

- Given the target sparsity pattern $\mathcal{S}\{\widetilde{L}+\widetilde{U}\}$, the goal is to iteratively generate incomplete factors fulfilling the ILU property

$$(\widetilde{L}\widetilde{U})_{ij} = a_{ij}, \quad (i,j) \in \mathcal{S}\{\widetilde{L}+\widetilde{U}\}$$

- Parallel computation using the constraints

$$\sum_{\substack{k=1 \\ (i,k),(k,j)\in\mathcal{S}\{\widetilde{L}+\widetilde{U}\}}}^{\min(i,j)} \tilde{l}_{ik}\,\tilde{u}_{kj} = a_{ij}, \quad (i,j) \in \mathcal{S}\{\widetilde{L}+\widetilde{U}\},$$

and the normalization $\tilde{l}_{ii} = 1$.

Incomplete factorizations

Fixed-point iterations for computing ILU factorizations

- Using the relations

$$\tilde{l}_{ij} = \left(a_{ij} - \sum_{k=1}^{j-1} \tilde{l}_{ik}\, \tilde{u}_{kj}\right) / \, \tilde{u}_{jj}, \quad i > j, \tag{64}$$

$$\tilde{u}_{ij} = a_{ij} - \sum_{k=1}^{i-1} \tilde{l}_{ik}\, \tilde{u}_{kj}, \quad i \leq j, \tag{65}$$

the approach can be formulated as a fixed-point iteration method of the form $w^{k+1} = f(w^k)$, $k = 0, 1, \ldots$, where w is a vector containing the unknowns $\tilde{l}_{ij}$ and $\tilde{u}_{ij}$. Each fixed-point iteration is called a sweep.

Incomplete factorizations

Fixed-point iterations for computing ILU factorizations

Algorithm (Fixed-point ILU factorization)

Input: *Matrix A, the target sparsity pattern $\mathcal{S}\{\widetilde{L}+\widetilde{U}\}$, and initial incomplete factors $\widetilde{L}$ and $\widetilde{U}$.*
Output: *Updated incomplete factors.*

```
Set $\tilde{l}_{ij}$ and $\tilde{u}_{ij}$ to initial values
for $sweep = 1, 2, \ldots$ do
    for $(i,j) \in \mathcal{S}\{\widetilde{L}+\widetilde{U}\}$ do
        if $i > j$ then
            Compute $\tilde{l}_{ij}$ using (64)
        else
            Compute $\tilde{u}_{ij}$ using (65)
        end if
    end for
end for
```

Incomplete factorizations

Ordering in incomplete factorizations

- Can have a positive effect on the robustness and performance of preconditioned Krylov subspace methods.
- The best choice of ordering for an incomplete factorization preconditioner may not be the same as for a complete factorization.
- When the natural (lexicographic) ordering is used, the incomplete triangular factors resulting from a no-fill ILU factorization can be highly ill-conditioned, even if the matrix A is well conditioned.
- Allowing more fill-in in the factors, for example, using ILU(1) instead of ILU(0), may solve the problem but it is not guaranteed.
- Minimum degree orderings: the rows (and columns) of the permuted matrix can have significantly different counts.
- A strategy is to specify that the permitted fill-in is proportional to the row/column counts of the complete factorization.

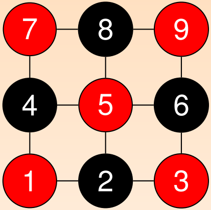
Incomplete factorizations

Ordering in incomplete factorizations

- Global orderings cut local connections within the graph of A.
- When used with incomplete factorizations, can lead to poor quality preconditioners.
- A global ordering that specifically targets incomplete factorizations is a red-black (or checker board) ordering.

Incomplete factorizations

Ordering in incomplete factorizations



$$
\begin{array}{c|ccccccccc}
 & 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 \\
\hline
1 & 4 & -1 & & -1 & & & & & \\
2 & -1 & 4 & -1 & & -1 & & & & \\
3 & & -1 & 4 & & & -1 & & & \\
4 & -1 & & & 4 & & & -1 & & \\
5 & & -1 & & -1 & 4 & -1 & & -1 & \\
6 & & & -1 & & -1 & 4 & & & -1 \\
7 & & & & -1 & & & 4 & -1 & \\
8 & & & & & -1 & & -1 & 4 & -1 \\
9 & & & & & & -1 & & -1 & 4
\end{array}
\qquad
\begin{array}{c|ccccccccc}
 & 1 & 3 & 5 & 7 & 9 & 2 & 4 & 6 & 8 \\
\hline
1 & 4 & & & & & -1 & -1 & & \\
3 & & 4 & & & & -1 & -1 & & \\
5 & & & 4 & & & -1 & -1 & -1 & -1 \\
7 & & & & 4 & & & -1 & & -1 \\
9 & & & & & 4 & & & -1 & -1 \\
2 & -1 & -1 & -1 & & & 4 & & & \\
4 & -1 & & -1 & -1 & & & 4 & & \\
6 & & -1 & -1 & & -1 & & & 4 & \\
8 & & & -1 & -1 & -1 & & & & 4
\end{array}
$$

Figure: *A model problem to illustrate a red-black ordering.*

Incomplete factorizations

Exploiting block structure

- Blocking methods for complete factorizations can be adapted to incomplete factorizations. The aim is to speed up the computation of the factors and to obtain more effective preconditioners.
- In a block factorization, scalar operations of the form

  $$\tilde{l}_{ik} = a_{ik}/\tilde{u}_{kk}$$

  are replaced by matrix operations

  $$\widetilde{L}_{ib,kb} = A_{ib,kb}\widetilde{U}_{kb,kb}^{-1},$$

  and scalar multiplications of entries of the factors are replaced by matrix-matrix products. When dropping entries, instead of considering the absolute values, simple norms of the block entries (such as the one-norm, max-norm or Frobenius norm) are used.
- An incomplete factorization can start with the supernodal structure of the complete factors.

Outline

1. Introduction
2. Introductory notation and terminology
3. Factorizations
4. Graphs and sparse matrices
5. Sparse matrices and data structures
6. Symbolic factorizations
7. Symbolic Cholesky factorization
8. Cholesky factorization
9. Sparse LU factorization
10. Stability, ill-conditioning, indefiniteness
11. Blocks in the input matrix
12. Sparse Least Squares and QR factorization
13. Reorderings
14. Algebraic preconditioning
15. Incomplete factorizations
16. Sparse approximate inverses

Sparse approximate inverses

Approximate inverse preconditioners

- Standard solves by substitution steps can present a computational bottleneck. In particular, in parallel computational environment.
- But it is M^{-1}, which represents an approximation of A^{-1}, that is applied by performing forward and back substitution steps
- Therefore, an alternative strategy to standard (incomplete) factorizations is to directly approximate A^{-1} by explicitly computing M^{-1}.

Sparse approximate inverses

Approximate inverse preconditioners

- But, there is a problem: The sparsity pattern of the inverse of an irreducible matrix A is dense, even when A is sparse.
- But, perhaps there is a way ...: although A^{-1} is fully dense, the following result shows this is not the case for the factors of factorized inverses.

Theorem

Assume the matrix A is SPD and let L be its Cholesky factor. Then $\mathcal{S}\{L^{-1}\}$ is the union of all entries (i, j) such that i is an ancestor of j in the elimination tree $\mathcal{T}(A)$.

- A consequence of this result is that L^{-1} need not be fully dense.
- Algorithmically, if A is SPD it may be advantageous to preorder A to limit the number of ancestors of vertices in $\mathcal{T}(A)$.
- For example, by ND applied to $\mathcal{S}\{A\}$ or to $\mathcal{S}\{A + A^T\}$.

Sparse approximate inverses

Basic approaches

- An obvious way: to compute an incomplete LU factorization of A and then perform an approximate inversion of the incomplete factors.
- But, two levels of approximation.

Sparse approximate inverses

Basic approaches

- Another straightforward approach is based on bordering.
- Let $A_j = A_{1:j,1:j}$ and its inverse factorization

  $$A_j^{-1} = W_j D_j^{-1} Z_j^T$$

  is known (W_j and Z_j are unit upper triangular matrices and D_j is a diagonal matrix).

  $$\begin{pmatrix} Z_j^T & 0 \\ z_{j+1}^T & 1 \end{pmatrix} \begin{pmatrix} A_j & A_{1:j,j+1} \\ A_{j+1,1:j} & a_{j+1,j+1} \end{pmatrix} \begin{pmatrix} W_j & w_{j+1} \\ 0 & 1 \end{pmatrix} = \begin{pmatrix} D_j & 0 \\ 0 & d_{j+1,j+1} \end{pmatrix},$$

  where for $1 \le j < n$

  $$w_{j+1} = -W_j D_j^{-1} Z_j^T A_{1:j,j+1}, z_{j+1} = -Z_j D_j^{-1} W_j^T A_{j+1,1:j}^T,$$

  $$d_{j+1,j+1} = a_{j+1,j+1} + z_{j+1}^T A_j w_{j+1} + A_{j+1,1:j} w_{j+1} + z_{j+1}^T A_{1:j,j+1}.$$

- Starting from $j = 1$, this suggests a procedure for computing the inverse factors of A. Sparsity can be preserved by dropping.

Approximate factorizations

Approximate inverse by bordering

Algorithm (Nonsymmetric inverse bordering algorithm)

Input: *Generally nonsymmetric A.*
Output: *A unit upper triangular matrix $\widetilde{Z}$ and diagonal matrix $\widetilde{D}$ such that $A^{-1} \approx \widetilde{Z}^T \widetilde{D}^{-1} \widetilde{W}^T$.*

```
Set $\widetilde{Z}_1 = (1)$, $\widetilde{W}_1 = (1)$, $\widetilde{D}_1 = (a_{11})$.
for $j = 2 : n$ do
    Set $\tilde{z}_j = -\widetilde{Z}_{1:j-1,1:j-1} \widetilde{D}^{-1}_{1:j-1,1:j-1} \widetilde{W}^T_{1:j-1,1:j-1} A_{1:j-1,j}$
    Set $\tilde{w}_j = -\widetilde{W}_{1:j-1,1:j-1} \widetilde{D}^{-1}_{1:j-1,1:j-1} \widetilde{Z}^T_{1:j-1,1:j-1} A^T_{j,1:j-1}$
    Set $\tilde{d}_j = A_{jj} + A^T_{1:j-1,j} \tilde{w}_j + A_{j,1:j-1} \tilde{z}_j + \tilde{w}_j^T A_{j-1} \tilde{z}_j$
    Set $\widetilde{Z}_j = \begin{pmatrix} \widetilde{Z}_{1:j-1,1:j-1} & \tilde{z}_j \\ 0 & 1 \end{pmatrix}$
    Set $\widetilde{W}_j = \begin{pmatrix} \widetilde{W}_{1:j-1,1:j-1} & \tilde{w}_j \\ 0 & 1 \end{pmatrix}$
end for
Set $\widetilde{Z} = \widetilde{Z}_n, \widetilde{W} = \widetilde{W}_n, \widetilde{D} = \widetilde{D}_n$
```

Sparse approximate inverses

Inverse by bordering: notes

- If A is symmetric, $W = Z$ and the required work is halved.
- Furthermore, if A is SPD then it can be shown that, in exact arithmetic, $d_{jj} > 0$ for all j and the process does not break down.
- The computation of Z and W are tightly coupled restricting the potential to exploit parallelism.
- This implies a potential problem with efficient implementation.

Sparse approximate inverses

Frobenius norm minimization: SPAI

- Denote $K = M^{-1}$.
- Use minimization of

$$\|I - AM^{-1}\|_F^2 = \|I - AK\|_F^2 = \sum_{j=1}^{n} \|e_j - Ak_j\|_2^2, \qquad (66)$$

  over all K with pattern $\mathcal{S}$.
- A left approximate inverse can be computed by solving a minimization problem for $\|I - KA\|_F = \|I - A^T K^T\|_F$.
- The problem reduces to least squares problems for the columns of K that can be computed independently and, if required, in parallel.

Sparse approximate inverses

Frobenius norm minimization: SPAI

- These least squares (LS) problems are all of small dimension when $\mathcal{S}$ is chosen to ensure K is sparse.
- Let $\mathcal{J} = \{i \mid k_j(i) \neq 0\}$ be the set of indices of the nonzero entries in column k_j. Further, denote $\mathcal{I} = \{m \mid A_{m,\mathcal{J}} \neq 0\}$.
- Let $\widehat{e}_j = e_j(\mathcal{I})$ be the vector of length $|\mathcal{I}|$ that is obtained by taking the entries of e_j with row indices belonging to $\mathcal{I}$.
- To solve the LS problem for k_j, construct the $|\mathcal{I}| \times |\mathcal{J}|$ matrix $\widehat{A} = A_{\mathcal{I},\mathcal{J}}$ and solve

$$\min_{\widehat{k}_j} \|\widehat{e}_j - \widehat{A}\,\widehat{k}_j\|_2^2. \tag{67}$$

- This can be done using QR factorization of $\widehat{A}$. Extending $\widehat{k}_j$ to have length n by zeros gives k_j.

Sparse approximate inverses

SPAI algorithm

- Construction: starting with a chosen column sparsity pattern $\mathcal{J}$ for k_j, construct $\widehat{A}$, solve (67) for $\widehat{k}_j$, set $k_j(\mathcal{J}) = \widehat{k}_j$ and define the residual vector
$$r_j = e_j - A_{1:n,\mathcal{J}}\widehat{k}_j.$$
- If $\|r_j\|_2 \neq 0$ then k_j is not equal to the j-th column of A^{-1} and a better approximation can be derived by augmenting $\mathcal{J}$.
- Augmentation: let $\mathcal{L} = \{l \,|\, r_j(l) \neq 0\}$ and define
$$\widetilde{\mathcal{J}} = \{i \,|\, A_{\mathcal{L},i} \neq 0\} \setminus \mathcal{J}. \tag{68}$$
- One or more candidate indices that can be added to $\mathcal{J}$ can be chosen. For example, such that most effectively reduce $\|r_j\|_2$.

Sparse approximate inverses

SPAI algorithm

- A possible heuristic is to solve for each $i \in \widetilde{\mathcal{J}}$ the minimization problem

$$\min_{\mu_i} \|r_j - \mu_i A e_i\|_2^2.$$

- This has the solution $\mu_i = r_j^T A e_i / \|A e_i\|_2^2$ with residual $\|r_j\|^2 - (r_j^T A e_i)^2 / \|A e_i\|_2^2$.
- Indices $i \in \widetilde{\mathcal{J}}$ for which this is small are appended to $\mathcal{J}$.
- The process can be repeated until either the required accuracy is attained or the maximum number of allowed entries in $\mathcal{J}$ is reached.

Sparse approximate inverses

SPAI algorithm

- Solving the unconstrained LS problem after extending $\widehat{A}$ to $A_{\mathcal{I}\cup\mathcal{I}',\,\mathcal{J}\cup\mathcal{J}'}$ is typically performed by updating the previous problems.
- Assume the QR factorization of $\widehat{A}$ is

$$\widehat{A} = A_{\mathcal{I},\mathcal{J}} = Q \begin{pmatrix} R \\ 0 \end{pmatrix} = \begin{pmatrix} Q_1 & Q_2 \end{pmatrix} \begin{pmatrix} R \\ 0 \end{pmatrix},$$

  where Q_1 is $|\mathcal{I}| \times |\mathcal{J}|$.
- The QR factorization of the extended matrix is

$$\begin{aligned} A_{\mathcal{I}\cup\mathcal{I}',\,\mathcal{J}\cup\mathcal{J}'} &= \begin{pmatrix} \widehat{A} & A_{\mathcal{I},\,\mathcal{J}'} \\ & A_{\mathcal{I}',\,\mathcal{J}'} \end{pmatrix} = \begin{pmatrix} Q & \\ & I \end{pmatrix} \begin{pmatrix} R & Q_1^T A_{\mathcal{I},\,\mathcal{J}'} \\ & Q_2^T A_{\mathcal{I},\,\mathcal{J}'} \\ & A_{\mathcal{I}',\,\mathcal{J}'} \end{pmatrix} \\ &= \begin{pmatrix} Q & \\ & I \end{pmatrix} \begin{pmatrix} I & \\ & Q' \end{pmatrix} \begin{pmatrix} R & Q_1^T A_{\mathcal{I},\,\mathcal{J}'} \\ & R' \\ & 0 \end{pmatrix}. \end{aligned}$$

Sparse approximate inverses

SPAI algorithm

- Q' and R' are from the QR factorization of the $(|\mathcal{I}'| + |\mathcal{I}| - |\mathcal{J}|) \times |\mathcal{J}'|$ matrix

$$\begin{pmatrix} Q_2^T A_{\mathcal{I},\mathcal{J}'} \\ A_{\mathcal{I}',\mathcal{J}'} \end{pmatrix}.$$

- Factorizing this matrix and updating the trailing QR factorization to get the new $\widehat{k}_j$ is much more efficient than computing the QR factorization of the extended matrix from scratch.
- Many variations of the basic approach.

Sparse approximate inverses

SPAI algorithm

Algorithm (**SPAI preconditioner (right-looking approach)**)

Input: *Nonsymmetric matrix A, a convergence tolerance $\eta > 0$, an initial sparsity pattern $\mathcal{J}_j$ and the maximum number nz_j of permitted entries for column j of K $(1 \leq j \leq n)$.*
Output: *$K \approx A^{-1}$ with columns k_j $(1 \leq j \leq n)$.*

```
for $j = 1 : n$ do                                              ▷ The columns may be computed in parallel
    Set $\mathcal{J} = \mathcal{J}_j$ and $\mathcal{I} = \{m \mid A(m, \mathcal{J}) \neq 0\}$, $\|r_j\|_2 = \infty$
    Construct $\widehat{A} = A_{\mathcal{I},\mathcal{J}}$ and solve (67) for $\widehat{k}_j$
    $r_j = e_j - A_{1:n,\mathcal{J}} \widehat{k}_j$
    while $|\mathcal{J}| < nz_j$ and $\|r_j\|_2 > \eta$ do
        Construct $\widetilde{\mathcal{J}}$ given by (68)                          ▷ $\widetilde{\mathcal{J}}$ is the candidate set
        Determine new indices $\mathcal{J}' \subset \widetilde{\mathcal{J}}$ to add to $\mathcal{J}$
        $\mathcal{I}' = \{m \mid A_{m,\mathcal{J}'} \neq 0\} \setminus \mathcal{I}$
        $\mathcal{I} = \mathcal{I} \cup \mathcal{I}'$ and $\mathcal{J} = \mathcal{J} \cup \mathcal{J}'$                      ▷ Augment the sparsity pattern
        Construct new $\widehat{A} = A_{\mathcal{I},\mathcal{J}}$ and new $\widehat{k}_j$             ▷ Update the QR factorization
        $r_j = e_j - A_{1:n,\mathcal{J}} \widehat{k}_j$
    end while
    $k_j(\mathcal{J}) = \widehat{k}_j$                ▷ Extend $\widehat{k}_j$ to $k_j$ by setting entries not in $\mathcal{J}$ to zero.
end for
```

SPAI algorithm

The example: The algorithm starts with $\mathcal{J}_1 = \{1, 2\}$.

$$A = \begin{pmatrix} 10 & -2 & & & \\ -1 & 10 & -2 & & \\ & -1 & 10 & -2 & \\ & & -1 & 10 & -2 \\ & & & -1 & 10 \end{pmatrix}, \widehat{A} = \begin{pmatrix} 10 & -2 \\ -1 & 10 \\ & -1 \end{pmatrix}, \widehat{k}_1 = \begin{pmatrix} 0.1020 \\ 0.0101 \end{pmatrix}, r_1 = \begin{pmatrix} 1.00 \times 10^{-4} \\ 1.00 \times 10^{-3} \\ 1.01 \times 10^{-2} \\ 0 \\ 0 \end{pmatrix}.$$

$$\widehat{A} = \begin{pmatrix} 10 & -2 & \\ -1 & 10 & -2 \\ & -1 & 10 \\ & & -1 \end{pmatrix}, \widehat{k}_1 = \begin{pmatrix} 0.1021 \\ 0.0104 \\ 0.0010 \end{pmatrix}, r_1 = \begin{pmatrix} 1.0 \times 10^{-5} \\ 1.1 \times 10^{-4} \\ 1.1 \times 10^{-3} \\ 1.0 \times 10^{-2} \\ 0 \end{pmatrix}, k_1 = \begin{pmatrix} 0.1021 \\ 0.0104 \\ 0.0010 \\ 0 \\ 0 \end{pmatrix}.$$

Figure: An illustration of computing the first column of a sparse approximate inverse using the SPAI algorithm with $nz_1 = 3$. On the top line is the initial tridiagonal matrix A followed by the matrix $\widehat{A}$ and the vectors $\widehat{k}_1$ and r_1 on the first loop of Algorithm. The bottom line presents the updated matrix $\hat{A}$ that is obtained on the second loop by adding the third row and column of A and the corresponding vectors $\widehat{k}_1$ and r_1 and, finally, k_1.

Sparse approximate inverses

SPAI algorithm

- When A is symmetric, there is no guarantee that the computed K will be symmetric. One possibility is to use $(K + K^T)/2$ to approximate A^{-1}.
- The SPAI preconditioner is not sensitive to reorderings of A. This has the advantage that A can be partitioned and reordered in whatever way is convenient, for instance to better suit the needs of a distributed implementation.
- The disadvantage is that orderings cannot be used to reduce fill-in and/or improve the quality of the approximate inverse.
- For instance, if A^{-1} has no small entries, SPAI will not find a sparse K, and because the inverse of a permutation of A is just a permutation of A^{-1}, no permutation of A will change this.

Sparse approximate inverses

FSAI preconditioner: SPD case

- The factorized sparse approximate inverse (FSAI) preconditioner for an SPD matrix A is defined as the product

  $$M^{-1} = G^T G,$$

  where the sparse lower triangular matrix G is an approximation of the inverse of the (complete) Cholesky factor L of A.
- Theoretically, a FSAI preconditioner is computed by choosing a lower triangular sparsity pattern $\mathcal{S}_L$ and minimizing

  $$\|I - GL\|_F^2 = tr\left[(I - GL)^T (I - GL)\right] \tag{69}$$

  over all G with sparsity pattern $\mathcal{S}_L$.

Sparse approximate inverses

FSAI preconditioner: SPD case

- Differentiating the formula with respect to the entries of G and setting to zero yields

$$(GLL^T)_{ij} = (GA)_{ij} = (L^T)_{ij} \quad \text{for all} \quad (i,j) \in \mathcal{S}_L. \tag{70}$$

- Because L^T is an upper triangular matrix while $\mathcal{S}_L$ is a lower triangular pattern, the matrix equation (70) can be rewritten as

$$(GA)_{ij} = \begin{cases} 0 & i \neq j, \quad (i,j) \in \mathcal{S}_L \\ l_{ii} & i = j. \end{cases} \tag{71}$$

Sparse approximate inverses

FSAI preconditioner: SPD case

- G is not available directly because L is unknown. Instead, $\overline{G}$ is computed such that

$$(\overline{G}A)_{ij} = \delta_{i,j} \quad \text{for all} \quad (i,j) \in \mathcal{S}_L, \tag{72}$$

where $\delta_{i,j}$ is the Kronecker delta function ($\delta_{i,j} = 1$ if $i = j$ and is equal to 0, otherwise).

- The FSAI factor G is then obtained by setting

$$G = D\overline{G},$$

where D is a diagonal scaling matrix.

- An appropriate choice for D is

$$D = [diag(\overline{G})]^{-1/2}, \tag{73}$$

so that

$$(GAG^T)_{ii} = 1, \quad 1 \leq i \leq n.$$

Sparse approximate inverses

FSAI preconditioner: SPD case

Theorem

Assume A is SPD. If the lower triangular sparsity pattern $\mathcal{S}_L$ includes all diagonal positions then G exists and is unique.

Proof.

Set $\mathcal{I}_i = \{j \mid (i,j) \in \mathcal{S}_L\}$ and let $A_{\mathcal{I}_i,\mathcal{I}_i}$ denote the submatrix of order $nz_i = |\mathcal{I}_i|$ of entries a_{kl} such that $k, l \in \mathcal{I}_i$. Let $\bar{g}_i$ and g_i be dense vectors containing the nonzero coefficients in row i of $\overline{G}$ and G, respectively. Using this notation, solving (72) decouples into solving n independent SPD linear systems

$$A_{\mathcal{I}_i,\mathcal{I}_i}\,\bar{g}_i = e_{nz_i}, \quad 1 \le i \le n,$$

where the unit vectors are of length nz_i. Moreover,

$$(\overline{G} A \overline{G}^T)_{ii} = \sum_{j \in \mathcal{I}_i} \delta_{i,j} \overline{G}_{ij} = \overline{G}_{ii} = (A^{-1}_{\mathcal{I}_i,\mathcal{I}_i})_{ii}.$$

This implies that the diagonal entries of D given by (73) are nonzero. Consequently, the computed rows of G exist and provide a unique solution. □ □

Sparse approximate inverses

FSAI preconditioner: SPD case

Algorithm (**FSAI preconditioner**)

Input: *SPD matrix A and lower triangular sparsity pattern $\mathcal{S}_L$ that includes all diagonal positions.*
Output: *Lower triangular matrix G such that $A^{-1} \approx GG^T$.*

```
for $i = 1 : n$ do
    Construct $\mathcal{I}_i = \{j \mid (i,j) \in \mathcal{S}_L\},\ A_{\mathcal{I}_i,\mathcal{I}_i}$ and set $nz_i = |\mathcal{I}_i|$
    Solve $A_{\mathcal{I}_i,\mathcal{I}_i}\,\bar{g}_i = e_{nz_i}$
    Scale $g_i = d_{ii}\bar{g}_i$ with $d_{ii} = (\bar{g}_{i,nz_i})^{-1/2}$        ▷ $\bar{g}_{i,nz_i}$ is the last component of $\bar{g}_i$
    Extend $g_i$ to the row $G_{i,1:i}$ by setting entries that are not in $\mathcal{I}_i$ to zero
end for
```

Sparse approximate inverses

FSAI preconditioner: SPD case

- Monotonicity property.

Theorem

Let L be the Cholesky factor of the SPD matrix A. Given the lower triangular sparsity pattern $\mathcal{S}_L$ that includes all diagonal positions, let G be the FSAI preconditioner computed using Algorithm above. Then any lower triangular matrix G_1 with its sparsity pattern is contained in $\mathcal{S}_L$ a $(G_1 A G_1^T)_{ii} = 1$ ($1 \leq i \leq n$) satisfies

$$||I - GL||_F \leq ||I - G_1 L||_F.$$

Sparse approximate inverses

FSAI preconditioner: SPD case

- The performance is highly dependent on the choice of $\mathcal{S}_L$.

Theorem

Let L be the Cholesky factor of the SPD matrix A. Given the lower triangular sparsity patterns $\mathcal{S}_{L1}$ and $\mathcal{S}_{L2}$ that include all diagonal positions, let the corresponding FSAI preconditioners computed using Algorithm 16.3 be G_1 and G_2, respectively. If $\mathcal{S}_{L1} \subseteq \mathcal{S}_{L2}$ then

$$||I - G_2 L||_F \leq ||I - G_1 L||_F .$$

FSAI preconditioner: general case

- The FSAI algorithm can be extended to a general matrix A. Two input sparsity patterns are required.
- First, lower and upper triangular matrices $\overline{G}_L$ and $\overline{G}_U$ are computed such that

$$(\overline{G}_L A)_{ij} = \delta_{i,j} \quad \text{for all} \quad (i,j) \in \mathcal{S}_L,$$
$$(A\overline{G}_U)_{ij} = \delta_{i,j} \quad \text{for all} \quad (i,j) \in \mathcal{S}_U.$$

- Then D is obtained as the inverse of the diagonal of the matrix $\overline{G}_L A \overline{G}_U$, and the final nonsymmetric FSAI factors are given by $G_L = \overline{G}_L$ and $G_U = \overline{G}_U D$. The computation of the two approximate factors can be performed independently.
- This generalization is well defined if, for example, A is nonsymmetric positive definite. There is also theory that extends existence to special classes of matrices, including M- and H-matrices.

Sparse approximate inverses

Determining a good sparsity pattern

- Input pattern is expected to filter out entries of A^{-1} that contribute little to the quality of the preconditioner.
- For instance, it might be appropriate to ignore entries with a small absolute value, while retaining the largest ones. But, locations of large entries in A^{-1} are generally unknown, and this makes the a priori sparsity choice difficult.
- A a banded SPD matrix: the entries of A^{-1} are bounded in an exponentially decaying manner along each row or column: there exist $0 < \rho < 1$ and a constant c such that for all i, j

  $$|(A^{-1})_{ij}| \leq c\rho^{|i-j|}.$$

  The scalars ρ and c depend on the bandwidth and $\kappa(A)$.
- A common choice for a general A is $\mathcal{S}_L + \mathcal{S}_U = \mathcal{S}\{A\}$, .
- An alternative strategy uses the Neumann series expansion of A^{-1}: using the pattern of a small power of A, i.e., $\mathcal{S}\{A^2\}$ or $\mathcal{S}\{A^3\}$.

Sparse approximate inverses

Factorized approximate inverses based on incomplete conjugation

- An alternative way: using incomplete conjugation (A-orthogonalization) in the SPD case and on incomplete A-biconjugation in the general case. For SPD matrices, the approach represents an approximate Gram-Schmidt orthogonalization that uses the A-inner product $\langle ., . \rangle_A$.
- Sparsity pattern not needed in advance.
- When A is a SPD matrix the AINV preconditioner is defined in the form

  $$A^{-1} \approx M^{-1} = ZD^{-1}Z^T,$$

  Z is unit upper triangular, D is a diagonal matrix with positive entries.
- Practical implementations need to employ sparse matrix techniques. The left-looking scheme computes the j-th column z_j of Z as a sparse linear combination of the previous columns $z_1, \ldots, z_{j-1}$. The key is determining which multipliers (the α's in Steps 4 and 5 of the two algorithms, respectively) are nonzero and

Sparse approximate inverses

Factorized approximate inverses based on incomplete conjugation

Algorithm (**AINV preconditioner (left-looking approach)**)

Input: *SPD matrix A and sparsifying rule.*
Output: *$A^{-1} \approx ZD^{-1}Z^T$ with Z a unit upper triangular matrix and D a diagonal matrix with positive diagonal entries.*

```
$[z_1^{(0)}, \dots, z_n^{(0)}] = [e_1, \dots, e_n]$        ▷ Initialise $Z$ to hold the columns of the identity matrix
for $j = 1 : n$ do
    for $k = 1 : j - 1$ do
        $\alpha = A_{k,1:n}\, z_j^{(k-1)} / d_{kk}$
        $z_j^{(k)} = z_j^{(k-1)} - \alpha z_k^{(k-1)}$
        Sparsify $z_j^{(k)}$        ▷ Drop entries from $z_j^{(k)}$
    end for
    $d_{jj} = A_{j,1:n}\, z_j^{(j-1)}$
end for
$Z = [z_1^{(0)}, \dots, z_n^{(n-1)}]$
```

Sparse approximate inverses

Factorized approximate inverses based on incomplete conjugation

Algorithm (**AINV preconditioner (right-looking approach)**)

Input: *SPD matrix A and sparsifying rule.*
Output: *$A^{-1} \approx ZD^{-1}Z^T$ with Z a unit upper triangular matrix and D a diagonal matrix with positive diagonal entries.*

```
1: $[z_1^{(0)}, \dots, z_n^{(0)}] = [e_1, \dots, e_n]$        ▷ Initialise $Z$ to hold the columns of the identity matrix
2: for $j = 1 : n$ do
3:     $d_{jj} = A_{j,1:n}\, z_j^{(j-1)}$
4:     for $k = j + 1 : n$ do
5:         $\alpha = A_{j,1:n}\, z_k^{(j-1)} / d_{jj}$
6:         $z_k^{(j)} = z_k^{(j-1)} - \alpha z_j^{(j-1)}$
7:         Sparsify $z_k^{(j)}$        ▷ Drop entries from $z_k^{(j)}$
8:     end for
9: end for
10: $Z = [z_1^{(0)}, \dots, z_n^{(n-1)}]$
```

Sparse approximate inverses

AINV preconditioner: general case

- In the general case, the AINV preconditioner is given by an approximate inverse factorization of the form

  $$A^{-1} \approx M^{-1} = W D^{-1} Z^T,$$

  where Z and W are unit upper triangular matrix and D is a diagonal matrix.
- Z and W are sparse approximations of the inverses of the L^T and U factors in the LDU factorization of A, respectively.

Sparse approximate inverses

AINV preconditioner: general case

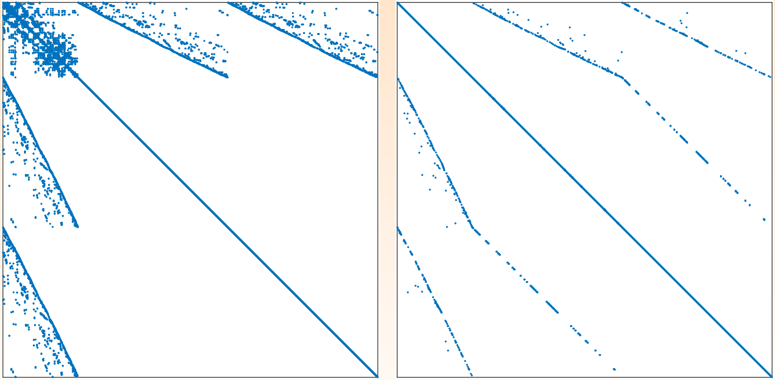
Algorithm (**Nonsymmetric AINV preconditioner (right-looking approach)**)

```
$[z_1^{(0)}, \ldots, z_n^{(0)}] = [e_1, \ldots, e_n]$ and $[w_1^{(0)}, \ldots, w_n^{(0)}] = [e_1, \ldots, e_n]$
for $j = 1 : n$ do
    $d_{jj} = (A_{1:n,j})^T z_j^{(j-1)}$ or $d_{jj} = A_{j,1:n}\, w_j^{(j-1)}$
    for $k = j + 1 : n$ do
        $\alpha = (A_{1:n,j})^T z_k^{(j-1)} / d_{jj}$
        $z_k^{(j)} = z_k^{(j-1)} - \alpha z_j^{(j-1)}$
        Sparsify $z_k^{(j)}$                              ▷ Drop entries from $z_k^{(j)}$
        $\beta = A_{j,1:n}\, w_k^{(j-1)} / d_{jj}$
        $w_k^{(j)} = w_k^{(j-1)} - \beta w_j^{(j-1)}$
        Sparsify $w_k^{(j)}$                              ▷ Drop entries from $w_k^{(j)}$
    end for
end for
$Z = [z_1^{(0)}, \ldots, z_n^{(n-1)}]$ and $W = [w_1^{(0)}, \ldots, w_n^{(n-1)}]$
```

Sparse approximate inverses

AINV preconditioner

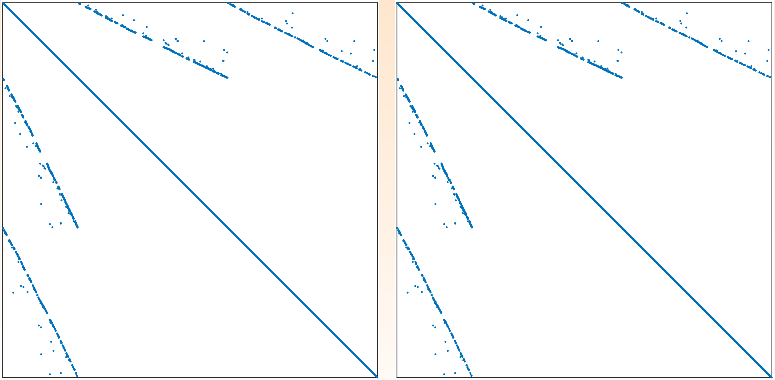
- Matrix A, AINV preconditioner



Sparse approximate inverses

AINV preconditioner

- ILUT, inverse ILUT



Sparse approximate inverses

SAINV: stabilization of the AINV method

- The following result is analogous to the SPD case.

Theorem

If A is a nonsingular M- or H-matrix then the AINV factorization of A does not break down.

- For more general matrices breakdown can happen because of the occurrence of zero d_{jj} or, in the SPD case, negative d_{jj}.
- In practice, exact zeros are unlikely but very small d_{jj} can occur (near breakdown), which may lead to uncontrolled growth in the size of entries in the incomplete factors and, because such entries are not dropped when using a threshold parameter, a large amount of fill-in.
- The next theorem indicates how breakdown can be prevented when A is SPD through reformulating the A-orthogonalization.

Sparse approximate inverses

SAINV: stabilization of the AINV method

Theorem

Consider AINV algorithm with no sparsification (Step 7 is removed). The following holds

$$A_{j,1:n}\, z_k^{(j-1)} \equiv e_j^T A z_k^{(j-1)} = \langle z_j^{(j-1)}, z_k^{(j-1)} \rangle_A, \quad 1 \le j \le k \le n.$$

Proof.

Because $AZ = Z^{-T}D$ and $Z^{-T}D$ is lower triangular, entries 1 to $j-1$ of the vector $Az_k^{(j-1)}$ are equal to zero. Z is unit upper triangular so entries $j+1$ to n of its j-th column $z_j^{(j-1)}$ are also equal to zero. Thus, $z_j^{(j-1)}$ can be written as the sum $z + e_j$, where entries j to n of th vector z are zero. The result follows. □ □

Sparse approximate inverses

SAINV: stabilization of the AINV method

Algorithm (**SAINV preconditioner (right-looking approach)**)

Input: *SPD matrix A and sparsifying rule.*
Output: *$A^{-1} \approx ZD^{-1}Z^T$ with Z a unit upper triangular matrix and D a diagonal matrix with positive diagonal entries.*

```
$[z_1^{(0)}, \dots, z_n^{(0)}] = [e_1, \dots, e_n]$
for $j = 1 : n$ do
    $d_{jj} = \langle z_j^{(j-1)}, z_j^{(j-1)} \rangle_A$
    for $k = j + 1 : n$ do
        $\alpha = \langle z_k^{(j-1)}, z_j^{(j-1)} \rangle_A / d_{jj}$
        $z_k^{(j)} = z_k^{(j-1)} - \alpha z_j^{(j-1)}$
        Sparsify $z_k^{(j)}$                              ▷ Drop entries from $z_k^{(j)}$
    end for
end for
$Z = [z_1^{(0)}, \dots, z_n^{(n-1)}]$
```

Sparse approximate inverses

From AINV to Cholesky

- The factors Z and D obtained with no sparsification can be used to compute the square root-free Cholesky factorization of A.
- The L factor of A and the inverse factor Z computed using AINV Algorithm without sparsification satisfy

$$AZ = LD \quad \text{or} \quad L = AZD^{-1}.$$

- Using $d_{jj} = \langle z_j^{(j-1)}, z_j^{(j-1)} \rangle_A$, and equating corresponding entries of AZD^{-1} and L gives

$$l_{ij} = \frac{\langle z_j^{(j-1)}, z_i^{(j-1)} \rangle_A}{\langle z_j^{(j-1)}, z_j^{(j-1)} \rangle_A}, \quad 1 \leq j \leq i \leq n.$$

- Thus, the SAINV algorithm generates the L factor of the square root-free Cholesky factorization of A as a by-product of orthogonalization in the inner product $\langle . , . \rangle_A$ at no extra cost and without breakdown.

Sparse approximate inverses

Stabilization-like for general A

- The stabilization strategy can be extended to the nonsymmetric AINV algorithm using the following result.

Theorem

Consider nonsymmetric AINV Algorithm with no sparsification (Steps 7 and 10 removed). The following identities hold:

$$A_{j,1:n}\, z_k^{(j-1)} \equiv e_j^T A z_k^{(j-1)} = \langle w_j^{(j-1)},\, z_k^{(j-1)} \rangle_A,$$

$$(A_{1:n,j})^T w_k^{(j-1)} \equiv e_j^T A^T w_k^{(j-1)} = \langle z_j^{(j-1)},\, w_k^{(j-1)} \rangle_A, \quad 1 \le j \le k \le n.$$

- The nonsymmetric SAINV algorithm obtained using this reformulation can improve the preconditioner quality but it is not guaranteed to be breakdown free.

Approximate factorizations

Approximate inverse by global iterations

Consider one-dimensional Newton-Raphson iterations to find a scalar value p which is the root of a given function f, that is

$$f(p) = 0.$$

The method approaches p by a sequence of approximations $p_0, p_1, \ldots$. Consider a tangent of f at p_k for some integer $k \geq 0$ in the following form

$$y = f'(p_k)p_k + b. \tag{74}$$

The tangent crosses $(p_k, f(p_k))$ and this can be put down as

$$f(p_k) = f'(p_k)p_k + b. \tag{75}$$

This implies

$$b = f(p_k) - f'(p_k)p_k \tag{76}$$

and we get a function of x given by

$$y = f'(x)x + f(p_k) - f'(p_k)p_k. \tag{77}$$

Approximate factorizations, splitting and preconditioning

Assume that the root is achieved at p_{k+1}. Then

$$0 = f'(p_{k+1})p_{k+1} + f(p_k) - f'(p_k)p_k \tag{78}$$

and therefore

$$p_{k+1} = p_k - \frac{f(p_k)}{f'(p_k)}. \tag{79}$$

For f beeing the function of the inverse given by

$$f(x) = 1/x - a$$

we have

$$p_{k+1} = p_k - \frac{1/p_k - a}{-1/p_k^2} = p_k(2 - ap_k). \tag{80}$$

Finding the matrix inverse in case it is well-defined:

$$G_{i+1} = G_i(2I - AG_i),\ i = 1, \ldots$$

for the sequence of non-factorized approximate inverses $G_0, \ldots$. The main problem: G is for irreducible A fully dense.

Approximate factorizations, splitting and preconditioning

Consider the computation of the j-th diagonal entry and assume the exactly computed quantities. The computation from the formula

$$d_j = A_{jj} + A^T_{1:j-1,j} w_j + A_{j,1:j-1} z_j + w_j^T A_{j-1} z_j \tag{81}$$

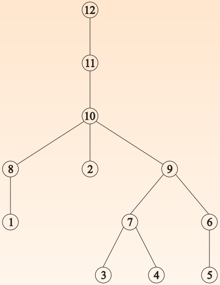
can be easily replaced by the mathematically equivalent formula which we used in the algorithms using biconjugation computation:

$$d_j = A_{j,1:j-1} z_j \text{ or } d_j = A^T_{1:j-1,j} w_j . \tag{82}$$

Notes on parallel approaches

1. Shared memory computers

- 1st level of parallelism: tree structure of the decomposition.
- 2nd level of parallelism: local node parallel enhancements.



- Both may/should be coordinated.
- **Tree parallelism** potential decreases towards its root.
- Potential for the local parallelism (larger dense matrices) increases towards the root.

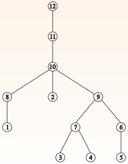
Decomposition and computer architectures: 1st level of parallelism

Two basic possibilities for the tree parallelism

- Dynamic task scheduling on shared memory computers
- Direct static mapping: subtree to subcube

1. Dynamic task scheduling on shared memory computers

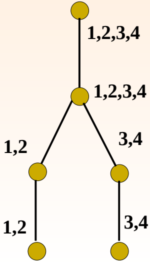
- Dynamic scheduling of the tasks
- Each processor selects a task
- Again, problem of elimination tree reordering
- Not easy to optimize memory, e.g., in the multifrontal method



Decomposition and computer architectures: 1st level of parallelism: II

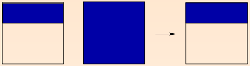
2. Direct static mapping: subtree to subcube

- Recursively map processors to the tree parts from the top
- Various ways of mapping.
- Note: In the SPD (non-pivoting) case the arithmetic work can be computed and considered
- Localized communication
- More difficult to share the work among processors in more complex models

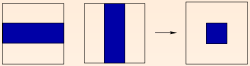


Decomposition and computer architectures: 2nd level of parallelism

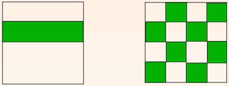
- Block Cholesky/LU factorization
- BLAS / parallel BLAS operations



1D partitioning



2D partitioning



1D and 2D block cyclic distribution

(Only illustrative figures for the talk!)

Decomposition and computer architectures: Distributed memory parallelism

Basic classical parallelization approaches (consider Cholesky)

- Fan-in approach
  - Demand-driven column-based algorithm
  - Required data are aggregated updates asked from previous columns
- bf Fan-out approach
  - Data-driven column-based algorithm
  - Updates are broadcasted once computed and aggregated
  - Historically the first approach; greater interprocessor communication than fan-in
- Multifrontal approach
  - Example: MUMPS