Matrix Decompositions for SLAM

Condition Numbers, Fill-in, and Multifrontal Cholesky

Hyungtae Lim, Ph.D.

SPARK Lab, Laboratory for Information & Decision Systems (LIDS), MIT

shapelim@mit.edu | fudxo5143+slam@gmail.com

Outline

At Every Gauss-Newton (GN) / Levenberg-Marquardt (LM) Iteration

After linearizing the residuals, the GN step reduces to:

$$\underbrace{(\boldsymbol{J}^\top \boldsymbol{J})}_{\boldsymbol{A} \in \mathbb{R}^{n \times n}} \boldsymbol{\Delta}^* = \underbrace{-\boldsymbol{J}^\top \boldsymbol{\epsilon}_0}_{\boldsymbol{b}}$$

$\boldsymbol{A} = \boldsymbol{J}^\top \boldsymbol{J}$ is **symmetric positive semi-definite (SPSD)**.

Scale in real SLAM:

- 1 pose (6 DoF): 6×6 system $\rightarrow$ trivial.
- $1{,}000$ poses: $6{,}000 \times 6{,}000$ system.
- $10{,}000$ poses + landmarks: $\sim 10^5 \times 10^5$ system.

Three questions for any solver

1. How **stable** is the solve?
2. How **fast** (big-O complexity)?
3. Does it exploit **sparsity**?

The choice of *decomposition* determines all three.
Wrong choice $\rightarrow$ numerical blow-up *or* $10{,}000\times$ slowdown.

Three Strategies at a Glance

Method	Operates on	Flops	Sensitivity	SLAM use
Dense QR	$\boldsymbol{A}$ directly	$O(mn^2)$	$\kappa(\boldsymbol{A})$	Marginalization, small n
Dense Cholesky	$\boldsymbol{A}^\top\boldsymbol{A}$	$O(n^3/3)$	$\kappa(\boldsymbol{A})^2$	Dense, well-conditioned
Sparse Multifrontal	$\boldsymbol{A}^\top\boldsymbol{A}$ (sparse)	$O(n)$–$O(n^{3/2})$	$\kappa(\boldsymbol{A})^2$	**SLAM default**

Spoiler:

- GTSAM and Ceres both default to **sparse multifrontal Cholesky**.
- QR appears during **marginalization** when variables become well-constrained.

"Sensitivity" preview

Sensitivity = how much a small measurement error gets amplified in the solution.

Lower is better. QR wins on stability; Cholesky wins on speed.

Warm-up: Small Measurement Error, Big Solution Error?

Consider two 2D systems (two lines, one intersection):

Well-conditioned (nearly perpendicular):

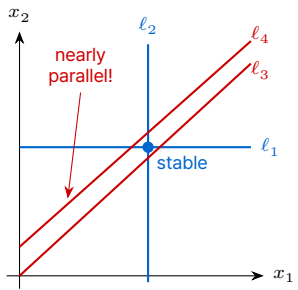
$$\begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix} \boldsymbol{x} = \begin{bmatrix} 2 \\ 3 \end{bmatrix} \Rightarrow \boldsymbol{x} = \begin{bmatrix} 2 \\ 3 \end{bmatrix}$$

Perturb $b_1 + 0.01 \rightarrow$ shift 0.01. (✓)

Ill-conditioned (nearly parallel):

$$\begin{bmatrix} 1 & 1 \\ 1 & 1.001 \end{bmatrix} \boldsymbol{x} = \begin{bmatrix} 2 \\ 2.001 \end{bmatrix} \Rightarrow \boldsymbol{x} = \begin{bmatrix} 1 \\ 1 \end{bmatrix}$$

Perturb $b_1 + 0.01 \rightarrow$ shift ≈ 10. (×)



Perpendicular lines → stable intersection.
Nearly parallel → tiny perturbation moves intersection far away.

The Condition Number $\kappa(A)$: Formal Definition

$$\kappa(\boldsymbol{A}) \;=\; \|\boldsymbol{A}\| \cdot \|\boldsymbol{A}^{-1}\| \;=\; \frac{\sigma_{\max}(\boldsymbol{A})}{\sigma_{\min}(\boldsymbol{A})}$$

where $\sigma_{\max}, \sigma_{\min}$ are the largest/smallest **singular values** of $\boldsymbol{A}$.

What it means: if $\boldsymbol{b}$ has relative error δ (measurement noise), the solution $\boldsymbol{x}$ can suffer relative error up to $\kappa(\boldsymbol{A}) \cdot \delta$.

$\kappa(\boldsymbol{A})$	Interpretation
1	Perfect (orthogonal matrix)
10^3	Lose $\sim$3 decimal digits
10^6	Lose $\sim$6 decimal digits
$\geq 10^{15}$	Numerically singular

Rule of thumb

IEEE double has ≈ 15 significant digits.

$\kappa(\boldsymbol{A}) = 10^k \Rightarrow$ lose k digits.

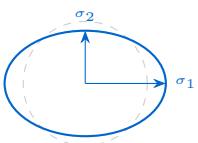
$\kappa = 10^{15}$: solution is *pure noise*.

For the ill-conditioned example:

$$\kappa\left(\begin{bmatrix} 1 & 1 \\ 1 & 1.001 \end{bmatrix}\right) \approx 2 \times 10^3$$

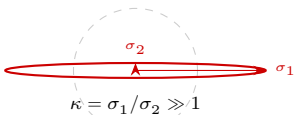
Geometric Intuition: Singular Values as Stretching Factors

The singular values σ_i describe how much $\boldsymbol{A}$ *stretches* space in each direction.



$\kappa = \sigma_1/\sigma_2 \approx 1.5$

Well-conditioned



$\kappa = \sigma_1/\sigma_2 \gg 1$

Ill-conditioned

$\boldsymbol{A}$ maps the unit ball to an **ellipsoid**. Semi-axes = singular values.

- Nearly degenerate $\boldsymbol{A} \to$ thin disk.
- Inverting maps disk back to ball: tiny changes in the "thin" direction get **hugely amplified**.
- κ = aspect ratio of the ellipsoid.

The Squaring Problem: $\kappa(\boldsymbol{A}^\top \boldsymbol{A}) = \kappa(\boldsymbol{A})^2$

Why? Singular values of $\boldsymbol{A}^\top \boldsymbol{A}$ are the *squares* of singular values of $\boldsymbol{A}$:

$$\kappa(\boldsymbol{A}^\top \boldsymbol{A}) = \frac{\sigma_{\text{max}}^2}{\sigma_{\text{min}}^2} = \kappa(\boldsymbol{A})^2$$

$\kappa(\boldsymbol{A})$	$\kappa(\boldsymbol{A}^\top \boldsymbol{A})$	Digits lost (double)
10^3	10^6	6 digits
10^7	10^{14}	14 digits!
10^8	10^{16}	Total precision loss

Why QR is more stable

Cholesky forms $\boldsymbol{A}^\top \boldsymbol{A}$ explicitly
$\Rightarrow$ sensitivity $\kappa(\boldsymbol{A})^2$.

QR works on $\boldsymbol{A}$ directly, never forms $\boldsymbol{A}^\top \boldsymbol{A}$
$\Rightarrow$ sensitivity only $\kappa(\boldsymbol{A})$.

With $\kappa(\boldsymbol{A}) = 10^7$: Cholesky loses 14 digits; QR loses 7.

When Does κ Blow Up in SLAM?

Short answer: when some variables are *very* well constrained while others are *barely* constrained — the system becomes lopsided.

Information $\leftrightarrow$ eigenvalue:

- For one variable, **information** $= 1/\text{variance}$.
- In $\boldsymbol{J}^\top \boldsymbol{J}$, each direction's information *is* an eigenvalue σ_i^2.
- More observations $\Rightarrow$ smaller variance $\Rightarrow$ **larger eigenvalue**.

Where SLAM goes lopsided:

- Recent pose — many observations $\Rightarrow \sigma_{\text{max}} \uparrow$
- Far landmark — few observations $\Rightarrow \sigma_{\text{min}} \downarrow$
- Mix them: $\kappa = \sigma_{\text{max}}/\sigma_{\text{min}} \gg 1$.

Intuitive analogy

One direction is pinned down tightly; another is almost free to drift. The problem "shape" is stretched — exactly the ill-conditioned case.

Why it matters for solver choice:

- Already $\kappa \gg 1$ from data.
- Cholesky forms $\boldsymbol{A}^\top \boldsymbol{A} \Rightarrow$ effective κ^2.
- Especially during **marginalization**: removing old poses compresses information onto the remaining ones.

Bottom line: mixed-certainty SLAM $\rightarrow$ high $\kappa \rightarrow$ **Use QR.**

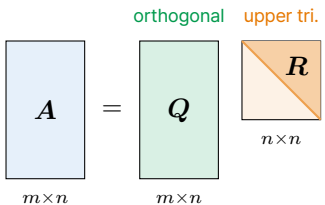
QR Decomposition: The Orthogonal-Triangular Split

Factorize $A \in \mathbb{R}^{m \times n}$ $(m \geq n)$ as $A = QR$:

$$Q \in \mathbb{R}^{m \times n},\ Q^\top Q = I \quad \text{(orthonormal columns)}, \qquad R \in \mathbb{R}^{n \times n} \text{ upper triangular}$$

Geometric meaning:

- Q: orthonormal basis for the column space of A. Multiplying by $Q^\top$ is a *rotation* — does **not** change distances or condition numbers.
- R: upper-triangular encoding of the "coordinates." Easy to invert via back-substitution.



Algorithm choices:

- **Householder reflections**: numerically stable, standard.
- **Givens rotations**: good for sparse A and streaming updates.

Since $Q^\top Q = I$, multiplying by $Q^\top$ preserves norms $\Rightarrow$ condition number of R equals that of A.

Solving $\min \|\boldsymbol{A}\boldsymbol{x} - \boldsymbol{b}\|^2$ **via QR**

Substitute $\boldsymbol{A} = \boldsymbol{Q}\boldsymbol{R}$:

$$\|\boldsymbol{A}\boldsymbol{x} - \boldsymbol{b}\|^2 = \|\boldsymbol{Q}\boldsymbol{R}\boldsymbol{x} - \boldsymbol{b}\|^2 = \|\boldsymbol{R}\boldsymbol{x} - \underbrace{\boldsymbol{Q}^\top \boldsymbol{b}}_{\tilde{\boldsymbol{b}}}\|^2 \quad \text{(multiply by } \boldsymbol{Q}^\top\text{, norm preserved)}$$

Minimum when $\boldsymbol{R}\boldsymbol{x} = \boldsymbol{Q}^\top \boldsymbol{b} \Rightarrow$ one **back-substitution** ($O(n^2)$). **Never form** $\boldsymbol{A}^\top \boldsymbol{A}$**!**

Contrast with normal equations

Normal eq. approach: $\hat{\boldsymbol{x}} = (\boldsymbol{A}^\top \boldsymbol{A})^{-1} \boldsymbol{A}^\top \boldsymbol{b}$
$\Rightarrow$ sensitivity $\kappa(\boldsymbol{A})^2$.

QR approach: $\hat{\boldsymbol{x}} = \boldsymbol{R}^{-1}(\boldsymbol{Q}^\top \boldsymbol{b})$
$\Rightarrow$ sensitivity $\kappa(\boldsymbol{A})$ only.

Dense QR complexity:

- Factorize: $O(mn^2)$
  ($= 2mn^2 - \frac{2}{3}n^3$ flops)
- Compute $\boldsymbol{Q}^\top \boldsymbol{b}$: $O(mn)$
- Back-substitute: $O(n^2)$
- **Total:** $O(mn^2)$

For square $m = n$: $O(n^3)$.
$\approx 2\times$ more flops than Cholesky.

Cholesky: Exploiting Symmetry and Positive Definiteness

For $\boldsymbol{A} \in \mathbb{R}^{m \times n}$, the matrix $\boldsymbol{A}^\top \boldsymbol{A}$ is always **symmetric positive semi-definite (SPSD)**; when full-rank, it is **SPD**. SPD matrices admit a unique lower-triangular factorization:

$$\boldsymbol{A}^\top \boldsymbol{A} = \boldsymbol{L}\boldsymbol{L}^\top, \qquad \boldsymbol{L} \in \mathbb{R}^{n \times n} \text{ lower triangular, positive diagonal}$$

Algorithm (column-by-column):

$$L_{jj} = \sqrt{[\boldsymbol{A}^\top \boldsymbol{A}]_{jj} - \textstyle\sum_{k<j} L_{jk}^2}$$

$$L_{ij} = \tfrac{1}{L_{jj}} \left([\boldsymbol{A}^\top \boldsymbol{A}]_{ij} - \textstyle\sum_{k<j} L_{ik} L_{jk} \right), \; i > j$$

Solve $\boldsymbol{A}^\top \boldsymbol{A}\, \boldsymbol{x} = \boldsymbol{A}^\top \boldsymbol{b}$ in two triangular steps:

1. Forward: $\boldsymbol{L}\, \boldsymbol{y} = \boldsymbol{A}^\top \boldsymbol{b}$ $\quad (O(n^2))$
2. Backward: $\boldsymbol{L}^\top \boldsymbol{x} = \boldsymbol{y}$ $\quad (O(n^2))$

Dense Cholesky complexity:

- Form $\boldsymbol{A}^\top \boldsymbol{A}$: $O(mn^2)$
- Factorize: $O(n^3/3)$
- Back-sub: $O(n^2)$
- **Total:** $O(n^3/3)$

Only **half** the matrix stored (symmetry).

Trade-off: works on $\boldsymbol{A}^\top \boldsymbol{A} \Rightarrow$ sensitivity $\kappa(\boldsymbol{A})^2$.

Dense QR vs. Cholesky: Summary

	Operates on	Dense flops	Sensitivity	Memory
Dense QR	$\boldsymbol{A}$ directly	$O(mn^2)$	$\kappa(\boldsymbol{A})$	$O(mn)$
Dense Cholesky	$\boldsymbol{A}^\top \boldsymbol{A}$	$O(n^3/3)$	$\kappa(\boldsymbol{A})^2$	$O(n^2/2)$

Use Dense QR when

- Maximum numerical stability required.
- Near-degenerate constraints.
- **Marginalization** in sliding-window SLAM.
- Small problem ($n < 500$).

Use Dense Cholesky when

- Problem is well-conditioned.
- Speed matters over extreme stability.
- Dense block fits in memory.
- **Not** a large sparse SLAM system.

For SLAM with $n > 500$, dense methods are impractical — we must exploit **sparsity**.

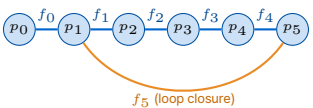
SLAM Jacobians Are Sparse

Each factor (measurement) involves only 1–2 variables.
$\Rightarrow$ each row of $\boldsymbol{J}$ has only a **few non-zeros**.

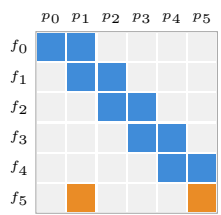
System	$\boldsymbol{J}$ size	Density
10 poses	60×60	$\sim$15%
100 poses	600×600	$\sim$2%
1 000 poses	$6{,}000\times6{,}000$	$<0.1\%$

Naive dense Cholesky on $6{,}000\times6{,}000$:
$O(6{,}000^3)\approx2\times10^{11}$ flops $\rightarrow$ unusable!

Pose graph (with loop closure)



Sparsity pattern of $\boldsymbol{J}$



Each f_r links 2 poses; loop closure f_5 adds a far-off pair.

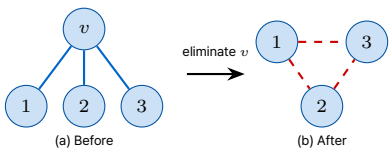
Prerequisite: Cholesky as Variable Elimination

Cholesky factorization processes one variable at a time, in a chosen order.

What happens at each step?

1. Pick the next variable v.
2. Use v's row to express v in terms of its neighbors $\rightarrow$ this becomes a column of $\boldsymbol{L}$.
3. **"Eliminate"** v: subtract v's contribution from all remaining rows. Now v no longer appears in the system.
4. Move on to the next variable.

Graph interpretation: $\boldsymbol{A} = \boldsymbol{J}^\top \boldsymbol{J}$ has a non-zero A_{ij} for each pair of variables that share a factor $\Rightarrow$ $\boldsymbol{A}$ *is* the variable adjacency graph.



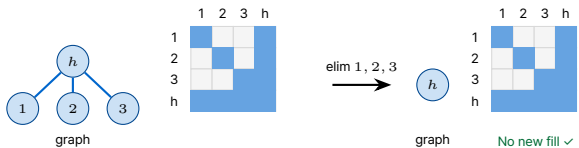
Key takeaway

Eliminating a variable v *removes* it from the graph, but **connects all of v's neighbors to each other**. Those new edges are exactly what we'll call **fill-in**.

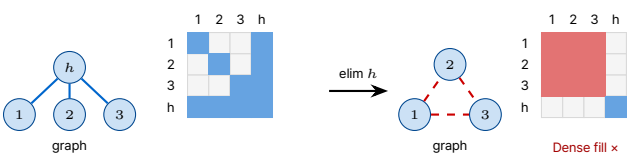
The Fill-in Problem: Ordering Changes Everything

Fill-in: zero entries of $\boldsymbol{A} = \boldsymbol{J}^\top \boldsymbol{J}$ that turn *non-zero* during elimination.
Setup: 4 variables in a "hub" graph — node h connects to $1, 2, 3$; nodes $1, 2, 3$ have *no* mutual edges.

Good order: eliminate tips $1, 2, 3$ first



Bad order: eliminate hub h first



Reading the picture:

- Graph $\equiv$ matrix sparsity.
- "Eliminate v": remove v from graph; v's neighbors become **pairwise connected**.

Hub-first $\rightarrow$ fill:
h's 3 neighbors get fully wired — 3 new edges = 3 fill-ins.
Tips-first $\rightarrow$ none:
Each tip's only neighbor is h — nothing to wire.

$\Rightarrow$ **Order matters. Fill-reducing orderings** (AMD, COLAMD) automate the choice.

AMD: Approximate Minimum Degree

Goal: find a permutation P such that factorizing $PAP^\top$ creates *minimal* fill-in.

The AMD algorithm:

1. Build the *adjacency graph* G: nodes = variables, edges = non-zero off-diagonal entries of $\boldsymbol{A}$.
2. **Greedy**: choose the node with the *fewest neighbors* (minimum degree).
3. Eliminate it; connect all its neighbors to each other (new clique edges = fill).
4. Repeat until all nodes eliminated.

"Approximate": use a fast lower-bound estimate of degree for speed (true degree recomputation would be $O(n^2)$).

Why minimum degree?

Eliminating a node with d neighbors creates at most $\binom{d}{2}$ new edges. Choosing d_{min} greedily minimizes worst-case fill at each step.

Not globally optimal, but highly effective in practice.

Used in: SuiteSparse, Eigen `SimplicialLLT`, MATLAB `symamd`.

COLAMD and METIS

COLAMD (Column AMD)

Applied to *columns* of $\boldsymbol{A}$ before forming $\boldsymbol{A}^\top \boldsymbol{A}$.

- Avoids computing $\boldsymbol{A}^\top \boldsymbol{A}$ explicitly.
- Better for rectangular systems ($m \gg n$).
- **Default in GTSAM and Ceres** (via CHOLMOD).

AMD (Symmetric)

Applied to $\boldsymbol{A}^\top \boldsymbol{A}$ directly.

- Standard for small-to-medium SLAM.
- Eigen's `SimplicialLLT` uses this.

METIS (Graph Partitioning)

- Recursively bisects the sparsity graph.
- Fill: $O(n \log n)$ for 2D meshes.
- Better for very large problems ($n > 10^5$).
- Used in: MUMPS, PaStiX.

In SLAM: **COLAMD** is the workhorse (GTSAM/Ceres default via SuiteSparse).
It reduces sparse Cholesky from $O(n^3)$ to near-$O(n)$ on SLAM chain graphs.

Effect of Ordering on Complexity

Graph structure	Ordering	Fill-in	Factorization	Example
Dense	—	$O(n^2)$	$O(n^3)$	Fully connected
2D grid ($\sqrt{n}\times\sqrt{n}$)	None	$O(n \log n)$	$O(n^{3/2})$	Dense grid map
2D grid	AMD/Nested dissect.	$O(n)$	$O(n^{3/2})$	Grid map
3D mesh	Nested dissect.	$O(n^{4/3})$	$O(n^2)$	Volumetric
SLAM chain	**Any**	$O(n)$	$\boldsymbol{O(n)}$	Sequential SLAM
SLAM general	**COLAMD**	$O(n)$–$O(n^{3/2})$	$\boldsymbol{O(n)} \sim \boldsymbol{O(n^{3/2})}$	Loop closure

SLAM graphs have favorable structure:
Each pose connects to only a few neighbors (local odometry + sparse loop closures). This gives a **banded / tree-like** sparsity pattern $\rightarrow$ near-$O(n)$ even with loop closures.

Real performance

$10{,}000$-pose SLAM:

- Dense Cholesky: $\sim$hours.
- Sparse + COLAMD: $\sim$**milliseconds**.

This is why GTSAM/Ceres can optimize large maps in real time.

Why Big-O Is Not the Whole Story

So far: Big-O told us how many arithmetic operations a solver does.
Next question: can the CPU actually execute those operations at full speed?

Same math, different inner loop:

- **Sparse column update:** load one non-zero, chase an index, update a far-away entry.
- **Dense block update:** load a small matrix block once, reuse it many times in cache.
- Modern CPUs are fast at arithmetic but slow at waiting for scattered memory.

Mental shift: not only "how many FLOPs?" but also "how many useful FLOPs per byte loaded?"

Two hardware words

Cache line: CPU fetches memory in chunks, typically 64 bytes. A random 8-byte read can waste most of that fetch.

Bandwidth-bound: speed limited by moving bytes from DRAM into CPU, not by multiply-add throughput.

Multifrontal idea: keep global sparsity; make the expensive inner loop dense.

Prerequisite: What Does *Peak FLOPs* Mean?

Peak FLOP rate: best-case arithmetic throughput on ideal dense code.

Achieved FLOP rate: throughput this solver actually sustains.

Toy number:

CPU peak	1 TFLOP/s
Plain sparse Cholesky	150 GFLOP/s
Utilization	$150/1000 = 15\%$

So "15%" means the arithmetic units are **under-fed**: the CPU could multiply/add faster, but operands arrive too slowly.

Important nuance

The CPU is not simply "idle." It is busy doing:

- memory address calculation,
- sparse index chasing,
- cache-miss waiting,
- branch / loop overhead.

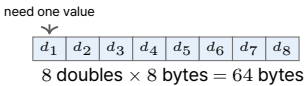
Those cycles do not count as useful FLOPs.

Key idea: FLOP count is not enough; sparse solvers also pay to move and locate data.

Prerequisite: The Missing Hardware Fact—CPUs Fetch Chunks

A CPU fetches memory in **cache lines** (commonly 64 bytes), not one number at a time.

One cache line:



- If the next 7 doubles are also used, the fetch is efficient.
- If they are unrelated, most of the line is wasted bandwidth.

Rough latency ladder:

Register	~ 1 cycle
L1 cache	~ 4 cycles
L2 cache	~ 10 cycles
L3 cache	~ 40 cycles
DRAM	~ 100–300 cycles

Dense code reuses cache lines.
Sparse scattered code keeps asking DRAM for new lines.

Why Sparse Column Updates Jump Around in Memory

Recap: plain sparse Cholesky builds the factor one column at a time:

$$\boldsymbol{A}^\top \boldsymbol{A} = \boldsymbol{L}\boldsymbol{L}^\top, \qquad \boldsymbol{L} = [\,\ell_1\ \ell_2\ \cdots\ \ell_n\,].$$

For column j, the code repeatedly does:

1. Read the row-index list for nonzeros in ℓ_j.
2. Read older columns that overlap those rows.
3. Accumulate numerical updates.
4. Write results into sparse row locations.

Even if each sparse column is stored compactly, the **rows touched by the update** can be far apart.

Mental model:

target rows: $3, 17, 420, 901, \ldots$

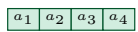
Why this hurts

The processor cannot stream through one nice array. It must follow row indices, jump to target entries, and wait on cache misses.

Why SIMD Does Not Save Plain Sparse Cholesky

SIMD = one instruction operates on several numbers at once.
It works best when data are contiguous.

Good SIMD case: dense contiguous data



one vector multiply-add

- Load adjacent values.
- Use all vector lanes.
- Reuse cache lines.

Sparse update case

- Values may be stored compactly.
- But the *destination rows* come from an index list.
- Updates become gather/scatter operations.
- Vector lanes wait because addresses are irregular.

Bottom line

Plain sparse Cholesky is often **memory-bound**:
$<20\%$ of peak FLOP rate is typical.

Why Multifrontal Cholesky Helps

Multifrontal Cholesky keeps the **global** matrix sparse.
It changes the **inner loop**: scattered sparse updates → dense block updates.

	Plain sparse Cholesky	Multifrontal Cholesky
Unit of work	one sparse column	one dense frontal matrix
Memory access	row-index chasing	contiguous dense block
Cache behavior	many misses	reuse the same block
SIMD/BLAS	hard to vectorize	`DPOTRF`, `DGEMM`, `DTRSM`
Global sparsity	preserved	preserved

The trick:

- Find columns with compatible sparsity.
- Group them into a **supernode**.
- Assemble a dense **frontal matrix**.

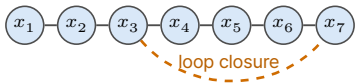
What changes

Same math, same solution. The hardware sees dense matrix operations, not scattered scalar updates.

Next: how the elimination tree and supernodes create those dense fronts.

The Elimination Tree on a SLAM Example (1/2)

SLAM scenario: 7 sequential poses connected by odometry, plus one loop closure $x_3 - x_7$.



Eliminate x_j in order $1, 2, \ldots, 7$. Cholesky connects every pair of *later* neighbors of x_j in $\boldsymbol{L}$ (= fill-in).

- x_1, x_2: only one later neighbor $\rightarrow$ no fill.
- x_3: later $\{x_4, x_7\} \rightarrow$ **fill** (x_4, x_7).
- x_4: $\{x_5, x_7\} \rightarrow$ fill (x_5, x_7).
- Wave continues at $x_5, x_6 \rightarrow$ row 7 nearly dense.

Punchline: the affected path $3 \rightarrow 4 \rightarrow 5 \rightarrow 6$ all send updates to the same later separator $x_7 \rightarrow$ a good candidate for a relaxed supernode/front (next slide).

Resulting $\boldsymbol{L}$ pattern:

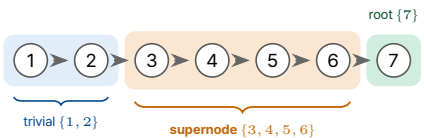
$$\boldsymbol{L} = \begin{pmatrix} \bullet & & & & & & \\ * & \bullet & & & & & \\ & * & \bullet & & & & \\ & & * & \bullet & & & \\ & & & * & \bullet & & \\ & & & & * & \bullet & \\ & & \circ & \circ & \circ & * & \bullet \end{pmatrix}$$

$\bullet$ diagonal, $*$ original, $\circ$ fill-in from loop closure.

Supernodes and the Frontal Matrix (2/2)

Recap: loop closure $x_3 - x_7$ creates a common separator x_7 along $3 \to 4 \to 5 \to 6$.

Supernode rule: after an ordering, merge consecutive columns with the same rows below the block. This is decided by symbolic L, not by raw loop endpoints.



Frontal matrix = dense block for the front + separator rows.
For relaxed front $\{3, 4, 5, 6\}$: four variables *plus* separator pose 7:

$F_{\{3,4,5,6\}} =$

	cols 3-6	col 7
rows 3-6	■	■
row 7	■	■

Not a rule: loop $x_n - x_m$ may yield $\{n, \dots, m-1\}$ only in a pure chain with natural ordering; other orderings/factors can split it.

- 6-DoF poses: 5×5 variable block $\Rightarrow$ 30×30 **dense** numeric front.
- Factor with `DPOTRF`; update pose 7 with `DGEMM`.
- Contiguous memory $\Rightarrow$ cache/SIMD friendly.

What “Multifrontal” Actually Means

The name = **multi-** + **-frontal**, from a 1970s/80s line asking: how do we organize sparse elimination so the inner loop is dense?

“Frontal” (single-front, 1970s): a *front* is the active row/column set being eliminated. Hold *one* dense frontal matrix; grow/shrink as you sweep through variables.

“Multi-frontal” (Duff & Reid, 1983): the elimination tree exposes *independent* subtrees, each with its own front. Process **many** fronts (any order, even in parallel); assemble children into parents bottom-up.

Yes — it is divide-and-conquer:

- **Divide:** tree splits problem into independent fronts.
- **Conquer:** factor each frontal matrix densely (BLAS).
- **Combine:** push each child’s Schur update onto its parent.

How one front works:

1. Gather original matrix entries involving the current supernode.
2. Add Schur-complement updates from already-factorized child fronts.
3. Factor the local dense frontal matrix with BLAS.
4. Send the remaining update to the parent front.

Connection to the previous slides

The graph is still sparse globally. The expensive numerical work happens inside each **dense** front, where cache reuse and SIMD are strong.

BLAS: The Engine Behind Multifrontal

BLAS = **B**asic **L**inear **A**lgebra **S**ubprograms. A standardized API with vendor-tuned implementations (Intel MKL, OpenBLAS, Apple Accelerate, NVIDIA cuBLAS), organized into **three levels** by operation shape.

Level	Example	FLOPs	Bytes	Intensity (FLOP/byte)
1 vector ↔ vector	`DAXPY`: $\boldsymbol{y} \mathrel{+}= \alpha \boldsymbol{x}$	$2n$	$\sim 24n$	~ 0.08
2 matrix ↔ vector	`DGEMV`: $\boldsymbol{y} \mathrel{+}= \boldsymbol{A}\boldsymbol{x}$	$2n^2$	$\sim 8n^2$	~ 0.25
3 matrix ↔ matrix	`DGEMM`: $\boldsymbol{C} \mathrel{+}= \boldsymbol{A}\boldsymbol{B}$	$2n^3$	$\sim 24n^2$	$\sim n/12$

Why Level 3 is special:

- FLOPs grow as n^3, data only as $n^2 \Rightarrow$ each loaded byte is **reused** $\sim n$ **times**.
- Intensity grows with $n \rightarrow$ stays in cache, hits $>$80% peak FLOP rate.
- Levels 1 & 2 are bandwidth-bound. Only Level 3 is compute-bound.

Multifrontal's BLAS recipe:

- `DPOTRF`: dense Cholesky inside one front.
- `DGEMM`: child-to-parent update (Level 3, workhorse).
- `DTRSM`: triangular solve inside front.

All Level 3 in the inner loop $\Rightarrow$ peak hardware utilization on every front.

Multifrontal Pipeline: Putting It All Together

Big picture: plain sparse Cholesky is bandwidth-bound; multifrontal turns it into a sequence of small *dense* solves so the CPU runs at peak speed.

Pipeline:

1. **Reorder** variables with COLAMD to minimize fill-in.
2. Build the **elimination tree**; identify supernodes.
3. For each node (bottom-up):
   - **Assemble** the frontal matrix from original entries + child contributions.
   - **Factorize** densely with BLAS-3 (`DPOTRF+DGEMM+DTRSM`).
   - **Push** the Schur-complement update onto the parent.

Parallelism for free: independent subtrees of the elimination tree can be processed concurrently — multifrontal is the basis of every multi-threaded sparse solver.

Why it is fast:

- Frontal matrices are contiguous in memory $\rightarrow$ cache-friendly.
- BLAS-3 hides memory latency.
- Same $O(n)$ as plain sparse Cholesky, but $5-10\times$ smaller constant.

Libraries (interview-ready):

- CHOLMOD (SuiteSparse): default in GTSAM and Ceres.
- Eigen `SimplicialLLT`: pure-template alternative.

GTSAM defaults to `MULTIFRONTAL_CHOLESKY`, not plain `SPARSE_CHOLESKY`.

Big-O Summary: All Methods Side by Side

Method	Flops	Sensitivity	Sparsity	Used in
Dense Cholesky	$O(n^3/3)$	κ^2	None	Tiny problems
Dense QR	$O(mn^2)$	κ	None	Max stability, small
Simplicial Chol. (no ordering)	$O(n^3)$	κ^2	Partial	(Avoid)
Simplicial Chol. + AMD/COLAMD	$O(n) \sim O(n^{3/2})$	κ^2	Full	Educational baseline
Multifrontal Chol. + COLAMD	$O(n)$ (chain)	κ^2	Full	**GTSAM default**
Sparse QR	$O(n^{3/2})$	κ	Full	Marginalization
Iterative (PCG/CGNR)	$O(n \cdot \text{iter.})$	—	Full	Very large scale

Axis check: AMD/COLAMD chooses the *ordering*; simplicial vs multifrontal chooses the *factorization implementation*.

Interview answer (Big tech / autonomous driving): the default is **sparse multifrontal Cholesky + COLAMD**, complexity $O(n)$–$O(n^{3/2})$ depending on loop-closure density.
Sparse QR (complexity $O(n^{3/2})$, sensitivity κ) is reserved for marginalization of well-constrained variables.

GTSAM: Multifrontal Cholesky as Default

Default solver flows through one parameter chain into every `optimize()` call.

`gtsam/nonlinear/NonlinearOptimizerParams.h`

```
// Default for GN and LM optimizers
LinearSolverType linearSolverType
  = MULTIFRONTAL_CHOLESKY;

// Selects the elimination function
GaussianFactorGraph::Eliminate
getEliminationFunction() const {
  switch (linearSolverType) {
  case MULTIFRONTAL_CHOLESKY:
    return EliminatePreferCholesky;
  case MULTIFRONTAL_QR:
    return EliminateQR;
  // ...
  }
}
```

`gtsam/nonlinear/NonlinearOptimizer.cpp`

```
// Called every GN/LM iteration
delta = gfg.eliminateMultifrontal(
  *cache,
  params.getEliminationFunction()
  // ^ EliminatePreferCholesky
)->optimize();
```

`gtsam/linear/GaussianFactorGraph.h`

```
// Hard-coded default elimination
static ... DefaultEliminate(...) {
  return EliminatePreferCholesky(
          factors, keys);
}
```

Every `graph.optimize()` / `optimizer.optimize()` call reaches `EliminatePreferCholesky` through the default parameter chain above.

GTSAM: QR for Marginalization

The `Marginals` class computes post-optimization covariances and supports QR for ill-conditioned cases (e.g., well-constrained variables in sliding-window marginalization).

`gtsam/nonlinear/Marginals.h`

```
class Marginals {
public:
  enum Factorization {
    CHOLESKY, // default, faster
    QR        // stable for marginals
  };
  // Explicitly request QR:
  Marginals(graph, solution,
            Marginals::QR);
};
```

`gtsam/nonlinear/Marginals.cpp`

```
void computeBayesTree() {
  if (factorization_ == QR)
    bayesTree_ =
      *graph_.eliminateMultifrontal(
        ordering, EliminateQR);
}

GaussianFactor::shared_ptr
marginalFactor(Key var) const {
  if (factorization_ == QR)
    return bayesTree_
      .marginalFactor(var,
                      EliminateQR);
}
```

When to use QR: marginalized variables are well-constrained (low uncertainty $\rightarrow$ large eigenvalues $\rightarrow$ high κ). Squaring via $\boldsymbol{A}^\top \boldsymbol{A}$ risks precision loss — use `Marginals::QR` for safety.

Ceres Solver: Linear Solver Type Enum

All solver choices are declared in a single enum.

`ceres-solver/include/ceres/types.h` (lines 57–91)

```
enum LinearSolverType {
  // Normal-equation solvers
  DENSE_NORMAL_CHOLESKY,
  DENSE_QR,
  SPARSE_NORMAL_CHOLESKY, // default

  // Schur-complement solvers
  // (BA-style E/F block split)
  DENSE_SCHUR,
  SPARSE_SCHUR,
  ITERATIVE_SCHUR,

  // Iterative
  CGNR
};
```

`ceres-solver/include/ceres/solver.h` (lines 325–330)

```
// Default linear solver:
LinearSolverType linear_solver_type =
#if defined(CERES_NO_SPARSE)
    DENSE_QR; // fallback
#else
    SPARSE_NORMAL_CHOLESKY; // default
#endif
```

With sparse libraries (SuiteSparse/CHOLMOD) present: **SPARSE_NORMAL_CHOLESKY**.

Next: Schur solvers need a bundle-adjustment-style E/F block partition.

Ceres Schur Solvers: What Are E-Blocks?

Schur solvers assume the Jacobian columns can be split into two groups:

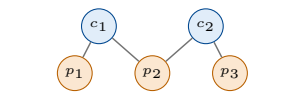
$$\boldsymbol{J} = [\,\boldsymbol{E}\;\boldsymbol{F}\,], \qquad \Delta = \begin{bmatrix} \Delta_E \\ \Delta_F \end{bmatrix}.$$

Meaning in Ceres:

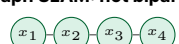
- **E-blocks**: variables eliminated first.
- **F-blocks**: variables kept in the reduced system.
- Schur is fast when $\boldsymbol{E}^\top \boldsymbol{E}$ is block diagonal, so each E-block can be inverted locally.

$$\boldsymbol{S} = \boldsymbol{F}^\top \boldsymbol{F} - \boldsymbol{F}^\top \boldsymbol{E}(\boldsymbol{E}^\top \boldsymbol{E})^{-1} \boldsymbol{E}^\top \boldsymbol{F}.$$

Bundle adjustment: bipartite



Pose graph SLAM: not bipartite



pose–pose factors:
no independent E group

Ceres: Auto-Selection — Schur → Cholesky

No E-blocks = no BA-landmark-like group to eliminate first.
A Schur solver is therefore the wrong family.

`internal/ceres/linear_solver.cc` (lines 51–73)

```
// Auto-select when no E-blocks exist
LinearSolverType
LinearSolverForZeroEBlocks(
    LinearSolverType type) {
  if (type == SPARSE_SCHUR)
    return SPARSE_NORMAL_CHOLESKY;
  if (type == DENSE_SCHUR)
    return DENSE_QR;
  if (type == ITERATIVE_SCHUR)
    return CGNR;
  return type; // unchanged
}
```

`internal/ceres/trust_region_preprocessor.cc`

```
// Triggered when no E-blocks present
if (IsSchurType(options.linear_solver_type)
    && no_e_blocks) {
  AlternateLinearSolverFor
    SchurTypeLinearSolver(&options);
  // Logs: "No E blocks. Switching from
  // SPARSE_SCHUR to
  // SPARSE_NORMAL_CHOLESKY."
}
```

- Pose graph: pose–pose factors.
- No camera–landmark bipartite split.
- Fallback: `SPARSE_SCHUR` → `SPARSE_NORMAL_CHOLESKY`.

Ceres: `SPARSE_NORMAL_CHOLESKY` Implementation

After that switch: Ceres delegates to a pluggable `SparseCholesky` backend.

`internal/ceres/sparse_normal_cholesky_solver.h`

```
class SparseNormalCholeskySolver {
 private:
  // Pluggable backend:
  // SuiteSparse/CHOLMOD, Eigen,
  // Accelerate (Apple), CUDA
  std::unique_ptr<SparseCholesky>
    sparse_cholesky_;

  // Computes A^T A and solves
  std::unique_ptr<InnerProductComputer>
    inner_product_computer_;
};
```

`internal/ceres/sparse_normal_cholesky_solver.cc`

```
// Constructor: create backend
SparseNormalCholeskySolver::
SparseNormalCholeskySolver(
    const LinearSolver::Options& opts)
    : options_(opts) {
  sparse_cholesky_ =
    SparseCholesky::Create(opts);
}

// Solve step (called each iteration)
summary.termination_type =
  sparse_cholesky_->FactorAndSolve(
    inner_product_computer_
      ->mutable_result(), // A^T A
    rhs_.data(),        // A^T b
    x,
    &summary.message);
```

`SparseCholesky::Create()` selects the backend at build time: SuiteSparse/CHOLMOD (default), Eigen SimplicialLLT, Apple Accelerate, or CUDA cuDSS.

Conclusion: When to Use Which Decomposition