TTK4551 Technical Cybernetics - Specialization Project

Martynas Smilingis

September 2025

Contents

Acronyms

AUR Lab	Applied Underwater Robotics Lab
AUV	Autonomous Underwater Vehicle
COLAMD	Column Approximate Minimum Degree
EKF	Extended Kalman Filter
GNSS	Global Navigation Satellite System
GTSAM	Georgia Tech Smoothing And Mapping
iSAM	Iterative Smoothing And Mapping
LAUV	Light Autonomous Underwater Vehicle
MAP	Maximum A Posteriori
SLAM	Simultaneous Localization And Mapping
SSS SLAM	Side Scan Sonar Simultaneous Localization And Mapping

1 Introduction

1.1 Goal

This specialization project studies how marine robots navigate on their own, with a focus on SLAM. The aim is to build a solid, practical understanding of modern navigation and SLAM algorithms, how current SLAM systems are structured, and which methods are used in practice. The work centers on sonar-based SLAM, in particular Side Scan Sonar (SSS) SLAM for AUVs, because side scan sonar is widely available on mature platforms and is useful for mapping large seabed areas. I will work with real AUV datasets (for eks from NTNUs AUR Lab platforms such as LAUV Harald) to evaluate how well SSS SLAM supports navigation tasks, and to see what it takes to run the pipeline close to real time. The project builds on prior work, including the 2023 master thesis by Vegard Haraldstad on a sidescan sonar pipeline [1] and the 2024 Intelligent Robots and Systems (IROS) paper on side scan sonar based landmark detection for underwater vehicles [2]. The plan is to study these methods, reimplement the key parts, adapt them where needed and make SSS SLAM run in real time on real hardware.

1.2 Motivation

Marine robots must make decisions without constant human input, often far from easy access. They need to know where they are and what surrounds them to move safely and do useful work. A prior map helps, but the ocean changes over time. Currents, waves, moving vessels, new structures, and shifting seabeds make static maps go out of date. GNSS data is weak or unavailable underwater, and dead reckoning drifts over time. Even in coastal areas, fjords and valleys can block signals, and ships operating in shallow water want to avoid the bottom with good margins. Because of this, robots need to build and update their own map while they localize in it, this is a SLAM problem. SLAM fuses sensors, handles loop closures to correct drift, and provides a consistent pose and an up to date map during the mission. Sonar is a key sensor for robust navigation underwater, and side scan sonar in particular provides wide swath, high resolution imagery that can reveal seafloor structure and landmarks suitable for data association. Focusing on SSS SLAM lets us study a concrete, relevant problem, how to extract stable features from side scan sonar, associate them across passes, close loops, and feed this information into a SLAM back-end that remains fast and stable over long runs. This matters both for precise mapping and for safe, efficient navigation when external positioning is unreliable.

The motivation is to run state of the art SLAM on real hardware in real time and see how it actually performs. We want to measure accuracy, robustness, and runtime on real AUV data, not just on papers or simulations, and learn what changes are needed to make it reliable at sea. In short, take modern SLAM, make it work on the robot, and judge it by field results.

1.3 Side Scan Sonar SLAM Architecture

SOme images here of basic overvies

Talk abit on SLAM

Then a picture of complex overview

Talk a bit more in depth on slam

2 Sonar Theory

Talk about acoustics
- Talk about Sonar
- Talk about camera and visual odometry stuff and how sonar is used

3 Hardware

Start by intro about AUV and ASV explanation That AUV data for building up the front end and backend and test with some data to verify that the algorithms work Then use this built up ssystem to mold and modify and optimize for ASV

Talk about AUV specs AUV sensors as well AUV Data set that it was collected

Then talk about ASV specs ASV sensors that are important Then talk a bit about ASV software pipeline

4 System Modeling

Introduction that for any navigation system it works best and is built on some assumptions about the movement of the robot and the sensor used, thsi is what we call motion model f() and measurement model h(), thsi is rigid body dynamics. In adition the robot needs to know where it is in relation to itself, its sensors and the world in a coherent and efficient manner, this is rigid body kinematics. For kinematics use the Euler represnetation as its consise and simpel and is videly used in the navigation of ships. For AUVs its more quarternion based that dominates, hwoever since we are doing only mapping euler will work as our dornes will not be maneuvering all the way. Even if euler angles give singularities by deciding how to repensent euler angles smrtly in standard navigation way ie NED represnetation, we can forego using quarternions wich in tehmeselves have their own cavieates. Never teh less we will still be using Quarternions as some sensors give our quarternions and some algorithms work better with quarternion reprensetation, because of that it is usefull to know how to handdle tem as well. Lastly we will have SO3 and SE3 groups as the SLAM map is built using SE3 represnetation, witch is an effective way of builing Envoroemnt over large data sets and works well with rendering as it is also used a lot in computer graphics. So Lie Grpups must be dicussed here as well

States = [pose, linear velocity, angle, angular velocity]

Introduction Kinematics - euler - quarternion - lie groups, SO(3) and SE(3) AUV modelling - Motion f() - Measurents h() ASV modelling - Motion f() - Measurents h()

5 State Estimation

intro to it Bayes filter KF EKF UKF + Alternative Sigma Point generation for better aproximation and stability Some other that might be uselful for later just to mention, like ESKF or UKF for system Idtentification for hydrostatic and parameters and better model aproximation.

6 Preintegration

Uses motion model f() here I think

For SLAM update step when a SOnar picture is made it usually takes 10 seconds or more, in this time span if we fed State estimate data directly into the optimizer we would generate hundreds of odometry factors that are useless as they are ujust connected to each other for the most part until new sonar image has been generated. This woudl mean optimizer part in the backend would have to sift and optimize for unecesarry hundreds of factors that dont contribute much by tehmselves.

Instead what smart peokle have figured out is that one can do method called preintegration where you use smart ways to sum up all the inertial mesurments to create 1 signle precice odometry factor that gets generated at the same time as the sonar pictre is fiched generating, so that we get a single odometry factor that can be conected with multiple landmark factors. This is significatly faster as now only the escential data is compressed withouth wasting optimizers time as well as still relaying all teh importnat information. Preintegration is therefore crucial step to making SLAM efficient

Alos talk about how odometry factors are generated here

7 Local Map Generation

All use State estimates and measuremnt model h() here

Also when generating map locally we build it from sonar 1D images to a 2D recontruction, this takes time and this image is what is then processed and fed into SLAM In adition next step we take with the last 1/3rd or the old image and generate a new image that is 2/3 new, this then becomes new 3/3 image that gets fed into SLAM and so forth This is done so that data asocaition we dont cut off crucial info between frames and Data Sdociation and landmark detection can extract features again here.

Swath Processing

Cartesian Mapping

Feature Extraction (Landmark Detection)

8 Data Association

Gating (Mahalanobis)
- matching/tracking
- Also talk about loop closures
- produce measurement factors for optimizer

9 Optimizers

9.1 Introduction

Optimizers are the engine behind the SLAM update step. Sensors add constraints, but the optimizer decides how the state moves to satisfy them. In practice we solve a Maximum A Posteriori (MAP) problem, we want the most likely trajectory (and landmarks) given all measurements and priors. With some assumptions we will discuss later like Gaussian noise and a first order linearization, MAP turns into nonlinear least squares problem that we solve iteratively. At each iteration we linearize the residuals around the current estimate and compute an increment that improves the state. This "correction" is what lets SLAM reduce drift, enforce loop closures, and keep the map consistent. [3]

There are two dominant families for doing state estimation in SLAM, filtering and smoothing. Filtering methods, like EKF-SLAM (Extended Kalman Filter) and particle-filter SLAM, maintain a rolling belief over the current state only (or a short window). They update online/live as measurements arrive by propagating the state and compressing all past information into the filter's covariance or a set of weighted particles. This is simple and has low memory, but it throws away structure in old constraints and it can become inconsistent after many linearizations, especially when revisiting places (loop closures) or when correlations span long time intervals. Particle filters can represent multi modal beliefs but scale poorly with dimension and often need heavy resampling and clever proposal distributions to avoid degeneracy, something that is difficult to achieve in practice. [4][5]

Smoothing methods keep a dense record of the variables we care about (eks: the whole robot trajectory and, if needed, landmarks) and all the measurement factors that tie them together. Instead of only "where am I now?", smoothing asks "what is the entire trajectory and map that best fits everything we have ever seen?". This global view tends to produce better accuracy and consistency, especially when closing loops or fusing many asynchronous sensors. Computationally, smoothing exposes sparse structure, meaning each measurement only touches a few variables, so the global normal equations are large but very sparse. Modern linear algebra plus careful data structures exploit that sparsity and outperform classical filters on realistic SLAM workloads. That is why smoothing has become the predominant approach in modern SLAM systems.

We keep an estimate of the unknowns, call it θ (robot poses, and landmarks). Each measurement gives an error, or residual, that says how far our current estimate is from what the sensor expects. Close to the current estimate, we can approximate how those residuals change if we nudge θ a little. This is a first order (linear) approximation. Stacking all residuals together, that approximation looks like this [6]:

$$r(\theta + \Delta\theta) \approx A\Delta\theta - b$$

Here A is the Jacobian, it tells us how each residual changes with each variable. The vector b is the residual at the current estimate (with a sign convention so the equation above points toward reducing error). The small vector $\Delta\theta$ is the "correction" we want to compute. [6]

The update step chooses $\Delta\theta$ that reduces all residuals as much as possible. The best practice here is to use MAP approach. Here if we assume measurement noise is Gaussian (it is not always true and we will later discuss how to solve for non Gaussian noise, for now this assumption will suffice). After linearizing the residuals around the current estimate, the MAP problem becomes a least-squares fit [6]:

$$\Delta\theta^\star = \arg\min_{\Delta\theta} \ \|A\Delta\theta - b\|^2 \tag{1}$$

Optimize this estimate so that change in $\Delta\theta^\star$ is equla to 0, ie linearize. When linearized, equation (1) can be simplified to a so called normal equation [6]:

$$A^T A\Delta\theta = A^T b$$

This equation system can be then be solved by Cholesky decomposition of $A^T A$ or by optimization algorithms we will be discussing down bellow. Solve this linear system for $\Delta\theta$, then update/correct the estimate [6]:

$$\theta = \theta + \Delta\theta$$

For stability on harder problems we can add Levenberg-Marquardt damping, but the core idea stays the same across these optimizer algorithms. [7]

Classical batch smoothing forms the full information matrix, eliminates variables in a chosen order, and solves for all states together. That is accurate but not ideal for online use. Every new measurement would, in principle, require rebuilding and refactoring a large system, with cost growing with mission length. Real robots need real-time behavior, so we prefer iterative, incremental smoothing that reuses previous computation. The idea is to keep the factorization of the linearized problem in a data structure that can be updated locally when new factors arrive, only touching the parts of the graph that actually change.

This is where Iterative Smoothing and Mapping (iSAM and iSAM2) methods come in. They exploit that SLAM data are very sparse and mostly locally connected, a new odometry or measurement links a pose to a neighbor pose or a nearby landmark, not to everything. iSAM maintains a square-root factor (via QR) and updates it incrementally using Givens rotations, with occasional reordering to control fill in. It keeps uncertainty queries fast and avoids full resolves, except when needed. iSAM2 goes further by expointing factor graphs and organizing the factorization into a Bayes tree data type (a directed tree of cliques). On new measurements, only the impacted cliques are relinearized and refactored, and variables are reordered incrementally. As a result, work scales with the local update rather than the entire graph, this makes update step "fluid".

In modern SLAM the hard part isn't "doing SLAM", it's solving the SLAM optimization fast as data grows. Most methods use the same MAP correction loop. Linearize, solve for $\Delta\theta^{\star}$, update θ. The real difference is how we represent and update the problem. Smart data structures and good variable ordering keep data structures sparse and decoupled, and solves quick. Meanwhile bad data structure representation of data causes slowdowns.

This is exactly why, for SLAM on marine vessels, especially AUVs with tight space, power, and compute budgets but strict real-time needs, iterative smoothing methods like iSAM2 are a strong fit. They reuse prior factorizations, add new measurements as local factors, relinearize and refactor only the affected cliques, and reorder variables incrementally. In practice that means low latency, bounded memory and CPU load, and accuracy close to batch solutions, even on long missions.

9.2 iSAM

9.2.1 Getting to SLAM update step

Before we can compute a good estimate, we need a simple model of how the robot moves and how sensors observe the world. We use a motion model for state evolution and a measurement model for sensor readings. States are x_i (robot poses), controls are u_i, and measurements z_k (landmarks). We stack all unknowns into θ (poses and landmarks).

Motion (process) model:

$$x_i = f_i(x_{i-1}, u_i) + w_i$$
$$w_i \sim N(0, Q_i)$$

Given the previous state x_{i-1} and control u_i, the next state x_i comes from a model f plus uncertainty noise in the model itself w_i. This uncertainty captures things like currents, slip, and actuator errors. f_i can be a discrete time dynamics update or use plain odometry. Assuming Gaussian w_i is a handy start so MAP becomes least squares. Later we can switch uncertainty model to robust or heavy tailed noise if needed.

Measurement model:

$$z_k = h_k(x_{i_k}, l_{j_k}) + v_k$$
$$v_k \sim N(0, R_k)$$

Each measurement z_k depends on state x_{i_k} and landmarks l_{j_k} transformed using measurement transform function $h_k(\cdot)$, this allows state estimate to become estimated measurement position. In addition this measurement has noise v_k witch we model ad Gaussian noise for simplifications later on when calculating.

Prior:

$$x_0 \sim N(\mu_0, \Sigma_0)$$

A prior anchors the graph (otherwise the problem is underdetermined up to a global transform). It can encode GPS at the start, a known dock pose, or simply a weak "zero" prior to fix gauge.

We want our predictions to match measurements. In a perfect world, every residual (prediction minus measurement) would be zero. In reality we have model errors and sensor noise, so residuals are nonzero. Estimation is about choosing the state update that makes all residuals as small and as statistically consistent as possible.

This is where MAP algorithm comes in. MAP (Maximum A Posteriori) is the principled way to fuse everything we know. A prior on the state, the motion model, and all measurements. It combines them through probability, weighting each residual by its uncertainty. With Gaussian noise, the negative log posterior becomes a sum of squared (weighted) residuals. That gives us a single objective to minimize, where more reliable terms (small covariance) count more. This is better than ad hoc weighting and naturally handles many sensors.

Our motion and measurement functions are nonlinear (angles, rotations, ranges). Minimizing the nonlinear MAP cost directly is hard. Linearization lets us solve it iteratively. At the current estimate we approximate the nonlinear functions by their first order Taylor expansion, solve a linear least squares problem for a small increment, update the estimate, and repeat. This is all shown in the iSAM paper [6] where linearized forms of our system becomes:

$$\begin{aligned} f_i(x_{i-1}, u_i) - x_i &\approx (F_i^{i-1}\Delta x_{i-1} - \Delta x_i) - a_i \\ F_i^{i-1} &:= \left.\frac{\partial f_i(x_{i-1}, u_i)}{\partial x_{i-1}}\right|_{x_{i-1}^0} \\ a_i &= x_{i-1}^0 - f_i(x_{i-1}^0, u_i) \end{aligned} \tag{2}$$

$$
\begin{aligned}
h_k(x_{i-1}, u_i) - z_k &\approx (H_k^{i_k} \Delta x_{i_k} - J_k^{j_k} \Delta l_{j_k}) - c_k \\
H_k^{i_k} &:= \left. \frac{\partial h_k(x_{i_k}, l_{j_k})}{\partial x_{i_k}} \right|_{(x_{i_k}^0, l_{j_k}^0)} \\
J_k^{j_k} &:= \left. \frac{\partial h_k(x_{i_k}, l_{j_k})}{\partial l_{j_k}} \right|_{(x_{i_k}^0, l_{j_k}^0)} \\
c_k &= z_k - h_k(x_{i_k}^0, l_{j_k}^0)
\end{aligned}
\tag{3}
$$

Plug the linearized odometry (2) and measurement (3) models into a single objective over the stacked increment vector $\Delta\theta$ (all pose and landmark updates). The goal is to pick the small change $\Delta\theta^*$ that jointly reduces all linearized residuals. Each factor becomes a linear row in the relevant increments.

For an odometry factor i, the linearized residual is

$$
r_i^{\text{odo}} = F_i^{i-1} \Delta x_{i-1} + G_i^i \Delta x_i - a_i,
$$

where F and G are the odometry Jacobians. Because odometry constrains the relative change between x_{i-1} and x_i, the block on Δx_i is $-I$ ($G_i^i = -I$). Here a_i is the current odometry prediction error.

For a measurement factor k connecting pose x_{i_k} to landmark l_{j_k}, the residual is

$$
r_k^{\text{meas}} = H_k^{i_k} \Delta x_{i_k} + J_k^{j_k} \Delta l_{j_k} - c_k,
$$

with H and J the measurement Jacobians with respect to the involved pose and landmark, and c_k the corresponding prediction error.

Each residual is measured with a Mahalanobis norm $\|r\|_\Sigma^2 := r^\top \Sigma^{-1} r$, using its own covariance, Λ_i for odometry and Γ_k for measurements. This matters because Mahalanobis distance "bakes in" uncertainty. Directions the sensor is confident about are penalized more. Noisy or correlated directions are penalized less, and the metric tilts along correlated axes. As a result, we judge errors not in plain Euclidean meters/radians but in "standard-deviation units" tailored to each factor.. Intuitively, this turns "Euclidean space + covariance" into Mahalanobis space, where the residual ellipses already encode the right weighting. That is why the covariance symbols appear inside the cost, uncertainty is not ignored, it's embedded in how distance is measured. With this, the whole objective of equation (4) is just "add up all these linearized residuals, each judged fairly in its own noise units, and pick the $\Delta\theta^*$ that makes the total smallest". Intuitively, we can think of each factor as a spring pulling on the variables. Mahalanobis scaling makes the springs stiff along low noise directions and soft along high noise ones, so the solution balances all pulls by their reliability.

Collecting all linearized factors with their covariances, the MAP update $\Delta\theta^*$ is obtained by minimizing the following Mahalanobis-weighted least-squares objective:

$$
\Delta\theta^* = \arg\min_{\Delta\theta} \left\{ \sum_{i=1}^{M} \|F_i^{i-1} \Delta x_{i-1} + G_i^i \Delta x_i - a_i\|_{\Lambda_i}^2 + \sum_{k=1}^{K} \|H_k^{i_k} \Delta x_{i_k} + J_k^{j_k} \Delta l_{j_k}) - c_k\|_{\Gamma_k}^2 \right\} \tag{4}
$$

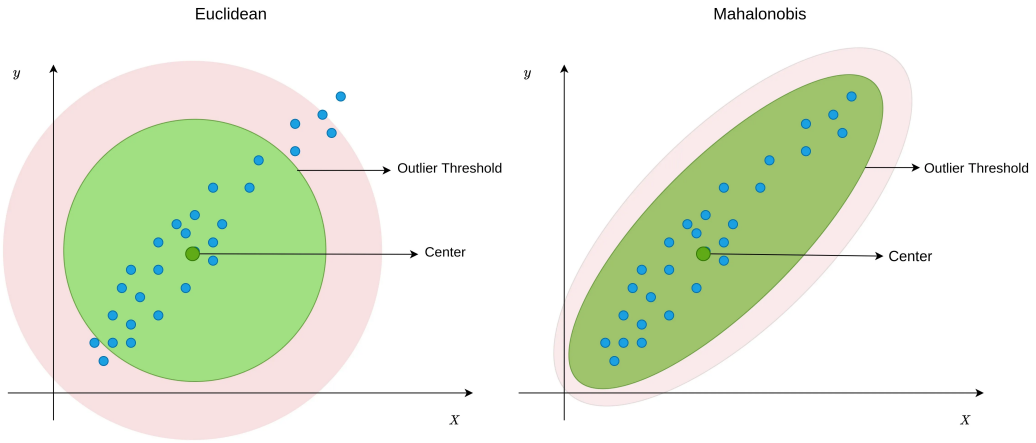


Figure 1: Euclidean vs. Mahalanobis residual contours. *Left:* isotropic (equal) weighting yields circular inlier regions. *Right:* a covariance Σ skews and scales the contours into an ellipse whose axes/tilt follow the noise correlations, whitening this with $\Sigma^{-1/2}$ maps this ellipse back to a circle.[8]

Mahalanobis distance is just "error measured in the units of its noise" (See Figure 1). If a residual has high variance, we shouldn't punish it as much, if two components are correlated, we shouldn't treat them as independent. That's what the covariance does. In the left plot (Euclidean), all directions are weighted equally so the inlier region is a circle. In the right plot (Mahalanobis), directions with low uncertainty are tighter and correlated axes tilt the ellipse. In our SLAM cost, each residual (process or measurement) is evaluated with its own covariance. Small reliable noises count more, whilst large noisy ones count less. When two parts of a measurement drift together, their error isn't along x or y alone, it's along some tilted direction. Mahalanobis tilts the "penalty shape" to match that direction. We are punished less along noisy directions and more where the sensor is precise.

Equation (4) is a sum of Mahalanobis residuals (process terms use Λ_i, measurement terms use Γ_k). To turn that into one clean least squares system, we "whiten" each residual so its noise is unit, for scalars divide by the standard deviation, for vectors apply the covariance's square root inverse to the residual and its Jacobians Σ^{-1}. After whitening, all errors are ordinary Euclidean ones, so we drop the covariance symbols, stack the Jacobians into one big sparse matrix A, stack the prediction errors into b, and solve the standard least squares problem (5).

$$\Delta\theta^\star = \arg\min_{\Delta\theta} \ \|A\Delta\theta - b\|^2 \tag{5}$$

Here, θ stacks all unknowns (robot poses x and landmarks l), A is the single large, sparse (whitened) measurement Jacobian formed by stacking the block Jacobians F, G, H, and J from the linearized motion and measurement models, and b is the stacked prediction error vector that collects the current odometry errors a and measurement errors c with a consistent sign convention. Intuitively, A describes how residuals change for small state perturbations, b encodes the present mismatch between predictions and measurements, and solving equation (5) yields the best local correction $\Delta\theta^\star$ used to update the estimate.

In the linearized setting, the optimal increment $\Delta\theta^\star$ is found by setting the gradient of the least squares objective to zero. This yields the normal equations according to iSAM paper [6]:

$$A^T A\Delta\theta = A^T b$$

Solving this system is typically performed using a numerically stable square root method (QR/Cholesky) rather than forming an explicit inverse. This gives the optimal correction $\Delta\theta^\star$. The state estimate is then updated as follows:

$$\theta \leftarrow \theta + \Delta\theta^\star$$

9.2.2 Incremental QR for fast updates (iSAM)

We solve the linearized SLAM subproblem by least squares. Solving the normal equations $(A^\top A)\Delta\theta = A^\top b$ with Cholesky can be fast but very unstable and ill conditioned as the problem grows (it squares the condition number and increases fill in). iSAM avoids this by working directly with the whitened Jacobian A using QR factorization, and by updating that factorization incrementally when new factors arrive.

Batch square root form (QR on the Jacobian) can be shown in iSAM paper [6] to be of form:

$$A \;=\; Q\begin{bmatrix}R\\0\end{bmatrix}, \qquad Q^\top Q = I, \qquad R\text{: upper triangular}$$

$$\begin{bmatrix}d\\e\end{bmatrix} \;=\; Q^\top b$$

$$\|A\Delta\theta - b\|^2 \;=\; \|R\Delta\theta - d\|^2 + \|e\|^2$$

The iSAM paper [6] shows that after QR we have:

$$A\Delta\theta - b \;=\; \begin{bmatrix}R\\0\end{bmatrix}\Delta\theta - \begin{bmatrix}d\\e\end{bmatrix}, \quad \Rightarrow \quad \|A\Delta\theta - b\|^2 = \|R\Delta\theta - d\|^2 + \|e\|^2.$$

Put simply, once we do QR, the error splits into two parts. To make the total error as small as possible, we make the first part zero by solving:

$$R\Delta\theta^\star = d \tag{6}$$

leaving $\|e\|^2$ as the (minimal) residual norm. If R has full rank, this linearized system has one singular unique solution $\Delta\theta^\star$.

In iSAM the matrix R is upper triangular, so we solve equation (6) by back substitution (no matrix inverse). This gives a fast, numerically stable way to compute the correction and update the state $\theta \leftarrow \theta + \Delta\theta^\star$ without heavy compute.

9.2.3 What is R? The square root information matrix

At the end of QR, the triangular factor R satisfies the following form:

$$R^\top R \;=\; A^\top A.$$

This means $A^\top A$ (the information matrix obtained by linearization) is represented by the "square root" R. Working with R keeps all the curvature of the problem but in a form that is easier to use and numerically safer because R is upper triangular, so computations reduce to cheap substitution methods instead of expensive matrix inverses. Uncertainty can also be extracted directly from R. The state covariance is given by:

$$\Sigma \;=\; (A^\top A)^{-1} \;=\; (R^\top R)^{-1},$$

We never build a dense inverse. When we need entries of the uncertainty Σ, we solve small triangular systems with $R^\top$ and R and read only the pose and pose to landmark blocks we care about. Since R is sparse and triangular, this is fast and stable, and we avoid forming $A^\top A$. (See 9.2.8 Data Association from R)

9.2.4 Matrix Factorization for building QR (Givens rotations)

We use *Givens rotations* to build an upper triangular factor R from the (whitened) Jacobian A by zeroing entries below the diagonal, one at a time. This yields a QR factorization without forming $A^\top A$ and without explicitly storing Q.

A Givens rotation is a 2×2 orthogonal transform applied to two rows (or two columns) to annihilate one chosen entry. Givens rotation matrix is defined as:

$$G(\varphi) = \begin{bmatrix}\cos\varphi & \sin\varphi\\ -\sin\varphi & \cos\varphi\end{bmatrix} \tag{7}$$

Start at the leftmost non zero column of the A matrix and sweep to the right, one column at a time. In each column, pick two rows, *"k"* (the current pivot row) and *"i"* (a row below it), and apply the small "rotate and combine" equation (7) so the entry under the diagonal in that column becomes zero. Only those two rows are mixed, the new row *"k"* becomes a bit of the old row *"k"* plus a bit of row *"i"*, and the new row *"i"* becomes a bit of the old row *"i"* minus a bit of row *"k"*. Repeat down the column until all subdiagonal entries are gone, then move to the next column on the right. (see Figure 2 down bellow for a visual of one Givens step)

As we sweep the columns of A, the matrix is transformed into an upper triangular form, this is R and we never need to build the full Q. Apply the same row rotations to b as you eliminate entries so the right hand side stays consistent. After the initial factorization of A matrix, new measurements don't require rebuilding A. We will illustrate later that we can just append the new (whitened) rows under the current R and apply a short sequence of the same row rotations to re triangularize R matrix again. In other words, updates operate directly on R (and b), we bypass A entirely for incremental steps.

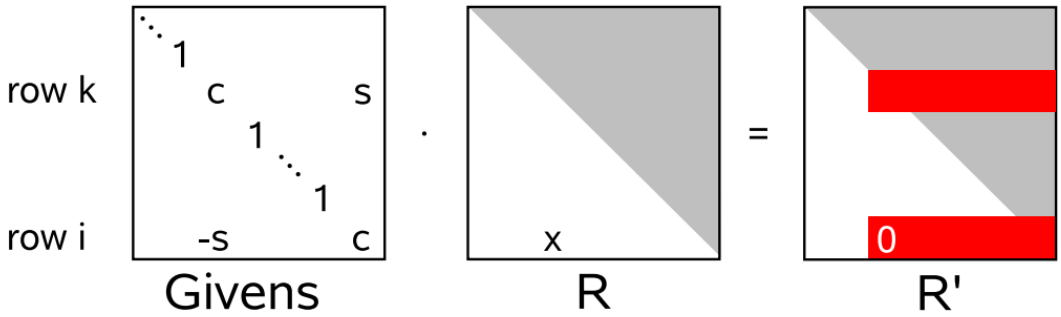


Figure 2: One Givens step in QR. The entry marked "x" is eliminated by rotating two rows, only the entries shown in red are modified, and the exact pattern depends on sparsity. Repeating this column wise (left to right) turns the matrix into an upper triangular R. Apply the same rotation to the b vector to keep the least squares system consistent.[6]

In order to make R upper triangular, we need to get perfect φ value to zero a single sub diagonal entry in preliminary matrix, either be it A matrix on batch step or R matrix on iterative steps. we choose a rotation angle φ from the two numbers we want to combine in the current column, the pivot $x = a_{kk}$ and the subdiagonal $y = a_{ik}$.

$$r = \sqrt{x^2 + y^2} = \sqrt{a_{kk}^2 + a_{ik}^2}$$
$$c = \cos\varphi = \frac{x}{r} = \frac{a_{kk}}{r}$$
$$s = \sin\varphi = \frac{y}{r} = \frac{a_{ik}}{r}$$

Solving for φ gives us the following answer, where $\alpha = x = a_{kk}$ and $\beta = y = a_{ik}$:

$$(\cos\varphi,\ \sin\varphi) = \begin{cases} (1,\ 0), & \text{if } \beta = 0, \\ \left(-\dfrac{\alpha}{\beta}\dfrac{1}{\sqrt{1+(\alpha/\beta)^2}},\ \dfrac{1}{\sqrt{1+(\alpha/\beta)^2}}\right), & \text{if } |\beta| > |\alpha|, \\ \left(\dfrac{1}{\sqrt{1+(\beta/\alpha)^2}},\ -\dfrac{\beta}{\alpha}\dfrac{1}{\sqrt{1+(\beta/\alpha)^2}}\right), & \text{otherwise.} \end{cases} \quad \text{with } \alpha := a_{kk},\ \beta := a_{ik}. \tag{8}$$

These coefficients in equation (8) give the same rotation as (7). They guarantee the (i, k) entry in the working matrix becomes zero, and they do it without changing lengths first for the two number pair $[x,\ y]^\top$ we rotate, and, when embedded, for the affected parts of the two rows (and the matching entries of b). In practice, embed $G_{(i,k)}(\varphi)$ so it acts only on rows k and i, and apply the same rotation to b to keep the least squares system consistent.

9.2.5 Incremental Updating

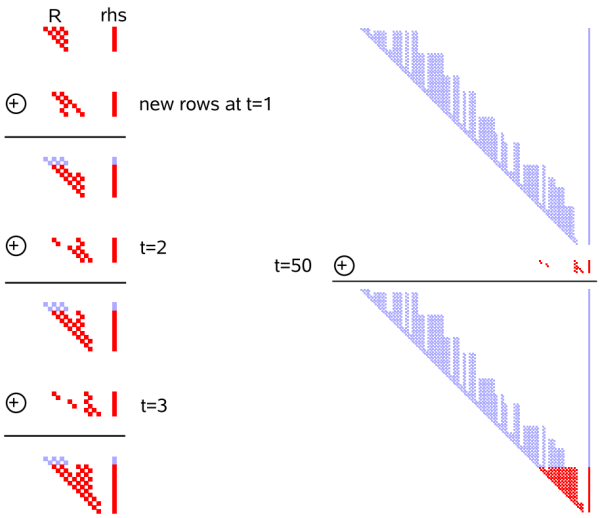


Figure 3: Incremental update of the factored system. A new whitened row $w^\top$ and RHS (Right Hand Side) entry γ are appended beneath the current R and d. A short sequence of Givens rotations restores the upper triangular form, yielding updated R' and d'. Unchanged entries are shown in light color, only a small stencil is touched each step, so update cost stays bounded.[6]

After the initial QR factorization, we maintain the solution in "square root" form, an upper triangular matrix R and a transformed right hand side d. Here, R is the triangular factor that satisfies $R^\top R = A^\top A$ (the Gauss Newton information), and d is the top part of $Q^\top b$. When a new measurement arrives, we first whiten it (divide by its standard deviation or apply the square root information of its covariance) so it has unit variance. The whitened measurement contributes a new row $w^\top$ to the Jacobian and a new scalar γ to the RHS (Right Hand Side). Notice that we do NOT rebuild A. Instead, we append $w^\top$ under the current R, and γ under the current d, which produces a system that is "almost" triangular but has one non triangular row at the bottom.

$$R' = \begin{bmatrix} R \\ w^\top \end{bmatrix}, \qquad d' = \begin{bmatrix} d \\ \gamma \end{bmatrix}$$

Next, we re-triangularize locally with Givens rotations (7). We only touch the columns where the new whitened Jacobian row $w^\top$ has nonzeros (i.e, the variables that this new factor actually connects to, such as a pose x_i or a landmark l_j). Starting from the leftmost such column, each rotation mixes the current pivot row with the new bottom row to kill one sub diagonal entry. We repeat until the entire bottom row is zero and the matrix is upper triangular again. The equation would look something like this:

$$\begin{bmatrix} R \\ w^\top \end{bmatrix} \xrightarrow{\text{Givens rotation on affected columns}} \begin{bmatrix} R' \\ 0 \end{bmatrix}$$

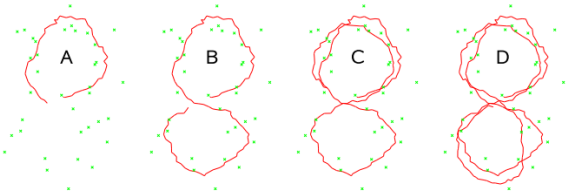
While we rotate the matrix, we apply the same rotations to the right hand side so that the least squares system stays consistent. Here d is the transformed RHS (Right Hand Side) before the update and γ is the new whitened RHS entry that pairs with $w^\top$. After the rotations, the top block becomes the updated RHS d' used for solving, and the final bottom entry becomes a small leftover error e_{new} that adds to the total residual.

$$\begin{bmatrix} d \\ \gamma \end{bmatrix} \xrightarrow{\text{same rotations}} \begin{bmatrix} d' \\ e_{\text{new}} \end{bmatrix}$$

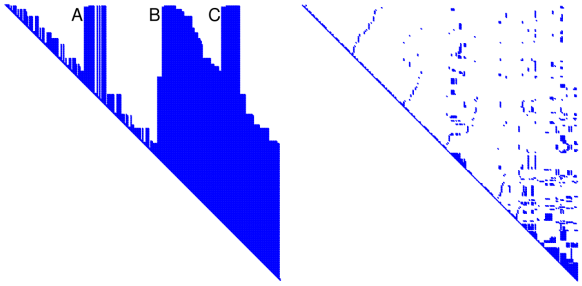
Intuitively, the new row $w^\top$ is "folded up" into the triangular structure by a short chain of 2x2 rotations that only touch the connected variables, everything else is left alone. We then get the correction by a fast back substitution on the updated matrix R' and vector d':

$$R' \Delta\theta^\star = d'$$

9.2.6 Loop Closure

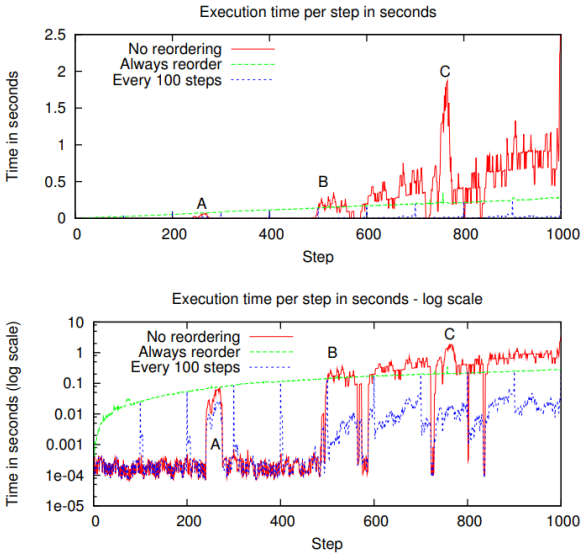


(a) Simulated double 8 loop at key loop closure moments.[6]



(b) Upper triangular factor R after several closures shows fill-in.[6]

(c) The same R after variable reordering (COLAMD) becomes sparser again.[6]



(d) Per step execution time for three strategies. A: no reordering, B: reorder every step, C: reorder every 100 steps, shown in linear (top) and log (bottom) scale. Periodic reordering (C) limits spikes and keeps runtime predictable between loop closures.[6]

Figure 4: Effect of loop closures and variable reordering (iSAM).[6]

Loop closures tie together far-apart parts of the trajectory (and landmarks), which makes previously separate columns interact. In QR terms this creates fill-in, R gets extra non zeros, so updates, back substitution, and selected covariance queries get slower and memory grows. We fix this by variable reordering. What we do is pick a new elimination order that preserves sparsity. In practice we run a heuristic like COLAMD (Column Approximate Minimum Degree) (often the block version for pose/landmark blocks) on the Jacobian's sparsity matrix A, then do batch factorization on the whole A matrix using rotations (7) with that order [6]. Reordering costs time because we rebuild the factor with a new permutation, but it pays back by making the next many updates cheap again.

Because reordering is expensive, we don't do it every step. Instead we do periodic reorder every N steps (eks, 50 - 200) so cost stays predictable. In marine AUV runs this keeps compute bounded. Between reorders incremental updates are fast and local. After a loop closure, we accept one spike (reorder + refactor), then return to low latency. Practical tips when doing this step is to keep poses as blocks (block COLAMD) to reduce fill-in, align reordering with planned relinearization passes, and monitor simple stats (nonzeros in R, update time) to decide when N is too small (wasting time reordering) or too large (letting fill in snowball).

9.2.7 Re-Linearization

Re-linearization keeps the local model honest. All the QR/update tricks we have gone through now assume the system is locally linear around the current estimate, but with angles, 3D motion, and nonlinear sensors that linearization drifts as the robot moves. If we never refresh it, increments stop being small, the optimizer biases the map, and loop closures can break the solution. The fix is to re-linearize, recompute Jacobians at the current state for the factors that matter. Doing this for every factor at every step is too expensive, so in practice we always linearize new factors (fresh odometry and measurements) and refresh older ones only when needed.

In iSAM the practical schedule is to perform incremental updates between maintenance cycles, then do a full re-linearization at a fixed interval N. Between cycles we do not re-linearize old factors, we only whiten and insert the new ones and update R incrementally. At the cycle boundary when we hit N steps, we re-linearize the entire problem at the current estimate (conceptually rebuild the full Jacobian

A), run a variable reordering to restore sparsity using COLAMD, and refactor using equation (7) to get a fresh triangular R. Thats why we usually bunch variable reordering with batch linearization in the same N step.

We must choose N carefully, by balancing freshness vs compute. If N is too large, the linearization point drifts far from reality, Jacobians no longer match the true geometry, corrections become biased, loop closures pull hard, and the map can warp (a classic "stale linearization" issue). If N is too small, we keep stopping to relinearize and refactor, burning CPU and power and hurting real-time throughput.

9.2.8 Data Association from R

For data association we often use a Mahalanobis based approach instead of plain nearest neighbour Approach. Nearest neighbour measures raw Euclidean distance and ignores sensor noise and correlations. Mahalanobis measures the innovation in the units of its uncertainty, so noisy directions count less, precise directions count more, and correlated components are handled correctly (See Figure 1). For a candidate match between current pose x_i and landmark l_j, form the innovation ν_k and score:

$$d_k^2 = \nu_k^\top \, \Xi_k^{-1} \, \nu_k, \qquad \Xi_k = J_k \, \Sigma \, J_k^\top + \Gamma_k$$

where $J_k = [\, H^{x_i} \;\; H^{l_j} \,]$ is the linearized measurement Jacobian, Γ_k is the sensor noise, and Σ is the state covariance.

This type of data association often uses gating with a chi-square test. The test keeps matches that are within $d_k^2 \le \chi^2_{m,\alpha}$ (right dimension m, chosen confidence α). From the survivors, we pick the "minimum cost" one (or solve a global assignment using d_{ij}^2 as the cost matrix if several features compete). One thing to note about this approach is that we don't need the full dense $\Sigma = (R^\top R)^{-1}$ to do our data association, we only require local covariances touched by the measurement. These types of covariances can be pull directly from the square root information matrix R. Here we can get pose, and pose to landmark terms are available live. Landmark terms on the other hand are either a approximate fast conservative estimate or computed exactly on demand. This keeps the association real-time for the most part. Here are the partitioned forms. Full state covariance (poses vs. landmarks):

$$\Sigma \;=\; \begin{bmatrix} \Sigma_{xx} & \Sigma_{xL} \\ \Sigma_{Lx} & \Sigma_{LL} \end{bmatrix}$$

where Σ_{xx} is pose to pose, Σ_{LL} is landmark to landmark, and $\Sigma_{xL} = \Sigma_{Lx}^\top$ is pose to landmark.

The 2×2 submatrix needed for a single candidate (x_i, l_j):

$$\Sigma_{\{x_i, l_j\}} \;=\; \begin{bmatrix} \Sigma_{x_i x_i} & \Sigma_{x_i l_j} \\ \Sigma_{l_j x_i} & \Sigma_{l_j l_j} \end{bmatrix}$$

Fast marginals from the square root factor (online/live)
In iSAM paper [6], they propose keeping the current pose last in the ordering. Then the covariances needed for association, the pose variance $\Sigma_{x_i x_i}$ and the pose to landmark cross terms $\Sigma_{x_i l_j}$ come straight from the square root factor R with two small triangular solves:

$$R^\top Y = B, \qquad RX = Y$$

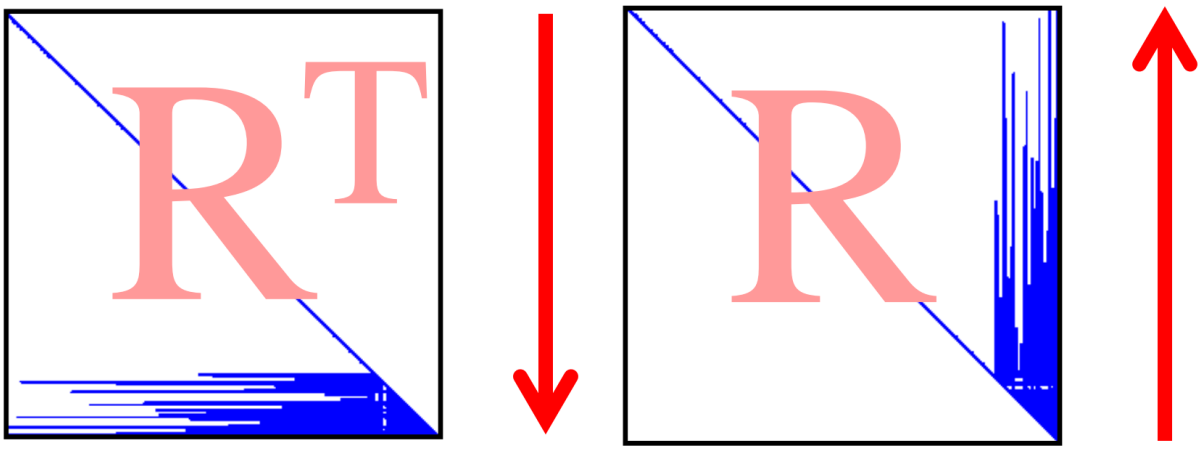
where $B = \begin{bmatrix} 0 \\ I_{d_x} \end{bmatrix}$ simply selects the last pose block of size d_x. Because R is upper triangular and B is zero above the last block, the forward solve gives:

$$Y = \big[\, 0, \ldots, 0, \;\; R_{ii}^{-1} \,\big]^\top$$

This Y preliminary matrix is the clue, where only d_x back substitutions are needed to get full X vector. Reading the result X yields, in one pass:

$$\Sigma_{x_i x_i} \;\; \text{(bottom–right block of } \Sigma) \qquad \Sigma_{l_j x_i} = \Sigma_{x_i l_j}^\top \;\; \text{for connected } l_j$$

Intuitively we "solve up" then "solve down" on R for the last pose, and we get exactly the columns of $\Sigma = (R^\top R)^{-1}$ that matter for gating in data association. This can be done every step without forming any dense inverses, only using R square root information matrix iteratively. (See Figure 5)



(a) $R^\top Y = B$ (forward substitution). Sweep "down" the matrix to form Y from a selector B that picks the last pose block.

(b) $RX = Y$ (back substitution). Sweep "up" the matrix to obtain the desired columns $X = \Sigma_{:,x_i}$ (pose and pose to landmark).

Figure 5: We grab the needed covariances in two quick steps: first solve "down" with $R^\top$, then solve "up" with R. We only touch entries near the last pose block, so each update stays fast and cheap.[6]

Conservative landmark covariances (online/live)
Exact landmark landmark blocks Σ_{jj} (or old pose to landmark $\Sigma_{(i-n)j}$) are expensive to extract at every step. Therefore in iSAM paper [6] they propose using a safe, conservative bound built from the current pose covariance and the measurement noise via the linearized back projection:

$$\tilde{\Sigma}_{jj} \;=\; \bar{J} \begin{bmatrix} \Sigma_{ii} & 0 \\ 0 & \Gamma \end{bmatrix} \bar{J}^\top$$

where $\bar{J}$ is the Jacobian of the local inverse measurement model, Σ_{ii} is the current pose covariance, and Γ is the measurement noise. This upper bounds landmark uncertainty (never over confident), is fast, and works well for online Mahalanobis gating. As more measurements of a landmark arrive, this bound typically tightens.

An important caveat to mention is that we usually don't know the true Γ (measurement noise) exactly. The iSAM recipe is to choose Γ conservatively so we don't get over confident. That keeps things safe but can make the gate too tight, causing the data association to reject more matches than it should. Later, we will discuss a more confident way to approximate Γ from measurement characteristics without being overly confident (eks: range/resolution for sonar and landmark confidence) so the gate is realistic without being risky.

Exact landmark covariances (on demand)
When we need accuracy (eks: on risky loop closure, conflicting hypotheses), Data Association can recover exact Σ_{jj} and $\Sigma_{(i-n)j}$ without forming the full dense inverse information matrix $\Sigma = (A^T A)^{-1} = (R^T R)^{-1}$. Because the covariance is the inverse of the information matrix:

$$\Sigma = (R^\top R)^{-1}, \qquad R^\top R \Sigma = I$$

We can get the needed entries without forming the full inverse by solving for two triangular matrixes:

$$R^\top Y = I, \qquad R\Sigma = Y$$

We only follow the "nonzeros" of R when solving. That means we don't touch the whole matrix, just the parts that matter. iSAM walks backwards along those non zero links and gives us exactly the covariance numbers σ_{ij} Data Association ask for. If R is mostly banded, this is near linear time. We should only use

this exact method when we really need it (eks: a few Σ_{jj} blocks to check on a loop closure). It's slower than the conservative shortcut, but still much faster than inverting the whole matrix.

Alternative way to finding Γ
There's another angle. Instead of being very conservative with our Γ measurement noise. Uncertainty used in Data Association should reflect the sensor, for example a sonar for scanning the sea floor. With sonar we can make the measurement noise grow with range and use the transducer's resolution to set per axis variances $N(0, \Sigma_{sonar})$. The landmark detection can also contribute its own uncertainty $N(0, \Sigma_{landmark})$. In practice we could combine "sensor noise" and "landmark estimate noise" to get the prediction uncertainty for the residual we score. In that case we can say:

$$\Gamma = Var(h(x, l)) = \Sigma_{sonar} + \Sigma_{landmark}$$

This can be more informative than a one size fits all setting. However we must be very careful with this approach, if we make these noises too optimistic, the gate in Data Association gets too tight, associations become brittle, and the system can go unstable. Never the less this is a lot better approach for building more accurate maps and estimating robots track.

9.2.9 Algorithm

At each time step we absorb new information, linearize around the current estimate, solve a small least squares, and update. Periodically we refresh linearization and variable order. Concretely:

1. **Add factors (whiten first):** take the new odometry/measurements and add them to the graph. "Whiten"" them so every residual has unit noise (as described before (5)).

2. **Linearize at the current guess θ:** turn the nonlinear motion/measurement models into local linear ones using the Jacobians in (2) and (3). This gives the summed Mahalanobis cost (4), which after whitening becomes one least squares problem (5).

3. **Keep a triangular system up to date (QR):** append the new (whitened) rows and apply a few Givens rotations (7) so the matrix stays upper triangular R. Update the right hand side b vector the same way (see "Incremental Updating").

4. **Solve for the small change:** because R is triangular, solve (6) by back substitution (fast) to get the correction .

5. **Update the estimate:** replace the old state with the improved one, $\theta \leftarrow \theta + \Delta\theta^{\star}$.

6. **Every N steps (maintenance):** refresh accuracy by re-linearizing all factors at the new θ, reorder variables (eks: COLAMD algorithm) to keep things sparse, and refactor with Givens (7) to get a clean R.

7. **Data Association update:** read the needed covariances from R (pose and pose to landmark live, landmark blocks conservative or exact on demand) and run Mahalanobis gating in Data Association for next matches.

9.2.10 Limitations

iSAM is fast between updates but has practical downsides. They come from the "data structure", not the underlying SLAM algorithm. iSAM keeps a single, global square root information matrix R (from $A^{\top}A$) and does periodic maintenance (reordering + re-linearization). This makes updates simple, but couples cost to global structure and variable ordering instead of just local changes.

- **Latency spikes at maintenance:** Periodic global variable reordering and re-linearization trigger stalls (especially after loop closures), since R must be refactored end to end.

- **Fill in growth between reorders:** Incremental QR on a fixed order accumulates fill in in R, touching more entries per update and increasing time/memory step by step.

- **All or nothing re-linearization:** iSAM typically refreshes many factors at maintenance even if most variables barely moved, wasting Jacobian recomputations.

- **Global refactor on ordering changes:** Any change to elimination order implies a large refactor of the global R, regardless of how small the new information is.
- **Broad marginal queries are costly:** Last pose and nearby cross terms can be pulled quickly from R, but wide Σ blocks (eks: many landmarks or older poses) require multiple triangular solves and can be costly.
- **Schedule sensitivity:** Choosing "every N steps" for reorder/re-linearize is heuristic, too small wastes time, too large lets fill in and linearization error grow, causing jitter and warp in the map.
- **Numerical robustness vs simplicity:** Working with A and R avoids explicit $A^\top A$, but long incremental runs plus fill in can still hurt conditioning and stability if ordering lags.

These limitations motivated **iSAM2**, which replaces the single monolithic R that is very static, with a Bayes tree representation that is dynamic and updates only the affected parts. We address iSAM2 and how it mitigates the issues above in the next chapter.

9.3 iSAM2

9.3.1 Introduction and Motivation

iSAM2 was developed to overcome the practical limitations of iSAM. The core optimization method in iSAM (nonlinear least squares solved through incremental QR factorization) is sound and provides accurate solutions. The bottlenecks arise not from the underlying algorithms, but from the static data structure used to maintain the problem. In iSAM the system is stored as a single global square root information matrix R. While this representation is compact and efficient for batch updates, it leads to several issues during incremental operation.

First, the R matrix is "global". Adding new factors or loop closures often changes many rows and columns, which requires expensive refactorization. This causes latency spikes, especially when loop closures occur and large parts of the trajectory suddenly become coupled. Second, relinearization in iSAM is also global. To maintain accuracy, the system periodically refreshes Jacobians for all factors, which forces full reconstruction of the R matrix. Third, variable reordering to reduce fill in and keep R sparse is again an all or nothing operation, with cost proportional to the entire problem size. Together, these properties mean that iSAM, while efficient between updates, still suffers from periodic heavy computation that disrupts real time performance.

iSAM2 addresses these limitations by introducing a new data structure, the "Bayes tree". Instead of representing the system as a static R matrix, iSAM2 leverages the factor graph formulation of SLAM, applies variable elimination, and interprets the resulting structure as a chordal Bayes net. This Bayes net can then be compactly represented as a Bayes tree, which retains all probabilistic information while enabling local updates. With the Bayes tree, adding new measurements or relinearizing states only modifies the affected cliques in the tree, leaving the rest untouched. This local property eliminates the global refactorization bottlenecks of iSAM, smooths out computation over time, and makes the algorithm scalable to large and long term mapping problems. [7, 9]

9.3.2 Factor Graphs

A factor graph is a bipartite graph that connects variable nodes (poses and landmarks) to factor nodes (priors, motion, and measurements). It encodes the same estimation problem as in iSAM, but makes sparsity and locality explicit because each factor touches only a few variables.

If we let the variables be robot poses $x_1, \ldots, x_M$ and landmarks $l_1, \ldots, l_N$, and let $\Theta = \{x_1, \ldots, x_M, l_1, \ldots, l_N\}$. According to the iSAM2 papers [7, 9] the posterior factorizes as

$$f(\Theta) = \prod_i f_i(\Theta_i),$$

Here, each factor f_i depends only on its adjacent variables Θ_i. The way we model each factor depends on the situation, however since we are dealing with real world where our odometry and measurement variables are uncertain, we should use a probabilistic model to represent factors. One of the easiest probabilistic models is Gaussian, which is easy to model and will work well later when working in Mahalanobis form to optimize the problem and solve it in simple manner (See Figure 1 for Mahalanobis form). Factor functions can therefore be modeled as follows:

$$\text{Prior:} \quad f_p(x_0) \;\propto\; \exp\Big(-\tfrac{1}{2}\|\mu_0 - x_0\|^2_{P_0}\Big),$$

$$\text{Odometry:} \quad f_i(x_{i-1}, x_i) \;\propto\; \exp\Big(-\tfrac{1}{2}\|f_i(x_{i-1}, u_i) - x_i\|^2_{Q_i}\Big),$$

$$\text{Measurement:} \quad f_k(x_{i_k}, l_{j_k}) \;\propto\; \exp\Big(-\tfrac{1}{2}\|h_k(x_{i_k}, l_{j_k}) - z_k\|^2_{R_k}\Big).$$

This is in a way very similar to how iSAM models the system, however here in iSAM2, the system is represented as factor graphs. This graph view exposes conditional independence directly in the topology. Each factor only connects nearby variables in time or space, which later yields a sparse linear system.

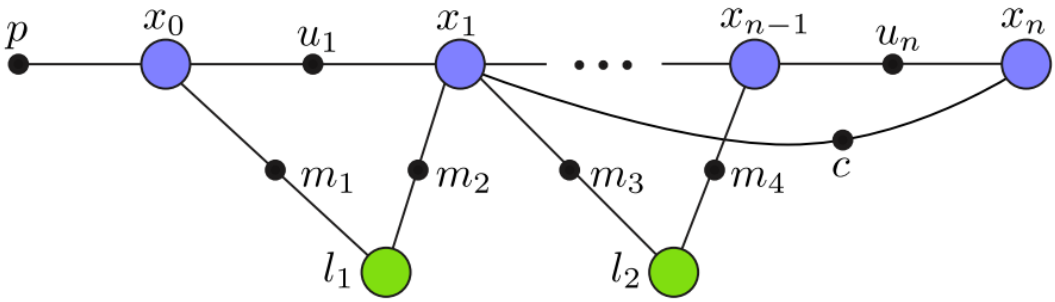


Figure 6: Picture from iSAM2 paper [7] describes factor graph formulation of the SLAM problem. Variable nodes (large circles) are poses $x_0, \ldots, x_n$ and landmarks l_1, l_2. Factor nodes (small solid circles) represent a prior p, odometry u_i, landmark measurements m_i, and a loop closure constraint c. We can represent any cost function, including factors that connect more than two variables.

This example shows how local connectivity induces sparsity. This will be useful when solving the optimization problem later down bellow. Each factor touches only its adjacent variables, so after linearization (to compute a local optimum) the Jacobian rows have nonzeros only in those columns. The resulting matrix is sparse, which helps the optimizer.

9.3.3 From Factor Graphs to the SLAM Optimization Problem

We represent SLAM with a factor graph $G = (\mathcal{F}, \Theta, \mathcal{E})$. There are two node types, factor nodes $f_i \in \mathcal{F}$ that encode pieces of information (prior, odometry, measurements, loop closures), and variable nodes $\theta_j \in \Theta$ that hold unknowns (poses, landmarks, calibration). An edge $e_{ij} \in \mathcal{E}$ is drawn only when factor f_i depends on variable θ_j. This wiring is important, missing edges mean independence, and the pattern of edges controls which variables interact in the estimation problem.

The graph specifies how a global objective splits into simple parts:

$$f(\Theta) \;=\; \prod_i f_i(\Theta_i),$$

Here Θ_i collects only the variables that touch factor f_i. In the SLAM setting, a prior p anchors the first pose, odometry factors u relate consecutive poses, landmark factors m couple a pose with a landmark, and loop closure factors c link poses that see the same place again (see Picture 6). The same framework can also handle factors that involve three or more variables, for example a factor that ties a pose to a landmark and a camera intrinsics block (calibration), or "separator" variables shared in cooperative mapping. The key point is that the graph can host any cost term as long as we state which variables it touches.

Under Gaussian measurement models as we defined in the previous subsection, each factor has the form:

$$f_i(\Theta_i) \;\propto\; \exp\Big(-\tfrac{1}{2}\,\|\,h_i(\Theta_i) - z_i\,\|^2_{\Sigma_i}\Big),$$

This Gaussian form will simplify calculations as Gaussian is nice to work with. Here $h_i(\cdot)$ predicts what the sensor should see from the current variables Θ_i, z_i is the actual measurement, and Σ_i is the measurement covariance. The notation $\|e\|^2_\Sigma \triangleq e^\top \Sigma^{-1} e$ is the squared Mahalanobis distance, which measures error in "units of its noise" (directions with low variance are penalized more). (See Picture 1)

To combine all information, we multiply the factor likelihoods. Products are awkward to optimize, so we take a negative logarithm to turn the product into a sum. Terms that do not depend on Θ drop out, and each Gaussian factor becomes a squared Mahalanobis residual weighted by its covariance. The result is one scalar objective that collects the prior, all odometry factors, all landmark measurements, and any loop closures. Small covariances make a factor count more, large covariances count less. In short, multiply the factors and take the negative log to get a single sum of squared errors. We write it compactly as:

$$\Theta^\star \;=\; \arg\min_\Theta \;\frac{1}{2}\sum_i \|\,h_i(\Theta_i) - z_i\,\|^2_{\Sigma_i}$$

All variables (poses, landmarks, and any auxiliary parameters) live inside the same cost, and only variables that share a factor appear together in a residual, which later yields a sparse linear system when we linearize.

This is our MAP function in nonlinear form.

We do not solve this nonlinear cost in one shot because $h_i(\cdot)$ measurement transform is nonlinear (because of angles, ranges, bearing-only sensors, etc...). Instead we solve it in similar fashion we solved iSAM system model, we linearize and solve iteratively. Choose a current estimate Θ^0 and look for a small update $\Delta\theta$ that improves the fit. For each factor we take a 1st-order Taylor expansion around Θ^0, which turns that factor into a simple linear residual in the increment $\Delta\theta$ that touches only its adjacent variables. After whitening by $\Sigma_i^{-1/2}$, we stack all linearized factors into one sparse least-squares problem:

$$\Delta\Theta^\star = \arg\min_{\Delta\Theta} \left(-log\left(f(\Delta\Theta)\right)\right) = \Delta\theta^\star = \arg\min_{\Delta\theta} \ \|A\,\Delta\theta - b\|^2 \tag{9}$$

Here $A \in \mathbb{R}^{m\times n}$ is the measurement Jacobian (one row block per factor), b stacks the whitened prediction errors, and $\Delta\theta$ is the n-dimensional increment. The sparsity pattern of A is dictated by the factor graph, a row has nonzeros only in the columns of the variables that appear in that factor.

What we can do here is solve this linear least squares problem in a numerically stable way. One route is the normal equations $A^\top A\,\Delta\theta = A^\top b$ and a Cholesky factorization $A^\top A = R^\top R$ followed by forward and back substitution to recover $\Delta\theta$. Another route is QR factorization on A, which yields an upper triangular system $R\,\Delta\theta = d$ that we solve by back substitution. Both routes are standard, in practice we prefer square-root methods (QR/Cholesky) for stability.

After solving for $\Delta\theta$, we update the state $\theta \leftarrow \theta + \Delta\theta$ and repeat, linearize, solve, update. This is Gauss-Newton. If the problem is difficult (poor linearization, strong nonlinearity), we add Levenberg-Marquardt damping and instead solve $(A^\top A + \lambda I)\Delta\theta = A^\top b$, which blends Gauss-Newton with a trust-region step to keep updates safe. We stop when $\Delta\theta$ is small or the cost no longer decreases.

This is the same least squares core as in iSAM. The difference is the data structure. iSAM works with the Jacobian A and its triangular factor R. iSAM2 keeps the factor graph as the main object, which carries the same information as A but in a graph form. We eliminate variables to get a Bayes net and store it as a Bayes tree. This lets us update only the cliques touched by new factors instead of refactoring everything. In the next subsection we will see how it is done.

9.3.4 From Factor Graphs to Bayes Networks

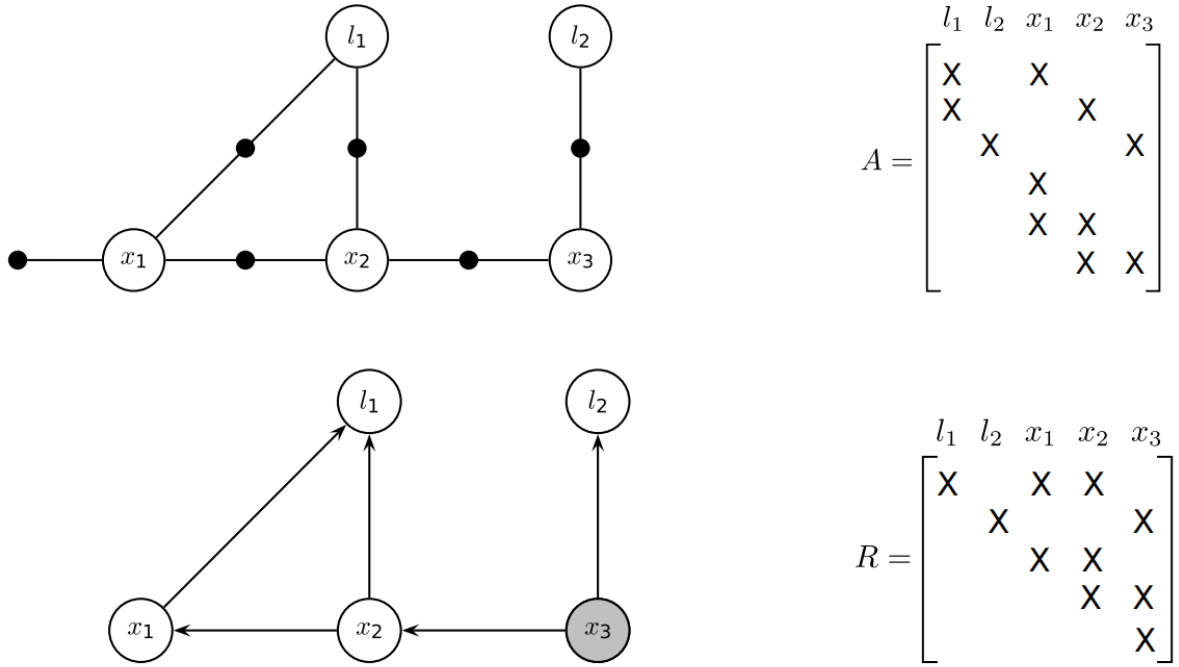


Figure 7: Picture from iSAM2 paper [7]. (Top) factor graph and associated Jacobian A for a small SLAM example with poses x_1, x_2, x_3, landmarks l_1, l_2, and a prior on x_1 (Bottom) the chordal Bayes net and the square root factor R obtained by eliminating in the order l_1, l_2, x_1, x_2, x_3. The last eliminated variable (the root) is shaded. [7]

Starting from the least squares form in (9) we can factor the (whitened) measurement matrix A exactly as in iSAM. We pick an elimination order (for the example in Picture 7 it is l_1, l_2, x_1, x_2, x_3) and run sparse QR with Givens rotations (7). This produces an orthogonal Q and an upper triangular R. Because R is triangular, solving by back substitution proceeds variable by variable in the chosen order. That solve can be read as a chain of simple Gaussian conditionals, one per variable, where the off diagonal entries to the right of each pivot indicate which previously eliminated variables that conditional depends on. If we draw arrows from those "parent" variables to the current one, the same structure becomes a directed graphical model. In short, for a chosen ordering, sparse QR on the factor graph yields an R that encodes a Bayes network, this is what the bottom panel of Picture 7 shows.

Rather than running sparse QR on the whitened Jacobian, we can reach the same end result by eliminating variables directly on the factor graph by bipartite elimination game methods instead [10]. Starting from (9), choose an order, then for each variable collect its adjacent factors, combine them, marginalize that variable out, and attach the resulting factor to the remaining neighbors. Repeat until all variables are removed. For any fixed order this purely graphical elimination produces exactly the same dependency pattern and the same square root information factor we would get from numeric QR factorization. This method directly forms a Bayes network with properties of chordal, and in the matrix view we obtain an upper triangular R with $R^\top R = A^\top A$ (see Picture 7). The advantage is that we never have to assemble A or apply Givens rotations (7) and do QR factorization. We stay on the graph, let the ordering control fill, and arrive at the chordal structure we want. In practice this is the same QR algebra, just done in a graph aware way that avoids unnecessary intermediate fill and work according to Good Column Orderings for Sparse QR Factorization paper [10].

This observation is the bridge to iSAM2. A chordal Bayes network groups naturally into cliques, so we can represent it as a Bayes tree. In the next subsections we use this tree, instead of one global R, to support local updates and avoid global refactorizations.

9.3.5 R as a Bayes tree data structure

A key outcome of variable elimination on the SLAM factor graph is a chordal Bayes network. Chordal means that in the moralized (undirected) view every long cycle has a shortcut edge, which keeps parents of a variable grouped in small cliques and prevents excessive fill in during elimination. This property is what makes the square root factor R sparse and tractable, and it also sets up a clean bridge to a tree representation of the same information. [7, 9]

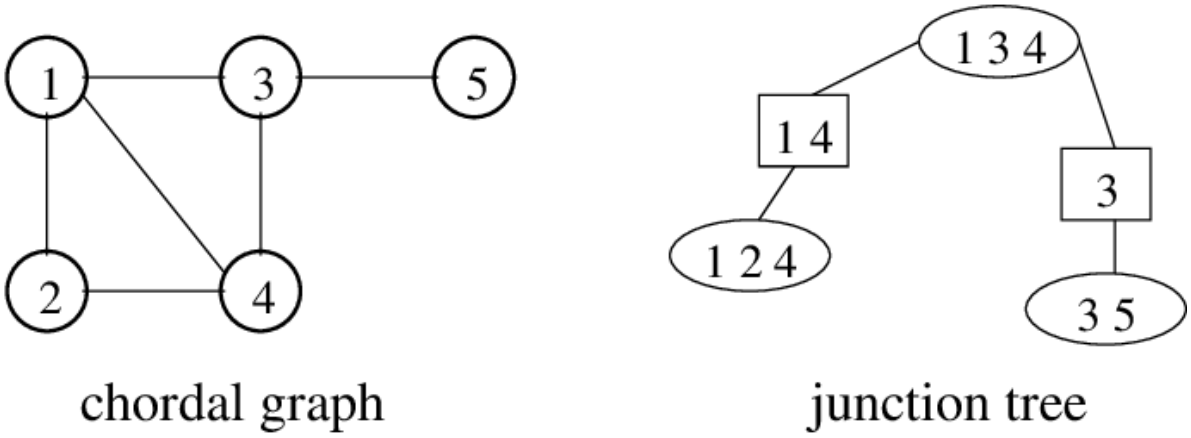


Figure 8: Picture from Robert Castelo paper [11] shows an example of a simple Chordal graph (Left side) and corresponding small cliques/junction tree (Right side). Eliminating in a good order produces a chordal Bayes net whose moralized graph can be grouped into cliques/junction trees. This is the structure that keeps R matrix sparse.

When we linearize and eliminate in some order, the resulting triangular system R still satisfies $R^\top R = A^\top A$, but each row block of R matrix now carries a specific meaning. Each row block of R belongs to the variable we just eliminated. That row says, in Gaussian form, how this variable depends on a few variables that were eliminated earlier. Think of those earlier variables as its parents. When we solve $R\Delta\theta = d$ by back substitution we start at the last row and move upward. That is the same as computing each variable from its parents in turn. So R is not just numbers in a triangle. It is the same set of conditional relationships as the Bayes network, written in matrix form.

Because the Bayes net is chordal, its conditionals naturally group into cliques. If we collect consecutive row blocks of R that share the same separator, each group is one clique. Connect two cliques when they share that separator, and we obtain a Bayes tree. Every clique node stores "frontal *child*|*parent* separator" conditionals, and every edge carries only the shared separator variables. The tree therefore encodes the same numeric information as R, but organized by locality rather than by a single global ordering.

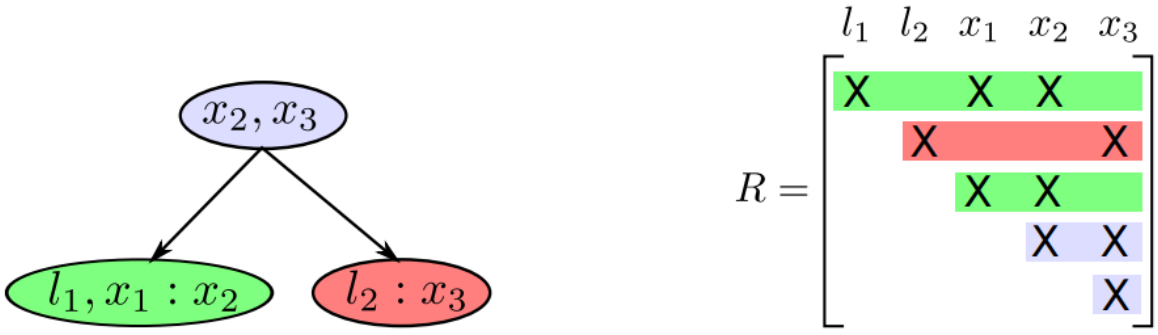


Figure 9: Picture from Bayes tree paper [9] shows Bayes tree (left) and its matching rows in the square root factor R (right). Colors indicate which contiguous row blocks of R belong to each clique.

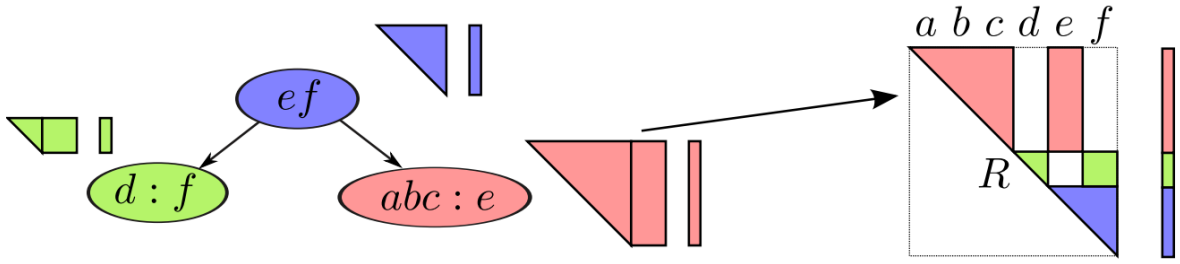


Figure 10: Picture from iSAM2 paper [7] shows each clique contains the conditional of its frontal variables given the separator. The same entries appear in the corresponding rows/columns of R. Back substitution on R mirrors evaluating the tree from leaves to root, while marginal queries follow the few cliques that touch the queried variables.

The takeaway is simple. Eliminating a factor graph gives a chordal Bayes net. Grouping its conditionals yields a Bayes tree. The square root factor R contains the very same conditionals in matrix form. Hence we can think of R as a Bayes tree data structure written as a matrix. iSAM2 stores and updates this Bayes tree directly instead of one global R, which preserves the numerical benefits of the square root form while enabling strictly local updates. This means when new measurements arrive or when some variables must be re-linearized, only the cliques on a small subtree are touched and the rest of the structure stays unchanged, making computational complexity manageable and the data set grows.

9.3.6 Incremental Updates directly on Bayes tree

In iSAM2 we never rebuild the whole system when new data arrives. A new odometry or landmark factor only touches a few variables in the factor graph, so only the matching subtree in the Bayes tree is modified. All other branches stay exactly the same. This is the key difference from iSAM, where adding a factor could force a global re-factorization of the single R matrix.

Two simple rules explain why updates stay local. First is that information flows upward in the tree during elimination, so changes propagate only from the touched variables toward the root. Second, a factor becomes active when the first variable in its local elimination order is eliminated, so only the paths from those variables up to the root can be affected, while unrelated subtrees remain untouched.

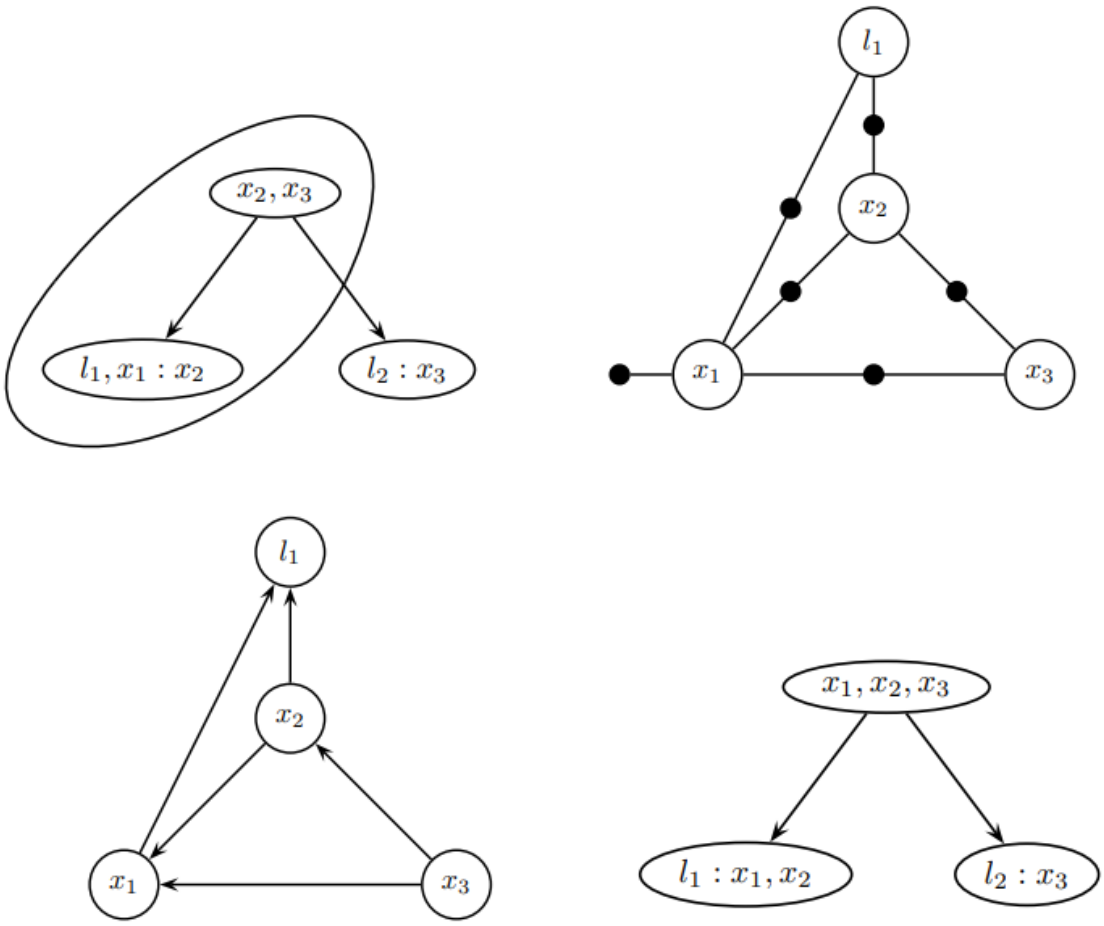


Figure 11: Picture taken from Bayes tree paper [9] shows how to update a Bayes tree with a new factor.
(Top left) Affected cliques when adding a factor between x_1 and x_3, the right branch is unaffected.
(Top right) Local factor graph rebuilt from those cliques plus the new factor.
(Bottom left) Local chordal Bayes net after elimination.
(Bottom right) New Bayes subtree with the untouched "orphan" subtree reattached.

One update works as follows. first, locate the cliques that contain the variables touched by the new factor and follow their ancestors up toward the root, this marks the only region that needs work, while the rest of the tree becomes "orphans" that stay valid and untouched. Next, convert the conditionals stored in those marked cliques back into a small local factor graph and insert the new factor. Then re-eliminate just this local graph (using the same graph aware elimination as discussed previously) to produce a new chordal Bayes net and its updated Bayes subtree. Finally, reattach the orphan subtrees at the proper separators. Only this small subtree changes, everything else is reused. (See Picture 11)

The result is incremental and predictable computation. iSAM2 edits only the cliques touched by the new information, runs a small local elimination, and solves by back substitution along that subtree. There are no global re-factorization spikes, and accuracy is maintained because we can also re-linearize only the variables inside the same affected region when needed.

9.3.7 Loop Closure and Incremental Reordering

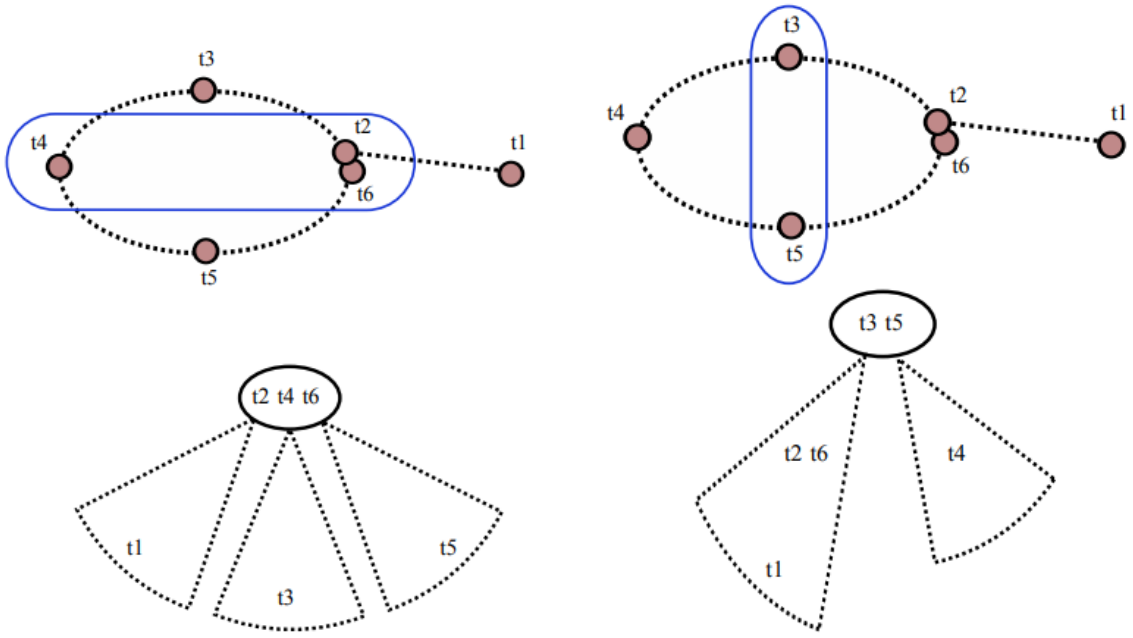


Figure 12: Picture taken from Bayes tree paper [9] shows loop closure with incremental reordering. Two batch optimal orderings (top) yield different Bayes trees (bottom). For online operation we prefer the ordering that keeps the newest variables near the root, so future updates affect only a small subtree.

Loop closures add a factor between two far apart poses. In a Bayes tree this never forces a global re-factorization. Only the cliques on the (unique) paths from those two pose cliques up to the root are affected, all other subtrees are untouched "orphans" and are reused as is. We pull just that small top region back into a local factor graph, add the loop closure factor, re-eliminate to obtain a new local chordal Bayes net (and Bayes subtree), and then reattach the orphan subtrees at the matching separators. This keeps updates local and predictable, unlike iSAMs global R re-factorization after loop closure. [7, 9]

A good variable ordering is still crucial because it controls fill in (clique sizes) during elimination. iSAM2 performs incremental reordering only over the affected variables, rather than periodic global reorderings. A simple and effective rule is to force the most recent variables (the ones new factors usually touch) to be eliminated last (i.e. near the root). Practically, this is implemented with a constrained COLAMD heuristic. Here we keep the newest pose blocks at the end of the order while letting COLAMD algorithm find a sparse order for the rest. The result is small, stable changes per step even when loops close. [9]

Batch orderings found by nested dissection (or similar heuristics) can look equally good if we solve the problem once, because they produce comparable sparsity. For online/live SLAM we care about the next update. We therefore prefer the ordering that leaves the newest poses at or very near the Bayes tree root, so the next odometry or loop closure factor only changes a small subtree. If the newest pose ends up deep in the tree, the same update must rewrite many cliques. This is the reason iSAM2 uses constrained reorderings, keeping recent variables last in the order (near the root) and let the heuristic arrange the rest. This keeps updates local and cheap. [9]

This constrained COLAMD heuristic will not yield a globally optimal ordering, however it reliably reduces fill in and the amount of work near the update, keeps recent variables near the root, and avoids large latency spikes. In practice it delivers close to batch sparsity as well as saving on compute time, and when needed we can reapply it only to the affected subtree so the cost stays proportional to that small region rather than the whole tree.

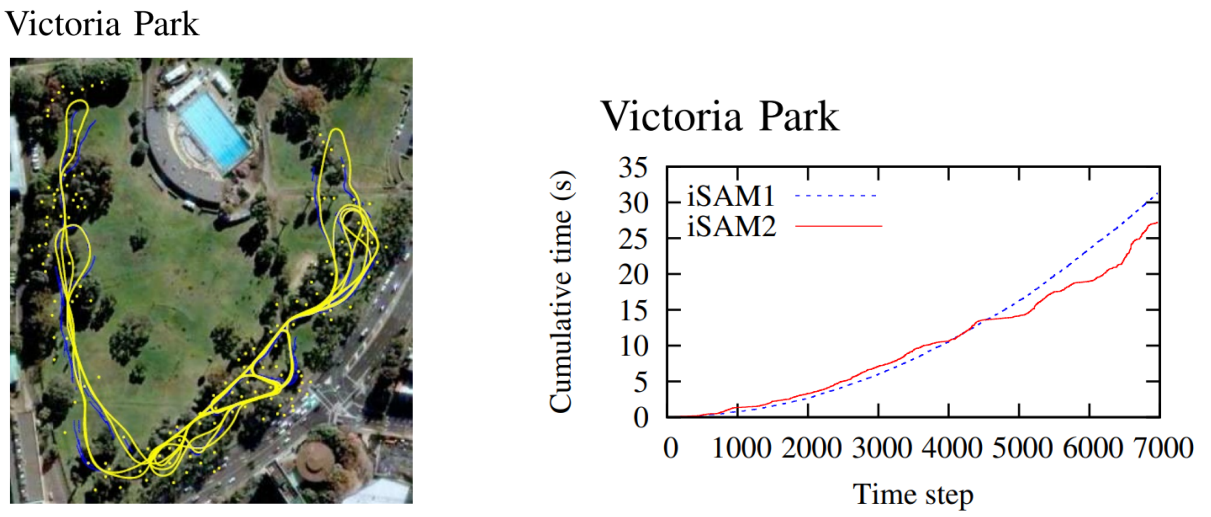
9.3.8 Fluid Re-Linearization

In iSAM2 we stop doing periodic global "re-linearize everything". Instead we only refresh (re-linearize) the parts of the problem that truly need it, right when they need it. The Bayes tree makes this easy, new measurements change only a small subtree, so we recompute just that piece and leave the rest of the tree alone. This keeps the math accurate without the big stalls that happened in iSAM after loop closures or long runs. [7, 9]

How it works in practice is simple. We always keep a running correction vector Δ from the latest linear solve. If a variable's change is tiny, we treat its current linearization point as "good enough" and do not touch it. If a variable's change is larger than a small threshold (call it β), we mark that variable for re-linearization. We then mark the cliques that contain those variables and their ancestors in the Bayes tree. Only those cliques are rebuilt, we go back to the original nonlinear factors for those cliques, recompute their Jacobians at the new linearization point, add cached "marginal" factors from untouched children, and eliminate again to update just the top of the tree. Everything else remains as it was. Note that this re-linearization threshold can be set per state. Positions (x, y) can use a looser (higher) threshold so they are refreshed less often, while the heading angle, being more nonlinear should use a tighter (lower) threshold. [9]

Solving for $\Delta\theta$ is also done only where needed. We first solve on the modified top of the tree. Then we walk into children only if any parent update exceeded a small propagation threshold (call it α). This "only follow when it matters" rule means we avoid needless work in distant parts of the map while still keeping accuracy where the robot is and where measurements arrived. Together, the two thresholds (β to decide who to re-linearize, and α to decide where to propagate the solve) give iSAM2 its fluid, real-time/online behavior. Accuracy when and where it matters, speed everywhere else.

This local, threshold strategy removes the need for heavy batch steps and keeps runtime smooth over long missions just like incremental reordering does the same. In essence what iSAM2 does is it exploits Factor graphs representations of the map and Bayes tree data structure to do dynamic variable reordering and re-linearization, no need for periodic batch calculations like in iSAM. On standard datasets (Victoria Park) the cumulative computation of iSAM2 grows much more slowly than iSAM, because re-linearization and re-elimination are confined to small subtrees rather than the full graph. And over time the discrepancy will only grow where iSAM will slow down whilst iSAM2 will hold its compute much more efficient. (See Picture 13)



Victoria Park SLAM map used in the original iSAM benchmarks.

Cumulative time vs. step for iSAM (periodic batch) and iSAM2 (fluid). iSAM2 stays faster as the dataset grows.

Figure 13: Picture from the Bayes tree paper [9]. In iSAM2, relinearization is fluid, only the affected cliques are recomputed and the rest of the tree is reused.

9.3.9 Sparse Factor Graphs

Over long missions the factor graph can accumulate many near duplicate constraints (revisits of the same place, repeated landmark sightings from similar viewpoints). This "Eiffel Tower effect" slowly densifies the graph and enlarges cliques in the Bayes tree, which increases update cost. The fix is sparsification. Here we keep the informative constraints and summarize or drop the redundant data points while preserving the important information flow. In practice this is done locally where data arrive. We retain the freshest odometry and a few diverse loop closures, and we replace discarded constraints by a light summary on the small separator set where they would have entered the tree. Intuitively, we keep the "shape" of the information at the boundary and forget interior details that are now redundant. Simple policies such as keyframing (keeping only selected poses as variables), pruning highly correlated measurements, and limiting per landmark observation count work well. The result is a graph that stays sparse, cliques that stay small, and a Bayes tree that remains cheap to update even in very long runs.

9.3.10 Beyond Gaussian Assumptions (Robust Estimators)

Pure Gaussian residuals are fragile in the face of outliers (bad data association, spurious loop closures, moving objects, changing environment). Robust estimators fix this by replacing the quadratic loss with a robust loss that grows slower than a square. Common choices include Huber (quadratic near zero, linear in the tails), Cauchy, or Tukey. In iSAM2 this is a drop in change at the factor level. Each robust loss yields a weight for its residual, updated as the estimate improves (iteratively re-weighted least squares). Factors that fit well keep high weight, inconsistent ones are down weighted, so they no longer dominate the solution. This makes incremental updates and fluid re-linearization safer because a single wrong constraint will not trigger large edits high up in the tree. For hard loop closures, one can also use switchable or graduated penalties that let the optimizer "turn off" a suspect factor until there is enough supporting evidence. The Bayes tree concept stays the same, only local factor weights and linearization adapt, so robustness comes with little extra complexity.

9.3.11 Data Association from the Bayes Tree

The same as in iSAM, we score candidate matches with a Mahalanobis distance, which measures the innovation in units of its uncertainty. However in iSAM2 the needed covariances are read directly from the Bayes tree without forming a dense matrix. Pose and nearby pose to landmark covariances are obtained efficiently by following only the few cliques that contain those variables and their separators. This keeps online/real-time gating fast for data association. Queries involving far away landmarks or many old poses may touch a larger portion of the tree and therefore cost more, but they remain practical on demand. In practice we combine a cheap, conservative bound (for routine gating) with exact small block queries when decisions are critical (eks verifying a loop closure). The result is reliable association with predictable compute cost during operation.

9.3.12 Algorithm

At each step we absorb new measurements, touch only the relevant cliques in the Bayes tree, solve for a small increment, and update the state. iSAM2 combines the linear update with fluid re-linearization, so work stays local and predictable. Concretely, one iteration looks like this (matching the structure summarized in the iSAM2 and Bayes tree papers [7, 9])

1. **Add new data:** Insert the new factors (odometry, measurements, loop closures) into the factor graph. If new states appear, add them to the estimate Θ.
2. **Mark what to refresh (fluid re-linearization):** Keep the last increment $\Delta\theta$. If a state moved more than a small threshold β, mark it for re-linearization. Only pass this mark to neighbors if the parent changed more than a smaller threshold α. This finds the small set that really needs work now.
3. **Build a small local problem:** Take just the cliques in the Bayes tree that touch the marked states (and their ancestors up to the root) and turn them back into a tiny factor graph, everything else becomes reusable "orphans".
4. **Order and eliminate locally:** Find a sparse order for this small graph using a constrained COLAMD (keep the newest states last, near the root), then eliminate to make a new local Bayes subtree.

5. **Reattach orphans:** Connect the untouched subtrees back at the correct separators. Only the edited subtree changed, the rest is reused.

6. **Solve where needed:** Back solve on the updated top of the tree to get a new increment $\Delta\theta$. Propagate the solve into children only when the parent's change is big enough (same α rule).

7. **Update the estimate:** Apply the increment on the whole factor graph, $\Theta \leftarrow \Theta \oplus \Delta\Theta$ (where $\theta = \Theta$ and $\Delta\theta = \Delta\Theta$). Keep $\Delta\Theta$ for the next steps re-linearization test.

8. **Keep variable ordering healthy (incremental):** When a loop closure grows cliques near the root, run constrained COLAMD again, but only on that small region to reduce fill and keep the newest states near the root.

9. **Data association update:** Fetch the needed local covariances from the Bayes tree, compute far away covariances only on demand, and compare predicted and observed features using our own chosen Data Association method.

9.3.13 Limitations

While iSAM2 removes the big computation spikes seen in iSAM, some practical and theoretical limits remain.

- **Ordering is heuristic, not optimal:** Choosing a variable order that minimizes fill in is NP-hard, so iSAM2 relies on constrained COLAMD and related heuristics. These give good, stable performance online but cannot guarantee the globally best sparsity.

- **Clique growth in dense areas.** Heavy revisiting of the same places ("Eiffel Tower effect") or many near duplicate constraints can enlarge cliques near the root. Updates remain local, but the cost of each local elimination grows with clique size. In long runs we often need sparsification/keyframing to keep the graph light.

- **Threshold tuning for fluid re-linearization:** The accuracy/speed trade off depends on two small thresholds (who to re-linearize and how far to propagate the solve). These must be tuned for the sensor and motion model. Too loose can delay accuracy, too tight does extra work.

- **Faraway covariances can be expensive:** Exact marginal/covariance queries are done by recursive message passing on the tree (dynamic programming style). Queries that span long paths or large separators touch more cliques and therefore cost more.

- **Gaussian least-squares core:** Outliers and non Gaussian effects are not handled by iSAM2 alone. Robust losses or switchable constraints must be added at the factor level to down weight bad data, otherwise accuracy can degrade during long missions.

- **Nonlinearity still matters:** Poor initial guesses or highly nonlinear measurements can require multiple Gauss-Newton/Levenberg-Marquardt steps. iSAM2 just makes each step local.

- **Engineering complexity and memory:** Compared to a single global R in iSAM, the Bayes tree in iSAM2 adds much more moving parts (cliques, separators, cached/orphan subtrees, incremental reordering). Correct, efficient implementations are more involved, and memory still grows with map size unless one prunes or summarizes.

In practice, most of these limits can be mitigated with careful design. Using keyframing/sparsification to cap clique size, robust losses or switchable constraints to handle outliers, and constrained incremental reordering to keep updates local. The main hurdle is engineering complexity. This is where the open source **GTSAM** library (from the Georgia Tech team behind iSAM/iSAM2) is invaluable. It ships a production quality Bayes tree/iSAM2 implementation, clean factor graph APIs, robust noise models, and utilities for ordering and re-linearization, making it a practical starting point for both research and deployment.

9.4 GTSAM

Georgia Tech Smoothing And Mapping (GTSAM) is a BSD-licensed C++ library that lets us model estimation problems as factor graphs and solve them efficiently with both batch optimizers and the incremental iSAM2/Bayes tree methods. The guiding idea is to express what we know as a product of small factors, each touching only a few variables, and then exploit sparsity and variable elimination to compute fast, stable MAP estimates. In practice this gives a single, uniform toolkit for SLAM (2D/3D), visual odometry/SLAM, structure from motion, calibration, and related inference tasks, all built on the same mathematical foundation described in the iSAM2 and Bayes tree papers [12, 7, 9].

Design philosophy: graph first, values separate: GTSAM cleanly separates the *"model"* from the *"state"*. A `NonlinearFactorGraph` stores our factors (priors, odometry, landmark/vision measurements, loop closures), while a `Values` container holds one current assignment to the unknowns (poses, landmarks, intrinsics, etc...). We can change the estimate without touching the graph, and we can add/remove factors without invalidating unrelated variables. This mirrors the math $f(\Theta) = \prod_i f_i(\Theta_i)$. The graph captures structure and sparsity. A particular Θ lets us evaluate or optimize. [12]

Core building blocks: Variables use compact *"keys"* (eks: `Symbol('x',i)` for pose x_i, `Symbol('l',j)` for landmark l_j). You build the problem by adding small, typed *"factors"*, each encoding one piece of sensor information plus its noise model. For example `PriorFactor<Pose2>`, `BetweenFactor<Pose2>`, `BearingRangeFactor2D`, camera factors like `GenericProjectionFactor<Cal3_S2>`, and many others. Noise models are explicit and first class (`noiseModel::Isotropic`, `noiseModel::Diagonal`, robust M-estimators), so units and weighting are clear and consistent. There are different ways to represent rotations in 3D space, however GTSAM has defined poses in Lie groups, this is a mathematical way to describe 3D space including rotation in an easy and intuitive to handle way. Because poses live on curved rotation/pose spaces (SE(2)/SE(3)), GTSAM updates them using *"local coordinates"* and a *"retraction"* operator. GTSAM computes a small 3D/6D increment in a flat tangent space and then maps it back to a valid pose, avoiding angle wrap around and keeping rotations proper. In practice, Gauss-Newton/Levenberg-Marquardt linearization methods "just works" with orientations, no ad hoc hacks needed. The end result is that writing SLAM code feels like drawing the factor graph, add one factor per measurement between the variables it touches, then optimize. GTSAM handles sparsity, ordering, and iSAM2s incremental updates under the hood. [12]

Batch optimization and linear algebra under the hood: In GTSAM we pose SLAM as a `NonlinearFactorGraph` plus an initial `Values`. At each optimizer step GTSAM linearize the factors at the current estimate to get a linear system witch we can solve for. Rather than forming one huge matrix to represent this linearized system, GTSAM solves this linear step by *variable elimination*. Choose an order, combine the factors that touch the next variable, eliminate it, and keep going. The result can be viewed as a Bayes net, grouping by shared separator variables yields a Bayes tree, through which the solution is recovered by simple back substitution. For online use, `iSAM2` keeps that Bayes tree and, when new measurements arrive or some states need re-linearization, it rebuilds only the small subtree that is affected and reuses the rest unchanged. Speed and memory depend on the elimination order, so GTSAM provides practical heuristics (eks: COLAMD and constrained COLAMD) that reduce fill in and keep the newest poses near the root so updates stay local. In short, we add small typed factors, GTSAM handles linearization and sparse elimination, and with `iSAM2` it updates only where needed. [12]

iSAM2 and the Bayes tree in GTSAM: For real-time use, GTSAM iSAM2 keeps a Bayes tree data structure, instead of one giant monolithic R matrix. When a new measurement arrives, it usually touches only a few variables, so iSAM2 edits just that small part of the tree, it pulls out the affected piece, re-solves that tiny subproblem, and snaps it back in place while leaving the rest untouched. Re-linearization is local and on-demand, only refreshing variables if it moves past a small (per-state) threshold, and only push the solve down the tree if a parent changed a lot. The variable order is maintained incrementally (via constrained COLAMD heuristics) so the newest poses stay near the root, which keeps future updates local and fast. In code we simply add factors and initial guesses, call `isam.update(...)`, and read out the current estimate (and local covariances) without the big compute spikes of global re-factorization. [12]

What using GTSAM looks like: Create a `NonlinearFactorGraph` and a `Values` with our first guesses. As new sensor data arrives, add the right factors (odometry, landmark/vision, loop closures) and insert

any new variables. For a batch solve, run `GaussNewtonOptimizer` or `LevenbergMarquardtOptimizer` and read the improved `Values`. For real-time use, we keep an `ISAM2` object, call `update(newFactors, newValues)` each step, then get the current best estimate with `isam.calculateEstimate()`. When we need uncertainty for data association or checks, ask for marginal covariances of just the variables we care about, no giant matrix inverses needed. We can freely mix `Pose2`/`Pose3`, `Point2`/`Point3`, camera calibration, and robust noise models in the same graph, it all fits the same pattern. [12]

Educational and practical: GTSAM is built to feel like the math we see in the iSAM and iSAM2 papers [6, 7]. Each measurement becomes a small, typed *"factor"*. Our current guesses live in a `Values` container that understands poses and rotations, and the solvers make sparsity and variable ordering visible. GTSAM comes with clear examples and MATLAB/Python bindings, so we can try ideas quickly and plot results without much setup. The focus is clarity and research, not necessarily efficiency. There are a lot of abstractions, and being pedagogical and educational comes before optimizations to the code and specific hardware for CPU and GPU maximum performance. The core methods from the iSAM and iSAM2 papers [6, 7] are exactly the same here behind the algorithms, square-root solving, variable elimination, Bayes trees, and iSAM2. GTSAM has powered real robots and vision systems, so for most projects GTSAM is "good enough" as a reliable back end we can read, extend, and trust. [12]

What GTSAM does NOT do (and how to fill the gap): GTSAM is a back-end optimizer, it does not detect features, track them, perform loop detection, or decide data association for you. Those front-end tasks live outside and feed GTSAM through factors. Robustness to outliers (bad matches, spurious loops, moving objects) is handled by choosing robust loss functions or switchable/graduated penalties at the factor level. Over very long runs, mitigate graph growth (the "Eiffel Tower" effect) with keyframing and sparsification so cliques stay small and updates remain local. For far away covariance queries, we must expect higher cost because more of the tree is touched, use of approximate/local covariances for routine gating and saving exact queries for critical decisions is the way to go. With clever design these gaps can be resolved. [12]

Takeaway: GTSAM provides a principled, factor graph centric way to model estimation problems and couples it with high quality batch and incremental solvers built on iSAM2 and the Bayes tree. It preserves the numerical strengths of square root factorization while delivering the locality we need for real-time operation. GTSAM is the practical implementation that lets us apply iSAM2 ideas immediately in our own SLAM systems. [12]

References

[1] Haralstad Vegard. "A side-scan sonar based simultaneous localization and mapping pipeline for underwater vehicles". Master's thesis. Norwegian University of Science and Technology (NTNU), 2023. URL: https://ntnuopen.ntnu.no/ntnu-xmlui/handle/11250/3086270 (visited on 09/13/2025).

[2] Hoff Simon, Andreas Hagen, Haraldstad Vegard, Reitan Hogstad Bjoornar, and Varagnolo Damiano. "Side-scan sonar based landmark detection for underwater vehicles". In: (Oct. 2024). URL: https://ntnuopen.ntnu.no/ntnu-xmlui/handle/11250/3172808 (visited on 09/13/2025).

[3] Cadena Cesar, Carlone Luca, Carrillo Henry, Latif Yasir, Scaramuzza Davide, and Neira José. "Past, Present, and Future of Simultaneous Localization and Mapping: Toward the Robust-Perception Age". In: (Dec. 31, 2016). URL: https://ieeexplore.ieee.org/document/7747236 (visited on 09/14/2025).

[4] Durrant-Whyte Hugh and Bailey Tim. "Simultaneous localization and mapping: part I". In: (June 30, 2006). URL: https://ieeexplore.ieee.org/document/1638022 (visited on 09/14/2025).

[5] Durrant-Whyte Hugh and Bailey Tim. "Simultaneous localization and mapping: part II". In: (Sept. 30, 2006). URL: https://ieeexplore.ieee.org/document/1678144 (visited on 09/14/2025).

[6] Kaess Michael, Ranganathan Ananth, and Dellaert Frank. "iSAM: Incremental Smoothing and Mapping". In: (Dec. 31, 2008). URL: https://ieeexplore.ieee.org/document/4682731 (visited on 09/14/2025).

[7] Kaess Michael, Johannsson Hordur, Roberts Richard, Ila Viorela, Leonard John, and Dellaert Frank. "iSAM2: Incremental smoothing and mapping with fluid relinearization and incremental variable reordering". In: (Aug. 18, 2011). URL: https://ieeexplore.ieee.org/document/5979641 (visited on 09/14/2025).

[8] Cansiz Sergen. "Multivariate Outlier Detection in Python: Multivariate Outliers and Mahalanobis Distance in Python". In: (Mar. 20, 2021). URL: https://medium.com/data-science/multivariate-outlier-detection-in-python-e946cfc843b3 (visited on 09/15/2025).

[9] Kaess Michael, Ila Viorela, Roberts Richard, and Dellaert Frank. "The Bayes Tree: An Algorithmic Foundation for Probabilistic Robot Mapping". In: (Jan. 2010). URL: https://www.cs.cmu.edu/~kaess/pub/Kaess10wafr.pdf (visited on 09/14/2025).

[10] Heggernes Pinar and Matstoms Pontus. "Finding Good Column Orderings for Sparse QR Factorization". In: (Sept. 2000). URL: https://www.researchgate.net/publication/2498292_Finding_Good_Column_Orderings_for_Sparse_QR_Factorization (visited on 09/25/2025).

[11] Castelo Robert. "The Discrete Acyclic Digraph Markov Model in Data Mining". In: (Jan. 2002). URL: https://www.researchgate.net/publication/46624302_The_Discrete_Acyclic_Digraph_Markov_Model_in_Data_Mining (visited on 09/25/2025).

[12] Dellaert Frank. "Factor Graphs and GTSAM: A Hands-on Introduction". In: (Sept. 2012). URL: https://repository.gatech.edu/server/api/core/bitstreams/b3606eb4-ce55-4c16-8495-767bd46f0351/content (visited on 09/25/2025).