TTK4551 Technical Cybernetics - Specialization Project

Martynas Smilingis

September 2025

Contents

Acronyms

ASV	Autonomous Surface Vessel
AUR Lab	Applied Underwater Robotics Lab
AUV	Autonomous Underwater Vehicle
COLAMD	Column Approximate Minimum Degree
DDS	Data Distribution Service
DOF	Degrees Of Freedom
ECEF	Earth-Centered, Earth-Fixed
EKF	Extended Kalman Filter
ESC	Electronic Speed Controller
ESKF	Error State Kalman Filter
GNSS	Global Navigation Satellite System
GTSAM	Georgia Tech Smoothing And Mapping
HDPE	High Density Polyethylene
IMU	Inertial Measurement Unit
INS	Inertial Navigation System
iSAM	Iterative Smoothing And Mapping
KF	Kalman Filter
LAUV	Light Autonomous Underwater Vehicle
LiDAR	Light Detection And Ranging
MAP	Maximum A Posteriori
MPC	Model Predictive Control
NED	North East Down
PPS	Pulse Per Second
PSM	Power Sense Module
PWM	Pulse Width Modulation
RL	Reinforcement Learning
ROS2	Robot Operating System 2
SBC	Single Board Computer
SE(3)	Special Euclidean Group in three dimensions
SLAM	Simultaneous Localization And Mapping
SLERP	Spherical Linear Interpolation
SO(3)	Special Orthogonal Group in three dimensions
SSS SLAM	Side Scan Sonar Simultaneous Localization And Mapping
UKF	Unscented Kalman Filter
WGS84	World Geodetic System 1984

1 Introduction

1.1 Goal

Maybe I should resturcture the Introduction Goal and Motivation maybe?...

Also add the FN bærekrafts mål I suppose?...

This specialization project focuses on navigation and Simultaneous Localization and Mapping (SLAM) for marine robots, with an emphasis on Side Scan Sonar based SLAM. The main objective is to study, reimplement, and optimize the core components of modern SSS SLAM pipelines to achieve real-time performance on real world AUV datasets, such as those collected from NTNU's AUR Lab platforms (for example LAUV Harald).

The work builds upon the 2023 masters thesis by Haraldstad [1] and recent research on side scan sonar landmark detection [2], which are heavily inspired by the earlier work of Hogstad, Bjørnar Reitan [3]. The projects scope is to implement and benchmark the essential components required for real-time SSS SLAM operation, validate them on existing datasets, and demonstrate that the processing pipeline can operate at or above the sonars frame rate while maintaining high mapping accuracy.

A second phase of the project focuses on embedded deployment on a autonomous surface vessel (ASV). The goal is to port and integrate the validated core pipeline on ship borne hardware, assess side scan sonar performance in coastal waters, and shift the work from algorithm design to practical system integration. This phase will use NTNU's MicroAmpere ASV.

1.2 Motivation

Reliable maritime navigation requires onboard estimation and mapping when external positioning is weak or unavailable. This project focuses on side scan sonar based simultaneous localization and mapping, SSS SLAM. SSS SLAM uses side scan sonar to estimate the vehicle pose while building a seafloor map at the same time. In SSS SLAM, sonar images drive feature extraction and data association, loop closures correct drift, and the SLAM back end fuses all measurements into a consistent trajectory and map. Marine robots operate with limited access and often far from support. They must know where they are and what surrounds them to move safely and do useful work. Static charts help, but the ocean changes over time. Currents, waves, moving vessels, new structures, and shifting seabeds make static maps go out of date. GNSS is weak or unavailable underwater, and dead reckoning drifts. Even in coastal areas, terrain can block signals, and shallow water operations require safe margins to the bottom. These factors make SSS SLAM a practical path to robust navigation and mapping.

Prior work [1] presented a pipeline for SSS SLAM but did not reach real time performance. Field deployment needs real time operation on real data with measured accuracy and robustness. The first motivation is to deliver a lean real time SSS SLAM implementation and to quantify performance on the same dataset used previously, so the results are directly comparable.

The second motivation is to move from offline studies to reliable system behavior at sea. The plan is to measure accuracy, robustness, and runtime on recorded AUV data, then prepare the pipeline for embedded use on a autonomous surface vessel (ASV). This shifts effort from algorithm design to practical integration on real hardware, including time synchronization, calibration, and stable runtime.

1.3 Side Scan Sonar SLAM Architecture

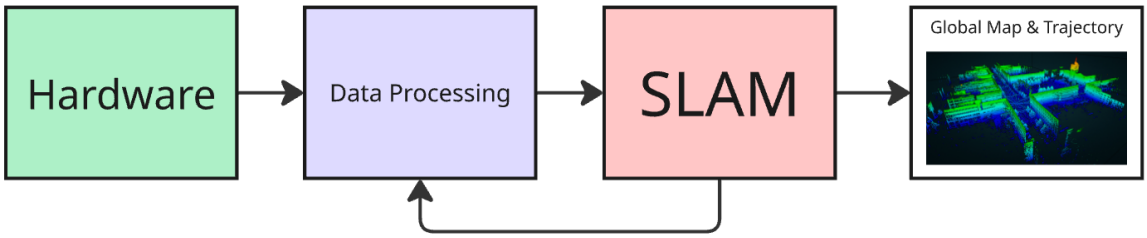


Figure 1: A simplified picture of SSS SLAM pipeline from raw sensor data to global map

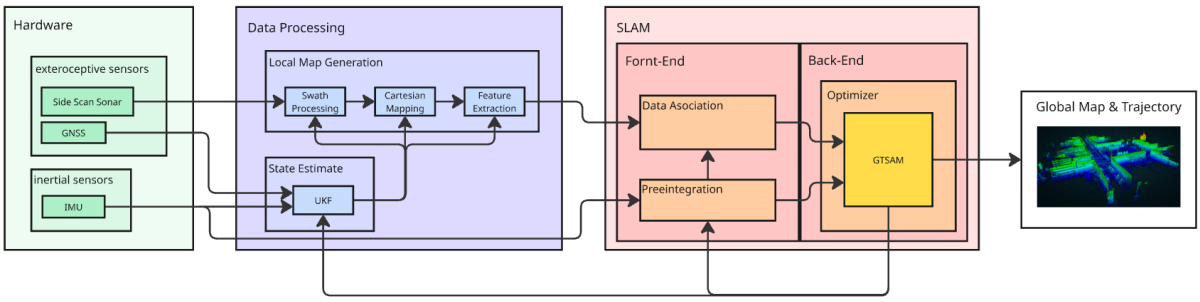


Figure 2: A picture of the whole SSS SLAM pipeline in detail from raw sensor data to global map

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

3.1 Micro Ampere ASV

Figure 3: A picture of microAmpere Autonomous Surface Vessel (ASV). Picture taken from Luan Cao Vo Tran master thesis on microAmpere ASV.[4]

This section focuses on the *"microAmpere Autonomous Surface Vessel (ASV)"* platform, which serves as the primary research and development base for this work. The information presented here is summarized from two detailed master theses, Luan Cao Vo Trans thesis on microAmpere hardware and integration [4], and Henrik Reimers thesis on control and system design [5]. More detailed technical descriptions can be found in the referenced theses.

In this section, only the most relevant aspects of the platform are presented, specifically those that are essential for integrating a side scan sonar payload and for enabling SSS SLAM. The focus is therefore on the hardware architecture, computing layout, and synchronization systems that directly affect sonar data acquisition and processing.

The microAmpere is an advanced Autonomous Surface Vessel (ASV) developed at NTNU as a modular platform for autonomous docking and maritime robotics research. It is part of the Autodocking25 project and represents the next evolution of the Blue Robotics BlueBoat platform. The vessel was redesigned to support high performance perception, navigation, and control tasks under realistic marine conditions. Its main purpose is to provide a flexible and open research platform for experimenting with autonomous operations such as docking, waypoint navigation, and sensor fusion at sea.

Physically, the vessel is based on the BlueBoat twin hull design, offering high stability and sufficient deck space for custom payloads. The hulls are constructed from High Density Polyethylene (HDPE), ensuring durability and resistance to saltwater corrosion. The superstructure houses one watertight enclosures containing the compute units, sensor interfaces, and power management systems. This enclosure is designed for quick maintenance and allows for modular upgrades for new hardware.

The system integrates a distributed computing architecture consisting of a *"Jetson Orin NX"* for GPU intensive perception tasks such as LiDAR and camera processing, and a *"LattePanda 3 Delta"* Single Board Computer (SBC) for real-time navigation, control, and communication handling. Both systems run ROS2 with synchronized clocks via GNSS and PPS hardware timing, achieving sub microsecond precision across devices. This allows accurate sensor timestamping and time alignment between the navigation, control, and perception subsystems.

The sensor suite includes dual GNSS antennas for accurate heading estimation, a high grade IMU for attitude and acceleration measurements, an *"Ouster OS1"* LiDAR for 3D environmental mapping, and *"StereoLabs ZED-X"* stereo cameras for visual perception and teleoperation. A dedicated power sensing board monitors voltage and current to ensure system safety, and an industrial *"Teltonika RUTX50"* router provides 4G/5G and GNSS connectivity, extending operational range far beyond line of sight limits. All components are interconnected through a structured internal wiring system with proper isolation, grounding, and surge protection to handle harsh marine environments.

The MicroAmperes software stack is fully containerized, combining real-time control loops with modular high level autonomy nodes. Each module communicates through standardized ROS2 topics and services, enabling independent algorithm development without system reconfiguration. This architecture also facilitates rapid iteration and field testing of advanced autonomy features such as Model Predictive Control (MPC), Reinforcement Learning (RL), and Simultaneous Localization and Mapping (SLAM).

In addition to autonomy research, the microAmpere platform is used for multi sensor data collection and algorithm benchmarking under controlled sea trials. The synchronized data from LiDAR, stereo vision, IMU, and GNSS enables high quality datasets for perception, sensor fusion, and state estimation studies. The vessels modular interfaces also make it suitable for integration with experimental payloads such as side scan sonar, underwater acoustic systems, or alternative control computers for specific research tasks.

From a systems engineering perspective, microAmpere demonstrates the effectiveness of distributed embedded computing in marine robotics. By separating perception and control workloads, developers can independently optimize performance for GPU heavy machine learning pipelines and deterministic control processes. The modular ROS2 setup also supports simulation-in-the-loop and hardware-in-the-loop testing, ensuring smooth transition from virtual models to field deployment.

According to Luan Cao Vo Tran master thesis on microAmpere ASV [4] and Henrik Reimers master thesis on microAmpere ASV [5], the platform has already undergone multiple field campaigns in Trondheim Fjord. These tests validated microAmpere ASVs hardware robustness, networking reliability, and autonomy stack. Its success has positioned it as a reference platform for NTNUs future autonomous surface vessel research, bridging efforts between the Department of Cybernetics and the AUR Lab.

Overall, the microAmpere ASV serves as a compact but powerful research platform, bridging the gap between simulation and real world autonomous marine systems. Its modular design, distributed computation, and precise timing infrastructure make it ideal for experimentation with advanced autonomy, perception, and control algorithms in challenging maritime environments.

3.2 Specifications

The microAmpere ASV was designed as a flexible and modular research platform, combining reliable marine hardware with powerful embedded computing and sensing capabilities. Its specification focuses on providing stable, high precision navigation and perception for real-time autonomy experiments in nearshore and harbor environments. The systems modularity allows researchers to swap components and integrate additional sensors without structural modification. The vessels configuration and components are summarized in Table 2 down bellow.

Table 2: Technical specifications of the microAmpere ASV.

Category	**Specification**
Platform Base	Blue Robotics BlueBoat twin-hull (HDPE construction)
Dimensions	Length: 1.8 m Width: 1.1 m Draft: 0.25 m
Weight	Approx. 45 kg (fully equipped)
Propulsion	Dual Blue Robotics T200 thrusters with ESC control, differential thrust for steering
Power Supply	Dual 6S Li-Ion batteries (22.2 V nominal, 20 Ah each), hot-swappable configuration
Compute Units	NVIDIA Jetson Orin NX (perception and LiDAR processing) LattePanda 3 Delta (navigation, control, and communication)
Operating System	Ubuntu 24.04 LTS with ROS2 Jazzy Jalisco
Networking	Teltonika RUTX50 router (4G/5G, Wi-Fi, and GNSS) with Tailscale VPN
Localization Sensors	Dual u-blox ZED-F9P GNSS modules with RTK correction Xsens MTi-680G high-precision IMU
Perception Sensors	Ouster OS1-64 LiDAR StereoLabs ZED-X stereo camera pair
Timing and Synchronization	PPS signal distribution and PTP-based synchronization via custom timing board
Power Monitoring	Custom power sensing PCB (voltage, current, and temperature feedback)
Communication Interfaces	CAN, Ethernet, USB 3.0, Serial (RS232/RS485), GPIO
Software Capabilities	Real-time navigation, control, diagnostics, and data logging Modular autonomy framework for MPC, RL, and SLAM integration
Field Operation	Endurance: $\sim$3 hours under typical load Max speed: $\sim$2.5 m/s Remote or fully autonomous operation modes

The specifications highlight the microAmpere balance between portability and computational power. The distributed computing setup provides sufficient resources for advanced autonomy tasks while maintaining low latency in the control loop. The integrated sensors enable precise positioning and rich perception, making it an ideal testbed for advanced control, SLAM, and sensor fusion algorithms in dynamic maritime environments.

3.3 Layout

The microAmperes internal and external layout is designed to achieve an optimal balance between functionality, modularity, and serviceability. The vessel is divided into three main physical sections, consisting of the two hulls and the central watertight enclosure. Each section has a specific role in the system: propulsion and power are located in the hulls, computation and communication in the watertight enclosure, and perception sensors are distributed around the vessel. This structure simplifies cabling, improves center-of-gravity stability, and provides a clean and organized platform for integrating research equipment.

The port hull houses the port side battery pack and Electronic Speed Controller (ESC) that drive the left thruster, while the starboard hull contains the mirrored setup together with a Power Sense Module responsible for voltage and current monitoring. Both hulls are interconnected through a reinforced cross tube that carries power, Pulse Width Modulation (PWM), and Ethernet lines to the watertight enclosure. This cross connection ensures a robust and low latency interface between propulsion, control, and sensor systems, while maintaining mechanical symmetry and balance across the hulls.

On the top deck, the vessel supports a LiDAR platform and stereo camera mounts designed to provide an unobstructed 360° field of view. These sensors enable simultaneous environmental mapping and perception. Additional top mounted features include power switch, diagnostic status LEDs, and the antennas for GNSS, 5G, and radio communication. The overall sensor placement was chosen to ensure optimal visibility, minimal interference between radio and optical systems, and ease of access during maintenance or calibration.

Externally, all antennas and sensors are symmetrically arranged to guarantee signal balance and redundancy. The vessel is equipped with dual GNSS antennas mounted at the rear of each hull for precise position and heading estimation, a 5G antenna for long range communication, and a radio antenna for local telemetry. The perception suite consists of two stereo cameras and a LiDAR unit mounted on the LiDAR platform. Together, these sensors provide full 360° situational awareness and deliver high quality data for SLAM, obstacle detection, and perception algorithms.

The internal layout focuses on structured modularity, with all high power and propulsion related systems isolated within the hulls, and all sensitive electronics centralized in the watertight enclosure. This separation improves electromagnetic compatibility and enhances maintainability during field operations. The vessels internal wiring is routed through marine grade cable glands, organized in cable harnesses with labeled connectors for quick service and component replacement.

In addition, active cooling fans are installed at the rear of the hulls and inside the watertight enclosure to maintain safe operating temperatures for both computation and power electronics. Proper thermal design and airflow direction have been validated through practical testing to ensure stability under extended mission durations and high computational loads.

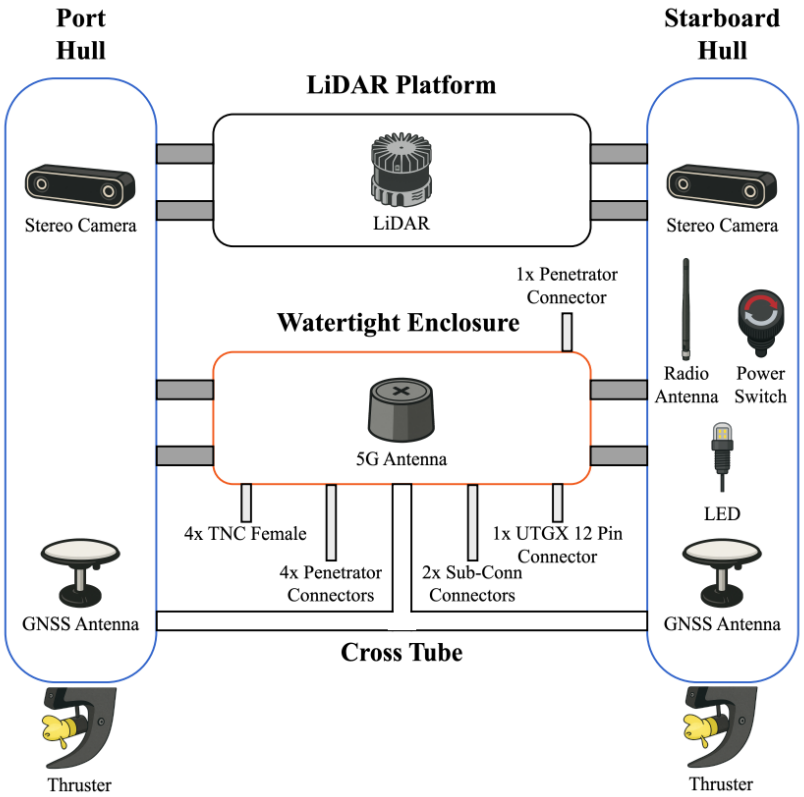


Figure 4: Top level overview of the microAmpere vessel showing the placement of cameras, antennas, thrusters, power switches, status LEDs, and LiDAR platform. Picture taken from Luan Cao Vo Tran master thesis on microAmpere ASV.[4]

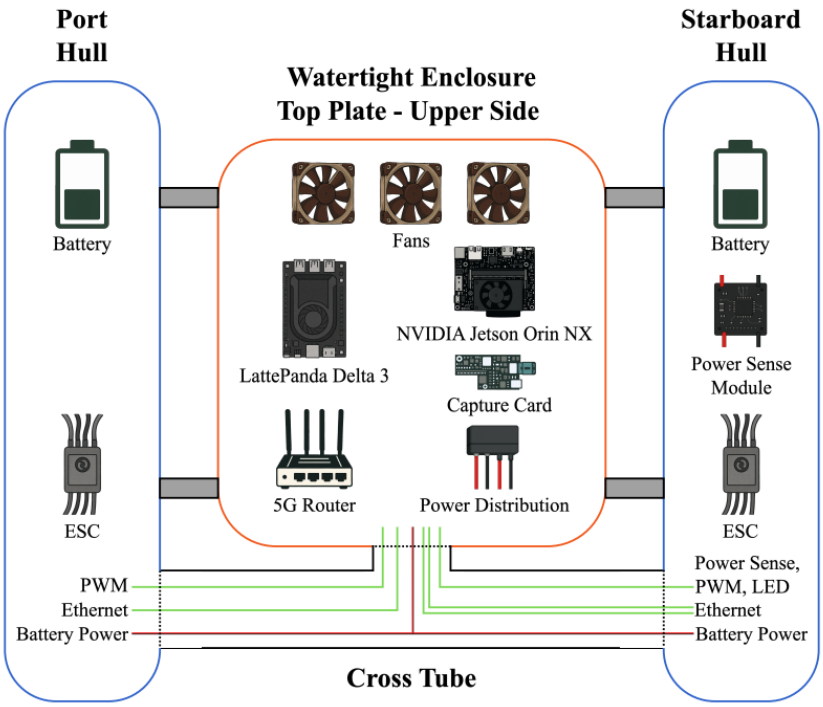


Figure 5: Bottom view of microAmpere showing the internal layout of batteries, ESCs, cooling fans, SBCs, capture card, and power distribution units for optimized performance and accessibility. Picture taken from Luan Cao Vo Tran master thesis on microAmpere ASV.[4]

Focusing on the watertight enclosure, this section contains all critical computational, power conversion, and synchronization electronics of the microAmpere system. The internal structure follows a two layer plexiglass mounting design with a top and bottom plate. This setup maximizes thermal efficiency, ensures proper airflow, and allows easy access for maintenance or hardware upgrades.

The top plate upper side holds the main compute stack, including the *"NVIDIA Jetson Orin NX"*, *"LattePanda 3 Delta"*, and the *"Teltonika RUTX50 5G router"*. These components are mounted alongside a power distribution module and a set of *"Noctua 60 mm cooling fans"* for optimal airflow and thermal stability during intensive computational loads. The placement of these units also provides clear cable routing paths and short communication links between processing and networking hardware.

The underside of the top plate contains the *"DC/DC converters"* rated at 30 W and 150 W, a *"fuse board"*, and the fan bracket assembly. These converters supply stable 12 V and 5 V lines to different subsystems including the compute boards, router, Sentiboard, and auxiliary electronics. The components are arranged for efficient heat dissipation and quick replacement in case of hardware maintenance.

The bottom plate top side integrates the primary navigation and synchronization modules. It includes the *"Xsens MTi-680G IMU"*, dual *"u-blox ZED-F9P GNSS receivers"*, and the custom *"Sentiboard"* used for precise hardware timestamping and signal synchronization. The stacked layout minimizes wiring complexity, reduces signal interference, and maintains clear access for diagnostics and firmware updates.

Overall, the watertight enclosure layout emphasizes modularity, cooling efficiency, and reliability. By separating compute, power, and navigation layers, the design reduces thermal coupling between components and improves system robustness during long duration field operations. This configuration allows the microAmpere to operate safely under variable weather and computational conditions while maintaining high data integrity across all subsystems.

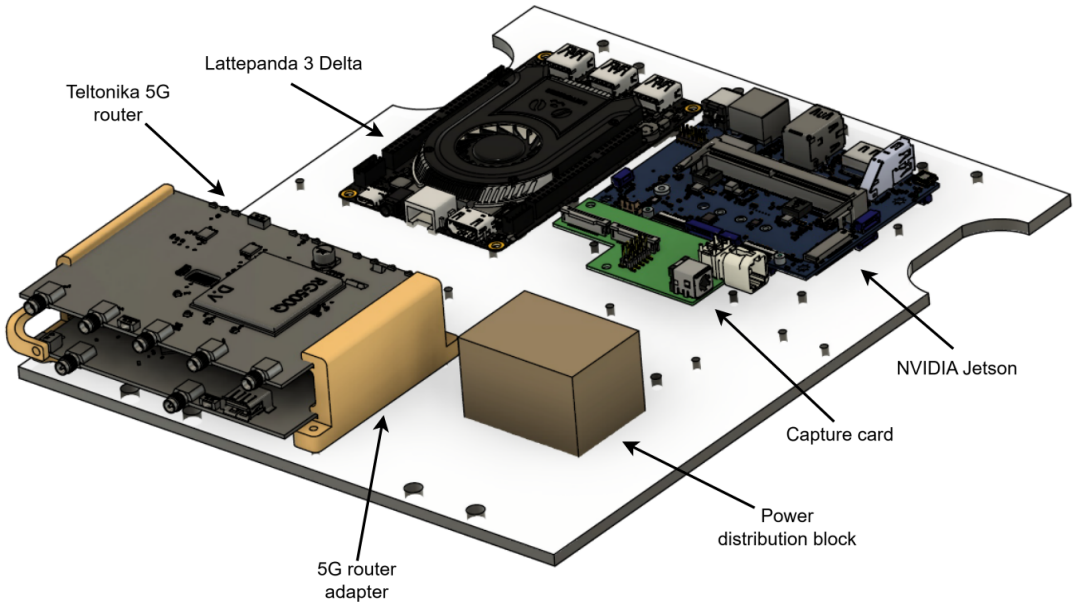


Figure 6: Watertight enclosure top plate (upper side) showing the main compute stack with Jetson Orin NX, LattePanda 3 Delta, and Teltonika RUTX50 router. Picture taken from Henrik Reimers master thesis on microAmpere ASV.[4]

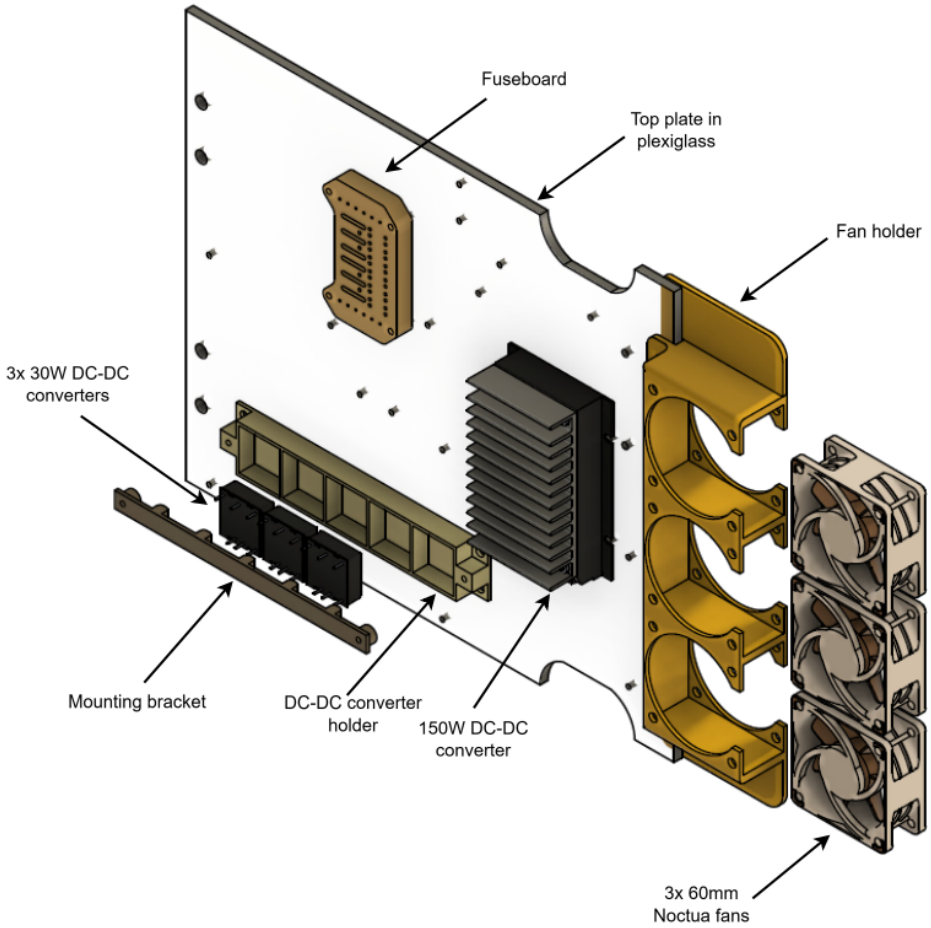


Figure 7: Underside of the top plate showing the DC/DC converters, fuse board, and cooling fans for power regulation and heat management. Picture taken from Henrik Reimers master thesis on microAmpere ASV.[4]

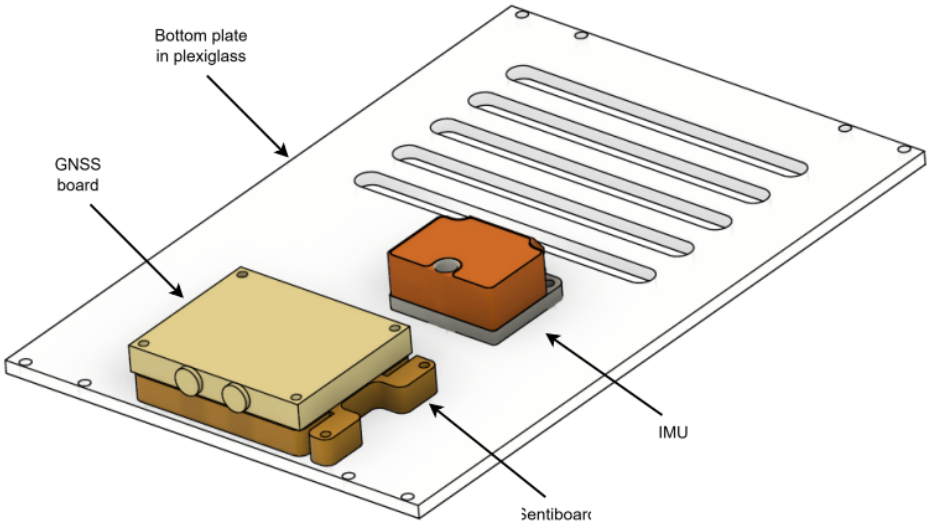


Figure 8: Bottom plate layout containing the IMU, Sentiboard, and GNSS receivers used for precise navigation and time synchronization. Picture taken from Henrik Reimers master thesis on microAmpere ASV.[4]

3.4 Sensors

3.4.1 Introduction

The microAmpere platform is equipped with a wide range of sensors supporting navigation, perception, and control tasks. For this work, the focus lies on the integration of a Side Scan Sonar and the development of an SSS SLAM. Two of the existing sensors play a key role in enabling this, the Inertial Measurement Unit (IMU) and the Global Navigation Satellite System (GNSS). These sensors together provide accurate state estimation and position tracking, forming the foundation required for precise sonar data alignment, georeferencing, and mapping.

3.4.2 Inertial Measurement Unit (IMU) for Navigation and Pose Estimation

The *"Sensonor STIM300"* is the primary Inertial Measurement Unit (IMU) onboard microAmpere. It is a tactical grade sensor providing precise angular rate and linear acceleration measurements for dead reckoning, attitude estimation, and control stabilization. Its compact and rugged design makes it well suited for marine robotics, where vibration, temperature variation, and magnetic disturbances are common. The unit integrates three high precision gyroscopes and three accelerometers, enabling full 3-axis motion sensing with excellent mechanical and thermal stability.

The STIM300 offers a $\pm 400\ ^\circ/s$ gyroscope range and $\pm 10\ g$ accelerometer range with 24-bit resolution, ensuring low noise and stable long term performance. It communicates via an RS-422 interface, operates from a 5 V supply, and supports sampling rates up to 2 kHz with deterministic timing. Its low bias instability and minimal drift allow reliable short term dead reckoning when GNSS signals are unavailable, making it a key component for autonomous control and future side scan sonar-based SLAM integration.

An overview of the IMU is shown in Figure 9, and its key specifications are summarized in Table 3.

Figure 9: Sensonor STIM300 high-performance IMU used for internal navigation and pose estimation. Picture taken from Henrik Reimers master thesis on microAmpere ASV.[4]

Table 3: Sensonor STIM300 IMU specifications summary.[6]

Parameter	Value
Gyroscope range	$\pm 400\ ^\circ/s$
Accelerometer range	$\pm 10\ g$
Resolution	24-bit
Max data rate	2 kHz
Interface	RS-422
Input voltage	5 V
Weight	55 g

3.4.3 GNSS for Positioning and Heading Estimation

For accurate positioning and heading estimation, microAmpere employs two *"u-blox ZED-F9P"* GNSS modules integrated on a Dual GNSS card from SentiSystems. These modules support Real-Time Kinematic (RTK) corrections for centimeter level positioning accuracy and operate in a moving base configuration to estimate heading from the relative position of two GNSS antennas.

The external *"SignalPlus"* GNSS antennas provide reliable satellite reception and minimize multipath effects common in harbor environments. With a baseline distance of approximately 0.7m between antennas, the system achieves heading accuracy of around 0.5°, while maintaining position accuracy down to 1.5m, or 0.01m with RTK corrections. This configuration ensures precise navigation and robust performance even under challenging marine conditions.

Together with the onboard IMU, the dual GNSS system forms the foundation for the vessels navigation, control, and future SSS SLAM integration. The Dual GNSS card and antennas are shown in Figure 10, and the system specifications are summarized in Table 4.

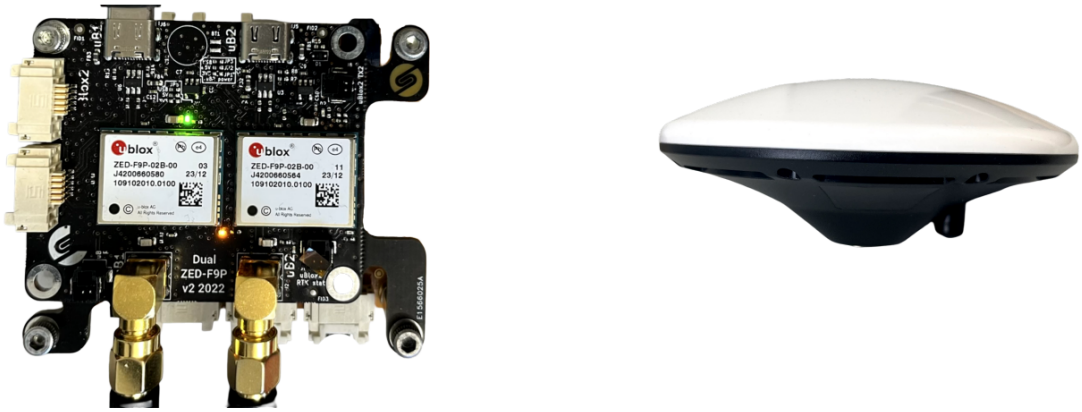


Figure 10: (Left) SentiSystems Dual GNSS card with two u-blox ZED-F9P modules. (Right) SignalPlus GNSS antennas used for position and heading estimation. Picture taken from Henrik Reimers master thesis on microAmpere ASV.[4]

Table 4: GNSS module specifications summary.[7]

Parameter	**ZED-F9P**
Position accuracy (CEP)	1.5 m (0.01 m with RTK)
Heading accuracy	0.4° (baseline 1.8 m)
RTK capability	Base + Moving Base
Satellite bands	L1, L2, L5
Max navigation rate	25 Hz (5 Hz moving base)

3.4.4 Side Scan Sonar for Environmental Perception and SLAM

The *"DeepVision OSM Ethernet Sonar System"* is the side scan sonar used onboard the microAmpere platform for seabed imaging, mapping, and SLAM development. Developed by DeepVision AB, this system operates with dual 680 kHz transducers utilizing chirp modulation to produce high resolution acoustic imagery. It supports a maximum operating depth of 100 m and can achieve image resolutions down to 1 cm depending on configuration settings.

The OSM sonar was previously evaluated by Hogstad, Bjørnar Reitan in his masters thesis at NTNU, where it was integrated on a BlueROV2 platform with an Error State Kalman Filter (ESKF) for autonomous navigation and acoustic mapping [3]. In that setup, the transducers were mounted symmetrically and angled downward at $\theta = 45°$, providing a vertical beam width of $\alpha = 60°$ and a horizontal beam width of $\phi = 0.7°$. This configuration allowed both wide area coverage and high spatial resolution across the seafloor.

The sonar connects via Ethernet to the onboard computer and is automatically recognized by the *"DeepView LT"* software for real-time imaging, control, and data logging. Its high operating frequency and narrow horizontal beam enable precise seafloor mapping, while the wide vertical beam ensures efficient coverage for environmental perception. This makes it highly suitable for integration into the microAmpere ASVs perception stack, supporting the development of SSS SLAM and environment aware autonomy.

An overview of the sonar system is shown in Figure 11, and its main specifications are summarized in Table 5.

Figure 11: DeepVision OSM Ethernet Side-Scan Sonar system with dual 680 kHz transducers used for seabed imaging and SLAM development. Picture adapted from Hogstads master thesis on BlueROV2 sonar integration.[3]

Table 5: DeepVision OSM Ethernet Sonar System specifications summary.[3]

Parameter	**Value**
Operating frequency	680 kHz (Chirp signal)
Resolution	Down to 1 cm
Maximum operating depth	100 m
Vertical beam width (α)	60°
Horizontal beam width (ϕ)	0.7°
Mounting angle (θ)	45°
Swath range	Up to 50 m per side (100 m total)
Interface	Ethernet
Software	DeepView LT
Manufacturer	DeepVision AB

3.5 Software Architecture

Although not strictly part of the hardware system, the software architecture is what connects, manages, and coordinates all onboard components. It defines how the sensors, actuators, and computational units communicate to form a complete autonomous system. The design follows a modular and distributed approach, separating computational loads across multiple processors to improve reliability, maintainability, and scalability. This structure also allows new hardware, such as the side scan sonar, to be integrated seamlessly without major reconfiguration of existing modules.

Each subsystem is assigned a specific computational domain with a well-defined purpose. The *"Navigation Unit"* on the *"LattePanda 3 Delta"* is responsible for low-level control, sensor fusion, and actuator management, interfacing directly with the IMU, GNSS, and thruster controllers. The *"Perception Unit"* on the *"NVIDIA Jetson Orin NX"* handles high bandwidth sensors such as LiDAR and stereo cameras, performing real-time perception, object detection, and mapping. The *"Ground Control Station"* provides remote supervision, telemetry visualization, and manual override when required. Together, these systems operate in a synchronized local Ethernet network, exchanging data through a standardized message layer.

The software stack runs on *"Ubuntu 24.04 LTS"* with the *"ROS2 Jazzy Jalisco"* distribution. ROS2 acts as the communication backbone, providing a real-time publish/subscribe framework through the Data Distribution Service (DDS) middleware. Each functional block in the system is implemented as an independent ROS2 node that communicates via topics, services, and actions. This decoupled design simplifies debugging, system upgrades, and integration of experimental modules for research.

A key part of the architecture is precise time synchronization between all onboard computers. The system uses a hardware signal called Pulse Per Second (PPS) from the GNSS modules to align the internal clocks of each computer. This signal marks the exact start of every second and is combined with the GNSS time data to keep all systems synchronized. A background process running in Linux automatically adjusts the system clocks based on this information, ensuring that all sensor data are timestamped with the same reference time. This sub-microsecond synchronization is essential for accurate sensor fusion, navigation, and SLAM.

An overview of the complete software architecture and node interconnections is shown in Figure 12 down below on the next page.

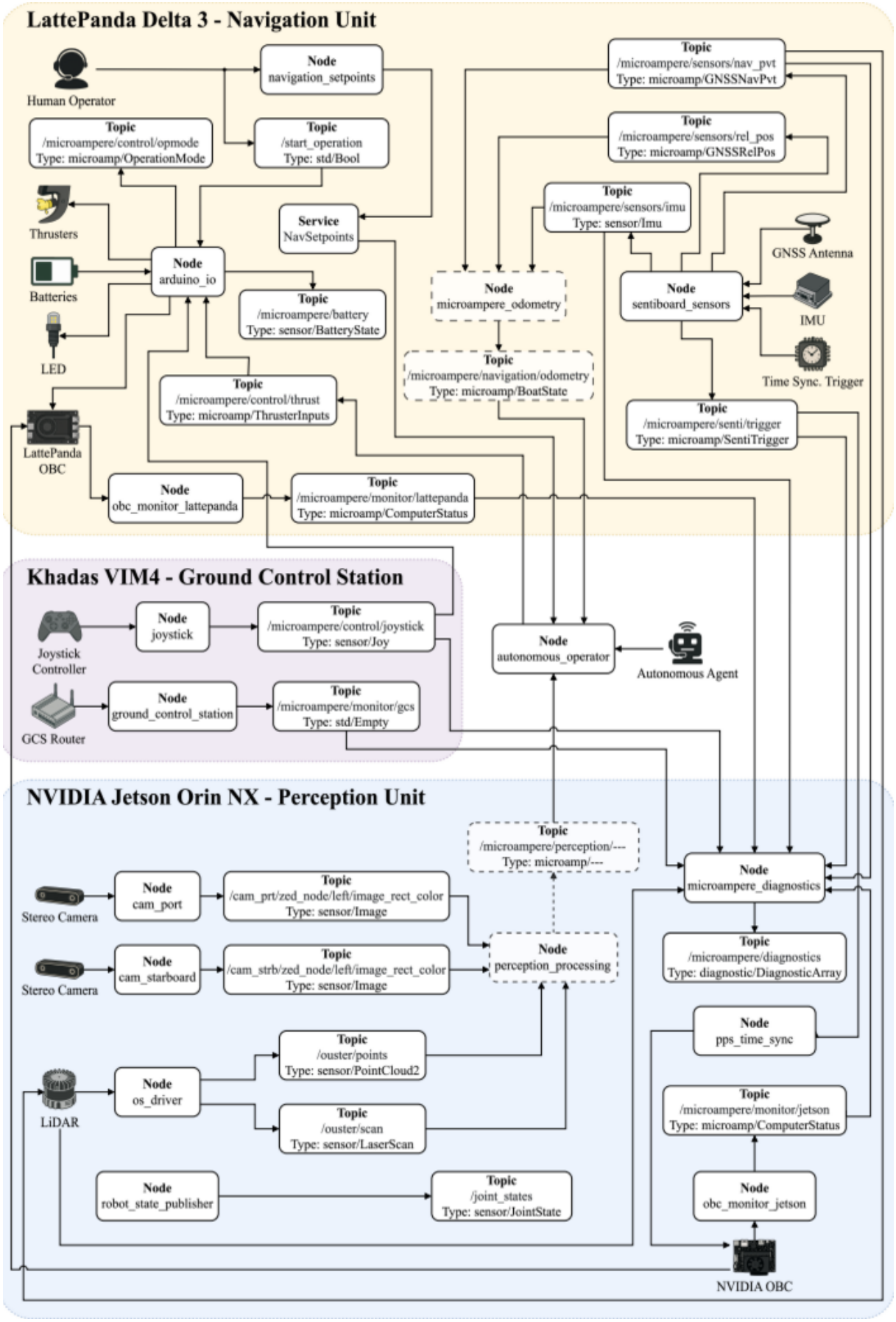


Figure 12: Overview of the software architecture for the microAmpere ASV, showing the distributed communication between perception, navigation, and control units. Picture taken from Henrik Reimers master thesis on microAmpere ASV.[4]

3.6 ROS2

The Robot Operating System 2 (ROS2) is a middleware framework designed for distributed robotic systems. It provides a modular communication layer that connects sensors, actuators, and computational nodes through a unified publish and subscribe architecture. ROS2 extends the original ROS1 with support for real time communication, security, and multi platform operation, making it suitable for both research and industrial use.

The structure of ROS2 is organized in several layers as shown in Figure 13.

The *"Operating System Layer"* provides hardware abstraction and supports multiple systems such as Linux, Windows, and macOS.

The *"ROS Middleware Layer (RMW)"* implements communication based on the Data Distribution Service (DDS) standard. DDS enables decentralized peer to peer communication without a central master, providing real time capabilities, reliability control, and quality of service policies for each data connection.

The *"ROS Client Library (RCL)"* defines the core functionality of ROS2 nodes in C, with language specific APIs such as `rclcpp` for C++ and `rclpy` for Python. This allows consistent behavior across different programming languages.

The *"User Code Layer"* contains the application specific nodes implemented by the developer. Each node handles one task such as sensor data processing, control, or estimation, and communicates with other nodes using topics, services, or actions. This modular approach improves scalability, debugging, and code reuse.

Overall, ROS2 provides a flexible communication framework that enables distributed systems to exchange data deterministically, synchronize timing, and scale across multiple computing units in real time applications.

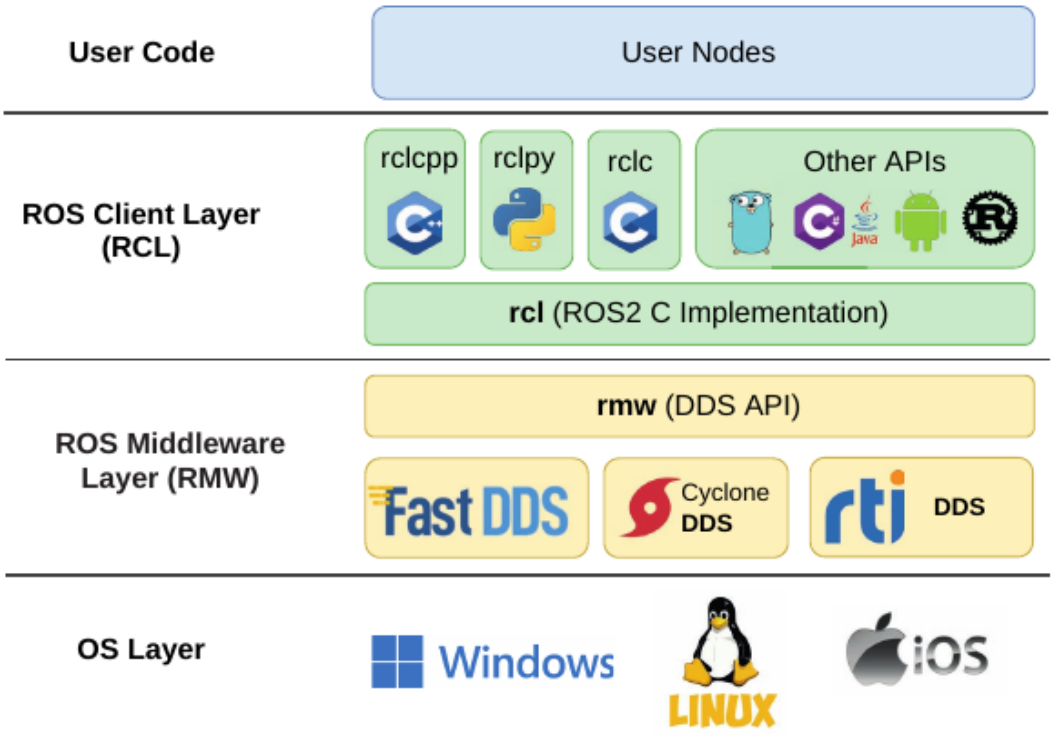


Figure 13: Layered structure of ROS2 showing the relation between the OS, middleware, client library, and user application layers.[8]

ROS2 is used as the main communication framework because it provides a decentralized, reliable, and real time capable architecture for robotic systems. Unlike ROS1, which depended on a single master node to manage communication, ROS2 uses the DDS middleware to automatically discover and connect nodes on the same network. This removes single points of failure and enables multiple computers to share data seamlessly. Each node in ROS2 can publish, subscribe, or request services without a central broker, which makes it ideal for distributed systems such as autonomous vehicles where perception, navigation, and control units must operate independently but still exchange synchronized information.

Communication in ROS2 is built on a publish and subscribe model. Nodes publish data on topics, and any node subscribed to that topic receives the data in real time. This approach decouples software components so that each one can be modified, restarted, or replaced without affecting others. For request and response type communication, ROS2 uses services, and for long running or feedback based tasks, it uses actions. Together these communication types allow flexible interaction between nodes, supporting both high frequency data streams such as IMU readings and low rate commands such as mission updates.

The middleware, based on DDS, handles message serialization, transport, and discovery automatically. DDS offers configurable Quality of Service (QoS) settings, allowing developers to control how data are delivered, stored, and synchronized. For example, high priority sensor data can be transmitted using reliable communication, while large perception messages can be sent using best effort mode to reduce latency. This control is essential in real time systems where timing and determinism are critical.

Time synchronization and deterministic execution are further supported through the ROS2 clock and timestamping mechanisms. Each message carries a precise time reference, which enables accurate sensor fusion and replay of recorded data. The system can integrate with external synchronization sources such as the GNSS Pulse Per Second (PPS) signal to align clocks between distributed nodes. This ensures that all data within the system share the same temporal reference, which is necessary for consistent estimation, control, and SLAM operations.

Another advantage of ROS2 is its modularity and reusability. Nodes are grouped into packages that can be developed, built, and deployed independently. The use of launch files written in Python allows flexible startup sequences, parameter configuration, and automatic interconnection of nodes. This makes ROS2 highly maintainable and adaptable to future upgrades, such as adding new sensors or switching computing hardware without rewriting the entire system.

Overall, ROS2 provides a robust, standardized, and industry ready framework for building distributed robotic systems. It forms the backbone of modern autonomous platforms by managing data flow, synchronization, and process isolation while maintaining real time performance and flexibility for research and development.

4 System Modeling

4.1 Introduction

Reliable navigation and mapping for an autonomous vessel depend on accurate mathematical models that describe how the vessel moves and how its sensors perceive the surrounding environment. These models form the foundation for state estimation, control, and SLAM algorithms, ensuring that all physical quantities such as position, velocity, and orientation are consistently defined and propagated over time.

The modeling framework is divided into two complementary parts, kinematics and dynamics. Kinematics focuses on the geometric description of motion without considering the forces that cause it. It defines how position, velocity, and orientation are represented and transformed between coordinate frames. Global reference frames such as the World Geodetic System 1984 (WGS84) and the Earth Centered Earth Fixed (ECEF) frame describe the Earth geometry and provide absolute positioning. Local frames such as the North East Down (NED) and Body frames define the vessel motion relative to its own orientation and surroundings. The transformation chain between these frames ensures that all measured and estimated quantities are consistently expressed and can be converted between global and local coordinates.

Dynamics extend the kinematic framework by describing how forces and moments act on the vessel to generate motion. This relationship forms the basis of the motion models used in navigation and estimation. Two main formulations are considered. The Marine Craft Model by Thor Inge Fossen [9] provides a comprehensive six degree of freedom (6 DOF) description that captures hydrodynamic effects, external disturbances, and control inputs, making it ideal for simulation and model based control. The Inertial Navigation System (INS) model, derived from the work of Edmund Brekke in Fundamentals of Sensor Fusion [10], offers a simplified sensor driven formulation that relies directly on IMU and GNSS data. This approach is better suited for real-time operation and SLAM applications, where computational efficiency and robustness to environmental uncertainty are prioritized over detailed hydrodynamic accuracy.

The complete mathematical framework connects these kinematic and dynamic representations into a unified structure. It defines how vessel states are represented, transformed, and propagated in time using rotation formulations such as Euler angles, quaternions, and Lie group representations. It establishes consistent conventions between global and local reference frames and expresses the relationships between linear and angular motion necessary for deriving velocities, accelerations, and time derivatives of position and orientation.

To achieve stable and accurate temporal evolution of the state, numerical solvers are used to integrate the underlying differential equations. Classical integration schemes such as Newton-Euler and Runge-Kutta methods are employed to propagate the vessel dynamics in discrete time while minimizing numerical drift and maintaining physical consistency. The choice of solver has a significant impact on model accuracy, especially in real-time applications where integration stability and computational efficiency determine overall system performance.

Together, these formulations provide a rigorous mathematical foundation for navigation and mapping in autonomous surface vessels. They enable consistent state representation, accurate propagation, and reliable fusion of multi-sensor information, which are all essential for achieving robust autonomy and for the integration of advanced perception systems such as SSS SLAM.

4.2 Orientation Representations

4.2.1 Euler Angles

Euler angles provide a simple and intuitive method for representing three dimensional orientation through a sequence of rotations about the principal axes. The orientation of a rigid body is described by three angles corresponding to yaw, pitch, and roll, which define the body's rotation relative to a fixed reference frame. These angles are widely used in navigation and control applications due to their clear geometric interpretation and direct relationship to measurable quantities such as heading and inclination.

In the North East Down (NED) convention commonly used for marine and aerial vehicles, the rotation sequence is typically defined as a rotation about the z-axis (yaw, ψ), followed by the y-axis (pitch, θ), and finally the x-axis (roll, ϕ). This corresponds to the rotation matrix:

$$R = R_x(\phi)R_y(\theta)R_z(\psi)$$

$$R_x(\phi) = \begin{bmatrix} 1 & 0 & 0 \\ 0 & \cos\phi & -\sin\phi \\ 0 & \sin\phi & \cos\phi \end{bmatrix}, \quad R_y(\theta) = \begin{bmatrix} \cos\theta & 0 & \sin\theta \\ 0 & 1 & 0 \\ -\sin\theta & 0 & \cos\theta \end{bmatrix}, \quad R_z(\psi) = \begin{bmatrix} \cos\psi & -\sin\psi & 0 \\ \sin\psi & \cos\psi & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

Combining these gives the full NED transformation from the body frame to the navigation frame:

$$R_b^n = \begin{bmatrix} c_\theta c_\psi & s_\phi s_\theta c_\psi - c_\phi s_\psi & c_\phi s_\theta c_\psi + s_\phi s_\psi \\ c_\theta s_\psi & s_\phi s_\theta s_\psi + c_\phi c_\psi & c_\phi s_\theta s_\psi - s_\phi c_\psi \\ -s_\theta & s_\phi c_\theta & c_\phi c_\theta \end{bmatrix}$$

where $s_\bullet = \sin(\bullet)$ and $c_\bullet = \cos(\bullet)$ for brevity. The NED frame transformation can be visualized in Figure 14, which illustrates the intrinsic yaw-pitch-roll rotation sequence following the NED convention. This sequence shows how the body frame is successively rotated about the global z-axes, y-axes, and x-axes to achieve the desired orientation.

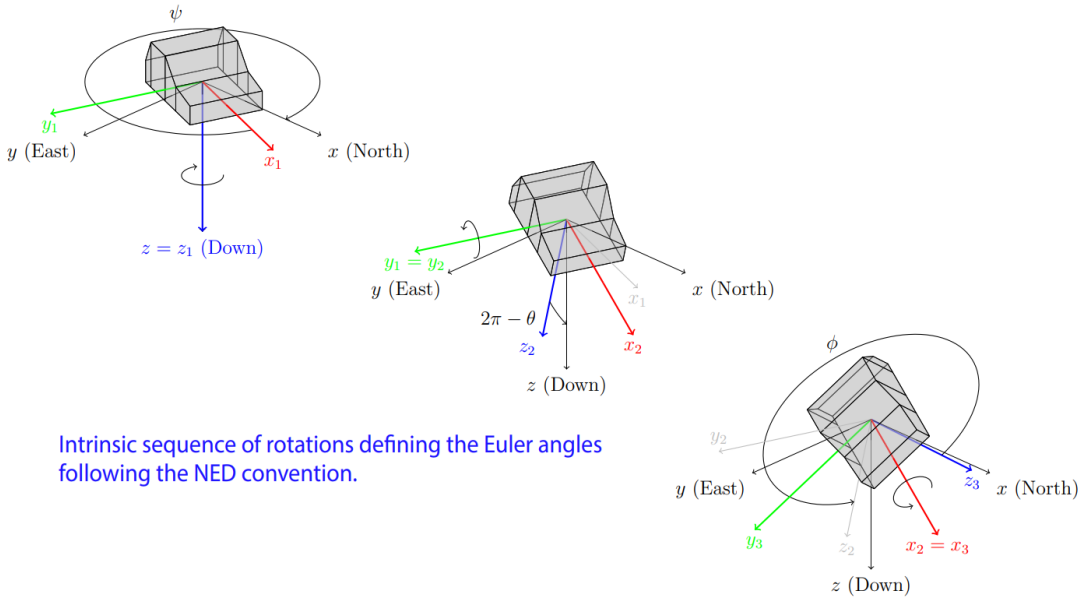


Figure 14: Intrinsic sequence of rotations defining the Euler angles following the NED convention. Figure taken from Edmund Brekke book on Fundamentals of Sensor Fusion.[10]

The main advantages of Euler angles are their simplicity, minimal parameter count, and intuitive physical meaning. Each angle directly corresponds to an observable rotational motion, which simplifies both system design and debugging. They also integrate naturally with classical control systems, where angular rates are easily expressed as time derivatives of Euler angles.

However, the Euler representation suffers from a singularity problem known as *"gimbal lock"*, which in the NED convention occurs when the pitch angle approaches $\theta = \pm 90°$. At this configuration,

the yaw and roll axes align, resulting in the loss of one rotational degree of freedom. In this state, small changes in pitch can cause disproportionately large or undefined changes in yaw and roll rates. Mathematically, the transformation matrix that relates Euler angle rates to body angular velocities becomes ill-conditioned, leading to singularities where the angular rate terms tend toward infinity. This introduces numerical instability and unreliable attitude estimates, which are particularly problematic for navigation and control systems relying on smooth and continuous angular motion.

The relationship between the body angular velocity vector $\boldsymbol{\omega}_b = [p \ q \ r]^T$ and the Euler angle rate vector $\dot{\boldsymbol{\Theta}} = [\dot{\phi} \ \dot{\theta} \ \dot{\psi}]^T$ for the NED yaw-pitch-roll convention is expressed as:

$$\begin{bmatrix} p \\ q \\ r \end{bmatrix} = \begin{bmatrix} 1 & 0 & -\sin\theta \\ 0 & \cos\phi & \sin\phi\cos\theta \\ 0 & -\sin\phi & \cos\phi\cos\theta \end{bmatrix} \begin{bmatrix} \dot{\phi} \\ \dot{\theta} \\ \dot{\psi} \end{bmatrix}$$

When the pitch angle approaches $\theta = \pm 90°$, the cosine term $\cos\theta$ tends to zero, causing the matrix to lose rank and become singular. As a result, any attempt to compute Euler rate derivatives or integrate attitude dynamics at this configuration leads to undefined or infinite angular velocities.

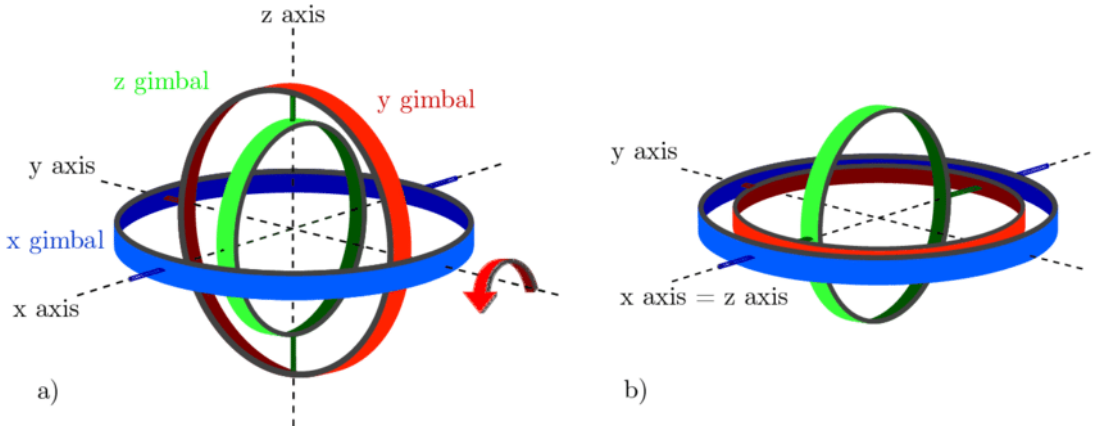


Figure 15: Illustration of gimbal lock where the yaw and roll axes coincide, causing loss of one rotational degree of freedom. Picture taken from a research paper that explains gimbal lock with illustrations.[11]

Despite this limitation, Euler angles remain a practical choice for surface and marine robotics, where pitch and roll angles typically remain small. Their intuitive interpretation and computational simplicity make them well suited for real-time estimation and control near level operating conditions. In the case of an ASV, large attitude excursions are highly unlikely, as excessive roll or pitch would indicate a loss of stability or complete capsizing. Under such operational constraints, the singularity at $\theta = \pm 90°$ is never encountered, making the Euler representation both sufficient and efficient for navigation and SLAM applications.

4.2.2 Quaternions

Quaternions provide a compact and singularity free alternative to Euler angles for representing three dimensional orientations. They extend complex numbers into four dimensions and are defined as a scalar and a three component vector

$$\mathbf{q} = \begin{bmatrix} q_w \\ q_x \\ q_y \\ q_z \end{bmatrix} = \begin{bmatrix} \cos\frac{\theta}{2} \\ \hat{u}_x \sin\frac{\theta}{2} \\ \hat{u}_y \sin\frac{\theta}{2} \\ \hat{u}_z \sin\frac{\theta}{2} \end{bmatrix}$$

where θ is the rotation angle and $\hat{u} = [\hat{u}_x \ \hat{u}_y \ \hat{u}_z]^T$ is the unit vector defining the rotation axis. The quaternion $\mathbf{q}$ therefore represents a rotation of θ radians about $\hat{u}$. To ensure that it encodes a valid rotation, it must satisfy the unit norm constraint

$$\|\mathbf{q}\| = \sqrt{q_w^2 + q_x^2 + q_y^2 + q_z^2} = 1$$

Normalization is typically enforced after each update step in numerical integration or filtering to avoid drift due to floating point errors. This is done by dividing the quaternion by its magnitude:

$$\mathbf{q}_{\text{normalized}} = \frac{\mathbf{q}}{\|\mathbf{q}\|} = \frac{1}{\sqrt{q_w^2 + q_x^2 + q_y^2 + q_z^2}} \begin{bmatrix} q_w \\ q_x \\ q_y \\ q_z \end{bmatrix}$$

A geometric interpretation of quaternions can be seen in Figure 16. Quaternions can be viewed as points on the unit four dimensional hypersphere S^3, composed of a real component and a three dimensional imaginary subspace spanned by (i, j, k). The real component represents the cosine of half the rotation angle, while the imaginary vector component encodes the rotation axis scaled by the sine of half the angle. Together, they define a rotation in three dimensional space through the relation $\mathbf{q} = \cos\frac{\theta}{2} + \hat{u}\sin\frac{\theta}{2}$. Each point on this hypersphere corresponds to a unique orientation of a rigid body, and continuous motion along its surface represents a smooth change in attitude without encountering any discontinuities or singularities.

When a vector $\mathbf{v}$ is rotated using $\mathbf{qvq}^{-1}$, the operation effectively applies a double rotation, first through $\mathbf{q}$ and then through its conjugate $\mathbf{q}^{-1}$, producing a single pure 3D rotation about the axis $\hat{u}$ by an angle θ. The scalar part $\cos(\theta/2)$ defines the rotation magnitude, while the vector part $\sin(\theta/2)\hat{u}$ specifies the rotation axis in the imaginary (i, j, k) space. This representation forms the basis for quaternion based attitude kinematics used in estimation and control.

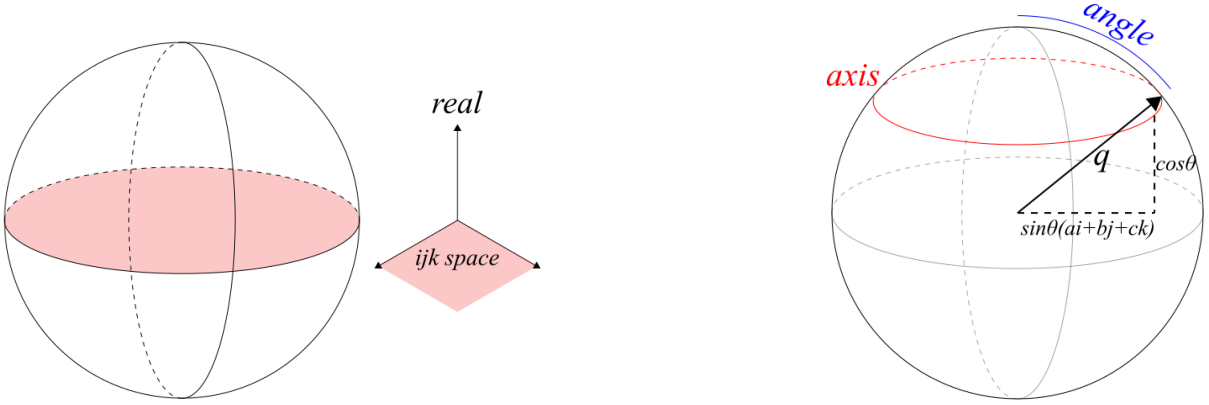


Figure 16: Geometric interpretation of quaternions on the unit hypersphere S^3. (Left) Decomposition into the real axis and imaginary (i, j, k) subspace. (Right) Axis-angle representation showing how a quaternion encodes rotation by θ about the unit axis $\hat{u}$. Picture taken from article that explains quarterion rotation in a intuitive way.[12]

Rotations using quaternions are performed through quaternion multiplication, which is associative but not commutative. Given two orientations $\mathbf{q}_1$ and $\mathbf{q}_2$, their combined rotation is expressed as

$$\mathbf{q}_{\text{combined}} = \mathbf{q}_2 \otimes \mathbf{q}_1,$$

where $\otimes$ denotes the quaternion product. The rotation of a vector $\mathbf{v}$ in three dimensional space can then be performed as

$$\mathbf{v}' = \mathbf{q} \otimes \mathbf{v}_q \otimes \mathbf{q}^{-1},$$

where $\mathbf{v}_q = [0 \ v_x \ v_y \ v_z]^T$ is the quaternion form of the vector $\mathbf{v}$. This operation applies the rotation represented by $\mathbf{q}$ to $\mathbf{v}$ in a smooth and continuous manner without encountering singularities.

For smooth orientation transitions, quaternions can be interpolated using Spherical Linear Interpolation (SLERP). Instead of blending components linearly, SLERP moves along the great circle that connects two orientations on the unit quaternion sphere S^3. This ensures that the interpolated motion follows a constant angular velocity and remains at a fixed distance from the sphere center, preserving rotation smoothness and avoiding distortion.

Given two normalized quaternions $\mathbf{q}_1$ and $\mathbf{q}_2$, the interpolation for $t \in [0, 1]$ is defined as

$$\text{SLERP}(\mathbf{q}_1, \mathbf{q}_2; t) = \frac{\sin((1-t)\Omega)}{\sin(\Omega)}\mathbf{q}_1 + \frac{\sin(t\Omega)}{\sin(\Omega)}\mathbf{q}_2,$$

where $\Omega = \cos^{-1}(\mathbf{q}_1 \cdot \mathbf{q}_2)$ represents the angle between the two quaternions. When $t = 0$, the result is $\mathbf{q}_1$, and when $t = 1$, the result is $\mathbf{q}_2$. Intermediate values of t trace the shortest rotation path between the two orientations.

This method is widely used in robotics, computer graphics, and navigation to generate smooth transitions between different orientations. SLERP ensures that intermediate orientations are evenly spaced in angular distance, resulting in constant rotational velocity, a property important for stable attitude control and estimation.

To ensure interpolation follows the shortest possible rotation path, many implementations adjust the sign of one quaternion when their dot product is negative. This avoids interpolation along the long arc (greater than 180°) and guarantees smooth motion along the minimal angular distance. Figure 17 illustrates this process, showing both the *"short"* and the *"natural"* interpolation paths along the quaternion sphere.

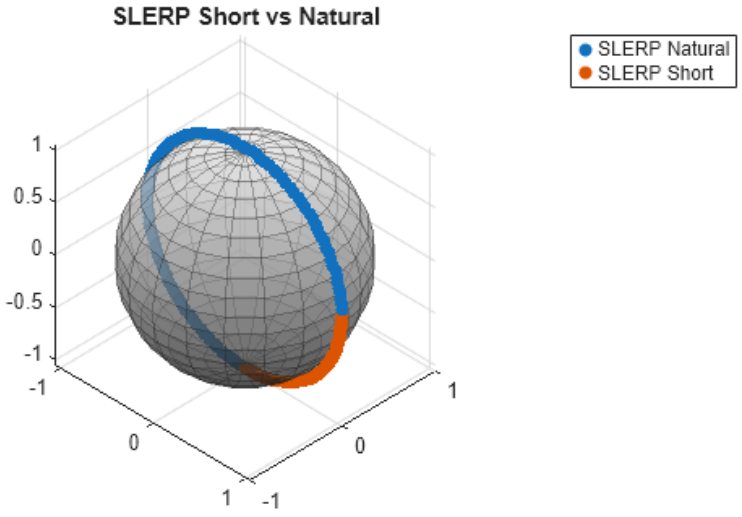


Figure 17: Spherical linear interpolation (SLERP) between two orientations $\mathbf{q}_1$ and $\mathbf{q}_2$ on the unit quaternion sphere. The *"short"* path minimizes angular distance, while the *"natural"* path follows the original direction without sign correction. Figure taken from MATLAB documentation for illustration purposes.[13]

In practical applications, quaternions are widely used in estimators and sensors where singularity free orientation handling is critical. The IMU on the microAmpere platform outputs attitude in quaternion form, which integrates naturally with sensor fusion algorithms such as the Error State Kalman Filter

(ESKF) discussed later in the report. These algorithms rely on quaternion algebra to represent small attitude perturbations efficiently and avoid numerical instability associated with Euler angle singularities.

The main advantages of quaternions include their smooth and continuous representation of orientation, absence of gimbal lock, and computational efficiency in rotation composition and interpolation. They maintain numerical stability under integration and are well suited for optimization and estimation algorithms. However, quaternions require one additional parameter compared to Euler angles and lack intuitive physical meaning, as their four components do not directly correspond to measurable angular quantities. Furthermore, they are less straightforward to interpret and manipulate mathematically, often requiring specialized operations such as normalization, conjugation, and noncommutative multiplication. Despite these challenges, their robustness and compatibility with modern sensor fusion and control frameworks make quaternions a preferred internal representation for real-time navigation and estimation systems.

4.2.3 Lie Groups and Manifolds

Lie groups provide a mathematical framework for representing continuous and smooth transformations such as rotation and translation in three dimensional space. They combine the properties of algebraic groups and differentiable manifolds, allowing both analytical manipulation and geometric interpretation of motion. In navigation, control, and robotics, Lie groups enable consistent handling of orientation and pose without the discontinuities or ambiguities present in minimal representations such as Euler angles.

The rotation group SO(3), called the Special Orthogonal Group, represents all possible 3D rotations. Each rotation is described by a 3×3 matrix R that satisfies

$$R^T R = I, \quad \det(R) = 1.$$

This group is smooth and continuous, meaning that small changes in orientation correspond to small movements on the manifold. To describe small or incremental rotations, SO(3) is associated with its tangent space, the Lie algebra $\mathfrak{so}(3)$.

Elements of $\mathfrak{so}(3)$ are skew-symmetric matrices that encode infinitesimal rotations. A rotation vector $\boldsymbol{\omega} = [\omega_x \ \omega_y \ \omega_z]^T$ can be written as

$$\boldsymbol{\omega}^\times = \begin{bmatrix} 0 & -\omega_z & \omega_y \\ \omega_z & 0 & -\omega_x \\ -\omega_y & \omega_x & 0 \end{bmatrix}.$$

This matrix form is simply another way of expressing the cross product $\boldsymbol{\omega} \times \mathbf{v}$, which rotates a vector $\mathbf{v}$ by a small angle.

The exponential map connects this local representation to an actual finite rotation on SO(3):

$$R = \exp(\boldsymbol{\omega}^\times),$$

And the inverse operation (the logarithmic map) retrieves the corresponding small rotation from R:

$$\boldsymbol{\omega} = \log(R).$$

These two functions allow smooth transitions between local angular velocity representations and full 3D orientations, which is highly useful for filters and optimizers for SLAM.

To include translation as well as rotation, the concept extends to SE(3), the so called Special Euclidean Group, which describes full 3D rigid body motion:

$$T = \begin{bmatrix} R & \mathbf{p} \\ 0 & 1 \end{bmatrix}, \quad T \in \mathrm{SE}(3) \tag{1}$$

Here R is orientation and $\mathbf{p}$ is position. This group provides a consistent mathematical way to represent a vehicles full pose and combine both rotational and translational motion.

In robotics and navigation, SO(3) and SE(3) are used to describe smooth and continuous motion in three dimensional space. SO(3) represents pure rotations, while SE(3) extends this to include both rotation and translation, forming the full rigid body pose. These groups define motion directly on the manifold, ensuring mathematically consistent and globally valid transformations without singularities.

The curved surface of the SE(3) manifold represents all possible poses of a rigid body in space. The tangent plane at the identity, denoted $\mathfrak{se}(3)$, represents small local motions that can be combined and integrated into full poses using the exponential map $\exp(\cdot)$. Conversely, the logarithmic map $\log(\cdot)$ converts a pose difference back into this local space. Together, these mappings allow smooth transitions between small local displacements and full 3D transformations, which is essential for analyzing and composing motion in robotics and control.

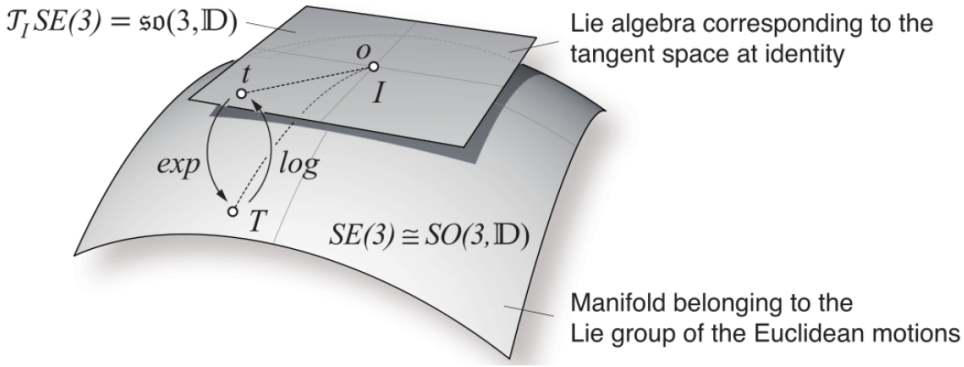


Figure 18: Visualization of the SE(3) manifold and its tangent space $\mathfrak{se}(3)$. Small motions are represented in the tangent space and mapped to the manifold through the exponential map $\exp(\cdot)$, while the logarithmic map $\log(\cdot)$ projects manifold elements back to the local linear space. Figure taken from Nikolay Atanasov lecture notes on Sensing & Estimation in Robotics.[14]

Lie group formulations are fundamental in modern SLAM systems because they provide a mathematically consistent way to represent and manipulate poses in three dimensions. When building and updating a map, each robot or camera pose is an element of SE(3), combining both rotation and translation into a single compact structure. This allows pose composition, inversion, and differentiation to be performed directly on the manifold without approximations or singularities. In practice, this means that motion updates, sensor transformations, and loop closure corrections can all be handled through clean matrix operations that are both efficient and numerically stable.

Using Lie groups also enables fast and scalable computation, a necessity when optimizing over thousands of poses and landmarks in large scale SLAM problems. The same mathematical principles are applied in computer graphics and 3D rendering, where SE(3) transformations are used to efficiently manipulate hundreds or thousands of objects and vertices in real time. By working on the manifold, transformations remain consistent regardless of scale or complexity, ensuring that rotations and translations compose correctly without distortion.

Overall, Lie group representations allow SLAM and mapping algorithms to combine geometry, motion, and optimization within a single unified framework. They ensure that the estimated trajectory and map remain globally consistent, even after many iterations of motion and correction, making them the foundation of accurate and robust 3D perception in robotics.

4.2.4 Handy Conversions

The following conversion formulas are taken directly from Edmund Brekkes book on *"Fundamentals of Sensor Fusion"* [10]. They are commonly used to convert between different orientation representations for navigation, control, and estimation.

Euler to Quaternion

$$\mathbf{q} = \begin{bmatrix} q_w \\ q_x \\ q_y \\ q_z \end{bmatrix} = \begin{bmatrix} \cos\frac{\phi}{2}\cos\frac{\theta}{2}\cos\frac{\psi}{2} + \sin\frac{\phi}{2}\sin\frac{\theta}{2}\sin\frac{\psi}{2} \\ \sin\frac{\phi}{2}\cos\frac{\theta}{2}\cos\frac{\psi}{2} - \cos\frac{\phi}{2}\sin\frac{\theta}{2}\sin\frac{\psi}{2} \\ \cos\frac{\phi}{2}\sin\frac{\theta}{2}\cos\frac{\psi}{2} + \sin\frac{\phi}{2}\cos\frac{\theta}{2}\sin\frac{\psi}{2} \\ \cos\frac{\phi}{2}\cos\frac{\theta}{2}\sin\frac{\psi}{2} - \sin\frac{\phi}{2}\sin\frac{\theta}{2}\cos\frac{\psi}{2} \end{bmatrix}$$

Quaternion to Euler

$$\begin{bmatrix} \phi \\ \theta \\ \psi \end{bmatrix} = \begin{bmatrix} \operatorname{atan2}\big(2(q_w q_x + q_y q_z),\, 1 - 2(q_x^2 + q_y^2)\big) \\ \operatorname{asin}\big(2(q_w q_y - q_z q_x)\big) \\ \operatorname{atan2}\big(2(q_w q_z + q_x q_y),\, 1 - 2(q_y^2 + q_z^2)\big) \end{bmatrix}$$

Euler to SO(3)

$$R_{from}^{to} = R_b^n = R_x(\phi)R_y(\theta)R_z(\psi)$$

$$R_x(\phi) = \begin{bmatrix} 1 & 0 & 0 \\ 0 & \cos\phi & -\sin\phi \\ 0 & \sin\phi & \cos\phi \end{bmatrix}, \quad R_y(\theta) = \begin{bmatrix} \cos\theta & 0 & \sin\theta \\ 0 & 1 & 0 \\ -\sin\theta & 0 & \cos\theta \end{bmatrix}, \quad R_z(\psi) = \begin{bmatrix} \cos\psi & -\sin\psi & 0 \\ \sin\psi & \cos\psi & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

Quaternion to SO(3)

$$R_{from}^{to} = R_b^n = \begin{bmatrix} 1 - 2(q_y^2 + q_z^2) & 2(q_x q_y - q_z q_w) & 2(q_x q_z + q_y q_w) \\ 2(q_x q_y + q_z q_w) & 1 - 2(q_x^2 + q_z^2) & 2(q_y q_z - q_x q_w) \\ 2(q_x q_z - q_y q_w) & 2(q_y q_z + q_x q_w) & 1 - 2(q_x^2 + q_y^2) \end{bmatrix}$$

These relationships provide a convenient reference for converting between orientation representations depending on the application, whether for computation, visualization, or integration with sensor data.

4.3 Reference Frames and Transformations

4.3.1 Basic Translation and Rotation Transformations

Rigid body motion in three dimensions can be expressed compactly using the homogeneous transformation matrix $T \in \mathrm{SE}(3)$ defined in Equation 1. These matrices combine rotation and translation into a single representation and are used to map coordinates between frames such as the navigation (NED), body, and sensor frames.

The pose of the body frame relative to the navigation frame is written as

$$T_b^n = \begin{bmatrix} R_b^n & \mathbf{p}_b^n \\ 0 & 1 \end{bmatrix}$$

where R_b^n is the rotation matrix from the body to the NED frame, and $\mathbf{p}_b^n$ is the position of the body origin expressed in NED coordinates.

A point $\mathbf{x}_b^b$ expressed in the body frame can be transformed to the NED frame as:

$$\mathbf{x}_b^n = T_b^n \begin{bmatrix} \mathbf{x}_b^b \\ 1 \end{bmatrix} = R_b^n \mathbf{x}_b^b + \mathbf{p}_b^n$$

Composing multiple transformations is performed through matrix multiplication. For instance, if T_s^b defines the transformation from a sensor frame to the body frame, then the sensor pose in the NED frame becomes

$$T_s^n = T_b^n T_s^b$$

The inverse transformation, which converts coordinates from NED back to the body frame, is given by

$$T_n^b = (T_b^n)^{-1} = \begin{bmatrix} (R_b^n)^T & -(R_b^n)^T \mathbf{p}_b^n \\ 0 & 1 \end{bmatrix}$$

This matrix formulation provides a clean and consistent way to represent spatial relationships between the navigation, body, and sensor frames. It is fundamental in robotics, navigation, and control, where accurate frame alignment and transformation chaining are essential for pose estimation and sensor fusion.

These homogeneous transformations form the foundation for defining global and local reference frames such as WGS84, ECEF, NED, and Body, which are described in the following sections

4.3.2 Global Reference Frames

Global reference frames provide the foundation for representing absolute positions on Earth and are essential for all GNSS based navigation and mapping systems. Since GNSS receivers provide position estimates in a global Earth fixed frame, while navigation and control systems typically operate in local frames such as NED or body coordinates, it becomes necessary to define consistent global reference models to enable accurate transformations between these coordinate systems.

The two main reference systems used for this purpose are the World Geodetic System 1984 (WGS84) geodetic model, which defines the Earths ellipsoidal shape and geodetic coordinates (latitude, longitude, altitude), and the Earth-Centered Earth-Fixed (ECEF) Cartesian frame, which expresses these same positions as 3D Cartesian coordinates centered at the Earths center of mass. These systems provide the common link between GNSS measurements and local navigation frames used in estimation and control.

WGS84
The World Geodetic System 1984 (WGS84) is the global geodetic reference used by all major GNSS systems. It defines an Earth-fixed ellipsoidal model that closely approximates the planets mean shape, accounting for the equatorial bulge caused by rotation. Although regional realizations such as *"ETRS89"* (European Terrestrial Reference System 1989) are used in Europe to reduce tectonic drift, the WGS84 reference remains the standard for GNSS based navigation in Norway and most marine applications, with differences between the two systems being only a few decimeters.

The WGS84 model defines the Earth as an oblate ellipsoid, flattened at the poles and expanded at the equator. It serves as the global geodetic reference for GPS and most modern satellite navigation systems. The ellipsoid parameters are defined as $a = 6378137.0$ m and $f = 1/298.257223563$, where a is the semi major axis and f is the flattening factor. The semi minor axis can then be computed as $b = a(1-f)$.

A position on Earth is defined by three geodetic coordinates. First the latitude φ, which is the angle between the equatorial plane and the ellipsoid normal. Then the longitude λ, which is the angle between the Greenwich meridian and the projection of the point onto the equatorial plane. Finally the altitude h, which is the height above the ellipsoidal surface. These coordinates describe a points global position and are the standard output format from all GNSS receivers.

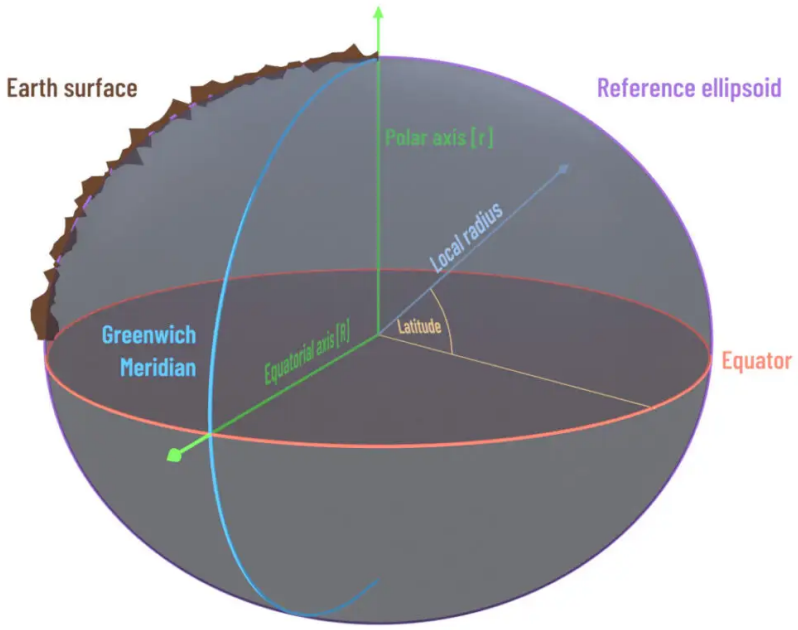


Figure 19: WGS84 ellipsoidal Earth model showing the relationship between latitude, longitude, and altitude. Figure taken from GeneSys Elektronik GmbH documentation.[15]

ECEF

The Earth-Centered Earth-Fixed (ECEF) frame is a three dimensional Cartesian coordinate system with its origin at the Earths center of mass. The x-axis passes through the intersection of the equator and the Greenwich meridian, the y-axis lies along the equator 90° east of the x-axis, and the z-axis points toward the North Pole. The ECEF frame rotates with the Earth and is commonly used for expressing GNSS positions in Cartesian form.

The transformation from geodetic coordinates WGS84 (φ, λ, h) to ECEF coordinates (x, y, z) is given by:

$$
\begin{aligned}
e^2 &= 2f - f^2, \\
N(\varphi) &= \frac{a}{\sqrt{1 - e^2 \sin^2 \varphi}}, \\
x &= (N(\varphi) + h) \cos\varphi \cos\lambda, \\
y &= (N(\varphi) + h) \cos\varphi \sin\lambda, \\
z &= \left[(1 - e^2)N(\varphi) + h\right] \sin\varphi
\end{aligned}
$$

where $N(\varphi)$ is the prime vertical radius of curvature and e is the first eccentricity of the ellipsoid. The

inverse conversion from ECEF to geodetic coordinates is typically solved iteratively due to the nonlinear nature of the equations.

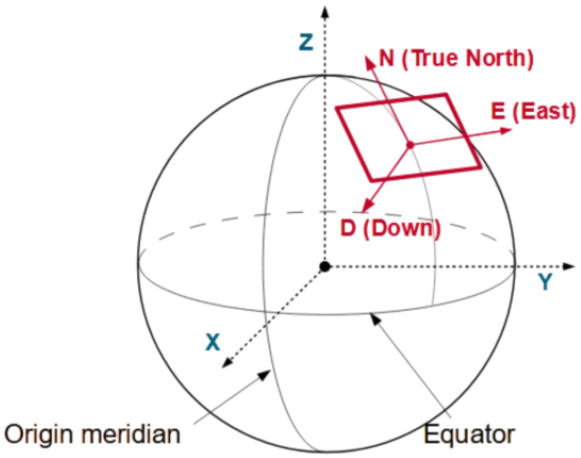


Figure 20: Earth-Centered Earth-Fixed (ECEF) coordinate system with origin at Earth's center and axes aligned with the equator and rotational axis. Figure taken from open source GNSS reference material.[16]

The WGS84 and ECEF frames together form the foundation of all global navigation and positioning systems. While WGS84 defines the geometric reference ellipsoid used by GNSS measurements, ECEF provides a Cartesian coordinate representation that is better suited for numerical computation, sensor fusion, and integration with local navigation frames such as NED or body fixed coordinates.

4.3.3 Local Reference Frames

Local reference frames are used to describe the motion and orientation of vehicles in a way that is intuitive and convenient for navigation, estimation, and control. In marine and aerial systems, the two most common local frames are the North-East-Down (NED) frame and the Body frame. These provide a consistent way to represent vehicle states such as position, velocity, and attitude relative to the Earth.

The NED frame is a local tangent frame fixed to a point on the Earths surface, with the x-axis pointing toward geographic north, the y-axis pointing east, and the z-axis pointing downward, perpendicular to the ellipsoid. This convention is widely used in navigation because it aligns naturally with compass directions and simplifies interpretation of GNSS and IMU data. The origin of the NED frame is typically defined at a reference point near the vehicles initial position, tangent to the Earth at that location.

The Body frame is attached to the vehicle itself, with its origin at the vehicles center of gravity or another defined point. The x-axis points forward along the vehicles longitudinal direction, the y-axis points to the right, and the z-axis points downward. All onboard sensor measurements, such as accelerations, angular velocities, and forces, are naturally expressed in this frame.

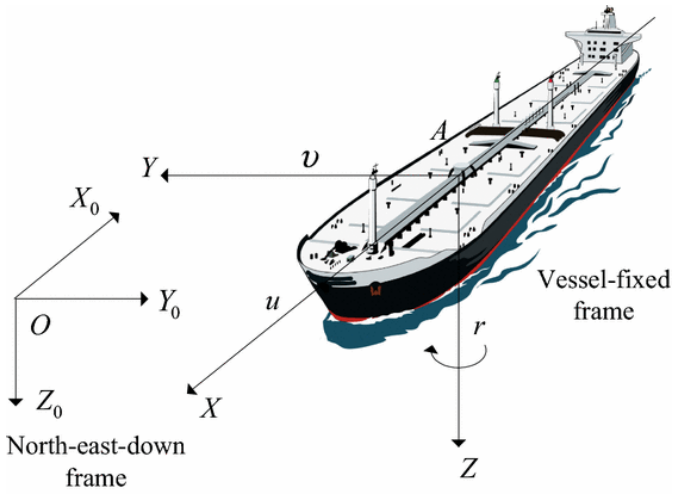


Figure 21: NED coordinate system attached to a marine vehicle. The illustration shows the local navigation axes and their relation to the body frame. Figure taken from a report on robust nonlinear control design for dynamic positioning of marine vessels.[17]

The relationship between the global and local reference frames follows the transformation chain:

$$\text{WGS84 (geodetic)} \rightarrow \text{ECEF (Cartesian)} \rightarrow \text{NED (local tangent)} \rightarrow \text{Body (vehicle-fixed)}$$

GNSS receivers typically provide positions in the WGS84 frame, expressed as geodetic coordinates (φ, λ, h), where φ is latitude, λ is longitude, and h is altitude above the reference ellipsoid. For computational purposes, these coordinates are first converted to the ECEF frame, which represents positions in Cartesian coordinates (x, y, z) relative to the Earth's center of mass.

To obtain a local navigation frame, the ECEF position is then rotated and translated into the NED frame, which defines a tangent plane fixed at a chosen reference point (φ_0, λ_0). The rotation from ECEF to NED is given by

$$R_e^n = \begin{bmatrix} -\sin\varphi_0 \cos\lambda_0 & -\sin\varphi_0 \sin\lambda_0 & \cos\varphi_0 \\ -\sin\lambda_0 & \cos\lambda_0 & 0 \\ -\cos\varphi_0 \cos\lambda_0 & -\cos\varphi_0 \sin\lambda_0 & -\sin\varphi_0 \end{bmatrix}$$

where φ_0 and λ_0 are the latitude and longitude of the local origin. The superscript n and subscript e indicate that this matrix transforms a vector expressed in the ECEF frame $\{e\}$ into the NED frame $\{n\}$.

Given the ECEF position of the vehicle $\mathbf{p}_{b/e}^e$ and the ECEF position of the NED origin $\mathbf{p}_{O/e}^e$, the vehicles position in NED coordinates is computed as

$$\mathbf{p}_{b/O}^n = \mathbf{p}_{b/e}^n + \mathbf{p}_{e/O}^n = R_e^n(\mathbf{p}_{b/e}^e - \mathbf{p}_{O/e}^e)$$

For onboard sensors, the position of a sensor $\{s\}$ relative to the vehicle body origin, expressed in the NED frame, is given by

$$\mathbf{p}_{s/b}^n = R_b^n \mathbf{p}_{s/b}^b$$

The absolute position of the sensor relative to the NED origin can then be found as

$$\mathbf{p}_{s/O}^n = \mathbf{p}_{s/b}^n + \mathbf{p}_{b/O}^n$$

where $\mathbf{p}_{s/b}^b$ is the known position of the sensor in the body frame, and R_b^n is the rotation matrix representing the vehicles attitude from the Body to the NED frame.

This formulation relates the sensor position to the global reference frame by first rotating the sensor offset from the body frame into the navigation frame, then translating it using the vehicles global position. The rotation matrix R_b^n is obtained from the vehicles orientation representation, typically parameterized using Euler angles or a unit quaternion, ensuring consistent alignment between the sensor,

body, and navigation coordinate frames.

Maintaining a consistent chain of transformations between these frames is essential for reliable navigation and estimation. In practice, GNSS delivers global positions in WGS84 or ECEF coordinates, while onboard IMU sensors provide measurements in the Body frame. Sensor fusion algorithms such as the Extended Kalman Filter (EKF) and Error State Kalman Filter (ESKF) depend on these transformations to express all quantities consistently within the NED frame used for state estimation and control.

4.4 Rigid Body Kinematics

4.4.1 Relevance for Navigation and Modeling

Rigid body kinematics provides the mathematical foundation for expressing motion, orientation, and acceleration consistently across coordinate frames. In navigation systems, these relationships allow the integration of measurements from inertial sensors, GNSS, and other sources within a unified dynamic model.

The transformation equations for position, velocity, and acceleration ensure that sensor data expressed in the body frame can be accurately related to the navigation frame, enabling correct estimation of the vehicles state. This consistency is essential for inertial navigation, attitude estimation, and sensor fusion, where body fixed measurements such as angular velocity and specific force must be mapped into global coordinates for integration and correction.

Moreover, the rigid body framework is fundamental for simulation and control system design. It allows modeling of vehicle dynamics, actuator response, and sensor placement with precise spatial relationships. The kinematic expressions derived in this section thus serve as the basis for dynamic modeling, state estimation, and motion prediction used in autonomous and navigation systems.

4.4.2 Position and Orientation

The position of a rigid body is represented by the vector $\mathbf{p}^n_{b/O}$, which denotes the location of the body frame origin expressed in the navigation (NED) frame. The orientation of the body is represented by the rotation matrix $R^n_b \in \mathrm{SO}(3)$, which maps a vector from the body frame $\{b\}$ to the navigation frame $\{n\}$.

The combined rigid body pose is described by the homogeneous transformation matrix $T^n_{b/O} \in \mathrm{SE}(3)$:

$$T^n_{b/O} = \begin{bmatrix} R^n_b & \mathbf{p}^n_{b/O} \\ 0 & 1 \end{bmatrix}$$

The kinematic relationship between the time derivative of position and the linear velocity is then given by

$$\dot{\mathbf{p}}^n_{b/O} = \mathbf{v}^n_{b/O}$$

where $\mathbf{v}^n_{b/O}$ is the velocity of the body origin expressed in the navigation frame.

4.4.3 Angular Velocity and Euler Angle Relationship

The angular velocity vector $\boldsymbol{\omega}^b_{b/O} = [p, q, r]^\top$ represents the instantaneous rotation rate of the body about its own axes, roll rate p about the x_b-axis (North), pitch rate q about the y_b-axis (East), and yaw rate r about the z_b-axis (Down).

When the body orientation is represented by ZYX Euler angles (yaw ψ, pitch θ, roll ϕ), the time derivative of the Euler angles relates to the body angular velocity through

$$\begin{bmatrix} \dot{\phi} \\ \dot{\theta} \\ \dot{\psi} \end{bmatrix} = T(\phi, \theta)\boldsymbol{\omega}^b_{b/O}$$

where the transformation matrix $T(\phi, \theta)$ for the NED convention is defined as

$$T(\phi, \theta) = \begin{bmatrix} 1 & \sin\phi \tan\theta & \cos\phi \tan\theta \\ 0 & \cos\phi & -\sin\phi \\ 0 & \sin\phi / \cos\theta & \cos\phi / \cos\theta \end{bmatrix}$$

This maps the body angular velocity to the Euler angle rates according to the NED convention.

The inverse relationship, expressing body angular velocity from Euler angle derivatives, is given by

$$\boldsymbol{\omega}^b_{b/O} = \begin{bmatrix} p \\ q \\ r \end{bmatrix} = T^{-1}(\phi, \theta) \begin{bmatrix} \dot{\phi} \\ \dot{\theta} \\ \dot{\psi} \end{bmatrix}$$

where

$$T^{-1}(\phi,\theta) = \begin{bmatrix} 1 & 0 & -\sin\theta \\ 0 & \cos\phi & \sin\phi\cos\theta \\ 0 & -\sin\phi & \cos\phi\cos\theta \end{bmatrix}$$

This formulation aligns with the NED coordinate convention and ensures correct mapping between angular velocities and Euler angle derivatives. The transformation matrix becomes singular at $\theta = \pm 90°$, corresponding to gimbal lock, where $\tan\theta$ diverges.

4.4.4 Angular Velocity and Quaternion Relationship

Quaternions provide a compact and singularity free representation of rotation. A unit quaternion $q = [q_0, q_1, q_2, q_3]^\top$ consists of one scalar and three vector components, often written as

$$q = \begin{bmatrix} q_0 \\ \mathbf{q}_v \end{bmatrix} = \begin{bmatrix} q_0 \\ q_1 \\ q_2 \\ q_3 \end{bmatrix}$$

where q_0 is the scalar part and $\mathbf{q}_v = [q_1, q_2, q_3]^\top$ is the vector part. The quaternion represents a rotation of angle θ around the unit axis $\hat{\mathbf{u}}$:

$$q = \begin{bmatrix} \cos\frac{\theta}{2} \\ \hat{\mathbf{u}}\sin\frac{\theta}{2} \end{bmatrix}$$

The time evolution of the quaternion is governed by the angular velocity vector $\boldsymbol{\omega}^b_{b/O} = [p, q, r]^\top$ measured in the body frame. The quaternion kinematics are given by

$$\dot{q} = \frac{1}{2}\Omega(\boldsymbol{\omega}^b_{b/O})q$$

where $\Omega(\boldsymbol{\omega}^b_{b/O})$ is the quaternion rate matrix:

$$\Omega(\boldsymbol{\omega}^b_{b/O}) = \begin{bmatrix} 0 & -p & -q & -r \\ p & 0 & r & -q \\ q & -r & 0 & p \\ r & q & -p & 0 \end{bmatrix}$$

Given the quaternion $q = [q_0, \mathbf{q}_v]^\top$ and its time derivative $\dot{q} = [\dot{q_0}, \dot{\mathbf{q}_v}]^\top$, the body angular velocity can be reconstructed directly as

$$\boldsymbol{\omega}^b_{b/O} = 2\big(q_0\dot{\mathbf{q}_v} - \dot{q_0}\mathbf{q_v} - \mathbf{q_v}^\times\dot{\mathbf{q}_v}\big).$$

Here, q_0 is the scalar part of the quaternion and $\mathbf{q} * v$ its vector part. This relation provides a direct and numerically stable way to compute the instantaneous angular velocity from quaternion derivatives while maintaining full consistency with rigid body rotational kinematics in the body frame. This formulation complements the quaternion rate equation which integrates angular velocity to update attitude over time. The quaternion must remain normalized, i.e. $|q| = 1$, to represent a valid rotation, and is therefore periodically renormalized during numerical integration as

$$q = \frac{q}{|q|}.$$

The corresponding rotation matrix from body to navigation frame is obtained as

$$R^n_b(q) = \begin{bmatrix} 1-2(q_2^2+q_3^2) & 2(q_1q_2-q_0q_3) & 2(q_1q_3+q_0q_2) \\ 2(q_1q_2+q_0q_3) & 1-2(q_1^2+q_3^2) & 2(q_2q_3-q_0q_1) \\ 2(q_1q_3-q_0q_2) & 2(q_2q_3+q_0q_1) & 1-2(q_1^2+q_2^2) \end{bmatrix}$$

This rotation matrix provides the mapping between the body and navigation frames, analogous to R^n_b obtained from Euler angles, but without the gimbal lock problem.

In practice, quaternion based orientation integration using $\dot{q} = \frac{1}{2}\Omega(\boldsymbol{\omega}^b_{b/O})q$ is preferred for real-time attitude propagation, while the equivalent rotation matrix or Euler angles are extracted only when needed for visualization or control.

4.4.5 Linear Velocity Relationship

The linear velocity of any point P fixed on a rigid body can be related to the velocity of a reference point O on the same body as

$$\mathbf{v}^b_{P/O} = \mathbf{v}^b_{b/O} + \mathbf{v}^b_{P/b} + \boldsymbol{\omega}^b_{b/O} \times \mathbf{p}^b_{P/b}$$

Here, $\mathbf{v}^b_{b/O}$ is the translational velocity of the body origin expressed in the body frame, $\mathbf{v}^b_{P/b}$ is the velocity of point P relative to the body (zero for fixed sensors), $\boldsymbol{\omega}^b_{b/O}$ is the body angular velocity, and $\mathbf{p}^b_{P/b}$ is the position of P relative to the body origin, both expressed in the body frame.

The cross product term $\boldsymbol{\omega}^b_{b/O} \times \mathbf{p}^b_{P/b}$ represents the additional linear velocity experienced by point P due to the rotational motion of the body. This term becomes more significant for points located farther away from the rotation axis, such as antennas or sensors mounted away from the vessels center of gravity.

In many applications, it is necessary to express all velocities in a common reference frame. The velocity of the body origin expressed in the navigation (NED) frame is obtained by rotating the body frame velocity using the rotation matrix R^n_b:

$$\mathbf{v}^n_{b/O} = R^n_b \mathbf{v}^b_{b/O}.$$

Conversely, the body frame velocity can be obtained from the navigation frame velocity by

$$\mathbf{v}^b_{b/O} = (R^n_b)^\top \mathbf{v}^n_{b/O},$$

since $(R^n_b)^\top = R^b_n$. The rotation matrix R^n_b is derived from the vehicles orientation, represented either by Euler angles or a unit quaternion.

Similarly, for a sensor or point P rigidly fixed to the vehicle (i.e., $\mathbf{v}^b_{P/b} = 0$), the velocity expressed in the navigation frame simplifies to

$$\mathbf{v}^n_{P/O} = R^n_b \left(\mathbf{v}^b_{b/O} + \boldsymbol{\omega}^b_{b/O} \times \mathbf{p}^b_{P/b} \right),$$

where $\mathbf{p}^b_{P/b}$ defines the sensors position in the body frame. This formulation provides a consistent way to compute the motion of any fixed point on the vehicle in both body and navigation frames, ensuring proper alignment between inertial and navigation states.

4.4.6 Angular Acceleration in Euler Representation

The angular acceleration $\boldsymbol{\alpha}^b_{b/O}$ describes the rate of change of the body's angular velocity vector $\boldsymbol{\omega}^b_{b/O} = [p, q, r]^\top$, which represents the instantaneous rotation of the body about its own x, y, and z axes. Physically, $\boldsymbol{\alpha}^b_{b/O}$ captures how fast the rotational motion itself is changing, and it directly relates to the torques acting on the body through the rotational dynamics equations.

When the body orientation is represented using Euler angles (ϕ, θ, ψ), the angular acceleration can be obtained by differentiating the Euler angle to angular velocity relationship:

$$\boldsymbol{\alpha}^b_{b/O} = T^{-1}(\phi, \theta, \psi)\ddot{\boldsymbol{\Theta}} + \dot{T}^{-1}(\phi, \theta, \psi)\dot{\boldsymbol{\Theta}}$$

where $\dot{\boldsymbol{\Theta}} = [\dot{\phi}, \dot{\theta}, \dot{\psi}]^\top$ and $\ddot{\boldsymbol{\Theta}} = [\ddot{\phi}, \ddot{\theta}, \ddot{\psi}]^\top$ are the first and second derivatives of the Euler angles. The first term represents the direct contribution from angular rate changes, while the second term captures coupling effects caused by the nonlinearity of rotational kinematics, for instance, when pitch or roll rates influence yaw acceleration.

In differential form, the time derivative of angular velocity is expressed as

$$\dot{\boldsymbol{\omega}}^b_{b/O} = \boldsymbol{\alpha}^b_{b/O}$$

which defines angular acceleration in the same frame as $\boldsymbol{\omega}^b_{b/O}$. To express angular acceleration in another reference frame, such as the navigation frame, it can be transformed using the rotation matrix:

$$\boldsymbol{\alpha}^n_{b/O} = R^n_b \boldsymbol{\alpha}^b_{b/O}$$

where R_b^n is the rotation from the body frame to the navigation (NED) frame.

In practical navigation and control systems, the angular acceleration $\boldsymbol{\alpha}_{b/O}^b$ is rarely measured directly. It is usually approximated by differentiating the body angular velocity $\boldsymbol{\omega}_{b/O}^b$ obtained from gyroscopes or inferred from rigid body dynamics. Maintaining consistent frame alignment between body and navigation frames is essential for accurate attitude propagation and torque computation.

4.4.7 Angular Acceleration in Quaternion Representation

The quaternion representation provides a singularity free alternative to Euler angles for describing rotational motion. The unit quaternion $q = [q_0, q_1, q_2, q_3]^\top$ defines the orientation of the body frame relative to the navigation frame, where q_0 is the scalar part and $[q_1, q_2, q_3]^\top$ is the vector part.

The time evolution of the quaternion is governed by the rotational kinematics equation:

$$\dot{q} = \frac{1}{2}\Omega(\boldsymbol{\omega}_{b/O}^b)q$$

where $\boldsymbol{\omega}_{b/O}^b = [p, q, r]^\top$ is the angular velocity in the body frame and $\Omega(\boldsymbol{\omega}_{b/O}^b)$ is the quaternion rate matrix:

$$\Omega(\boldsymbol{\omega}_{b/O}^b) = \begin{bmatrix} 0 & -p & -q & -r \\ p & 0 & r & -q \\ q & -r & 0 & p \\ r & q & -p & 0 \end{bmatrix}$$

Differentiating this expression gives the quaternion based angular acceleration:

$$\ddot{q} = \frac{1}{2}\Omega(\boldsymbol{\alpha}_{b/O}^b)q + \frac{1}{2}\Omega(\boldsymbol{\omega}_{b/O}^b)\dot{q}$$

which captures both the direct contribution of angular acceleration $\boldsymbol{\alpha}_{b/O}^b$ and the coupling effect from the current angular velocity.

In vector form, the time derivative of angular velocity is expressed as

$$\dot{\boldsymbol{\omega}}_{b/O}^b = \boldsymbol{\alpha}_{b/O}^b$$

which defines angular acceleration in the same frame as $\boldsymbol{\omega}^b$. To express angular acceleration in another reference frame, such as the navigation frame, it can be transformed using the rotation matrix:

$$\boldsymbol{\alpha}_{b/O}^n = R_b^n \boldsymbol{\alpha}_{b/O}^b$$

where R_b^n is the rotation matrix from body to navigation frame.

To ensure q remains a valid rotation, it must satisfy the unit norm constraint:

$$\|q\| = 1$$

which is typically enforced by periodic normalization during numerical integration like so:

$$q = \frac{q}{\|q\|}$$

This quaternion based formulation is computationally efficient and avoids the gimbal lock issue inherent in Euler angle representations. For this reason, quaternion propagation is widely preferred for representing orientation and modeling angular dynamics in navigation and control systems.

4.4.8 Linear Acceleration

The linear acceleration of a point P on a rigid body consists of translational, tangential, centripetal, and coriolis components. It is expressed as

$$\mathbf{a}_{P/O}^b = \mathbf{a}_{b/O}^b + \mathbf{a}_{P/b}^b + \boldsymbol{\alpha}_{b/O}^b \times \mathbf{p}_{P/b}^b + \boldsymbol{\omega}_{b/O}^b \times (\boldsymbol{\omega}_{b/O}^b \times \mathbf{p}_{P/b}^b) + 2\boldsymbol{\omega}_{b/O}^b \times \mathbf{v}_{P/b}^b$$

where $\mathbf{a}_O$ is the acceleration of the reference point (often the body origin or center of mass), $\boldsymbol{\alpha}$ is the angular acceleration vector, $\boldsymbol{\omega}$ is the angular velocity vector, $\mathbf{r}_{OP}$ is the position vector from the reference point O to the point P, and $\mathbf{v}_{rel}$ is the velocity of P relative to the rotating frame. The first term represents the translational acceleration of the reference point, the second term the tangential acceleration caused by the change in rotational rate, the third term the centripetal acceleration directed toward the axis of rotation, and the fourth term the coriolis acceleration arising from relative motion within the rotating frame.

When the body fixed frame is used, the total acceleration of a point expressed in the navigation (NED) frame is related through the rotation matrix as

$$\mathbf{a}_b^n = R_b^n \mathbf{a}_b^b,$$

and conversely,

$$\mathbf{a}_b^b = (R_b^n)^\top \mathbf{a}_b^n.$$

These transformations ensure consistent representation between body frame sensor measurements and navigation frame quantities.

In inertial navigation systems, accelerometers measure the specific force, which is the total acceleration excluding gravity, given by

$$\mathbf{f}^b = \mathbf{a}_b^b - R_n^b \mathbf{g}^n,$$

where $\mathbf{g}^n = [0, 0, g]^\top$ is the gravity vector in the NED frame. The specific force $\mathbf{f}^b$ is the quantity directly measured by IMUs and forms the basis for estimating velocity and position through numerical integration.

4.4.9 Linear Acceleration

The linear acceleration of a point P on a rigid body consists of translational, tangential, centripetal, and Coriolis components. It is expressed in the body frame as

$$\mathbf{a}_{P/O}^b = \mathbf{a}_{b/O}^b + \mathbf{a}_{P/b}^b + \boldsymbol{\alpha}_{b/O}^b \times \mathbf{p}_{P/b}^b + \boldsymbol{\omega}_{b/O}^b \times (\boldsymbol{\omega}_{b/O}^b \times \mathbf{p}_{P/b}^b) + 2\boldsymbol{\omega}_{b/O}^b \times \mathbf{v}_{P/b}^b$$

Here, $\mathbf{a}_{b/O}^b$ is the translational acceleration of the body origin, $\mathbf{a}_{P/b}^b$ is acceleration of point P on sensor relative to body, $\boldsymbol{\alpha}_{b/O}^b$ is the angular acceleration, $\boldsymbol{\omega}_{b/O}^b$ is the angular velocity, $\mathbf{p}_{P/b}^b$ is the position of point P relative to the body origin, and $\mathbf{v}_{P/b}^b$ is the velocity of P relative to the body frame.

Each term represents a specific physical contribution to the total acceleration. The first term $\mathbf{a}_{b/O}^b$ describes the translational acceleration of the body origin. Second term $\mathbf{a}_{P/b}^b$ denotes the local acceleration of P relative to the body frame, which becomes zero for fixed sensors, while $\boldsymbol{\alpha}_{b/O}^b \times \mathbf{p}_{P/b}^b$ represents the tangential acceleration caused by changes in rotational rate. The term $\boldsymbol{\omega}_{b/O}^b \times (\boldsymbol{\omega}_{b/O}^b \times \mathbf{p}_{P/b}^b)$ accounts for the centripetal acceleration directed toward the axis of rotation, and $2\boldsymbol{\omega}_{b/O}^b \times \mathbf{v}_{P/b}^b$ captures the Coriolis acceleration arising from relative motion within the rotating body.

For a sensor rigidly mounted on the body, where $\mathbf{v}_{P/b}^b = 0$ and $\mathbf{a}_{P/b}^b = 0$, the equation simplifies to

$$\mathbf{a}_{P/O}^b = \mathbf{a}_{b/O}^b + \boldsymbol{\alpha}_{b/O}^b \times \mathbf{p}_{P/b}^b + \boldsymbol{\omega}_{b/O}^b \times (\boldsymbol{\omega}_{b/O}^b \times \mathbf{p}_{P/b}^b).$$

This formulation is widely used to compute the acceleration at sensor locations such as IMUs or GNSS antennas mounted away from the vehicles center of mass.

To express the acceleration in the navigation (NED) frame, the transformation is performed using the rotation matrix R_b^n:

$$\mathbf{a}_{P/O}^n = R_b^n \mathbf{a}_{P/O}^b,$$

and conversely,

$$\mathbf{a}_{P/O}^b = (R_b^n)^\top \mathbf{a}_{P/O}^n.$$

In inertial navigation systems, accelerometers measure the specific force, which is the total acceleration excluding gravity:

$$\mathbf{f}^b = \mathbf{a}_{b/O}^b - R_n^b \mathbf{g}^n,$$

where $\mathbf{g}^n = [0, 0, g]^\top$ is the gravity vector in the NED frame. The specific force $\mathbf{f}^b$ is the quantity directly measured by IMUs and serves as the basis for estimating velocity and position through numerical integration.

4.5 ASV Motion and Measurement Models

Fossen's 6-DOF marine craft model (as one formulation).

INS-based kinematic model (as your chosen implementation).

GNSS (as the aiding/measurement model, 2 antenas can get position and Heading).

And the comparison/discussion between Fossen vs. INS (why INS chosen). Basically select good tool for the job you want to do, even though Fossen model is great, for our use case in SLAM a INS model will suffice, especially i combination with aiding measurements. Fossens is good but INS is simpler and in SLAM case for building a local map for Data Processing step it is more than god enough. like yes in this case Fossne might be more accurate if we had more system parameters that we know but then we would have to estimate system parameters, and for that we need system identification at that point. And then we have to consider is it worth doing full system identification for 1% maybe more performance? SO jesjes INS model it is because of it.

when talking on ins midel Presentation AND the book for designing the model is very nice ESKF DO NOT USE CONSTANT BIAS model because dificult to readjust (!!!Dangerous!!!), better is Wiener process ORRRR if we know something about inertial sensor use Gaus Markov Model BEST :D (Aka when INS chapter talk about bias modeling, the constant (BAD), wiener (Overly pesimistic), Gaus Markov (More realistic)) esrrir state modeling is there to keep track of different between nominal and ground truth state

INS: p is v in world frame v is acel in worldframe (thats why transform R) + gravity q is quarternion propagate explain biases models choice build from constant to wirner to gaus markov basically need to do more modeling

Chapter on: - Fossen Marineccraft model as motion model - Inertial Navogation System and motion model - GNNS as aiding measuremnt model

4.6 Numerical Solvers

Expricit and Impricit solvers, focus only on explicit solvers as tehy are real-time, while implicit is forstable simulations only. Explicit is good enough for navigation use case

X.Y.1 Newton–Raphson Method

For solving nonlinear equations (used in implicit models or iterative equilibrium solving).

Brief explanation: derivative-based root finding, convergence properties.

Example: solving steady-state forces or trim conditions in ASV model.

X.Y.2 Runge–Kutta Integration (RK4)

For numerical time integration of dynamic equations (e.g., Fossen's 6-DOF model).

Derivation of RK4 update rule.

Discussion on trade-offs: stability vs. accuracy vs. computational cost.

Possibly mention comparison with Euler or semi-implicit schemes.

5 State Estimation

intro to it Bayes filter KF + Talk about propagation Newton and also RK4 method :) EKF ESKF UKF (WIth bias and all :D f() IMU model for that, + since quarternion, have to normalize after update the quarternion angle yesyes) + 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. TUNING (NIS and other stuff to keep in mind) State Variables at the end ie the ESKF variables that wahta we end up with

EKF vs ESKF (1 kind of state vs 2 kinds of states) IMPORTANT: ESKF Covariance is of error state NOT nominal state!!!!! explain why error state better than EKF

ESKF: A and G are just aprocimations of DISCRETIZATION, but you can get a better dicreteziced system using kayley hamilton for these error matrix system. however more run time witch might not be good, so thats why usually we use the A and G defined already lol

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, the problem is posed as a Maximum A Posteriori (MAP) problem, seeking the most likely trajectory and landmarks given all measurements and priors.With standard assumptions such as Gaussian noise and first order linearization, the MAP problem becomes a nonlinear least squares problem solved iteratively. At each iteration, the residuals are linearized around the current estimate and an increment that improves the state is computed. This "correction" is what lets SLAM reduce drift, enforce loop closures, and keep the map consistent. [18]

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. [19][20]

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.

An estimate of the unknowns is maintained, denoted θ (robot poses, and landmarks). Each measurement yields a residual that indicates how far the current estimate is from the expected sensor reading. Close to the current estimate, a small nudge in θ gives an approximate change in the residuals. This is a first order (linear) approximation. Stacking all residuals together, that approximation looks like this [21]:

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

Here A is the Jacobian, it describes 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" to be computed. [21]

The update step chooses $\Delta\theta$ that reduces all residuals as much as possible. The best practice here is to use MAP approach. In this section, measurement noise is assumed Gaussian (this is an approximation, not always true, and non Gaussian cases are discussed later). After linearizing the residuals around the current estimate, the MAP problem becomes a least-squares fit [21]:

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

Optimize this estimate so that change in $\Delta\theta^\star$ is equla to 0, ie linearize. When linearized, equation (2) can be simplified to a so called normal equation [21]:

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

This equation system can be then be solved by Cholesky decomposition of $A^T A$ or by optimization algorithms that will be discussing down bellow. Solve this linear system for $\Delta\theta$, then update/correct the estimate [21]:

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

For stability on harder problems Levenberg-Marquardt damping can be added, however the core idea stays the same across these optimizer algorithms. [22]

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 iterative methods are prefer, 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 lies in how the problem is represented and updated. 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 computing a good estimate, defining simple models for robot motion and sensor observations is crucial. The motion model describes state evolution, and the measurement model describes sensor readings. States are x_i for robot poses, controls are u_i, and measurements are z_k for landmarks. 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 this uncertainty model can be switched to robust or heavy tailed noise model 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 is modeled as 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.

Predictions should match measurements. In a perfect world, every residual (prediction minus measurement) would be zero. In practice, model errors and sensor noise make the residuals 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.

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 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 [21] where linearized forms of the system becomes:

$$\begin{aligned} f_i(x_{i-1}, u_i) - x_i &\approx (F_i^{i-1} \Delta x_{i-1} - \Delta x_i) - a_i \\ F_i^{i-1} &:= \left. \frac{\partial f_i(x_{i-1}, u_i)}{\partial x_{i-1}} \right|_{x_{i-1}^0} \\ a_i &= x_{i-1}^0 - f_i(x_{i-1}^0, u_i) \end{aligned} \tag{3}$$

$$
\begin{aligned}
h_k(x_{i-1}, u_i) - z_k &\approx (H_k^{i_k} \Delta x_{i_k} - J_k^{j_k} \Delta l_{j_k}) - c_k \\
H_k^{i_k} &:= \left. \frac{\partial h_k(x_{i_k}, l_{j_k})}{\partial x_{i_k}} \right|_{(x_{i_k}^0, l_{j_k}^0)} \\
J_k^{j_k} &:= \left. \frac{\partial h_k(x_{i_k}, l_{j_k})}{\partial l_{j_k}} \right|_{(x_{i_k}^0, l_{j_k}^0)} \\
c_k &= z_k - h_k(x_{i_k}^0, l_{j_k}^0)
\end{aligned}
\tag{4}
$$

Plug the linearized odometry (3) and measurement (4) 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, the errors are not judged 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 (5) 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, factor can be visualized as 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:

$$
\Delta\theta^* = \arg\min_{\Delta\theta} \left\{ \sum_{i=1}^{M} \|F_i^{i-1} \Delta x_{i-1} + G_i^i \Delta x_i - a_i\|_{\Lambda_i}^2 + \sum_{k=1}^{K} \|H_k^{i_k} \Delta x_{i_k} + J_k^{j_k} \Delta l_{j_k}) - c_k\|_{\Gamma_k}^2 \right\} \tag{5}
$$

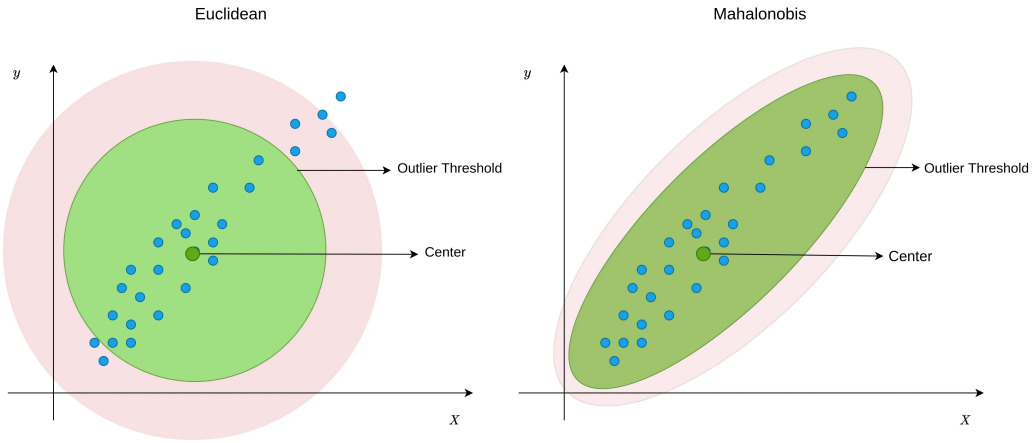


Figure 22: Euclidean vs. Mahalanobis residual contours. *Left:* isotropic (equal) weighting yields circular inlier regions. *Right:* a covariance Σ skews and scales the contours into an ellipse whose axes/tilt follow the noise correlations, whitening this with $\Sigma^{-1/2}$ maps this ellipse back to a circle.[23]

Mahalanobis distance is just "error measured in the units of its noise" (See Figure 22). If a residual has high variance, it should be penalized less. If two components are correlated, they should not be treated 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 SLAM cost function, 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. Penalties are smaller along noisy directions and larger where the sensor data is precise.

Equation (5) 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, first step is to "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 the covariance symbols can be dropped, stack the Jacobians into one big sparse matrix A, stack the prediction errors into b, and solve the standard least squares problem (6).

$$\Delta\theta^\star = \arg\min_{\Delta\theta} \|A\Delta\theta - b\|^2 \tag{6}$$

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 (6) 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 [21]:

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

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)

The linearized SLAM subproblem is solved 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 [21] 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 [21] shows that after QR the equation is:

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

Put simply, once QR factorization is performed, the error splits into two parts. To make the total error as small as possible, set the first term to zero and solve:

$$R\Delta\theta^\star = d \tag{7}$$

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 equation (7) is solved 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:

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

A dense inverse is never built. In reality, when entries of the uncertainty Σ are needed, solve small triangular systems with $R^\top$ and R, and read only the pose blocks and the pose to landmark blocks of interest. Since R is sparse and triangular, this is fast and stable, and it avoids forming $A^\top A$. (See 9.2.8 Data Association from R)

9.2.4 Matrix Factorization for building QR (Givens rotations)

Here *Givens rotations* is used 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:

$$G(\varphi) = \begin{bmatrix} \cos\varphi & \sin\varphi \\ -\sin\varphi & \cos\varphi \end{bmatrix} \tag{8}$$

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 (8) 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 23 down bellow for a visual of one Givens step)

As the algorithm sweeps the columns of A, the matrix is transformed into the upper triangular form, this is R, and the full Q doesn't need to be formed to get to the result. 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. Later it is shown that new whitened rows can be appended beneath the current R, then a short sequence of row rotations re-triangularizes R. In other words, updates operate directly on R and b, and A is bypassed for incremental steps.

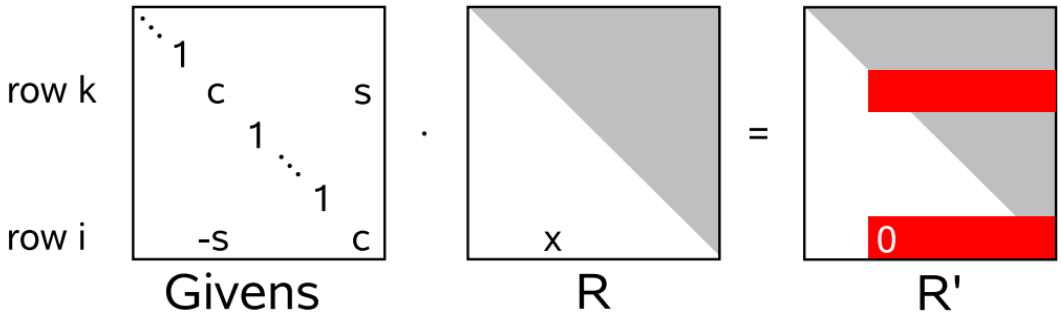


Figure 23: One Givens step in QR. The entry marked "x" is eliminated by rotating two rows, only the entries shown in red are modified, and the exact pattern depends on sparsity. Repeating this column wise (left to right) turns the matrix into an upper triangular R. Apply the same rotation to the b vector to keep the least squares system consistent.[21]

In order to make R upper triangular, φ value must be chosen precisely to zero out a single sub diagonal entry in preliminary matrix, either be it A matrix on batch step or R matrix on iterative steps. The rotation angle φ is computed from the two entries 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 the following answer, where $\alpha = x = a_{kk}$ and $\beta = y = a_{ik}$:

$$(\cos\varphi,\ \sin\varphi) = \begin{cases} (1,\ 0), & \text{if } \beta = 0, \\ \left(-\dfrac{\alpha}{\beta}\dfrac{1}{\sqrt{1+(\alpha/\beta)^2}},\ \dfrac{1}{\sqrt{1+(\alpha/\beta)^2}}\right), & \text{if } |\beta| > |\alpha|, \\ \left(\dfrac{1}{\sqrt{1+(\beta/\alpha)^2}},\ -\dfrac{\beta}{\alpha}\dfrac{1}{\sqrt{1+(\beta/\alpha)^2}}\right), & \text{otherwise.} \end{cases} \quad \text{with } \alpha := a_{kk},\ \beta := a_{ik}. \tag{9}$$

These coefficients in equation (9) give the same rotation as (8).

Givens rotations guarantee that the (i, k) entry of the working matrix becomes zero, and they preserve lengths. First the two numbers $[x,\ y]^\top$ are rotated. When embedded in the full matrix, the same rotation is applied to the affected parts of the two rows and to 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

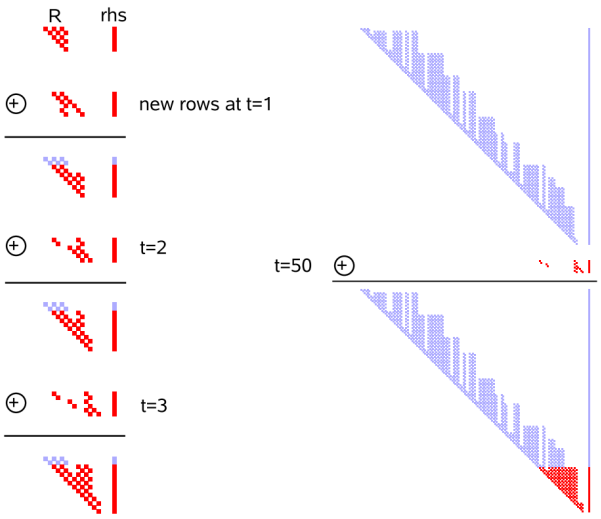


Figure 24: Incremental update of the factored system. A new whitened row $w^\top$ and RHS (Right Hand Side) entry γ are appended beneath the current R and d. A short sequence of Givens rotations restores the upper triangular form, yielding updated R' and d'. Unchanged entries are shown in light color, only a small stencil is touched each step, so update cost stays bounded.[21]

After the initial QR factorization, 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, 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 A is NOT rebuild. Instead, 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, re-triangularize locally with Givens rotations (8). Only touching 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. Repeat the process until the entire bottom row is zero and the matrix is upper triangular again. The equation would look something like this:

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

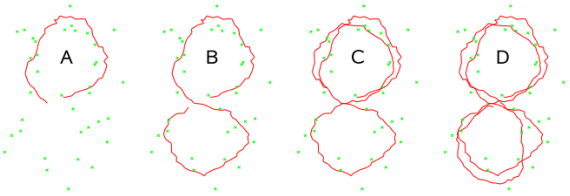
While the matrix is rotates, apply the same rotations to the right hand side so that the least squares system stays consistent. Here d is the transformed RHS (Right Hand Side) before the update and γ is the new whitened RHS entry that pairs with $w^\top$. After the rotations, the top block becomes the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

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

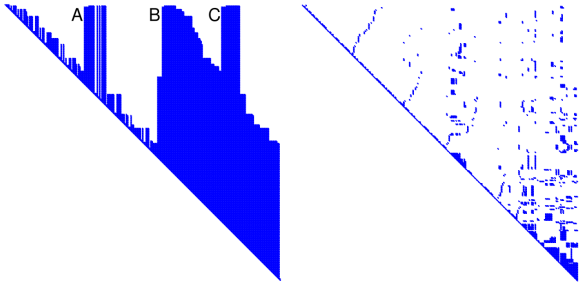
Intuitively, the new row $w^\top$ is "folded up" into the triangular structure by a short chain of 2x2 rotations that only touch the connected variables, everything else is left alone. Then get the correction by a fast back substitution on the updated matrix R' and vector d':

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

9.2.6 Loop Closure

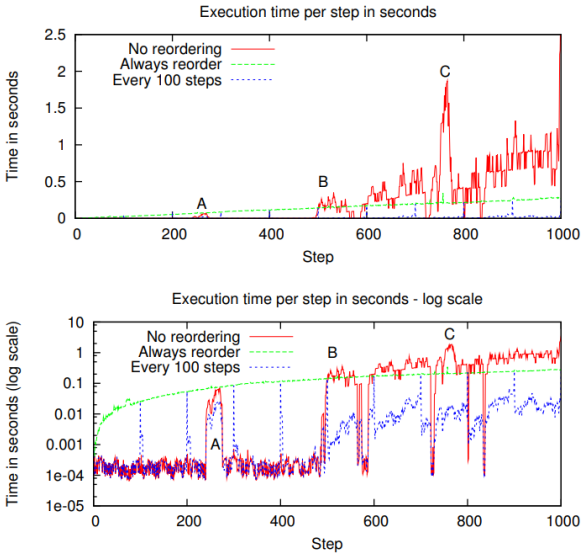


(a) Simulated double 8 loop at key loop closure moments.[21]



(b) Upper triangular factor R after several closures shows fill-in.[21]

(c) The same R after variable reordering (COLAMD) becomes sparser again.[21]



(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.[21]

Figure 25: Effect of loop closures and variable reordering (iSAM).[21]

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. This can be fixed by variable reordering. The goal is to pick a new elimination order that preserves sparsity. In practice 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 (8) with that order [21]. Reordering costs time because the factor must be rebuilt with a new permutation, but it pays back by making subsequent updates cheap again.

Because reordering is expensive, it is not done at every step. Instead, reordering is performed periodically every N steps (eks, 50 - 200) so the cost stays predictable. In marine AUV and ASV runs this keeps compute bounded. Between reorders incremental updates are fast and local. After a loop closure, one spike occurs (reorder + refactor), then the system returns 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 valid. The QR and update methods assume the system is locally linear around the current estimate, but with angles, three dimensional motion, and nonlinear sensor data, that approximation drifts as the robot moves. If it is not refreshed, increments grow, the optimizer biases the map, and loop closures can fail. The remedy is to re-linearize and 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 new factors are always linearized, and older ones are refreshed only when needed.

In iSAM the practical schedule is to run incremental updates between maintenance cycles, then perform a full re-linearization every N steps. Between cycles, old factors are not re-linearized, new factors are whitened and inserted, and R is updated incrementally. At the cycle boundary after N steps, the entire problem is re-linearized at the current estimate, the full Jacobian A is rebuilt conceptually, variables are reordered with COLAMD to restore sparsity, and the system is refactored using equation (8) to obtain

a fresh triangular R. For this reason variable reordering is usually grouped with batch re-linearization at the same N step.

N must be chosen 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, the system stops often to re-linearize and refactor, wasting CPU and power, and reducing real-time throughput.

9.2.8 Data Association from R

For data association, Mahalanobis based approach is often used 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 22). 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, 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 the full dense $\Sigma = (R^\top R)^{-1}$ is not required for data association, only the local covariances influenced by the measurement. These covariances can be extracted directly from the square root information matrix R. Pose blocks and pose to landmark blocks are available online as well. 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 [21], 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:

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

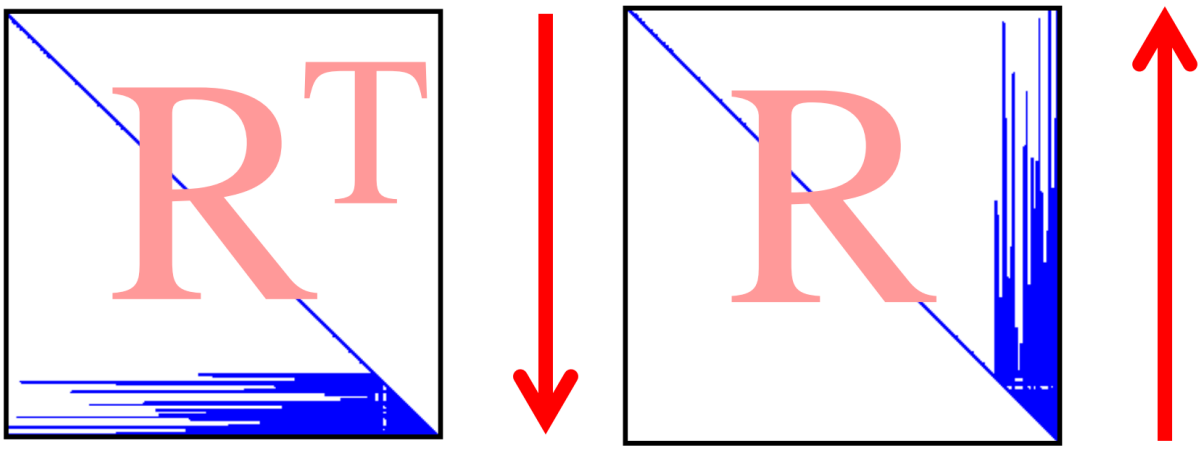
where $B = \begin{bmatrix} 0 \\ I_{d_x} \end{bmatrix}$ simply selects the last pose block of size d_x. Because R is upper triangular and B is zero above the last block, the forward solve gives:

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

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 "solve up" then "solve down" on R for the last pose, and 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 26)



(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 26: Grab the needed covariances in two quick steps: first solve "down" with $R^\top$, then solve "up" with R. Only touch entries near the last pose block, so each update stays fast and cheap.[21]

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 [21] 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 the true Γ (measurement noise) is usually unknown. The iSAM recipe is to choose Γ conservatively so the algorithms doesn't get overly confident. That keeps things safe but can make the gate too tight, causing the data association to reject more matches than it should. Later, a more reliable way to approximate Γ from measurement characteristics is described. It avoids overconfidence and uses cues such as sonar range and resolution and landmark confidence, so the gate is realistic and not risky.

Exact landmark covariances (on demand)
When accuracy is needed (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:

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

Needed entries can be extracted without forming the full inverse by solving for two triangular matrixes:

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

The solve uses only the "nonzero" entries of R, so computation touches only the parts that matter, not the whole matrix. 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. This exact method

should only be used when its really needed (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 Γ measurement noise. Uncertainty used in Data Association should reflect the sensor, for example a sonar for scanning the sea floor. With sonar the measurement noise can grow with range, and the transducer resolution can set per axis variances $N(0, \Sigma_{sonar})$. The landmark detection can also contribute its own uncertainty $N(0, \Sigma_{landmark})$. In practice these are combined to form the prediction uncertainty for the residual that is scored. In that case the effective covariance is:

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

This can be more informative than a one size fits all setting. However, this approach requires care. If the noise terms are too confident, the data association gate becomes too tight, matches become brittle, and the system can become unstable. Nevertheless, it often yields more accurate maps and a better estimate of the robot's track.

9.2.9 Algorithm

At each time step absorb new information, linearize around the current estimate, solve a small least squares, and update. Periodically 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 (6)).
2. **Linearize at the current guess θ:** turn the nonlinear motion/measurement models into local linear ones using the Jacobians in (3) and (4). This gives the summed Mahalanobis cost (5), which after whitening becomes one least squares problem (6).
3. **Keep a triangular system up to date (QR):** append the new (whitened) rows and apply a few Givens rotations (8) 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 (7) 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 (8) 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. [22, 24]

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.

Let the variables be robot poses $x_1, \dots, x_M$ and landmarks $l_1, \dots, l_N$, and let $\Theta = \{x_1, \dots, x_M, l_1, \dots, l_N\}$. According to the iSAM2 papers [22, 24] 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. How each factor is modeled depends on the situation. Because odometry and measurements are uncertain in the real world, factors should be represented with a probabilistic model. 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 22 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.

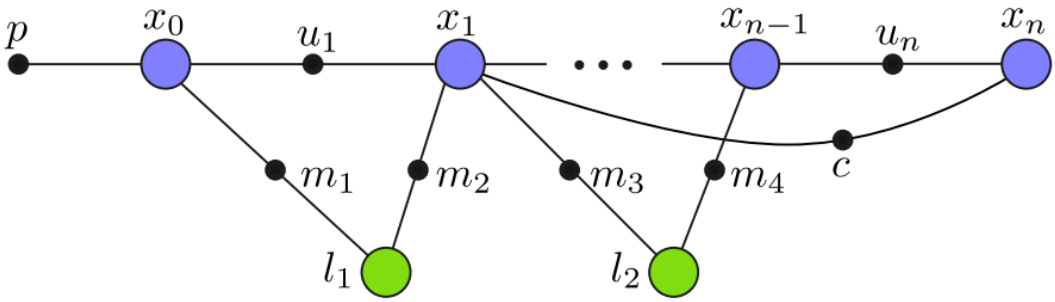


Figure 27: Picture from iSAM2 paper [22] describes factor graph formulation of the SLAM problem. Variable nodes (large circles) are poses $x_0, \ldots, x_n$ and landmarks l_1, l_2. Factor nodes (small solid circles) represent a prior p, odometry u_i, landmark measurements m_i, and a loop closure constraint c. This approach 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

Here SLAM is represented 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 27). 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 it states which variables it touches.

Under Gaussian measurement models as 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 22)

To combine all information, simply multiply the factor likelihoods. Products are awkward to optimize, instead 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. This can be write in compact form as:

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

Here *Theta* collects all unknowns, such as poses, landmarks, and other parameters. Each factor h_i depends only on a small subset Θ_i, so each residual couples only those variables. After linearization this gives a sparse system, because most variables do not appear together in any single residual.

This is our MAP function in nonlinear form.

This nonlinear cost is not solved in one shot because the measurement transform $h_i(\cdot)$ is nonlinear (because of angles, ranges, bearing-only sensors, etc...). As in the iSAM system, linearize around the current estimate and solve iteratively. Choose a current estimate Θ^0 and look for a small update $\Delta\theta$ that improves the fit. For each factor 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 $\Sigma_i^{-1/2}$, stack all linearized factors into one sparse least-squares problem:

$$\Delta\Theta^\star = \arg\min_{\Delta\Theta} \left(-log\left(f(\Delta\Theta)\right)\right) = \Delta\theta^\star = \arg\min_{\Delta\theta} \ \|A\,\Delta\theta - b\|^2 \tag{10}$$

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.

Here the linear least squares problem is solved in a numerically stable manner. 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 can be solved by back substitution. Both routes are standard, in practice QR factorization methods are preferred for stability.

After solving for $\Delta\theta$, 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), 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. 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 from iSAM but in a graph form. Eliminate variables to get a Bayes net and store it as a Bayes tree. This lets iSAM2 update only the cliques touched by new factors instead of refactoring everything. The next subsection shows how this is done.

9.3.4 From Factor Graphs to Bayes Networks

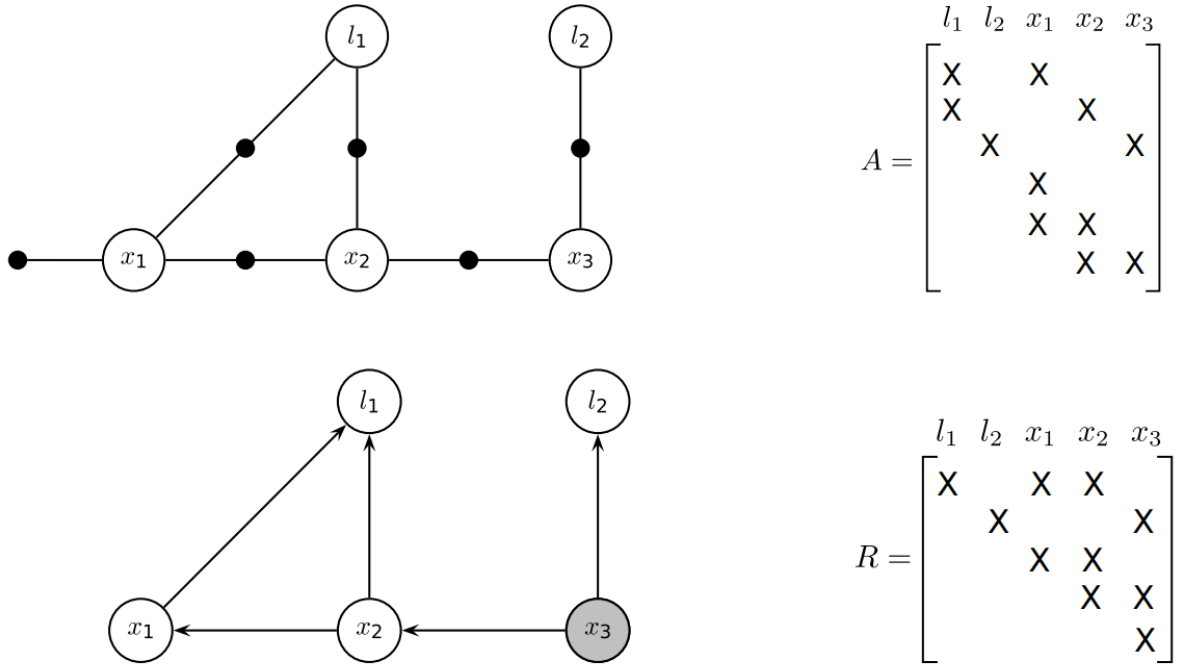


Figure 28: Picture from iSAM2 paper [22]. (Top) factor graph and associated Jacobian A for a small SLAM example with poses x_1, x_2, x_3, landmarks l_1, l_2, and a prior on x_1 (Bottom) the chordal Bayes net and the square root factor R obtained by eliminating in the order l_1, l_2, x_1, x_2, x_3. The last eliminated variable (the root) is shaded. [22]

Starting from the least squares form in (10), the whitened measurement matrix A is factored as in iSAM. An elimination order is chosen, (for the example in Picture 28 it is l_1, l_2, x_1, x_2, x_3), and sparse QR with Givens rotations (8) is applied. 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. Drawing arrows from the parent variables to the current variable turns the same structure into 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 28 shows.

Instead of running sparse QR on the whitened Jacobian, the same result can be obtained by eliminating variables directly on the factor graph using bipartite elimination game methods [25]. Starting from (10), 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 ordering, the purely graphical elimination produces the same dependency pattern and the same square root information factor as numeric QR factorization. This procedure forms a Bayes network with chordal properties, and in the matrix view yields an upper triangular R with $R^\top R = A^\top A$ (see Picture 28). The advantage is that A does not need to be assembled, and Givens rotations and explicit QR factorization are avoided. The computation stays on the graph, the ordering controls fill, and the desired chordal structure is obtained. 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 [25].

This observation is the bridge to iSAM2. A chordal Bayes network groups naturally into cliques, this can be represented as a Bayes tree. The next subsections uses this tree, rather than a single global R, to support local updates and avoid global re-factorizations.

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. [22, 24]

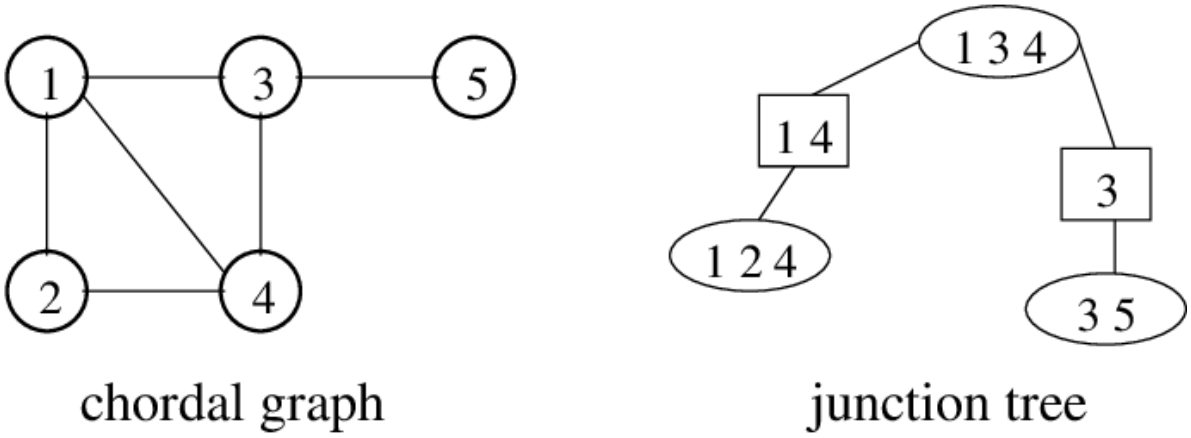


Figure 29: Picture from Robert Castelo paper [26] shows an example of a simple Chordal graph (Left side) and corresponding small cliques/junction tree (Right side). Eliminating in a good order produces a chordal Bayes net whose moralized graph can be grouped into cliques/junction trees. This is the structure that keeps R matrix sparse.

When linearizing and eliminating 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 that was 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 solving $R\Delta\theta = d$ by back substitution, 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. Consecutive row blocks of R that share the same separator form a clique. Cliques that share a separator are connected to form a Bayes tree. Each clique stores frontal conditionals, child given the parent separator, and each edge carries only the shared separator variables. The tree encodes the same numeric information as R, but it is organized by locality rather than by a single global ordering.

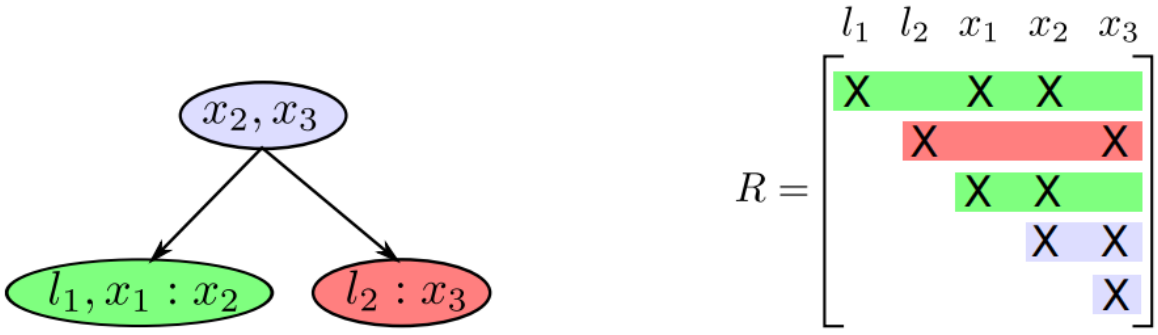


Figure 30: Picture from Bayes tree paper [24] shows Bayes tree (left) and its matching rows in the square root factor R (right). Colors indicate which contiguous row blocks of R belong to each clique.

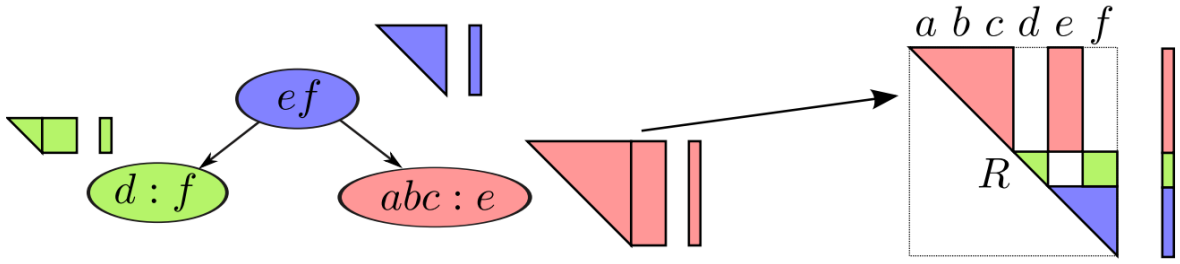


Figure 31: Picture from iSAM2 paper [22] shows each clique contains the conditional of its frontal variables given the separator. The same entries appear in the corresponding rows/columns of R. Back substitution on R mirrors evaluating the tree from leaves to root, while marginal queries follow the few 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. Thus R can be viewed as a Bayes tree data structure in matrix form. 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

iSAM2 never rebuilds 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.

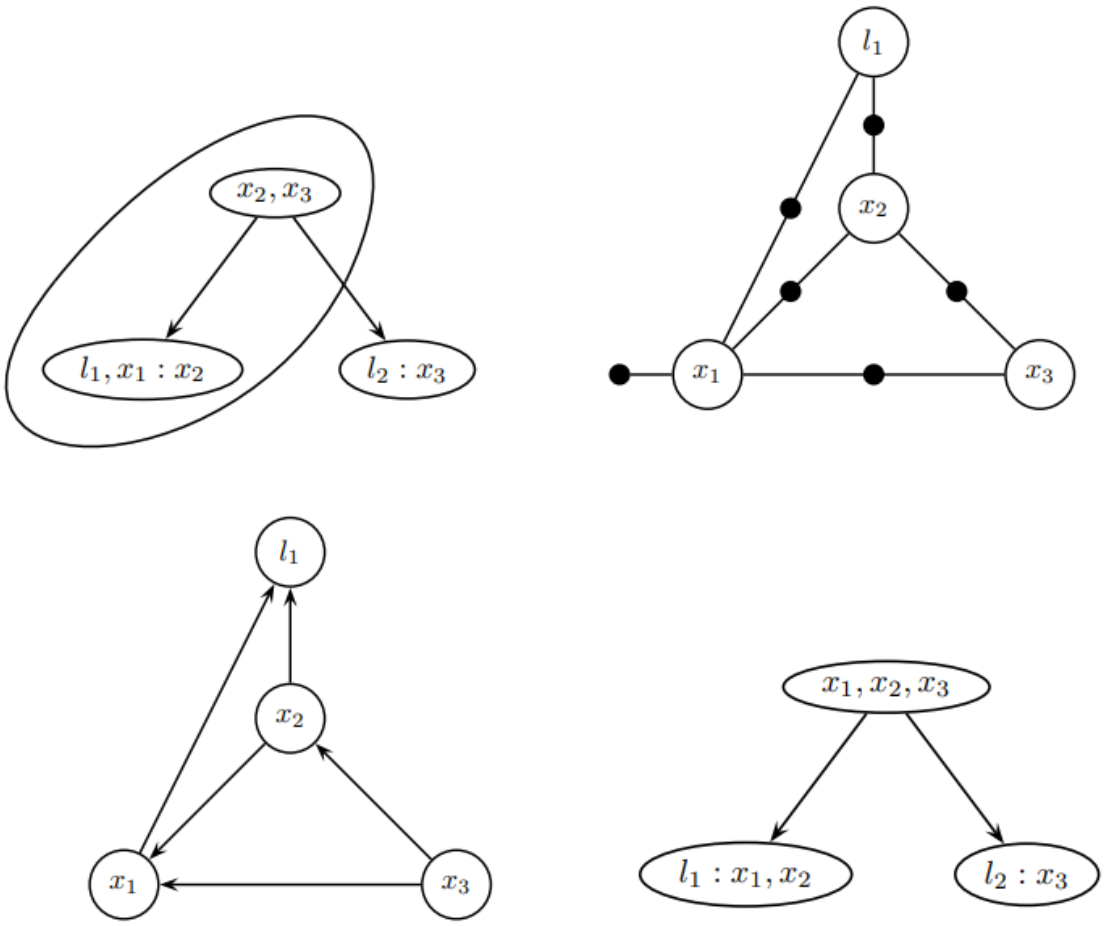


Figure 32: Picture taken from Bayes tree paper [24] shows how to update a Bayes tree with a new factor.
(Top left) Affected cliques when adding a factor between x_1 and x_3, the right branch is unaffected.
(Top right) Local factor graph rebuilt from those cliques plus the new factor.
(Bottom left) Local 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 32)

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 by re-linearizing only the variables in the affected region when needed.

9.3.7 Loop Closure and Incremental Reordering

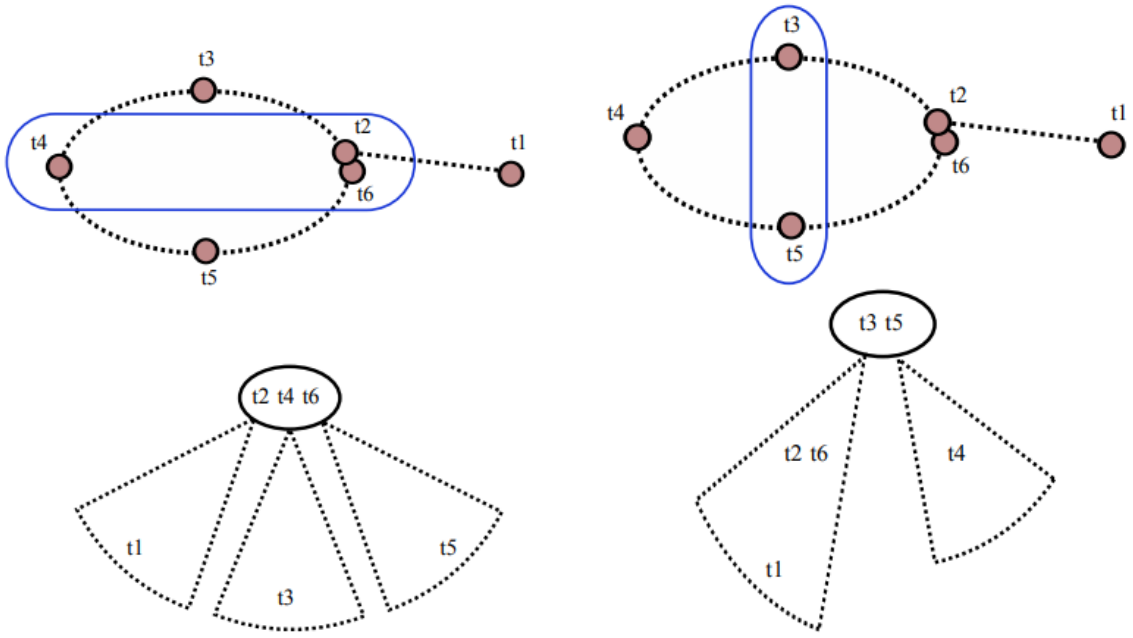


Figure 33: Picture taken from Bayes tree paper [24] shows loop closure with incremental reordering. Two batch optimal orderings (top) yield different Bayes trees (bottom). For online operation, the ordering should keep the newest variables near the root, so future updates affect only a small subtree.

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. The affected top region is converted back to a local factor graph, the loop closure factor is added, and the region is re-eliminated to produce a new local chordal Bayes net and Bayes subtree. The unchanged subtrees are then reattached at the matching separators. This keeps updates local and predictable, unlike iSAMs global R re-factorization after loop closure. [22, 24]

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 the newest pose blocks are kept at the end of the order while letting COLAMD algorithm find a sparse order for the rest. This produces small, stable updates at each step, even when loops close. [24]

Batch orderings found by nested dissection (or similar heuristics) can look equally good for a one time solve because they produce comparable sparsity. For online SLAM the next update matters. The preferred ordering therefore leaves the newest poses at or near the Bayes tree root, so the next odometry or loop closure factor changes only a small subtree. If the newest pose lies deep in the tree, the same update must rewrite many cliques. For this reason iSAM2 uses constrained reorderings, keeping recent variables last in the order near the root and letting the heuristic arrange the rest. This keeps updates local and cheap. [24]

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 while saving compute time, and when needed it is reapplied 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

iSAM2 stops doing periodic global "re-linearize everything". Instead iSAM2 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 iSAM2 recomputes 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. [22, 24]

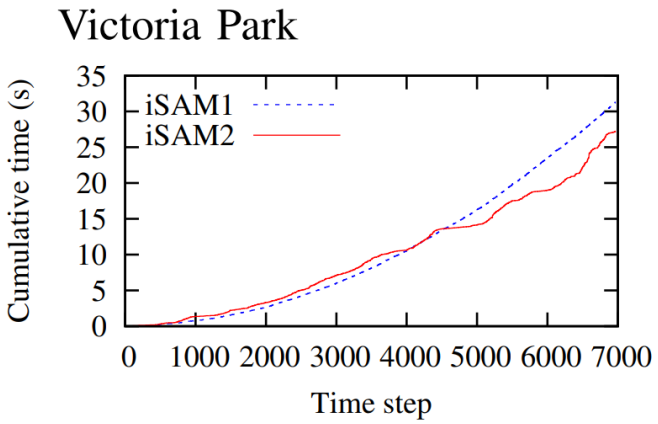
How it works in practice is simple. Always keep a running correction vector Δ from the latest linear solve. If a variable's change is tiny, treat its current linearization point as "good enough" and do not touch it. If a variable's change is larger than a small threshold (call it β), mark that variable for re-linearization. Then mark the cliques that contain those variables and their ancestors in the Bayes tree. Only those cliques are rebuilt, 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. [24]

Solving for $\Delta\theta$ is also done only where needed. First solve on the modified top of the tree. Then walk into children only if any parent update exceeded a small propagation threshold (call it α). This "only follow when it matters" rule 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 34)

Victoria Park

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 34: Picture from the Bayes tree paper [24]. 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 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. Retain the freshest odometry and a few diverse loop closures, and replace discarded constraints by a light summary on the small separator set where they would have entered the tree. Intuitively, 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, iSAM2 scores 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 this combines 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 iSAM2 absorbs 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 [22, 24])

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 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, sparsification/keyframing is often needed 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 for modeling estimation problems as factor graphs and solving them efficiently with batch optimizers and with incremental iSAM2 and Bayes tree methods. The core idea is to represent knowledge as a product of small factors, each involving only a few variables, and then exploit sparsity and variable elimination to compute fast and stable MAP estimates. In practice this gives a single, uniform toolkit for SLAM in 2D and 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 [27, 22, 24].

Design philosophy: graph first, values separate: GTSAM cleanly separates the *"model"* from the *"state"*. A `NonlinearFactorGraph` stores factors (priors, odometry, landmark/vision measurements, loop closures), while a `Values` container holds one current assignment to the unknowns (poses, landmarks, intrinsics, etc...). The estimate can be changed without touching the graph, and factors can be added or removed without invalidating unrelated variables. This mirrors the math formulation $f(\Theta) = \prod_i f_i(\Theta_i)$. The graph captures structure and sparsity. A particular Θ provides an assignment that can be evaluated or optimized. [27]

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. [27]

Batch optimization and linear algebra under the hood: In GTSAM, SLAM is posed as a `NonlinearFactorGraph` with initial `Values`. At each optimizer step, GTSAM linearize the factors at the current estimate to obtain a linear system. 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, add small typed factors, GTSAM handles linearization and sparse elimination, and with `iSAM2` it updates only where needed. [27]

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 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. [27]

What using GTSAM looks like: Create a `NonlinearFactorGraph` and a `Values` with first initial 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, keep an `ISAM2` object, call `update(newFactors, newValues)` each step, then get the current best estimate with `isam.calculateEstimate()`.When uncertainty is required for data association or validation, compute marginal covariances only for the variables that matter, no giant matrix inverse needed. `Pose2`/`Pose3`, `Point2`/`Point3`, camera calibration, and robust noise models can be mixed in the same graph, and all follow the same pattern. [27]

Educational and practical: GTSAM is designed to match the mathematics in the iSAM and iSAM2 papers [21, 22]. Each measurement becomes a small, typed *"factor"*. Current guesses live in a `Values` container that understands poses and rotations, and the solvers make sparsity and variable ordering visible. GTSAM provides clear examples and MATLAB and Python bindings, so ideas can be tested quickly and results plotted with minimal 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 [21, 22] 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 that can be read, extended, and trusted. [27]

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 distant covariance queries, expect higher cost because more of the tree is touched. Use approximate or local covariances for routine gating, and reserve exact queries for critical decisions. With clever design these gaps can be resolved. [27]

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 needed for real-time operation. GTSAM provides a practical implementation of iSAM2 ideas for direct use in SLAM systems. [27]

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] Bjørnar Reitan Hogstad. "Side-Scan Sonar Imaging and Error-State Kalman Filter Aiding Unmanned Underwater Vehicle (UUV) to Autonomy". Master's thesis. Norwegian University of Science and Technology (NTNU), Feb. 2022. URL: https://ntnuopen.ntnu.no/ntnu-xmlui/handle/11250/3032962 (visited on 10/11/2025).

[4] Cao Vo Tran Luan. "Design and Implementation of microAmpere: An Advanced USV for Autonomous Docking Research". In: (Jan. 2025).

[5] Reimers Henrik. "NMPC and blueboat hardware". In: (Jan. 2025).

[6] Safran Navigation & Timing. "STIM300 Datasheets". In: (May 2025). URL: https://safran-navigation-timing.com/document/stim300-datasheets/ (visited on 10/11/2025).

[7] u-blox. "ZED-F9P Module". In: (May 9, 2025). URL: https://www.u-blox.com/en/product/zed-f9p-module (visited on 10/11/2025).

[8] Rosell Jan, Palomo Leopold, and Avellaneda. "Appendix: ROS 2 - Introduction to ROS". In: (2023). URL: https://sir.upc.edu/projects/rostutorials/appendixRos2/index.html (visited on 10/15/2025).

[9] Thor Inge Fossen. "Fossen's Marine Craft Model". In: (). URL: https://fossen.biz/html/marineCraftModel.html (visited on 10/12/2025).

[10] Brekke Edmund. *Fundamentals of Sensor Fusion: Target Tracking, Navigation and SLAM*. NTNU, 2020.

[11] Zeitlhöfler Julian. "Nominal and observation-based attitude realization for precise orbit determination of the Jason satellites". In: (June 2019). URL: https://www.researchgate.net/figure/llustrates-the-principle-of-gimbal-lock-The-outer-blue-frame-represents-the-x-axis-the_fig4_338835648 (visited on 10/12/2025).

[12] David. "How Quaternions Produce 3D Rotation". In: (June 2019). URL: https://penguinmaths.blogspot.com/2019/06/how-quaternions-produce-3d-rotation.html (visited on 10/12/2025).

[13] MATLAB. "SLERP - Spherical Linear Interpolation". In: (). URL: https://se.mathworks.com/help/nav/ref/quaternion.slerp.html (visited on 10/12/2025).

[14] Atanasov Nikolay. "ECE276A: Sensing & Estimation in Robotics: Lecture 12: SO(3) and SE(3) Geometry and Kinematics". In: (). URL: https://natanaso.github.io/ece276a2020/ref/ECE276A_12_SO3_SE3.pdf (visited on 10/12/2025).

[15] Wagner Tobias. "Position information and their coordinate systems WGS84 and ETRS". In: (Mar. 25, 2024). URL: https://genesys-offenburg.de/support/application-aids/gnss-basics/position-information/ (visited on 10/12/2025).

[16] SBG Systems. "NED (North-East-Down) Frame". In: (). URL: https://www.sbg-systems.com/glossary/ned-north-east-down/ (visited on 10/12/2025).

[17] Xin Hu. "Robust nonlinear control design for dynamic positioning of marine vessels with thruster system dynamics". In: (Oct. 2018). URL: https://www.researchgate.net/figure/North-east-down-frame-and-vessel-fixed-frame-26_fig1_325388618 (visited on 10/12/2025).

[18] 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).

[19] 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).

[20] 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).

[21] 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).

[22] 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).

[23] 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).

[24] 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).

[25] 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).

[26] 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).

[27] 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).