Localization and mapping

From SLAM to EKF

Roberto Masocco
roberto.masocco@uniroma2.it

University of Rome Tor Vergata
Department of Civil Engineering and Computer Science Engineering

June 12, 2024

TOR VERGATA
UNIVERSITY OF ROME
School of Engineering

Roadmap

Roadmap

The perception problem
Definition

To be able to operate autonomously, a robot must continuously answer the following questions:

- **Where am I?**
- **What is this place?**

Thus, it must be able to **perceive** the environment, gathering information useful to:

- **localize itself**, *i.e.*, continuously estimate its **pose** as **both position and orientation** in 3D space;
- **map the environment**, *i.e.*, build a **representation** of the environment useful to **navigate** within it.

The perception problem
Challenges

The perception problem is challenging because:

- the environment may be **partially observable**, *i.e.*, the robot can only perceive a **subset** of it, and need to update its information in real time;
- the environment may be **dynamic**, *i.e.*, it can change over time;
- measurements are always subject to **noise**.

The perception problem is usually solved by **sensor fusion**, *i.e.*, combining information from **multiple sensors** to obtain a more **accurate** and **reliable** estimate of the environment, possibly accounting for **sensor faults**.

The perception problem

Tools for the job

The tools that robots use to gather **measurements** from the environment are called **sensors**.

They can be classified as:

- **proprioceptive**, *i.e.*, measuring robotic interaction with the environment (*e.g.*, **encoders**, **GPS**, **IMUs**);
- **exteroceptive**, *i.e.*, measuring the environment itself (*e.g.*, **cameras**, **LiDARs**, **radars**);
- **interoceptive**, *i.e.*, measuring the robot's internal state.

As any other measurement tool, sensors are based on **physical principles** and **energy exchanges**, translating the information they gather into **electrical signals** that can be acquired and/or processed by a computer.

They are usually characterized by at least:

- a **digital** or **analog encoding** of the measurement;
- a **frame of reference** in which the measurement is expressed;
- **accuracy** and **uncertainty** parameters.

The perception problem
Tools for the job

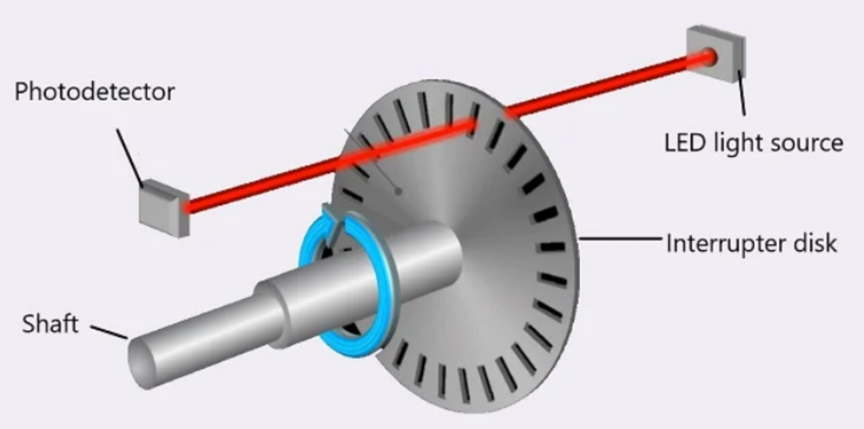

Figure 1: Rotary encoder working principle.

The perception problem

Tools for the job

Figure 2: GPS module for drones.

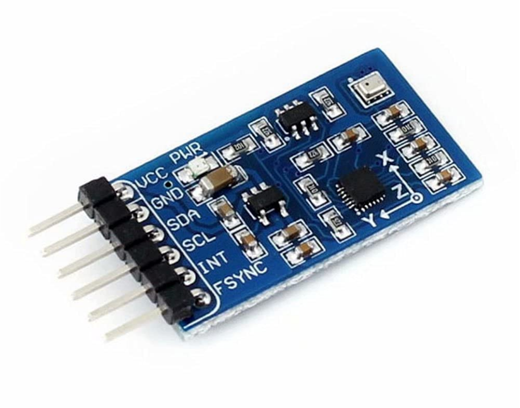

Figure 3: Inertial Measurement Unit (IMU).

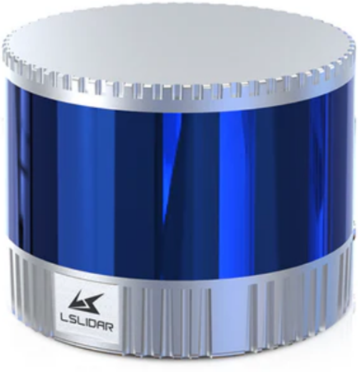

Figure 4: Light Detection and Ranging (LiDAR) sensor.

Figure 5: ZED 2i stereo camera.

Roadmap

The mapping problem

Definition

Using **exteroceptive sensors**, a robot can gather information about the environment, which can be used to build a **map** of it.

A **map** is a **representation** of the environment, in a format that the robot can **understand**, **parse**, and **store**.

The utimate goal of mapping is twofold:

- to enable the robot to **localize itself** within the environment;
- to enable **safe navigation** of the robot within the environment.

The mapping problem
Challenges

Given the utility requirement of a map, the mapping problem must be **continuously solved in real time**.

Thus, it is challenging because:

- routines must be **efficient**, and run at a sufficiently **high rate**;
- the map must be **accurate**, and **reliable**;
- the map must be in a format that is as much **easy to load and parse** as possible, taking up as little **memory** as possible;
- the map must stay **up-to-date**, and **consistent** with the environment.

The mapping problem
Tools for the job

The most important tool for the mapping problem is the **occupancy grid**, a representation of the environment as a **grid** of **cells**, each of which is **occupied** or **free**.

The occupancy grid is a **probabilistic** representation, where each cell is associated with a **probability** of being occupied or free.

The occupancy grid is usually built using **LiDAR** or **camera** depth data, and is updated in real time as the robot moves.

The occupancy grid is the most common representation for **local** and **global** maps.

To efficiently store an occupancy grid, **tree-like data structures** are often employed (*e.g.*, **octrees**).

The mapping problem

Tools for the job

The second most important class of tools are **navigation algorithms**, which use the map to plan a **safe** and **efficient** path for the robot to follow.

The definition of such algorithms involves **geometry**, as well as **optimization** and **search** techniques.

They usually rely on two mathematical subjects:

- **topology**, to define the **connectivity** of the map (*e.g.*, Voronoi tessellation);
- **graph theory**, to define the **best way** of moving from one free cell to another (*e.g.*, Dijkstra's, $A^\star$ algorithms).

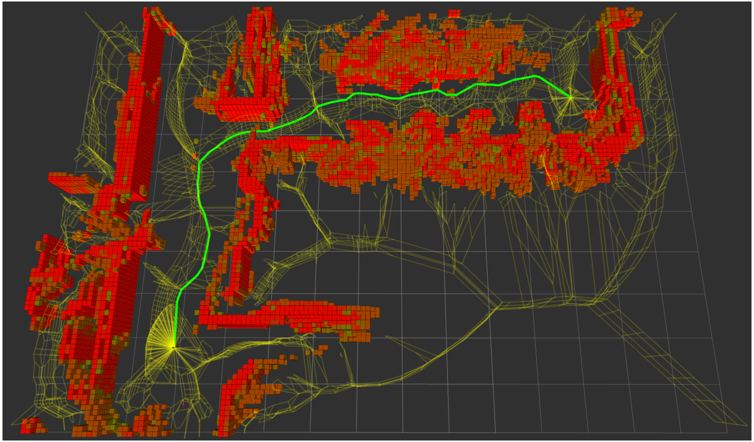

Figure 6: Mapping and navigation algorithms execution.

Roadmap

1. The perception problem
2. The mapping problem
3. Simultaneous Localization And Mapping
4. Common interfaces
5. The tf2 library
6. The Extended Kalman Filter

Simultaneous Localization And Mapping

Definition

The **SLAM** problem is a **chicken-and-egg** problem: a robot must **localize itself** within an environment, while **mapping** it.

The ultimate goal of SLAM is to enable the robot to **navigate** within the environment, while **updating** the map as it moves.

The robot must be able to **efficiently** and **accurately** build a map of the environment, while **localizing itself** within it, with respect to either the origin of the map or the starting point of the robot itself (*i.e.*, with either some or none **prior notion of the environment**).

To solve this problem, **sensor fusion** techniques are often employed, mixing data coming from **heterogeneous sensors** and accounting for **sensor faults**.

Typically, SLAM algorithms rely on recognizable **features** of the environment to build the map and detect motion.

While a SLAM system builds a **map** of the environment, it can detect whether it is exploring a zone that it has **already visited**.

When this happens, the system can **close the loop** by **matching** the current "view" with a **previous one**, thus **correcting** the map and the robot's pose.

This process is called **loop closure** and is a key feature of SLAM algorithms.

Loop **detection** and **closing** must also be performed in real time, as well as the subsequent **corrections**; efficient optimization algorithms and data structures are crucial.

Figure 7: Loop closure of the ORB-SLAM2 algorithm.

Sensors:

- **LIDARs** for direct environment mapping through depth information;
- **Cameras** to infer the environment structure through image processing;
- **IMUs** to account for the robot's motion and correct sampled data;
- **GNSS** to have a slow, but reliable global position estimate.

Plus all the **algorithms** and the **mathematical tools** we discussed in the context of **mapping**.

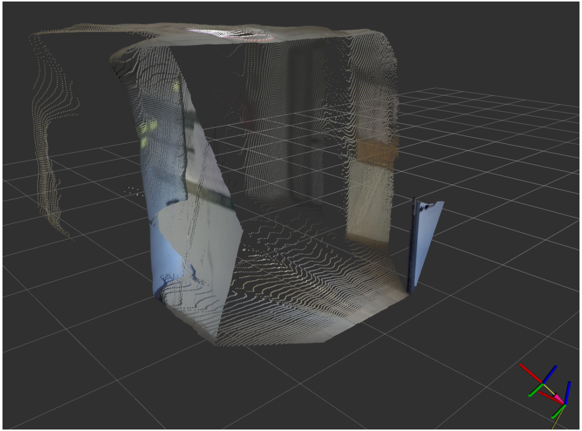

Figure 8: 3D point cloud generated by a stereoscopic camera.

Cartographer is a **2D** SLAM algorithm developed by Google.

Meant for **offline** floor plan generation, it has been ported in **ROS** for mobile robot navigation.

It used **LiDAR** laser scans, corrected by **IMU** data, to build 2D **submaps** of the **surrounding** environment.

Such maps are then used to **build and update** a **global map**, gluing submaps together and optimizing the result.

Features are particular **environment structures** that are used to detect loop closures and estimate the robot's trajectory.

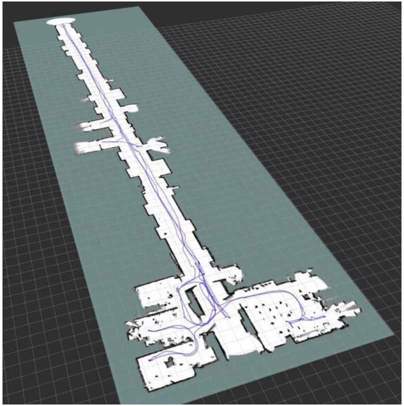

Figure 9: Execution of the Cartographer SLAM algorithm.

Visual SLAM

ORB-SLAM

ORB-SLAM is a **visual** SLAM algorithm that uses **monocular** or **stereo** cameras to build a map of the environment.

It relies on **binary ORB features** to detect and match points in the environment, building a **covisibility graph** of **keyframes**: relevant views for pose estimation.

The graph, and thus, the map and the camera pose estimate, are optimized with **bundle adjustment**.

Latest versions also include **IMU** samples in the bundle adjustment cost function.

Figure 10: ORB-SLAM2 algorithm execution on a ROS 2 dataset (bag).

Roadmap

Common interfaces

A standard ROS 2 installation offers many **interface packages** (*i.e.*, messages), to provide **standard data types** to communicate sensor measurements and related data.

The most important are:

- `sensor_msgs`, for **sensor measurements**;
- `geometry_msgs`, for **geometric data**;
- `nav_msgs`, for **navigation data**.

It is suggested to **always use these message types**, plus common best practices, to ensure full interoperability between sensor drivers and localization and mapping algorithms.

Try to `ros2 interface show` these messages to understand their structure!

`sensor_msgs`

Common interfaces for sensors

- `Imu`
- `JointState`
- `CameraInfo` and `Image`
- `LaserScan`
- `PointCloud2`
- `Temperature`
- `NavSatFix`
- `Illuminance`
- ...

geometry_msgs

Common interfaces for algebraic data

- Vector3Stamped
- QuaternionStamped
- PoseWithCovarianceStamped
- TwistWithCovarianceStamped
- TransformStamped (used by tf2!)
- AccelWithCovarianceStamped
- ...

nav_msgs

Common interfaces for navigation data

- Odometry (Header, body (child) frame ID, PoseWithCovariance, TwistWithCovariance)
- Path
- OccupancyGrid
- ...

Roadmap

Rigid transformations

When a robot moves in space, it is important to keep track of its **position** and **orientation** with respect to a **reference frame**.

Sensors measuring this information, as well as many more, are **mounted** on the robot, in fixed positions and orientations.

To process these measurements, they must first be **transformed** from the **body frame** into a common reference frame, usually called:

- **world frame** (world origin), in the case of **global localization**;
- **local frame**, or **odom frame** (robot starting point), in the case of **local localization**.

Such **rigid transformations** are **isometries**. They must be applied to almost every sensor measurement, and are usually **composable**.

We would like the middleware to provide tools to do this almost automatically...

The tf2 library

`tf2` is the **standard ROS 2 library** to handle rigid transformations.

It allows to:

- **broadcast** and **listen** to transformations, thanks to appropriate buffering subscribers and publishers;
- **transform** any kind of sensor data from one frame to another, making efficient computations in C++ code relying on the `Eigen` mathematical library;
- broadcast **robot descriptions** from `URDF` files, listing links and joints and how they are connected, resulting in a **tree-like structure**;
- **command-line tools** to inspect the current tree status, and broadcast custom transformations.

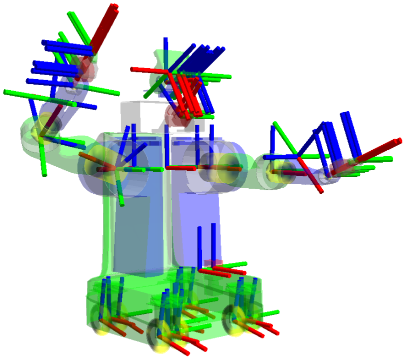

Figure 11: Example of robot description with tf2.

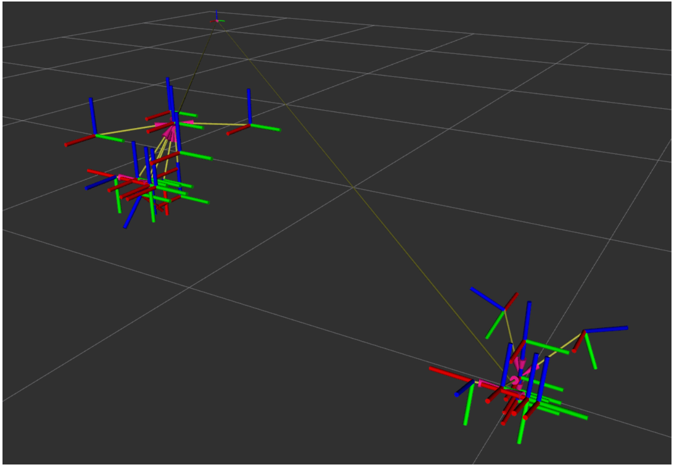

Figure 12: Broadcasted robot descriptions, plus real-time transformations given by localization systems.

Roadmap

Odometric reconstruction

An elementary sensor fusion technique

Suppose you have a robot equipped with m **sensors** producing **measurements** q_i, $i = 1, \dots, m$, and a **motion model** that predicts the **state** of the robot x_k at time k given the state at time $k-1$ and the control input u_k applied to the robot.

You can use these data to perform an **odometric reconstruction**: integrating the motion model with sensor data and the **estimated state** of the robot.

Suppose you have a robot equipped with m **sensors** producing **measurements** q_i, $i = 1, \ldots, m$, and a **motion model** that predicts the **state** of the robot x_k at time k given the state at time $k-1$ and the control input u_k applied to the robot.

You can use these data to perform an **odometric reconstruction**: integrating the motion model with sensor data and the **estimated state** of the robot.

Measurements are affected by noise, that this technique does not account for.

Noise may even be due to **physical phenomena** like slippage, vibrations, or sensor faults.

Suppose that $m = 2$. At any given time, the **measurement** of a quantity x can be modeled as a **Gaussian random variable** centered around the **true value** of x with a certain **standard deviation**:

- $q_1 \sim \mathcal{N}(x, \sigma_1^2)$;
- $q_2 \sim \mathcal{N}(x, \sigma_2^2)$.

Having distinct sensors makes these variables **independent**. This is reasonable because:

- of well-known results, *e.g.*, the Central Limit Theorem;
- the **average** is the real value of x if we rule out **systematic errors**, by, *e.g.*, calibrating the sensor;
- the **variance** models the **dispersion** of the measurements around the central value.

Note that, by relying on **independence**, **Bayes' theorem**, and the **properties** of the Gaussian distribution:

- $q_1 \sim \mathcal{N}(x, \sigma_1^2) \Rightarrow x \sim \mathcal{N}(q_1, \sigma_1^2)$;
- $q_2 \sim \mathcal{N}(x, \sigma_2^2) \Rightarrow x \sim \mathcal{N}(q_2, \sigma_2^2)$;

i.e., x **is also a Gaussian random variable** centered around a measurement.

Modelling noise
Combining variables

Similarly, we can combine the two measurements q_1 and q_2 to obtain a **better estimate** of x, as $x \sim \mathcal{N}(q, \sigma^2)$, where:

- $q = \frac{\sigma_2^2 q_1 + \sigma_1^2 q_2}{\sigma_1^2 + \sigma_2^2}$;
- $\sigma^2 = \frac{\sigma_1^2 \sigma_2^2}{\sigma_1^2 + \sigma_2^2}$;

i.e., x is a new Gaussian random variable centered around a **weighted average** of the measurements, with a **variance** that decreases as the **uncertainty** of the measurements decreases.

Combining two measurements

Step by step

Suppose you have to update an **estimate** $\hat{x}$ of a quantity x and its **variance** Σ^2 with two measurements q_1 and q_2:

1. Measurement q_1 arrives: $\hat{x}_1 = q_1$, $\Sigma_1^2 = \sigma_1^2$.
2. Measurement q_2 arrives.
3. $\hat{x}_{1,2} = \frac{\Sigma_1^2 q_2 + \sigma_2^2 \hat{x}_1}{\Sigma_1^2 + \sigma_2^2} = \hat{x}_1 + \frac{\Sigma_1^2}{\Sigma_1^2 + \sigma_2^2}(q_2 - \hat{x}_1)$.
4. $\Sigma_{1,2}^2 = \frac{\Sigma_1^2 \sigma_2^2}{\Sigma_1^2 + \sigma_2^2} = (1 - \frac{\Sigma_1^2}{\Sigma_1^2 + \sigma_2^2})\Sigma_1^2$.

Suppose now that you can receive m measurements in **subsequent steps**. Then:

$$\hat{x}_{k+1} = \hat{x}_k + V_{k+1}(q_{k+1} - \hat{x}_k), \tag{1}$$

$$\Sigma_{k+1}^2 = (1 - V_{k+1})\Sigma_k^2, \tag{2}$$

where:

$$V_{k+1} = \frac{\Sigma_k^2}{\Sigma_k^2 + \sigma_{k+1}^2}, \tag{3}$$

and $V_{k+1} \in (0, 1)$. It is a **sampled approximation**.
Note what happens when the new $k+1$-th measurement is either **very accurate** ($\sigma_{k+1}^2 \ll \Sigma_k^2$) or **very noisy** ($\sigma_{k+1}^2 \gg \Sigma_k^2$).

Suppose you have a **dynamic system** with a **discrete-time model**:

$$x_{k+1} = ax_k + bu_k + \omega_k, \tag{4}$$

with $\omega_k \sim \mathcal{N}(0, \sigma_\omega^2)$, and **measurements that depend on the state**:

$$z_k = cx_k + \nu_k, \tag{5}$$

where $\nu_k \sim \mathcal{N}(0, \sigma_\nu^2)$ is the **noise**.

Initialize the **estimate** as $\hat{x}_0$ and the **covariance** as P_0.
Then, relying on the same properties as the previous cases:

$$\hat{x}^-_{k+1} = a\hat{x}_k + bu_k, \tag{6a}$$

$$P^-_{k+1} = a^2 P_k + \sigma^2_\omega, \tag{6b}$$

$$K_{k+1} = \frac{P^-_{k+1} c}{c^2 P^-_{k+1} + \sigma^2_\nu}, \tag{6c}$$

$$\hat{x}_{k+1} = \hat{x}^-_{k+1} + K_{k+1}(z_{k+1} - c\hat{x}^-_{k+1}), \tag{6d}$$

$$P_{k+1} = (1 - K_{k+1}c)P_{k+1}. \tag{6e}$$

(6a)-(6b) compose the **prediction step** of the **a priori estimate**, (6c) is the **Kalman gain**, and (6d)-(6e) are the **correction step**, applying the **innovation term** to compute the **a posteriori estimate**.

The **vectorial case** is a **generalization** of the scalar case, using **vectors** and **covariance matrices**.
Suppose that now you have this model:

$$x_{k+1} = Ax_k + Bu_k + \omega_k, \tag{7}$$

where A and B are **matrices**, $\omega_k \sim \mathcal{N}(0, Q_\omega)$, and the **measurement** is:

$$z_k = Cx_k + \nu_k, \tag{8}$$

where $\nu_k \sim \mathcal{N}(0, Q_\nu)$.

Kalman filter

The vectorial case

By applying the same reasoning as before, with appropriate algebraic manipulations:

$$\hat{x}_{k+1}^{-} = A\hat{x}_k + Bu_k, \tag{9a}$$

$$P_{k+1}^{-} = AP_kA^T + Q_\omega, \tag{9b}$$

$$K_{k+1} = P_{k+1}^{-}C^T(CP_{k+1}^{-}C^T + Q_\nu)^{-1}, \tag{9c}$$

$$\hat{x}_{k+1} = \hat{x}_{k+1}^{-} + K_{k+1}(z_{k+1} - C\hat{x}_{k+1}^{-}), \tag{9d}$$

$$P_{k+1} = (I - K_{k+1}C)P_{k+1}^{-}, \tag{9e}$$

where I is the identity matrix.

Suppose you have a **nonlinear model**:

$$x_{k+1} = f(x_k, u_k, \omega_k), \tag{10a}$$
$$z_k = h(x_k, \nu_k), \tag{10b}$$

with random variables as before.

In this case, we can resort to a **linearization** to **reapply the same reasoning**, but we lose **convergence properties**.

Nonetheless, the **Extended Kalman Filter** (EKF) is a **widely used** heuristic state estimation technique, achieving **good results** in many practical scenarios.

Consider two subsequent time instants k and $k+1$.
Given the **current estimate** $\hat{x}_k$ at time k and the **prediction** $\hat{x}_{k+1}^-$ at time $k+1$, we can write:

$$f(x_k, u_k, \omega_k) \approx f(\hat{x}_k, u_k, 0) + F_k(x_k - \hat{x}_k) + W_k\omega_k, \tag{11a}$$

$$h(x_{k+1}, \nu_{k+1}) \approx h(\hat{x}_{k+1}^-, 0) + H_{k+1}(x_{k+1} - \hat{x}_{k+1}^-) + L_{k+1}\nu_{k+1}, \tag{11b}$$

where $F = \partial f / \partial x$, $W = \partial f / \partial \omega$, $H = \partial h / \partial x$, and $L = \partial h / \partial \nu$ are Jacobian matrices.
Grouping some **known terms**, these lead to a **linear system**:

$$x_{k+1} = F_k x_k + f_0(\hat{x}_k, u_k) + W_k\omega_k, \tag{12a}$$

$$z_{k+1} - h_0(\hat{x}_{k+1}^-) = H_{k+1}x_{k+1} + L_{k+1}\nu_{k+1}, \tag{12b}$$

where (12b) can be considered a measurement, subtracting a **known term** from the **true measurement**.

By applying the same reasoning as before, with appropriate algebraic manipulations:

$$\hat{x}_{k+1}^{-} = f(\hat{x}_k, u_k, 0), \tag{13a}$$
$$P_{k+1}^{-} = F_k P_k F_k^T + W_k Q_\omega W_k^T, \tag{13b}$$
$$K_{k+1} = P_{k+1}^{-} H_{k+1}^T (H_{k+1} P_{k+1}^{-} H_{k+1}^T + L_{k+1} Q_\nu L_{k+1}^T)^{-1}, \tag{13c}$$
$$\hat{x}_{k+1} = \hat{x}_{k+1}^{-} + K_{k+1}(z_{k+1} - h(\hat{x}_{k+1}^{-}, 0)), \tag{13d}$$
$$P_{k+1} = (I - K_{k+1} H_{k+1}) P_{k+1}^{-}. \tag{13e}$$

It is a **good compromise** between **accuracy** and **computational complexity**.